Obsah

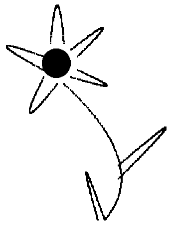
Předmluva

Skriptum *Paralelní architektury a programy* je primárně určeno jako učební text pro předmět *Paralelní programování* využívaný zejména v druhé etapě studijního programu oboru *Informatika a výpočetní technika.* Další předpokládané využití je pro předmět *Programové struktury* zařazený do studijního programu *první etapy inženýrského studia na FAV.* Skriptum může sloužit jako doplňková literatura i pro další obory inženýrského studia na FAV (zejména Kybernetika a řídicí technika a Matematicko-fyzikální inženýrství) i na dalších inženýrských fakultách ZČU.

Čtení skripta vyžaduje znalosti na úrovni základního kursu informatiky (předměty *Počítače a programování 1 a 2, Programovací techniky, Teoretická informatika*) v první etapě inženýrského studia na FAV, zejména pak alespoň rámcovou znalost programovacích jazyků **C**, **Pascal** a **Fortran**. Dále se předpokládají znalosti principů operačních systémů. Pro čtenáře, kteří neabsolvovali předmět *Základy operačních systémů*, jsou zařazeny pasáže objasňující principy řízení paralelního výpočtu na úrovni operačního systému a pojetí procesů v operačním systému **Unix**. První kapitola skript je přehledová a poskytuje určitou všeobecnou orientaci v problematice paralelních počítačů a paralelních výpočtů.

Autoři se podíleli na zpracování skript následovně : *Doc.Ing.Karel Ježek, CSc.* zpracoval kapitoly 2 a 3. *Přemysl Matějovic, prom. mat.* zpracoval kapitolu 6 a podkapitoly 1.6 a 4.4. *Doc.Ing.Stanislav Racek, CSc.* zpracoval kapitoly 1 (kromě 1.6), 4 (kromě 4.4), a 5.

Autoři děkují Dr.Ing. Pavlu Šmrhovi (ZČU/LPS) za provedení recenze skript a Helence Benešové (ZČU/KIV) za neobyčejnou péči, s kterou se věnovala typografické úpravě velké části skript. Chyby, které se ve skriptech nepochybně vyskytnou, určitě nebudou její vinou.



Kapitola 1

Paralelní počítače a programové modely

1.1 Základní pojmy

Zvyšování výpočetního výkonu počítačů s klasickou architekturou Neumannova typu je možné zejména vyšší hustotou integrace součástek a využitím elektronické technologie umožňující zvýšit rychlost operací (např. ECL). Obě uvedené možnosti narážejí v současné době na fyzikální meze dané zejména konečnou rychlostí šíření elektrického signálu a nutností odvádět tepelné ztráty z malého objemu integrovaných obvodů. Jednou z cest dalšího zvyšování výpočetního výkonu je využití možností paralelního zpracování dat na všech úrovních výpočetního systému.

Terminologicky vymezíme *procesor* jako obecný výpočetní prvek charakterizovaný množinou operací, které je schopný provádět. Dále je též využíváno kvůli stručnosti označení procesoru obvyklou zkratkou CPU (*Central Processing Unit*), i když složka *central* je v této zkratce již archaismem. Procesor může využívat vlastní (lokální) operační paměť (a pak se označení *procesor* obvykle přenáší i na dvojici procesor - paměť) a musí mít určité (obvykle omezené) možnosti pro styk s okolím. Samotný procesor (bez spojení s dalšími prvky) zpravidla není schopný samostatné funkce. Typickým příkladem procesorů jsou mikroprocesory řady I8086.

Naproti tomu *počítačový systém* (též *výpočetní systém*, dále kvůli stručnosti jen *počítač* nebo *paralelní počítač*) zahrnuje jeden nebo několik procesorů (i různého typu), komunikační vazby mezi procesory a dalšími prvky systému, mechanickou konstrukci a napájecí zdroje, může obsahovat ještě systémovou (globální) operační paměť, řídicí jednotky periferií různého typu, vlastní periferní zařízení (např. vnější paměť), a komunikační vazby na okolí (počítačová síť). Počítačový systém je integrován do funkceschopného celku základním řídicím programem (*operační systém*, dále využíváme zkratku OS), který zároveň poskytuje základní uživatelské rozhraní (angl. *interface*) pro práci s počítačem.

V rámci této publikace se budeme zabývat pouze případy paralelního zpracování dat, které nejsou zajištěny na úrovni elektroniky a tedy nejsou transparentní při vytváření programového vybavení pro příslušný počítač. Technické aspekty konstrukce paralelních počítačů jsou probírány zejména v rámci předmětů *Architektury číslicových systémů* a *Speciální architektury počítačů*.

Omezíme se na paralelní počítače, které jsou logickou extenzí von Neumannova modelu číslicového počítače a jejich programování je tudíž možné rozšířením a doplněním klasických metod programování. U počítačů, které budeme uvažovat, je výpočetní výkon rozložen (distribuován) na větší počet ($N > 1$) procesorů, které jsou nějakým způsobem spojeny do jednoho funkčního celku.

Paralelní výpočet nějaké *aplikační úlohy* (též *zakázka*, angl. *job*) probíhající podle jejího *programu* je strukturován na prvky (tj. dílčí výpočty), které jsou zpravidla označovány jako výpočetní procesy, zkráceně *procesy*. Jiná používaná označení jsou *úkoly* (angl. *task*) nebo *vlákna* (angl. *thread*). Procesy *spolupracují* na dosažení nějakého cíle (výsledku) celého výpočtu. Pojem procesu bude dále precizován v kapitole 2, podobně jako některé další zde používané pojmy. Prozatím budeme proces intuitivně chápat jako samostatnou výpočetní aktivitu (tj. činnost) schopnou spolupracovat s jinými procesy na dosažení celkového cíle výpočtu. Spolupráce procesů bývá též označována jako *interakce*. Elementární formou interakce je *synchronizace* (tj. zajištění správné návaznosti operací), komplikovanější formou je *zasílání zpráv* nebo *sdílení dat* (tj. výměna informace mezi procesy).

V souvislosti s paralelním programováním je třeba dosti důsledně rozlišovat pojmy *program* a *proces*. *Program je statický popis výpočetního postupu* (nemění se, nemá stav). Naproti tomu *proces je aktivita probíhající v čase* podle "svého" programu a její *stav* je určen aktuálním místem v programu a okamžitými hodnotami zpracovávaných dat. Proces nemusí existovat po celou dobu výpočtu, typicky je na začátku výpočtu spuštěn základní proces, který postupně vytváří či ruší jiné procesy (tj. počet procesů se obecně mění v průběhu zpracování úlohy). Podle téhož programu může (ve stejné době) probíhat větší počet procesů, přičemž každý proces má "svoje" data (se stejnou strukturou jako ostatní procesy, ale s jiným obsahem). *Alokace* (fyzické umístění) programového kódu a dat procesů v průběhu výpočtu je ovlivněno základním uspořádáním (architekturou) paralelního počítače (viz dále 1.2.).

Paralelní výpočet je tedy dynamicky strukturován na *procesy*, naproti tomu *paralelní program* je staticky strukturován na *programy* procesů a případné deklarace sdílených (též společných či globálních) dat.

Operační systém (zkratka OS) multiprocesorového počítače musí (na základě interpretace příkazů z terminálu) zajistit počáteční zavedení programového kódu do sdílené či distribuované paměti a spuštění výpočtu (v jednom či několika procesorech). Dále formou volání *jádra operačního systému* obvykle poskytuje prostředky pro paralelní interakci několika uživatelských úloh (tzv. *multitasking*) či lépe pro řízení paralelního výpočtu na úrovni jedné uživatelské úlohy (tzv. *multithreading*, v tomto případě procesy (označované jako *vlákna*) sdílí *kontext* úlohy - zejména data a otevřené soubory). Jednodušší situace nastává v případě, kdy multiprocesorový počítač je v určitém čase výlučně přidělen pro výpočet jediné úlohy. Pro tento případ také platí údaje o *urychlení* (angl. *speedup*) výpočtu uváděné pro různé typy paralelních algoritmů. Ve většině případů (např. superpočítač Alpha Digital na ZČU) je ale výkonný (a drahý) multiprocesorový počítač provozován ve *víceuživatelském režimu* činnosti (složitější operační systém), kdy paralelizovaná úloha soupeří o zdroje výpočetního systému (procesory, paměť, periferní zařízení) s úlohami jiných uživatelů (tj. v určitém čase existují v systému procesy (resp. vlákna) patřící k různým úlohám). V tomto případě může urychlení výrazně proklesnout proti teoreticky odvozeným hodnotám.

1.2 Základní typy paralelních počítačů

Výpočetní systémy s distribuovaným výpočetním výkonem se zhruba dělí na:

- **Počítačové sítě** (též *vícepočítačové* či *multipočítačové* systémy).
  Jsou vytvořeny spojením několika počítačů komunikačními linkami. Jsou místně rozlehlé, často heterogenní a jednotlivé prvky jsou zcela „suverénní“ v tom smyslu, že v rámci sítě poskytují jen omezenou množinu přesně definovaných služeb. Ostatní činnost je jejich „interní“ záležitostí. Počítače – prvky sítě tedy typicky nespolupracují na řešení jedné aplikace. *Počítačová síť může být ale programově přizpůsobena pro distribuované řešení jedné aplikace a v tomto případě ji lze zahrnout do kategorie paralelních počítačů* (a využívat některé dále uváděné metody a principy paralelního programování).

- **Paralelní počítače**
  Jsou vytvořeny spojením většího počtu výpočetních a dalších prvků do jednoho funkčního celku. Jsou místně kompaktní (například v jedné skříni) a zpravidla homogenní (výpočetní prvky stejného typu). Mají integrované řízení na úrovni operačního systému - tedy jeden OS, který může být *distribuovaný* – (jeho programový kód se vyskytuje na více místech v systému). OS musí poskytovat základní úroveň služeb umožňujících paralelní výpočet jedné aplikace.

Specifické problémy programování počítačových sítí jsou probírány v jiných předmětech. Většina těchto problémů se řeší na vyšších úrovních abstrakce s využitím základních principů paralelního programování. Zde se proto budeme zabývat zejména druhou uvedenou kategorií distribuovaných výpočetních systémů s tím, že řada uvedených přístupů a metod je platná i pro aplikace počítačových sítí.

Nejstarší klasifikaci paralelních počítačů provedl Flynn již v roce 1966. Základní prvky v této klasifikaci jsou:

1. SISD – Single Instruction, Single Data (stream).
   Klasická architektura (von Neumann) – jeden program („proud“ instrukcí) zpracovává jeden „proud“ dat.

2. SIMD – Single Instruction, Multiple Data.
   Tato architektura představuje vektorové a maticové procesory (tatáž

instrukce se provádí ve všech procesorech současně, ale každý procesor má svá jedinečná data).

3. MIMD – Multiple Instruction, Multiple Data.
   Jedná se o *multiprocesor*, několik „proudů“ instrukcí (procesů) zpracovává každý svůj „proud“ dat.

Časem se jednotlivé složky této klasifikace rozšířily a vznikly nové, viz např. [Hlav94]. Jedná se například o následující architektury:

- data flow procesory,
- systolické sítě,
- asociativní procesory.

Některé z nich lze zařadit s větší či menší přibližností do Flynnovy klasifikace (např. asociativní procesory do SIMD). Většina těchto nových architektur prozatím nepřekročila práh laboratoří.

Z komerčně úspěšných paralelních systémů lze vydělit dvě hlavní kategorie:

- **Vektorové paralelní počítače.**
  Jedná se například o systémy Cray-1S a CDC Cyber 205. Tyto počítače mají technicky realizované instrukce pro operace s vektory čísel. Realizace instrukcí je provedena paralelně (zřetězené zpracování, vektor představuje „proud“ dat, tatáž operace je rozpracována současně nad několika prvky proudu). Uvedené systémy se využívají hlavně pro náročné numerické výpočty. Hlavním programovacím prostředkem je programovací jazyk **Fortran** překládaný speciálním překladačem, který dokáže analyzovat cykly v programu (viz dále kap.6) a využít vektorové instrukce. Pro programátora je paralelismus tohoto typu transparentní a může je považovat za systémy typu SISD.

- **Multiprocesorové počítače.**
  Jsou vytvořeny spojením několika procesorů s klasickou (von Neumann) architekturou a dalších prvků do jednoho funkčního celku. Jedná se o architekturu označovanou ve Flynnově klasifikaci jako MIMD. Multiprocesory se využívají komerčně v řadě aplikací, například jako řídicí počítače, výkonné databázové servery nebo počítače pro náročné numerické výpočty. Rozlišujeme multiprocesory *se sdílenou pamětí*

(též *těsně vázané*, angl. *tightly coupled*) a *s distribuovanou pamětí* (též *volně vázané*, angl. *loosely coupled*). První typ multiprocesorových počítačů má *sdílenou* (tj. přímo adresovatelnou na úrovni instrukcí procesoru) celou operační paměť, nebo alespoň její část. Druhý typ multiprocesorových počítačů je realizován vhodným komunikačním propojením (zpravidla *sériovými linkami*) procesorů s vlastní (tj. *lokální*) pamětí.

Dále uváděné prostředky a metody paralelního programování předpokládají realizaci buď na počítači typu SISD (tj. klasická jednoprocesorová architektura typu von Neumann, viz např. kap. 3, paralelní procesy v jednoprocesorovém systému) *nebo multiprocesorovém počítači typu MIMD.*

Poznámka:

Klasifikace multiprocesorových systémů (a paralelních počítačů obecně) není zcela jednotná. Je to způsobeno rychlým vývojem v této oblasti, řadou možných hledisek klasifikace a určitým možným významovým překrytím některých pojmů (např. *procesor* versus *počítač*, *počítačová síť* versus *paralelní počítač* ap.). Navíc u složitějších architektur mohou být prvky v kategorii *procesor* (resp. *počítač*) do sebe hierarchicky „vnořeny“.

Dále uvedeme systémovou charakteristiku uvažovaných paralelních počítačů v kategorii MIMD. Technické detaily ponecháme do specializovaných předmětů. Správné chápání systémových principů a omezení příslušné architektury je nezbytné zejména v analytické fázi aplikace (tj. výběr hardware a volba programového modelu).

1.3 Multiprocesory se sdílenou pamětí

Multiprocesory se sdílenou pamětí můžeme dále dělit na:

- *Homogenní* – výpočetní [1] procesory jsou stejného typu, mají stejnou množinu instrukcí.
- *Nehomogenní* – procesory jsou různého typu.

[1] V architektuře počítače se mohou vyskytnout ještě specializované procesory určené například k provádění I/O operací, které mají na systémové sběrnici stejné postavení jako procesory určené k provádění obecných výpočtů.

Homogenní multiprocesory můžeme ještě dělit na:

- *Symetrické* – všechny procesory v systému mají stejné možnosti práce (například přístup ke sdíleným periferním prostředkům). Operační systém zachází se všemi procesory stejně.
- *Asymetrické* – procesory nemají stejné možnosti.

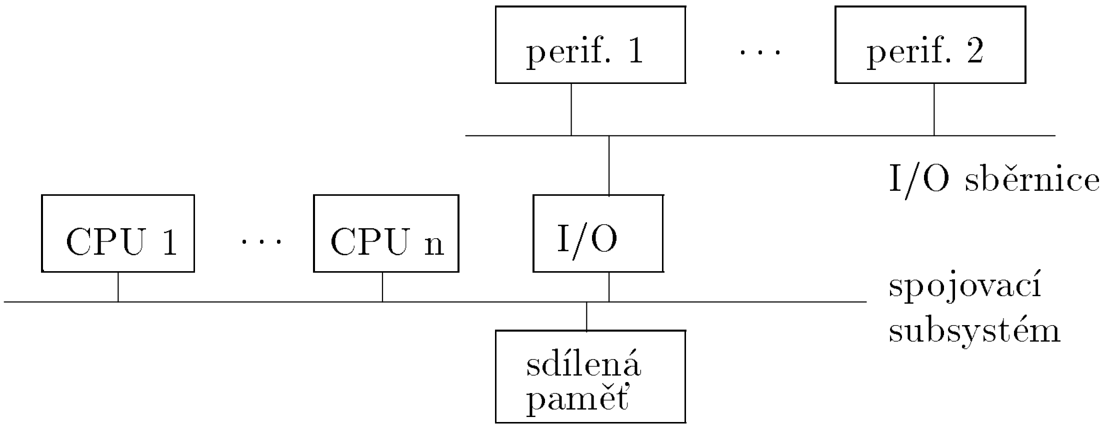
> Kritickým prvkem multiprocesorové architektury se sdílenou pamětí je evidentně spojovací subsystém (typicky *paralelní* sběrnice). Na propustnosti subsystému silně závisí počet procesorů, které lze do systému připojit.

1.3.1 Symetrický multiprocesor

Symetrická multiprocesorová architektura je schematicky znázorněna dále na obr. 1.1. Všechny univerzální procesory v systému (tj. CPU 1, CPU 2, ...) mají stejné možnosti využití zdrojů a z hlediska OS jsou rovnocenné. Kód i data OS jsou alokovány ve sdílené paměti. Žádný proces v paralelně počítané aplikaci tedy není vázán na konkrétní procesor a v průběhu svého „života" může „vystřídat" několik procesorů. V architektuře tohoto typu jsou dobře realizovatelné všechny základní programové modely (tj. MPMD, SPMD i MPSD).

Výhody

- **Jednoduchost** řízení na úrovni operačního systému. Naprosto stejný programový text jádra může být vykonáván (paralelně) všemi procesory. Jádro operačního systému při většině svých akcí nemusí rozlišovat jednotlivé procesory a může s nimi zacházet anonymně (tj. nemusí je nijak adresovat). *Některé aspekty řízení výpočtu na úrovni jádra operačního systému v symetrickém multiprocesorovém počítači jsou dále rozebírány v kap.4.*
- **Univerzálnost.** Libovolný výpočet dekomponovatelný na paralelní procesy je v této architektuře dobře realizovatelný. Sdílená paměť umožňuje lehce a efektivně implementovat všechny formy interakce procesů (formy interakce viz dále v kap.2). *Omezením dekompozice je pouze „rozměr" procesů* – pochopitelně se nevyplatí dynamicky



Obr. 1.1: Symetrická multiprocesorová architektura

vytvářet procesy pro výpočty trvající dobu (strojový čas) srovnatelnou s dobou, kterou jádro potřebuje pro založení a zrušení procesu. Dynamickou režii paralelního výpočtu lze redukovat vytvořením statické množiny procesů existujících po celou dobu výpočtu (viz dále Sequent, PPL, kap.4).

- **Spolehlivost.** V systému existuje N - násobná redundance procesorů, při výpadku procesoru často postačí pouze provést restart právě běžícího procesu. Centralizovaná paměť sice představuje ze spolehlivostního pohledu „úzký profil“, ale lze ji zejména proti nejčastějším poruchám úspěšně zabezpečit samoopravným kódem.

Nevýhody

- **Výkonnost.** Centralizované uložení veškeré informace ve sdílené paměti způsobuje velké zatížení spojovacího subsystému a z toho vyplývající degradaci výkonnosti (přístupové konflikty) pro větší počet procesorů. Možnosti oslabení této nevýhody jsou například:
  - doplnění lokální rychlé vyrovnávací paměti procesoru (*cache memory*), ale za cenu komplikací v hardware (problém s udržením identické informace vyskytující se na více místech v systému),
  - využití lokální ROM paměti procesoru k uložení frekventovaného společného programového kódu (např. jádra OS),

– distribuce externích přerušení přednostně na nečinné (momentálně zahálející) procesory.

I přes uvedená opatření nepřekračuje počet procesorů v komerčně úspěšných systémech tohoto typu (viz např. dále Sequent, kap.4) několik desítek.

Aplikace

Komerčně nejúspěšnější systémy tohoto typu (např. Sequent, Digital, file-servery Sun-Sparc, vyvíjené file-servery SGI) se využívají jako výkonné a spolehlivé databázové servery zajišťující dobrou dobu odezvy (typicky 1 s) pro přístup z velkého počtu (až několik tisíc pro Sequent) terminálů. Využívá se možnost rozkladu složité operace nad databází na elementární paralelně vykonatelné akce (transakce).

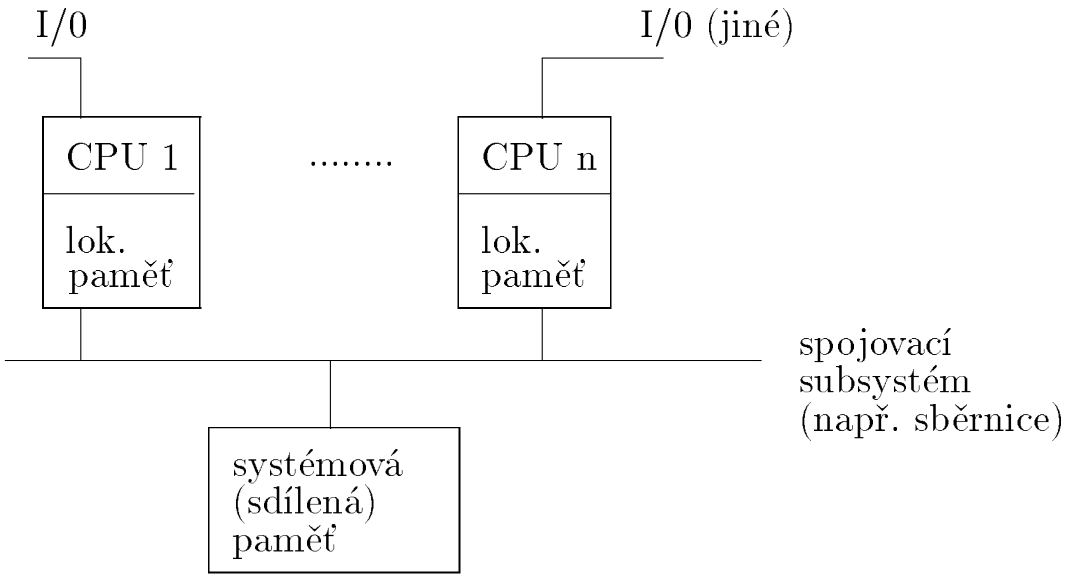
Další aplikační oblastí jsou časově i paměťově náročné numerické výpočty vědecko-technického charakteru. Nejčastěji se jedná o řešení (numerickou integraci) soustav obyčejných diferenciálních rovnic nebo numerické řešení parciálních diferenciálních rovnic.

1.3.2 Asymetrický multiprocesor

Všechny procesory (CPU) jsou stejného typu, každý využívá (kromě sdílené) vlastní lokální paměť a může mít realizované vazby na vlastní periferní zařízení. Z hlediska řízení na úrovni operačního systému již nelze takto definované procesory považovat za rovnocenné. Programový kód OS může být společný pro všechny procesory (alokován v sdílené paměti nebo distribuován mezi sdílenou paměť a lokální paměti procesorů). V krajním případě může mít každý procesor vlastní OS alokovaný v jemu přístupné lokální paměti. Struktura systému je blokově znázorněna na obrázku 1.2.

Asymetrie

je způsobena různými funkčními schopnostmi procesorů. Operační systém nemůže alokovat kterýkoliv výpočet (proces) na libovolný procesor, musí brát v úvahu požadavky procesu na zdroje systému. V krajním případě může mít každý procesor vlastní operační systém.



Obr. 1.2: Struktura asymetrického multiprocesoru

Výhody

Je možné docílit snížené zatížení spojovacího subsystému rozvržením procesů (tj. kódu jejich programů a jejich lokálních dat) do lokálních pamětí – ve sdílené paměti lze ponechat pouze globální data nutná pro spolupráci procesů.

Nevýhody

Složité řízení na úrovni operačního systému (OS musí rozlišovat procesory) – například složité je zajištění přenosu z periferie připojené k CPU 1 na periferii připojenou k CPU 2. Dále z hlediska spolehlivosti se nevyužívá redundance procesorů (každý má nějaké specifické schopnosti).

Aplikace

Zřejmě nebude možné architekturu tohoto typu úspěšně využívat jako univerzální (tj. vhodnou pro širší spektrum aplikací). Využití uvedených výhod při minimalizaci nevýhod je velmi dobře možné v aplikacích, kde *existuje stabilní množina procesů a je možné provést jejich pevné rozvržení na jednotlivé procesory.*

Typicky se jedná o aplikace počítačového řízení průmyslových (i jiných) procesů v reálném čase. Jednotlivé procesory počítače mohou vykonávat specifické úkoly realizované množinou procesů alokovaných na procesor, potřebné programy jsou umístěny v lokální paměti procesorů (mohou být i v ROM), data potřebná pro činnost procesů mohou být distribuována mezi lokální paměť procesorů a sdílenou systémovou paměť (globální data potřebná pro koordinaci činnosti v celém systému). Možnosti I/O jednotlivých procesorů mohou být přizpůsobeny potřebám alokovaných procesů.

Řada průmyslových mikropočítačových stavebnic podporuje uvedenou architekturu. Klasickou stavebnicí je SBC (Single Board Computers) firmy Intel. Procesorové desky představují kompletní mikropočítač (procesor např. I8086, paměť RAM i ROM, periferní obvody pro digitální a sériový I/O) jsou ve skříni počítače vázány paralelní sběrnicí (např. MULTIBUS 1) navzájem (logika sběrnice řeší arbitraci konfliktů) a s dalšími „pasivními" moduly (sdílená paměť, deska A/D převodníků ap.)

1.4 Multiprocesory s distribuovanou pamětí

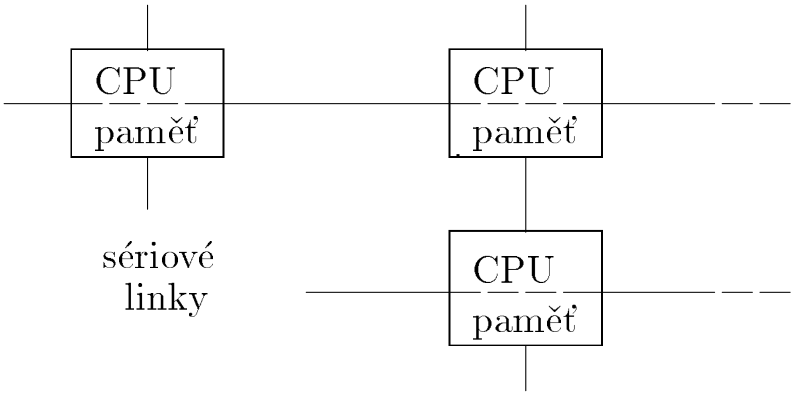
Volně vázaná multiprocesorová architektura obsahuje jako základní stavební prvky procesory doplněné vlastní (lokální) pamětí vázané mezi sebou rychlými sériovými linkami. Ve struktuře není přítomna žádná sdílená (tj. strojovou instrukcí procesorů přímo adresovatelná) paměť. Lokální paměti obsahují programový kód a data pro proces (procesy) alokovaný na příslušném prvku sítě. Procesy vzájemně komunikují zasíláním zpráv po sériových linkách. Prvky procesorové sítě jsou v podstatě kompletní počítače (s omezenými I/O možnostmi).

Struktura systému je schematicky naznačena na obr 1.3.

Na multiprocesorovém počítači s distribuovanou pamětí jsou realizovatelné všechny dříve uvedené základní programové modely, tj. MPMD, SPMD i MPSD.

Poznámky:

- Ve struktuře není explicitně žádný „úzký profil" z hlediska výkonnosti. Je zřejmě možné síť rozšiřovat na velký rozměr přidáváním dalších prvků.
- Veškerá výměna informace se děje prostřednictvím zasílání zpráv mezi procesy alokovanými na jednotlivé procesory systému. Při dekompozici výpočtu na paralelní procesy musí být „globální" data výpočtu



Obr. 1.3: Struktura multiprocesoru s distribuovanou pamětí

rozptýlena v síti (buď vědomě pomocí alokačních příkazů nebo automaticky kompilátorem – viz dále nCUBE v kap.5).

- Rozvržení procesů na jednotlivé prvky sítě musí být provedeno před výpočtem (typicky při překladu paralelního programu). Dynamická „realokace“ procesů v síti je přinejmenším problematická.
- Zavedení programů do jednotlivých prvků musí být rovněž provedeno před zahájením výpočtu.
- Konkrétní topologie sítě bude nepochybně vhodná pouze pro omezené spektrum aplikací.
- Typické používané topologie jsou dvourozměrná (cyklicky uzavřená) mřížka nebo n-rozměrná krychle.

Architektura může být dále doplněna výkonným „organizačním“ procesorem, který nebude stejného typu jako prvky sítě (lze použít např. RISC pracovní stanici). Tento procesor zajišťuje jednak I/O možnosti pro celý systém (vazba na počítačové sítě, vnější paměť), umožňuje vývoj programů pro prvky sítě (křížové prostředky) a zajišťuje celkovou organizaci výpočtu (zavedení programů do prvků sítě, převzetí a interpretaci výsledků). Z jiného úhlu pohledu lze pak organizační procesor chápat jako počítač (*host system*) vybavený výkonným „numerickým koprocesorem“.

V diskutovaném případě bude operační systém „distribuován“ na části rezidující v organizačním procesoru (např. nějaká extenze **Unix**u) a identické části rezidující ve všech prvcích sítě.

Výhody

- **Výkonnost.** Hlavní (potenciální) výhodou proti dříve uváděným multiprocesorovým architekturám je využitelnost i v aplikacích vyžadujících tzv. *masívní paralelismus*, tj. rozprostření výpočtu na stovky až tisíce procesorů. Je odstraněna degradace výkonnosti způsobená konflikty na přístupu do sdílené paměti. Navíc může být výrazně omezena režie spojená se zakládáním procesů (nevytváří se za běhu programu) a s přepínáním kontextu (v každém prvku sítě může být jen jeden nebo několik málo procesů).

Nevýhody

- **Aplikační citlivost** (tj. opak univerzálnosti). Konkrétní topologie sítě není univerzálně vhodná pro všechny aplikace (paralelní algoritmy). Proto využití výše uvedených výhod je „podmíněné" vhodnou volbou topologie sítě, vhodným alokováním procesů na prvky sítě a optimalizací jejich komunikace. Uvedené rozhodování komplikuje vytváření programů. Část práce spojené s vytvářením paralelní aplikace lze přenést na překladač využívaného jazyka a dále lze využít různé knihovny podpůrných funkcí (viz dále např. knihovny pro nCUBE v kap. 5.3).

Aplikace

Výhodně lze systémy diskutovaného typu použít pro jednoúčelové aplikace, kde jak topologie sítě, tak rozvržení výpočetních procesů mezi uzly sítě je pevně dáno a nemění se. Jedná se například o využití sítí složených z Transputerů pro monitorování a řízení průmyslových procesů. Síť může být přizpůsobena a naprogramována pro konkrétní průmyslový proces. Podobně například pro zpracování rozsáhlých souborů dat získávaných současně z mnoha zdrojů lze konstruovat specializované sítě (geofyzikální experimenty, rozpoznávání obrazů snímaných radarem ap.). Lze očekávat využití v oblasti řešení úloh umělé inteligence (např. rozpoznávání mluvené řeči v reálném čase, rozpoznávání obrazů ap.). Další již realizovaná využití jsou v oblasti počítačové grafiky (visualizace dat, grafické transformace, animace ap.). Komerčně úspěšné systémy nCUBE jsou podobně jako symetrické multiprocesory využívány též jako výkonné databázové servery.

Další oblastí využití multiprocesorových systémů s distribuovanou pamětí jsou numericky náročné vědecko-technické výpočty. Zde je možné najít úlohy vyžadující (resp. umožňující) „masivní paralelismus“. Lze odhadnout, že největší naději na urychlení mají úlohy paralelizovatelné na relativně samostatné procesy (tj. malá interakce s ostatními procesy) alokovatelné v různých prvcích výpočetní sítě. Jako příklad lze uvést *proudové (angl. pipe-line) zpracování dat*, kdy zpracovávané datové záznamy „proudí“ sítí procesorů a každý prvek sítě nad nimi provádí specifickou operaci (analogie pásové výroby). V tomto ideálním případě je komunikace omezena na minimum a komunikují jen sousední prvky v síti.

1.5 Paralelizace výpočetních postupů

1.5.1 Základní modely paralelní dekompozice

Vlastnímu programování paralelní programové aplikace většinou předchází fáze analýzy, ve které se nejprve rozhodujeme o nějakém základním přístupu, který použijeme pro *dekompozici výpočtu* (tj. dekompozici činnosti) na složky - procesy. Zároveň musíme řešit s tím související problém *dekompozice dat* využívaných k výpočtu. Toto rozhodování nemusí být ovlivněno dostupnou multiprocesorovou architekturou, protože dále uváděné modely dekompozice lze zpravidla programově realizovat na obou hlavních uvažovaných architekturách, tj. multiprocesorech se sdílenou a distribuovanou pamětí. Protože paralelizací složitých nebo objemem velkých činností se lidé zabývají odedávna, lze nalézt četné ”nepočítačové” analogie dále uváděných přístupů.

Model MPMD

První možný přístup k paralelní dekompozici výpočtu je označován zkratkou MPMD (*Multiple Program Multiple Data*). Jedná se o dekompozici výpočtu na relativně samostatné činnosti, z nichž některé mohou být vykonávány souběžně (tj. paralelně). Tento přístup využijeme *pro relativně složitou činnost a málo ”objemná” data* (tj. důraz dáváme na dekompozici činnosti, data potřebná pro dílčí činnost k ní logicky přiřadíme – používá se označení *lokální data* (procesu)) [2]. Jednotlivé dílčí činnosti vzniklé dekompozicí mohou být samostatně naprogramovány (Multiple Program) a odpovídající výpočetní procesy pracují každý se svými lokálními daty (Multiple Data) strukturně odlišnými od dat jiných procesů. Provedenou dekompozici lze obvykle vyjádřit precedenčním grafem (viz známé metody CPM, PERT z operační analýzy), ve kterém hrany modelují dílčí činnosti (mohou mít jako váhu dobu trvání činnosti) a uzly modelují požadavky na synchronizaci činností. Využitím modelu MPMD nemusí být sledováno jen výkonnostní hledisko (tj. urychlení výpočtu). V mnoha případech (např. řízení v reálném čase, simulační programy) se tento model využívá s ohledem na lepší strukturování programu.

Programový model MPMD lze implementovat jak na jednoprocesorovém počítači, tak na multiprocesorech různého typu. Například program vytvořený v programovacím jazyce **ADA** podle modelu MPMD je logicky paralelní, ale může být vytvořen (a odladěn) na jednoprocesorovém počítači. Po přenesení (beze změn ve zdrojovém textu, vše potřebné zařídí překladač) na symetrický multiprocesor budou jeho části (úkoly, angl.task) probíhat fyzicky paralelně a výpočet bude rychlejší. Z historického pohledu byl model MPMD teoreticky rozpracován jako první (Dijkstra, Hoare, Wirth), protože je smysluplně využitelný i na jednoprocesorovém počítači.

Model SPMD

Druhá možnost globálního přístupu k paralelizaci výpočtu je označována zkratkou SPMD (*Single Program Multiple Data*), nebo též jako *dekompozice podle dat*. Jedná se o případ, kdy *relativně jednoduchá* činnost je prováděna nad *objemově rozměrnými daty*. Zpracovávaná množina dat (typicky homogenní pole prvků s jednoduchým typem, jednorozměrné nebo vícerozměrné) se v tomto případě rozdělí na m částí. Vytvoří se k procesů ($k \leq m$) pracujících podle stejného programu (Single Program) a každý z těchto procesů samostatně zpracuje jednu nebo několik strukturně podobných (ale hodnotami různých) částí dat (Multiple Data).

V tomto druhém případě je sledováno výhradně výkonnostní hledisko - tedy cílem je *urychlení výpočtu* (angl. *speedup*). Už při výchozí analýze aplikace zamýšlíme fyzicky paralelní výpočet, na rozdíl od funkční dekompo-

[2] Snažíme se přitom minimalizovat potřebu tzv. *globálních dat*, která "nepatří" k žádnému procesu.

zice, kdy vytvořený program (s eventuelními malými modifikacemi) může probíhat pseudoparalelně na jednoprocesorovém počítači nebo fyzicky paralelně na multiprocesoru určitého typu. Využití modelu SPMD je vázáno předpokladem funkční ekvivalence procesorů (schopných vykonávat tentýž program). Pro lepší představu uvedeme příklady využití modelu SPMD:

Násobení matic – každý prvek (nebo řádek či sloupec) výsledné matice lze počítat jedním procesem, popřípadě několik (málo) procesů se může nějak podělit o výpočet všech prvků (řádků, sloupců).

Geometrická dekompozice – využívá se při paralelizaci iteračního numerického řešení parciálních diferenciálních rovnic. Oblast řešení rovnice se (geometricky) rozdělí na části, výpočet řešení v jednotlivých částech provedou (podle stejného programu) dedikované procesy.

Model MPSD

Dalším (třetím) základním přístupem k paralelizaci výpočtového problému v uvažovaných počítačových architekturách je zřetězené zpracování (dat) [3]. Odpovídající programový model můžeme v analogii k předchozím označit jako MPSD (*Multiple Program Single Data*). Jedná se o zpracování rozsáhlého "proudu" datových prvků, přičemž nad jednotlivými prvky jsou postupně vykonávány nějaké (libovolně složité) operace. Operace je možné svěřit různým (specializovaným) procesům a vykonávat je paralelně pro několik prvků datového proudu. Analogií modelu MPSD je průmyslová pásová výroba.

Hybridní modely

Uvedená kategorizace (jako ostatně každá jiná) je pouze hrubá. Jako ukázku uvedeme případ aplikace, která má zpracovat proud zakázek (příkazů) z okolí systému, přičemž jednotlivé zakázky lze počítat nezávisle (angl. označení stylu výpočtu jako *demand driven*). V multiprocesorovém systému lze jednotlivé zakázky zpracovávat paralelně na různých procesorech systému. Pokud každý procesor v systému umí vybrat (například z fronty ve sdílené paměti) a zpracovat kteroukoliv zakázku (tj. všechny procesory vykonávají stejný program - *cykl: čekej zakázku, zpracuj zakázku*), jedná se o přímé využití modelu SPMD.

[3] Je třeba odlišovat od *zřetězeného zpracování instrukcí programu*, což je (pro programátora transparentní) hardwarově realizované současné zpracování několika následujících strojových instrukcí v procesoru.

Pokud jeden (řídicí) proces rozděluje požadavky na specializované (řízené) procesy (alokované na různých procesorech) a sbírá jejich výsledky, jedná se o využití modelu MPMD (různé programy procesů zúčastněných na výpočtu) [4]. Speciální případ, kdy všechny podřízené procesy mají stejný program, se označuje jako výpočetní model *processor farm* - řídící proces je "farmer" ostatní procesy jsou "workers". Tento výpočetní model (jak už ostatně vyplývá z názvu) má opět četné analogie v organizaci paralelní lidské činnosti, je průhledný a dobře programovatelný. Jako příklad jeho využití uveďme namátkou: simulační výpočet funkčních závislostí – řídicí proces zadává podřízeným výpočet jednotlivých bodů hledané funkce, podřízené procesy řeší tentýž simulační model pro různé zadávané parametry.

1.5.2 Základní metody interakce procesů

Každý z výše uvedených modelů dekompozice musí být realizován s využitím konkrétního způsobu interakce procesů, nebo jinak řečeno: *po výběru globálního modelu paralelní dekompozice se musíme rozhodnout pro konkrétní způsob (model) interakce procesů existujících za běhu úlohy.* Při tomto rozhodování (na rozdíl od volby modelu dekompozice) jsme už výrazně ovlivněni architekturou multiprocesorového počítače a konkrétními programovacími prostředky, které poskytuje operační systém (knihovny funkcí, programovací jazyky a jejich vývojová prostředí).

Nezávisle na použitém modelu dekompozice výpočtu, modelu interakce procesů a předmětu výpočtu bychom se při analýze a programování konkrétní paralelní aplikace měli snažit o realizaci následujících vlastností paralelního programu:

- *minimální interakce (co největší samostatnost) paralelních procesů.* Častá interakce má negativní vliv na výkonnostní ukazatele výpočtu (tj. zpomaluje výpočet), zejména pak u počítačů s multiuživatelským

[4] Principiálně je možné využívat všechny výše uváděné základní modely paralelní dekompozice v rámci jedné úlohy (postupně, prostřídaně, popřípadě i vnořeně). Například stavba rodinného domku je paralelizována na dílčí činnosti v základní úrovni způsobem MPMD (různé navazující či souběžně vykonatelné činnosti), některé (objemem práce velké) činnosti mohou být *urychleny* způsobem SPMD (fasáda).

režimem činnosti nebo sdíleným komunikačním mediem (např. Ethernet - viz dále PVM, podkap.5.2.). Kromě toho je složitá a nepřehledně řešená interakce procesů zdrojem nepříjemných *run-time* chyb (tzv. *časově závislé chyby* – nenastanou vždy, těžko se odhalují). V této souvislosti ještě zdůrazníme současný trend hledání a výzkumu tzv. *totálně asynchronních* (např. iteračních) paralelních algoritmů (viz např. [Bert89]) u kterých je minimální potřeba vzájemné synchronizace a jiné interakce.

- *rychlostní nezávislost výpočtu*, tj. interakce procesů musí být (správně) řešena bez ohledu na (obecně neznámou, ovlivněnou např. typem či různým zatížením procesorů) rychlost jednotlivých procesů. Prakticky to znamená zvažovat pro každý bod interakce (místo v programu procesů) všechny možné stavy zúčastněných procesů.

  Rychlostně nezávislý výpočet je ukončen v konečném čase (tj. nedojde k tzv. *zablokování*, angl. *deadlock*) a dá (pro stejná vstupní data) vždy stejný výsledek. Jako formální model pro analýzu možných stavů paralelních činností se využívají *Petriho sítě*.

- *nezávislost na počtu dostupných procesorů* (vlastnost označovaná též jako *scalability*). Toto je aktuální zejména při využití modelu SPMD (pro MPMD automaticky řeší jádro OS svojí technikou přidělování procesorů procesům připraveným k výpočtu). Počet dostupných procesorů je možné zadat ve vstupních datech programu nebo (lépe) zjistit v inicializační části programu voláním příslušné funkce jádra OS.

Základní modely interakce procesů se liší způsobem výměny informace mezi spolupracujícími procesy. Jedná se o:

- *sdílení dat* (angl. *data sharing*),
- *komunikaci pomocí zpráv* (angl. *message passing*).

Obě uvedené techniky umožňují též synchronizaci procesů principiálně realizovatelnou testováním změny obsahu sdílených dat v prvním případě a výměnou zpráv s nulovým informačním obsahem v případě druhém.

Sdílení dat

Technika sdílení dat je realizovatelná pouze na multiprocesorových počítačích se sdílenou pamětí. Procesy využívají ke vzájemné výměně informací sdílené datové struktury (analogie: *více úředníků, jedna kartotéka*). *Procesy se nemusí vzájemně "znát".* Je ovšem třeba nějakým způsobem vyloučit současnou (tj. chaotickou) změnu sdílených dat více procesy (angl. *mutual exclusion*). Části programu, v rámci kterých může dojít ke konfliktnímu přístupu k datům, se označují jako *kritické sekce*. K zabezpečení kritických sekcí se využívají tzv. nízkoúrovňové programové konstrukty (*semafory, paměťové zámky, bariéry*). Jejich použití je ale nepřehledné a může vést k výskytu obtížně odhalitelných chyb za běhu programu. Nízkoúrovňové prostředky zpravidla poskytuje jádro operačního systému prostřednictvím specializovaných volání jádra OS.

V průběhu času byly vyvinuty bezpečnější a přehlednější strukturované způsoby interakce paralelních procesů spolupracujících nad sdílenými daty. Nejznámější jsou *monitory* a *rendezvous*. Využití obou uvedených způsobů přichází v úvahu zejména ve specializovaném programovacím jazyce (viz dále **ADA** v kap. 3).

Komunikace zasíláním zpráv

Technika *komunikace zasíláním zpráv* je přirozená (a jedině možná) v multiprocesorových systémech s distribuovanou pamětí, je ji možné ale realizovat i v multiprocesorových systémech se sdílenou pamětí (emulace zasílání zpráv přes sdílenou paměť). Je tedy *na rozdíl od sdílení dat* univerzální technikou interakce procesů ve vztahu k užívaným multiprocesorovým architekturám. *Této programovací technice tedy dáváme přednost, záleží–li nám na přenositelnosti vytvářeného programu* [a].

[a] Problém je v tom, že využití techniky sdílení dat pro multiprocesorovou architekturu se sdílenou pamětí často vede k přehlednějším a rychlejším programům (sdílení dat má menší časovou režii).

Základní dělení způsobů komunikace je na *synchronní* (výměna zprávy proběhne až když jsou dva komunikující procesy označované jako *odesilatel* a *příjemce* připraveni ke komunikaci) a *asynchronní* (odesilatel vkládá zprávu do vyrovnávací paměti a příjemce si ji vyzvedne z téže či jiné vyrovnávací paměti až ji bude potřebovat). Další možné dělení je podle způsobu *adresace procesů* použitého ve zprávách: *symetrické* - zpráva obsahuje adresu (identifikaci) odesilatele i příjemce, *asymetrické* - zpráva obsahuje jen adresu příjemce, *nepřímé* - zpráva obsahuje adresu *vyrovnávací paměti* či *komunikačního kanálu* (tj. příjemce a odesilatel se vzájemně "nemusí znát jménem").

1.5.3 Úloha operačního systému

S ohledem na široké spektrum konkrétních multiprocesorových architektur a operačních systémů je obtížné obecně specifikovat požadované funkce operačního systému multiprocesorového výpočetního systému. Operační systém symetrického multiprocesorového počítače se nemusí příliš lišit od víceuživatelského operačního systému konvenčního počítače. Kód operačního systému se může vyskytovat pouze v jednom exempláři a na jednom místě (sdílená paměť). Odlišnosti se mohou v ideálním případě omezit jen na jádro operačního systému. Množina volání jádra musí být rozšířena o prostředky pro založení a interakci procesů, musí být modifikován základní mechanismus přidělování procesoru procesům připraveným k výpočtu a musí být modifikován mechanismus obsluhy přerušení. Často se jedná o nějakou modifikaci **Unix**u, jako příklady lze uvést operační systémy **Unix** SVR4 (označovaný jako Irix) pro multiprocesory Power Challenge firmy SGI nebo DYNIX pro počítače Sequent Symmetry (viz dále ukázku paralelizace výpočtu s využitím knihovny PPL).

Operační systém pro volně vázaný multiprocesorový počítač je složitější záležitostí. Každý prvek výpočetní sítě je vybaven vlastní kopií jádra operačního systému, které řídí výpočet v něm probíhající a komunikuje s dalšími prvky systému. Výpočetní síť je zpravidla doplněna organizačním procesorem (*host procesor*), který zprostředkuje vnější uživatelské rozhraní mutiprocesorového počítače (např. grafický vstup/výstup, síťové služby ap.) - "tváří se" tedy vůči uživateli jako konvenční OS. Organizační procesor může být stejného nebo odlišného typu jako prvky výpočetní sítě, v obou případech ale musí být vybaven částí OS funkčně odlišnou proti prvkům výpočetní sítě. Významná odlišnost proti konvenčním operačním systémům zřejmě spočívá v tom, že kód operačního systému se vyskytuje na více místech ve výpočetním systému, přičemž jednotlivé části nemusí být funkčně shodné. Operační systém musí umožnit alokaci a zavedení programového kódu jednotlivých procesů úlohy na různé procesory systému a měl by podporovat jednotný způsob logického adresování (číslování) prvků výpočetní sítě, který by byl pokud možno nezávislý na fyzickém umístění prvků v síti.

V této hrubé charakteristice operačních systémů multiprocesorových počítačů je třeba se ještě zmínit o programech, které dokáží integrovat lokální počítačovou síť do jednoho volně vázaného multiprocesorového počítače. Pravděpodobně nejznámějším programovým prostředkem v této kategorii je PVM (*Parallel Virtual Machine*), který umožní využít obecně heterogenní lokální síť unixovských pracovních stanic k provedení paralelního výpočtu jedné úlohy. Paralelní výpočet přitom běží "na pozadí" a neomezuje běžné uživatele použitých pracovních stanic. Zpravidla jedna pracovní stanice organizuje výpočet. V každém použitém prvku výpočetní sítě je alokován démón (PVMD), který dokáže na základě povelu zprostředkovaného PVMD řídící stanice zavést konkrétní program, odstartovat příslušný výpočetní proces a zprostředkovat jeho komunikaci s okolím. Program PVM byl vytvořen v Oak Ridge National Laboratory v USA a je volně dostupný na Internetu. Detailnější informace týkající se využití PVM je uvedena dále v podkap. 5.2.

1.5.4 Programovací prostředky pro paralelizaci výpočtu

Specializovaný programovací jazyk

Jedná se o jazyk, který obsahuje prostředky pro paralelní programování přímo jako svoji nedílnou součást. Jako příklad programovacího prostředku spadajícího do této kategorie lze uvést jazyk **ADA**, který je vhodný zejména pro realizaci základního modelu MPMD (tj. funkční dekompozice - více procesů fungujících podle různých programů). Podrobnější popis jazyka je uveden dále v podkap. 3.3. Hlavní aplikační oblastí je programování *real–time* aplikací (např. řízení technologií v průmyslu, řízení lodí, letadel, radarových systémů ap.) [5].

ADA využívá k interakci procesů mechanismus nazývaný *dostaveníčko* (*rendezvous*). Tento mechanismus spočívá ve společném provedení téhož

[5] Překladač Digital ADA je na ZČU dostupný všem uživatelům superpočítače Alpha Digital.

programového kódu dvěma procesy a je natolik univerzální, že (spolu s příkazem pro nedeterministický výběr alternativy) mimo jiné umožňuje realizaci všech dříve využívaných primitivnějších způsobů interakce (semafory, kritické sekce, monitory, schránky pro asynchronní komunikaci ap.).

Knihovna externích funkcí

Další možností je použití *univerzálního programovacího jazyka* s prostředky pro paralelní programování poskytované operačním systémem ve formě *knihovny externích funkcí.* Teoretické výhody jsou na straně využití specializovaného programovacího jazyka (zejména možnost využívání strukturovaných paralelních konstruktů, vyšší spolehlivost programů s ohledem na kontroly realizované překladačem, nezávislost na operačním systému ap.). Praxe ale zatím spíše preferuje využívání standardních programových prostředků (typicky jazyk **C** a nějaká modifikace OS **Unix**) doplněných nějakou nadstavbou pro paralelní programování. Do standardního prostředí (nebo blízkého standardu) se lépe přenáší už hotové programy a různé účinné ladicí prostředky částečně eliminují "nebezpečnost" využívání primitivních konstruktů pro paralelizaci výpočtu.

Jako příklad prostředků tohoto typu dostupných na ZČU lze uvést knihovnu externích funkcí DECthreads pro symetrický multiprocesor fy Digital. Volání funkcí z knihovny umožňuje v programech psaných v jazyce **C** nebo **Fortran** realizovat vytváření a interakci paralelních procesů (tzv. *vláken*, angl. termíny *threads*, *multithreading*), které (na rozdíl od procesů v OS **Unix**) *sdílí kontext úlohy* (zejména data a otevřené soubory).

S využitím DECthreads lze realizovat všechny výše uváděné základní modely paralelní dekompozice a oba modely interakce procesů (zasílání zpráv je emulováno přes komunikační struktury ve sdílené paměti).

Dalším dostupným programovacím prostředkem je knihovna PPL pro počítač Sequent popisovaná dále v podkap. 4.3.

V kategorii programovacích prostředků pro multiprocesorové počítače s distribuovanou pamětí lze na ZČU využívat programový nástroj PVM (*Parallel Virtual Machine*), který umožňuje využít obecně heterogenní lokální síť unixovských pracovních stanic k provedení paralelního výpočtu jedné úlohy (tedy softvarově integruje několik nezávislých počítačů do jednoho paralelního počítače). Paralelní výpočet přitom běží "na pozadí" a neomezuje běžné uživatele použitých pracovních stanic. Zpravidla jedna pracovní stanice organizuje výpočet. V každém použitém prvku výpočetní sítě je alokován *démón* (PVMD), který dokáže na základě povelu zprostředkovaného PVMD řídící stanice zavést konkrétní program, odstartovat příslušný výpočetní proces a zprostředkovat jeho komunikaci s okolím.

Prostřednictvím PVM lze opět realizovat všechny výše uvedené základní modely paralelní dekompozice, z hlediska modelu interakce jsme ovšem omezeni jen na asynchronní komunikaci pomocí zpráv. Pro PVM přirozený (a zhusta využívaný) je model *farmer-workers*, jehož ideální využití má malou komunikační režii (jedna zpráva–příkaz pro "dělníka", jedna zpráva-odpověď s výsledkem).

Program PVM byl vytvořen v Oak Ridge National Laboratory v USA a je volně dostupný na Internetu. Na ZČU je PVM instalován na skupině cca deseti pracovních stanicích SG (univerzitní laboratoř CAD) a na přibližně stejném počtu stanic SUN (laboratoře KIV).

Paralelizační překladač

Třetí možností (nejpříjemnější pro aplikačního programátora) je *využití univerzálního programovacího jazyka s paralelizačním překladačem.* Automatická paralelizace je výhodná proto, že velké množství sériových algoritmů, které již byly zapsány a vyzkoušeny v sériovém tvaru, je takto možno relativně "bezpracně" převést do paralelního tvaru. Je ovšem třeba počítat s tím, že postup zdaleka nemusí být přímočarý.

Využitelné paralelizační překladače byly zkonstruovány zejména pro programovací jazyk **Fortran** (viz dále podkap. 4.4), což je dáno jeho relativní syntaktickou jednoduchostí a dominantním využitím pro složitější numerické výpočty. Vývoj **Fortran**u zatím vyvrcholil programovacím jazykem **HPF** (*High Performance Fortran*) [6]. Tento jazyk je určen k programování náročných numerických aplikací. V těchto výpočtech se využívají zejména regulární datové struktury (matice, vektory), které může překladač jednoduše distribuovat do lokálních pamětí jednotlivých procesorů

[6] Na ZČU je k dispozici Digital Fortran pro superpočítač Alpha Digital, přičemž překladač splňuje poslední normu **HPF**.

systému [7]. Detailnější popis jazyka je proveden dále v kap. 6.

HPF je programovým prostředkem rozšiřujícím možnosti základního jazyka (**Fortran 90**) pro paralelní dekompozici výpočtu způsobem SPMD v multiprocesorové architektuře s distribuovanou pamětí. Programátor má možnost zvolit rozměry procesorové sítě (direktiva `PROCESSORS`) a ovlivnit distribuci regulárních datových struktur do prvků sítě (zejména direktivy `ALIGN` a `DISTRIBUTE`). Dále má možnost "uvolnit" striktně sekvenční provádění operací (typicky iterace v cyklu) příkazem `FORALL`. *Překladač sám zařídí veškerou potřebnou komunikaci spojenou s paralelním prováděním tohoto příkazu nad distribuovanými daty.*

1.6 Hodnocení výkonnosti

Dále zavedeme některé důležité pojmy používané v souvislosti s paralelními algoritmy a s volbou příslušného programového modelu:

- **Složitost** (complexity, worst-case complexity) je funkce $f(n)$, jejíž hodnota je pro konkrétní algoritmus úměrná maximální době jeho výpočtu, maximum se bere přes všechny možné vstupy algoritmu s rozměrem n (např. rozměr matice, počet čísel v paralelně realizovaném součtu ap.). Například sekvenční algoritmus pro součet n čísel má složitost $f(n) = n$, protože maximální doba výpočtu je úměrná počtu čísel n. Složitost se označuje písmenem O, v tomto případě tedy $O(n)$ znamená, že doba výpočtu je lineárně závislá na počtu čísel n. Pro paralelní součet prováděný na $p = n/2$ procesorech je složitost $O(log\ n)$, kde logaritmus je s libovolným základem.

- **Cena** (angl. cost) paralelního algoritmu je složitost násobená počtem použitých procesorů, například pro paralelně prováděný součet n čísel je cena $n/2\ .\ (log\ n)$. Cena je úměrná celkovému strojovému času všech použitých procesorů.

[7]Implicitně je předpokládána multiprocesorová architektura s distribuovanou pamětí, která dovoluje tzv. *masivní paralelismus* (tj. řádově stovky či tisíce relativně samostatných procesorů) a jako (skrytá) technika interakce je využita *komunikace zasíláním zpráv.* Na multiprocesorech se sdílenou pamětí jsou lokální paměti (logických) procesorů a způsob komunikace programově emulovány.

- **Urychlení** (angl. speedup) S je obvykle vyhodnocováno jako poměr doby výpočtu nejlepšího známého sekvenčního algoritmu a doby výpočtu paralelního algoritmu na témže (paralelním) počítači, využíváme-li p procesorů.

- **Účinnost** (angl. efficiency) je *urychlení* dělené počtem použitých procesorů. Například nejlepší sekvenční algoritmus se počítá na uvažovaném paralelním počítači 10 s, výpočet hodnoceného paralelního algoritmu pro $p = 4$ procesory trvá 5 s. *Urychlení* je 2.0 a účinnost je 0.5.

1.6.1 Amdahlův zákon

Kvalitní paralelní algoritmy musí nepochybně vykazovat dostatečné urychlení při dostatečné účinnosti (tj. využití procesorů). Dosažitelné urychlení je pro určitý algoritmus omezeno tzv. *Amdahlovým zákonem.* Tento zákon bere v úvahu, že výpočet zpravidla nelze paralelizovat úplně, určitá část musí být provedena sekvenčně. Označíme–li tuto část f, pak pro výpočet probíhající na p procesorech je dosažitelné urychlení S limitováno podle vzorce

$$S \ \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Amdahlův zákon byl po dlouhou dobu hlavním argumentem pochybovačů o užitečnosti masívně paralelních systémů, kteří zdůrazňovali omezení urychlení hodnotou $\frac{1}{f}$. Tyto obavy však vycházely z nesprávné extrapolace pro velký počet procesorů a z těchto ne zcela oprávněných předpokladů:

- byl uvažován jen tradiční způsob tzv. *postupné* paralelizace sekvenčních programů. Při tomto způsobu jsou nejprve nalezena místa s největšími výpočetními nároky. V dalších etapách jsou pak víceméně tradičními způsoby výpočty v těchto místech paralelizovány, dokud není dosaženo uspokojivých výsledků. Ostatní, z tohoto přístupu nezajímavá místa, jsou, aniž jim byla věnována dostatečná péče, prohlášena za nenapravitelně sekvenční. Jistě, jejich urychlení by možná vyžadovalo mnohem větší úsilí, ale nelze ho vyloučit jednou provždy. Vždyť pokud jedna část vyžaduje sekvenční zpracování, nemusí to ještě nutně znamenat, že ostatní procesory musí zahálet. Z těchto důvodů se proto postupná paralelizace mnohdy nazývá *částečná* a tím se dostáváme vlastně k jádru problému. Je nutné si uvědomit, že k problémům paralelních výpočtů nelze přistupovat na úrovni existujících programů, nýbrž na úrovni paralelních algoritmů.

- druhým, mnohdy zastřeným předpokladem je *neměnnost* podílu f. Použití paralelních počítačů umožňuje s rostoucím počtem procesorů řešení stále rozměrnějších úloh. Podle praxe se přitom zároveň ukazuje, že objem sekvenčních výpočtů narůstá velmi pomalu a podíl f se tedy velmi rychle zmenšuje.

Je tedy nutné přeneseně uvažovat závislost podílu f na počtu procesorů. Označme dále $\overline{f}(p)$ časový podíl sekvenčního výpočtu, který je „obvyklý“ pro rozměry úloh řešených p procesory. Podíl výpočtu části vhodné pro paralelizaci je pak $1 - \overline{f}(p)$ a tuto část můžeme v ideálním případě urychlit p-krát. Urychlení oproti výpočtu jedním procesorem[8] je tedy omezeno hodnotou

$$S \leq p * (1 - \overline{f}(p)) + \overline{f}(p)$$

Je tedy vidět, že při platnosti našich předpokladů, není využitelnost masívně paralelních systémů předem nijak omezena.

Nicméně, „nelimitní“ Amdahlův zákon je užitečný při analýze výkonu paralelních aplikací, pokud nás zajímá efektivnost paralelizace.

Anomální urychlení

Při zběžném pohledu se zdá samozřejmým, že urychlení výpočtů nemůže být lepší než lineární podle počtu procesorů. Tato úvaha však neuvažuje, že více procesorů pohromadě může poskytovat nejen akumulaci výkonu, nýbrž i dalších zdrojů. Vedle běžných, vcelku logických odlehčení, např. rozdělení paměťového prostoru na více menších částí může velmi omezit nutnost odkládání na disky, lze někdy pozorovat i urychlení *superlineární*, tedy lepší než lineární. Nejčastěji se tak děje v těchto dvou případech:

- Efekt cache-paměti. Rozdělením výpočtu mezi více procesorů může dojít za příznivých podmínek k daleko častějšímu uplatnění lokálních cache-pamětí. Každý lokální výpočet je pak prováděn rychleji než v případě výpočtu jedním procesorem. Pokud algoritmus sám o sobě

[8]Pokud by ovšem pro tak rozměrný výpočet jednoprocesorové prostředí poskytovalo dostatečné zdroje, např. paměť.

vykazoval dobré urychlení, může pak být urychlení celkové lepší než lineární.

- Anomálie při vyhledávání. Paralelizované prohledávací algoritmy metodou „uřezávání“[9] pracují také velmi často rychleji, než by odpovídalo lineárnímu urychlení. Zde je to hlavně díky tomu, že paralelizovaný algoritmus je vlastně odlišný od sekvenčního a umožňuje většinou rychlejší upřesňování průběžného výběrového kriteria.

1.6.2 Rozdělení celkového výpočetního času

Amdahlův zákon je poplatný době svého vzniku a hodí se spíše pro systémy se sdílenou pamětí. Při složitějších architekturách je pro posouzení možností paralelizace nutné sledovat celkový čas výpočtu detailněji. Obvykle lze rozlišovat tyto hlavní položky: vlastní (procesorový) výpočetní čas, čas strávený komunikací a čas spotřebovaný čekáním. Zbývající položka, *administrativní režie výpočtu*, tj. čas potřebný pro zakládání a rušení procesů, je spíše jednorázová a projevuje se především u méně objemných výpočtů. Podle specifických vlastností použité počítačové architektury můžou být některé části dominantní.

Výpočetní čas procesoru

Zde není asi zapotřebí zvláštní vymezení. Tento čas závisí zejména na použitém procesoru a paměťovém systému. Nelze však uvažovat jednotlivé procesory odděleně, jak jsme již viděli, zpracování více procesory může ovlivnit zřetězování instrukcí, způsob přístupu do paměti, využití cache-paměti. Nelze tedy paradoxně vyloučit závislost rychlosti výpočtu jednoho procesoru na počtu procesorů.

Komunikační čas

Tímto časem rozumíme čas, který úloha potřebuje pro vysílání a příjem zpráv. Je zvykem rozlišovat mezi komunikací *interprocesorovou* a *intraprocesorovou*. Pokud však není komunikace intraprocesorová vysoce optimalizována, není z dnešního pohledu jeden způsob výrazně lepší než druhý: moderní komunikační technologie nevyžadují pro obsluhu interprocesorové

[9] z angl. *branch-and-bound*

komunikace výrazně delší čas než čas pro přepínání kontextu při komunikaci intraprocesorové.

Časové prodlevy

Tento čas nelze dost dobře vymezit, většinou není explicitně zachycen v algoritmu. Důvody časových prodlev jsou hlavně tyto dva: nedostatek práce a nedostupnost dat. O co obtížněji lze krátit výpočetní a komunikační čas procesoru, o to snáze dochází k narůstání časových prodlev. I když lze někdy tyto problémy řešit pomocnými prostředky (rozdělování zátěže, překrývání výpočtu a komunikace), naznačují velké časové prodlevy nevhodnost použitého algoritmu.

1.6.3 Testy výkonnosti

Porovnání výkonnosti paralelních systémů lze ihned rozdělit na dvě části: výpočetní a databázovou.

Při testech výkonnosti z hlediska databázového zpracování jsou testovány spíše všechny komponenty systému (CPU, I/O, vyvážení paměti). Lze jen velmi těžko odhadovat jakou vahou se na výsledcích těchto testů podílí výkonnost paralelního výpočetního subsytému, a proto se touto oblastí již dále zabývat nebudeme. Jinak je zde však situace mnohem jasnější, protože téměř všichni výrobci se podřídili skupině testů podle TPC[10].

Panuje velmi mnoho názorů na testy výpočetního výkonu jednoprocesorových počítačů, o to je situace složitější pro systémy paralelní. Zatímco při hodnocení jednoprocesorových systémů se výrobci čas od času mlčky podřídí nějakému *de facto* standardu, v současné době to jsou testy podle SPEC95, u paralelních počítačů k podobnému sjednocení dojde jen velmi málokdy. Možná je to také způsobeno dost úzkým okruhem výrobců paralelních systémů.

Výkonnost paralelních systémů je posuzována, s ohledem na nejčastější použití pro náročné technické a vědecké výpočty, téměř výhradně podle výkonnosti v pohyblivé řádové čárce. Nejčastější se lze setkat s těmito způsoby hodnocení:

[10]Transactions Performance Council

- teoretický špičkový údaj (TPP[11]). Zde je výkonnost odvozena podle návodu výrobce z technických parametrů základního procesoru a násobením počtem procesorů. Tento údaj je téměř bezcenný, měl by spíše sloužit ke kontrole ztrát při reálném provozu.

- relativní výkonnost podle skupiny aplikací. Zde jsou asi nejvíce respektovány testy NPB[12] tvořené skupinou 8 aplikací z výpočtové mechaniky tekutin a pracování signálu. Zvláštní důraz je kladen na zajištění takových podmínek, které umožňují extrapolaci výsledků pro vymezenou skupinu programů. Není tedy zejména prováděna žádná optimalizace.

- největší kolekci výsledků se podařilo shromáždit pro testy mírně nesprávně nazvané `LINPACK`. Zkušebním testem je řešení soustavy řídkých lineárních rovnic. Jsou rozeznávány tři úrovně testů:

  – test řešení soustavy 100 rovnic pevně určeným fortranským programem (v tomto programu jsou užity procedury z knihovny LINPACK, zde je asi původ jména testu). Provozovatel testu může použít fortranský kompilátor podle své libovůle, je povolena jakákoliv optimalizace umožněná kompilátorem. Volby kompilátoru musí být uveřejněny spolu s výsledky testu. Pro běžného uživatele je tento test asi nejprůkaznější.

  – tzv. LINPACK 1K. Řešení soustavy 1000 rovnic, vše ostatní je na libovůli vykonavatele testu: numerický algoritmus, použitý programovací jazyk, pouze výsledky musí být identické s testovacími. Výkonost se určuje podle pevného počtu aritmetických operací. Tento test tedy vypovídá i o schopnostech programátorů ve výrobním závodě a asi příznivým způsobem ovlivňuje optimalizované knihovny dodávané s kompilátorem.

  – tzv. MP LINPACK. Testovací problém je stejný jako výše, je však uvolněna i jeho rozměrnost. Vykonavatel testu si může zvolit nejvhodnější počet rovnic a přizpůsobit ho počtu procesorů, přístupu do paměti apod.. Tento test dává představu o reálných omezeních daného systému. Zvláštní postavení má systém, který

[11] z angl. *Theoretical Peak Performance*
[12] NAS Parallel Benchmark

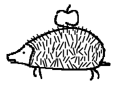
> má podle tohoto testu největší výkon. V daném okamžiku představuje maximální reálné možnosti paralelních technologií. Tuto ohraničující výkonnost lze dlouhodobě porovnávat a používat ke sledování trendů vývoje.

Je trochu šťastnou náhodou, že v době psaní těchto řádků, byla poprvé podle testu LINPACK MP překonána trochu magická hranice **1 Tflops**[13]. Tento výkon (přesněji 1.06 Tflops) byl dosažen 12. prosince 1996 v Sandia National Laboratory (USA) na paralelním počítači se 7264 procesory Intel Pentium Pro, 200 MHz. Druhý systém, dokonce běžně komerčně dosažitelný, který by tuto hranici mohl dnes překonat, by byl plně osazený počítač CRAY T3E–900 s 2048 procesory Alpha 450 MHz, u kterého byl prokázán trvalý výkon 504 Mflops/procesor.

[13] 1 Tflops = 10^{12} operací v pohyblivé řádové čárce za sekundu

Kapitola 2

Paralelní výpočetní procesy



Pro paralelní procesy se používá také název *souběžné procesy.* V angl. literatuře jsou označovány přívlastky concurrent, parallel, simultaneous.

Označení *paralelní* budeme chápat ve smyslu *logicky paralelní.* Upozorňujeme tím na okolnost, že program konstruovaný jako logicky paralelní může být vykonáván buď na jednom procesoru (tzv. pseudoparalelně), nebo na víceprocesorovém systému (tj. skutečně paralelně). V této kapitole zavedeme základní pojmy související s paralelními výpočty a probereme primitivní formy interakce paralelních procesů.

2.1 Programy a procesy

Pro rozlišení pojmů **program a proces** použijme analogie divadla. Scénáře jednotlivých rolí jsou obdobou programů, vystoupení herců odpovídají procesům. Herce samotné lze považovat za procesory. Např. pí. Bohdalová (procesor) realizuje proces Křemílek a proces Vochomůrka pseudoparalelním způsobem. Soubor opery (multiprocesor) realizuje množinu procesů nazvaných Prodaná nevěsta, prováděných podle programů napsaných Smetanou a libretistou.

Pojmy, které používáme v souvislosti s procesem:

OPERACE – elementární, na námi uvažované úrovni dále nedělitelná činnost při zpracování dat.

SEKVENČNÍ VÝPOČETNÍ PROCES (zkráceně proces) – uspořádaná množina operací, vyjadřující posloupnost operací v čase. Proces představuje činnost, a proto je charakterizovatelný v daném časovém okamžiku určitým stavem.

PARALELNÍ VÝPOČET – výpočet členěný do několika spolupracujících sekvenčních výpočetních procesů.

Odpovídající pojmy používané v souvislosti s programem:

INSTRUKCE – (ve smyslu strojová instrukce) je povel k provedení operace nebo popis této operace.

PROGRAM SEKVENČNÍHO VÝPOČTU – pevná posloupnost instrukcí. Program je formální popis činnosti, v čase výpočtu se nemění, a proto nemá žádný stav. Podle jednoho reentrantního programu může být prováděno i více procesů.

PARALELNÍ PROGRAM – formální popis paralelního výpočtu.

Pod pojmem program zahrnujeme jak program sekvenčního výpočtu, tak i paralelní program.

Pokud nebude uvedeno jinak, chápeme nadále proces jako sekvenční činnost. Formálně jej lze popsat dvojicí

$$P = \{P, D\}, \text{ kde}$$

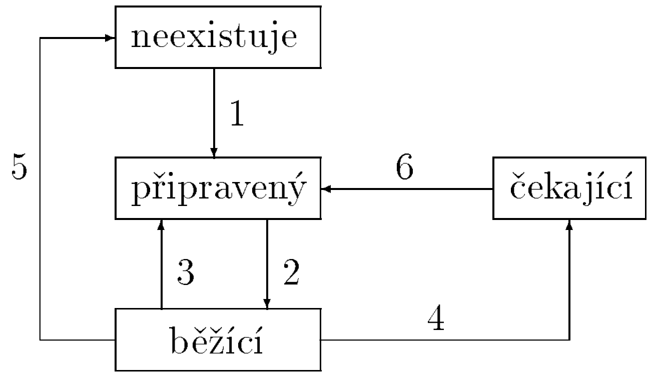
P je program, dle kterého proces probíhá,
D jsou lokální data procesu.

Proces může využívat některá data společně s dalšími procesy. Tato data se označují jako *sdílená* či *globální data* paralelního výpočtu (angl. *shared data*).

Jak bylo již uvedeno, nachází se proces v určitém časovém okamžiku v nějakém stavu. Stav procesu zahrnuje veškerou informaci o minulosti výpočtu, která zahrnuje:

- místo v programu kam dospěl výpočet (obsah čítače instrukcí),
- obsahy všech ostatních registrů,
- konkrétní hodnoty lokálních dat procesu.

Z hlediska řídícího algoritmu paralelního výpočtu (je jím obvykle jádro OS) se rozlišují pouze makrostavy. Typické makrostavy a přechody mezi nimi ukazuje obrázek 2.1.



Obr. 2.1: Diagram stavů procesu a mezistavových přechodů

Popis přechodů mezi stavy:

1 – vytvoření procesu (aktivitou OS nebo jiného procesu),

2 – spuštění procesu (provádí OS, vybírá proces z fronty připravených procesů, existují různé algoritmy výběru)

3 – přerušení procesu (provádí OS),

4 – běžící proces vyžadoval něco, co zatím není k dispozici (obecně splnění nějaké podmínky),

5 – proces došel na konec svého programu,

6 – je splněna podmínka, na kterou proces čekal.

Poznámky:

- Každý proces má v datové oblasti řídícího algoritmu svůj informační záznam (angl. *control block*, CB), který obsahuje údaje důležité z hlediska řídícího algoritmu (zejména údaje o stavu procesu).
- V souvislosti s výměnou běžícího procesu se používá pojmu *přepnutí kontextu*. Při něm se provede uložení stavových údajů přerušeného procesu do CB přerušeného procesu a naplnění registrů na základě stavových údajů spouštěného procesu z jeho CB.
- Z hlediska souvislé doby provádění rozlišujeme:

a) procesy s dlouhou životností (s dlouhou dobou výpočtu). Pak nejsou výrazné nároky na rychlost přepnutí kontextu. Např. typická doba pro OS **Unix** je 100 mikrosekund.

b) procesy s krátkou životností (s krátkou dobou výpočtu). V takovém případě záleží výrazně na časové režii potřebné k vytvoření, spuštění a ukončení procesu. Pro tento případ je účelné využít k realizaci výpočtu speciální architektury (pro transputery je typická doba přepnutí kontextu 1 mikrosekunda). Vyplatí se pak vytvářet paralelní procesy i pro vykonání jednotlivých příkazů vyššího programovacího jazyka (viz dále **Occam**).

2.2 Základní formy interakce

Procesy vytvořené v rámci jednoho programu spolu obvykle nějakým způsobem spolupracují za účelem splnění určitého cíle výpočtu.

Formy interakce rozdělíme na:

- základní (jsou obvykle implementovány na úrovni jádra OS jako volání služeb jádra),

- strukturované (dokonalejší, zaváděné ve vyšších programovacích jazycích). Využívají základních forem interakce. Jedná se např. o monitory (**MODULA**) nebo rendezvous (**ADA**). Koncepty monitoru a rendezvous budou vysvětleny v kapitole dedikované strukturovaným formám interakce.

Základní formy interakce jsou:

- synchronizace,

- sdílení dat (a dalších zdrojů systému),

- předávání dat (např. komunikace zasíláním zpráv),

- přímé ovlivňování stavu procesu (např. vytvoření a aktivace procesu z jiného procesu).

2.2.1 Synchronizace

Princip:

j-tá operace procesu A může být provedena až po ukončení k-té operace procesu B.

Důsledek:
Proces A se synchronizuje na událost (podmínku), kterou je ukončení k-té operace procesu B. Přitom mohou nastat dva případy:

- A se zdrží (B ještě neukončil k-tou operaci),
- A se nezdrží (B již ukončil k-tou operaci).

V paralelním výpočtu složeném z několika procesů zpravidla nelze předem stanovit časové poměry výpočtu. Proto nejsme schopni rozhodnout, nastane-li první nebo druhý případ. Obvykle je třeba počítat s oběma možnostmi. V paralelním výpočtu se běžně předpokládá , že každý proces probíhá předem nepredikovatelnou rychlostí (jedná se o tzv.*rychlostně nezávislý výpočet*).

Při správně vyřešené synchronizaci paralelních procesů je zaručeno úspěšné dokončení výpočtu nezávisle na rychlosti vykonávání jednotlivých procesů. Proto může být tentýž program úspěšně provozován na jednoprocesorovém počítači (tj. pseudoparalelně), nebo na víceprocesorové struktuře (tj. fyzicky paralelně).

K řešení synchronizace úloh se využívá programové konstrukce nazývané *semafor* (zavedl Dijkstra v r.1968). Pro každý synchronizační účel se zřizuje zvláštní semafor.

Příklad datové struktury semaforu znázorňuje obr.2.2.

Položky *head* a *tail* ukazují na začátek a konec fronty procesů čekajících na tento semafor. Nulová hodnota čítače *count* znamená zavřený semafor, nenulová hodnota znamená otevřený semafor. Nabývá-li čítač hodnot 0, 1, označujeme takový semafor jako „binární“. Jeho prostřednictvím lze vyjadřovat obsazenost (0) či použitelnost (1) jím chráněného sdíleného zdroje. Pro synchronizaci dvou procesů je použitelný binární semafor. Pokud semafor chrání přístup ke skupině n sdílených zdrojů, jeho čítač nabývá hodnot z intervalu $0 \ldots n$.

count	celočíselný čítač
head	
tail	

Obr. 2.2: Datový obsah semaforu

Nad semafory jsou definovány základní operace P a V s významem:

V ... nastavení (otevření) semaforu,

P ... čekání na semafor.

Operace V má následující sémantiku:

a) Pokud je neprázdná fronta procesů čekajících na otevření semaforu, je proces z čela fronty převeden do stavu připravený a hodnota čítače *count* zůstane nezměněná.

b) Pokud žádný proces nečeká, inkrementuje se hodnota *count*.

Operace P rovněž zahrnuje dvě možnosti:

a) Pokud je $count > 0$, dekrementuje se *count* a ve výpočtu pokračuje proces, který provedl operaci P.

b) Pokud je *count* nulový, je volající proces zařazen do fronty čekajících procesů a místo něj je spuštěn jiný (připravený) proces. Dojde tedy k přepnutí kontextu.

V dalším výkladu budeme předpokládat existenci operací P a V, realizovaných jako volání jádra OS.

Před prvním použitím je nutné semafor nastavit. Předpokládejme tedy ještě existenci operace *INITSEM* (*semafor* , c), kde *semafor* je odkaz na semafor a c je výchozí hodnota čítače, představující počet semaforem chráněných prostředků.

Předpokládáme nedělitelnost (atomicitu) operací P a V. Je-li operace jednou započata, proběhne bez přerušení až do konce. Není možné, aby proces A a proces B prováděly současně obě operace P i V nad stejným semaforem.

Z uvedeného je zřejmé, že proces čeká na otevření semaforu v neaktivním stavu (a nespotřebovává přitom čas procesoru). Druhou možností je aktivní čekání (busy wait) v programově realizované přerušitelné čekací smyčce. Tento druhý způsob se využívá pro tzv. paměťové zámky (memory lock) – viz dále. Operace uzamčení a odemknutí zámku odpovídají operacím P a V nad semaforem. Rozdíl je jen ve způsobu čekání. Čekat v programové smyčce se vyplatí tehdy, je-li typická doba čekání kratší než doba přepnutí kontextu.

Poznámky:

1. Uvedené synchronizační operace jsou elementární a s jejich pomocí se dají implementovat další složitější formy interakce procesů, o kterých pojednáme později.

2. Operace se semafory jsou nepřehledné, obtížně kontrolovatelné překladačem, proto je při komplikovanější synchronizaci značné nebezpečí zablokování (angl. *deadlock*).

3. Pokud může čítač semaforu nabývat jen hodnot 0/1, jedná se o binární semafor. Jeho význam je podobný signálu.

Operaci	*VYŠLI(signal)*	-	odpovídá	*V(semafor)*,
	ČEKEJ(signal)	-	odpovídá	*P(semafor)*.

   Rozdíl semaforu a signálu je ve způsobu jejich použití. Signál synchronizuje dva procesy, z nichž jeden používá operaci *VYŠLI* a druhý operaci *ČEKEJ*. Proces pracující se semaforem používá jak P, tak i V operaci.

4. Formálním aparátem umožňujícím ověřit logicky správné řešení synchronizace (tj. vylučující zablokování) jsou Petriho sítě.

2.2.2 Sdílení dat a zdrojů

Zdroji (prostředky) obhospodařovanými systémem rozumíme soubory, periferní jednotky apod.

Sdílení dat je typickou technikou v systémech obsahujících globální (sdílenou) paměť, přístupnou ze všech procesorů a procesů na nich probíhajících. Sdílení dat je např. použitelné v symetrických multiprocesorech řady Sequent, ale není možné v transputerových strukturách.

Používané pojmy:

sdílená data (shared data)

– data, která logicky patří k sobě a jsou společná několika procesům. Jejich zápis musí být nedělitelnou operací. Např. proměnná typu float je tvořena čtyřmi byty, které se musí přepisovat (a číst) nedělitelně.

vzájemné vyloučení (mutual exclusion)

– pokud může pracovat několik paralelních procesů se sdílenou datovou strukturu, je třeba vyloučit současný přístup těchto procesů ke sdíleným k datům.

kritická sekce (critical region)

– sekvence příkazů pracujících se sdílenou datovou strukturou (je třeba vyloučit současný přístup k datům).

Programové ošetření kritické sekce lze provést pomocí binárního semaforu.

Příklad 2.1

Uvažujme situaci, kdy dva procesy sdílí datový záznam.

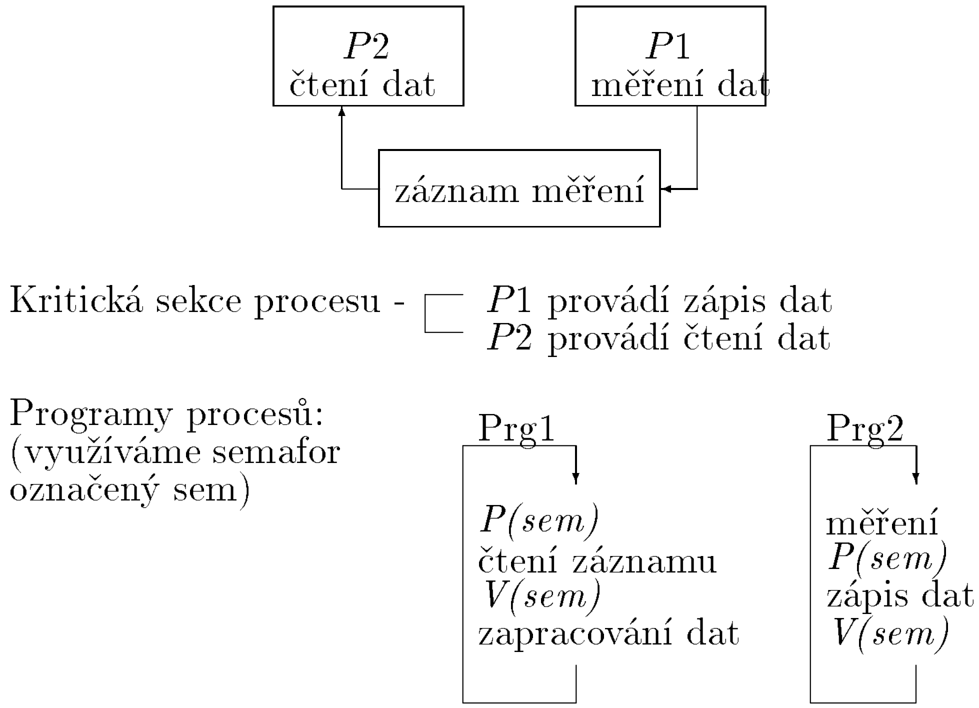
Proces $P1$ ... měří údaje z reálného světa. Probíhá cyklicky s nějakou frekvencí opakování.

Proces $P2$ ... čte naměřené údaje. Probíhá cyklicky, s jinou frekvencí než $P1$.

Spolupráci procesů a náznak jejího programového řešení ukazuje obr.2.3.

Před vytvořením procesů $P1, P2$ je nutné provést počáteční nastavení semaforu příkazem:

INITSEM(sem, 1)



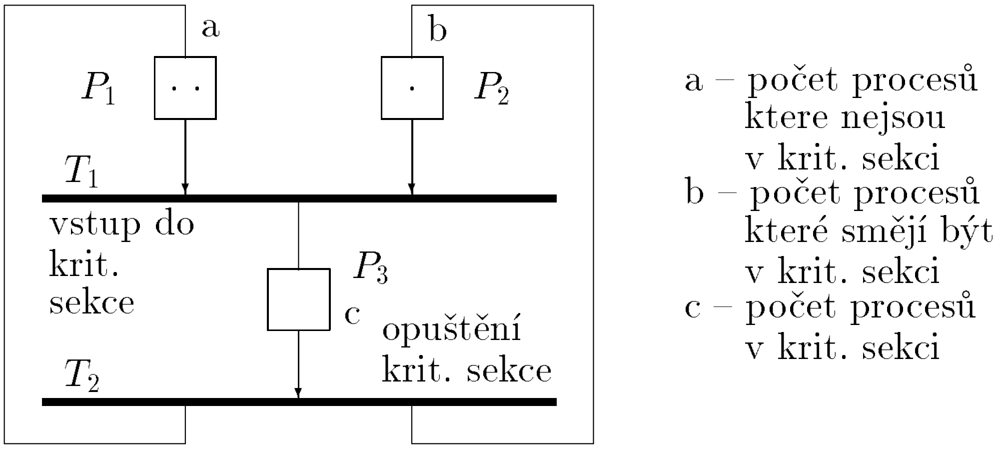
Obr. 2.3: Spolupráce procesů měření (zápisu) a čtení dat

Poznámky:

1. Při složitějších synchronizačních úlohách je obtížné zajistit správné umístění příkazů P a V.

2. Pro ověření správnosti synchronizace (speciálně pro ověření, že nedojde k zablokování) jsou používány Petriho sítě. Uceleně o nich pojednáme později.
   Petriho síť uvedeného příkladu ilustruje obr. 2.4.

   Stav sítě je určen „hodnocením“míst P_1, P_2, P_3 tečkami. Obr. 2.4 uvádí počáteční ohodnocení. Přechody T_1, T_2 jsou možné, pokud všechna vstupní místa přechodu mají nenulové ohodnocení. Uskutečněný přechod vede do stavu, ve kterém je ohodnocení všech vstupních míst přechodu dekrementováno a ohodnocení všech vstupních míst inkrementováno. *Analýzou všech možných stavů sítě lze odhalit stavy, ze kterých nelze nikam přejít (tzv. mrtvé stavy).*



Obr. 2.4: Petriho síť příkladu 2.1

2.2.3 Předávání dat a zpráv

Předávání dat a zpráv lze provádět prostřednictvím:

- asynchronní komunikace,
- synchronní komunikace.

Při asynchronní komunikaci odesílající proces vloží zprávu do vyrovnávací paměti a pokračuje ve své činnosti. Přijímající proces zprávu vybere z vyrovnávací paměti v okamžiku kdy ji potřebuje a pokračuje ve své činnosti. Pokud zpráva není připravena, proces čeká na její zaslání. Při dostatečně velké vyrovnávací paměti **nedochází ke zdržení procesů** (předp. stejné střední frekvence zápisu a čtení).

Synchronní komunikace znamená přímou účast obou procesů na výměně zprávy. Implementace synchronní komunikace zpravidla nevyžaduje vyrovnávací paměť. Jeden z obou účastníků musí počkat, až bude druhý připraven k předání zprávy. Synchronní způsob komunikace umožňuje jednak synchronizaci činností spojenou s výměnou dat, nebo prostou synchronizaci, pokud je informační obsah vyměňované zprávy nulový.

Jiným možným rozlišováním způsobů komunikace je dělení dle způsobu adresace komunikujících procesů. Rozlišujeme :

- Symetrickou adresaci. V operacích pro odeslání a příjem zprávy se uvádí jména obou komunikujících procesů. Prakticky se nepoužívá.

- Asymetrickou adresaci. V operaci pro odeslání zprávy se uvádí pouze jméno příjemce zprávy. Od synchronní komunikace s asymetrickou adresací je odvozen koncept rendezvous ADY.

- Nepřímou adresaci. V operacích pro odeslání a příjem zprávy se neuvádí jména odesilatele ani příjemce. Komunikace se uskutečňuje přes logický komunikační kanál (nebo tzv. schránku, angl. mailbox), jehož jméno je známé oběma účastníkům komunikace. Tento způsob je použit např. v jazyce **Occam**.

Základní operace pro komunikaci s nepřímou adresací mají tvar:

SEND (kanál, zpráva)

→ adresa v lokální paměti
odesílajícího procesu

TAKE (kanál, zpráva)

→ adresa v lokální paměti
přijímacího procesu

Podrobnější rozbor uvedených možností lze nalézt v lit. [Pláš91].

2.3 Strukturované prostředky interakce procesů

Doposud diskutované programové prostředky pro vzájemnou spolupráci paralelních procesů lze charakterizovat jako nízkoúrovňové. To, že jsou implementovány voláním funkcí jádra OS, má za důsledek jejich relativní nebezpečnost. Překladač totiž nemůže být nápomocen při vyhledávání logických chyb v synchronizaci a komunikaci procesů. Uveďme základní důvody nebezpečnosti a nedokonalosti nízkoúrovňových prostředků:

- Opomene-li se provedení operace V pro uvolnění semaforu (otevření), dojde k zablokování systému.

- Opomene-li se provedení operace P pro obsazení semaforu (čekání), může dojít k interferenci dat.

- Semafory resp. signály nebrání možnosti skoku do kritické sekce, s následnou interferencí sdílených dat.

- Není možné naprogramovat alternativní akce pro případ, že semafor je obsazen.
- Není možné čekat na uvolnění jednoho z více semaforů.

Chyby, které není schopen odhalit překladač, se projeví až při výpočtu. Jejich detekce bude navíc ztížena náhodností výskytu chyb, ovlivňovanou rychlostí výpočtu jednotlivých procesů. Takové chyby označujeme jako *časově závislé.*

Zvýšení bezpečnosti paralelních programů je možné dosáhnout zahrnutím programovacích prostředků pro paralelní zpracování přímo do vyššího jazyka (v podobě strukturovaných konstrukcí vyjadřujících interakci procesů). K nejznámějším patří *monitor* a *rendezvous*, s jejichž podobou se seznámíme v programovacích jazycích **MODULA** a **ADA**.

2.3.1 Monitory

Monitor je programový modul, určený pro práci v paralelním prostředí. V monitoru jsou zapouzdřena sdílená data spolu s procedurami, které tato data zpřístupňují. Procedury monitoru mají speciální vlastnost: Vždy pouze jeden proces může v jednom časovém okamžiku aktivně provádět monitorovou proceduru v daném monitoru.

Ideu monitoru zavedli nezávisle na sobě Brinch Hansen (1973) a Hoare (1974).

Monitory jsou vyšší jazykovou konstrukcí pro implementaci vzájemně výlučného přístupu ke sdíleným zdrojům. Lze jimi chránit tedy nejen data sdílená více procesy, ale i např. zajistit řádné sdílení periferního zařízení.

Vlastnosti monitorů:

- V okamžiku, kdy proces provádí monitorovou proceduru, může jeden, několik, ale také žádný z procesů požadovat služby monitoru.
- Požaduje-li proces provedení monitorové procedury v době, kdy jiný proces je monitorem obsluhován, je žadatel pozastaven a umístěn do fronty.
- V okamžiku, kdy proces opouští monitor, je tato fronta prohledána a není-li prázdná, je aktivován proces z jejího čela.

- Proces, který právě provádí monitorovou proceduru, může žádat někdy i takovou službu, kterou monitor není schopen okamžitě poskytnout. V takovém případě vyvolá uvnitř monitorové procedury operaci *wait(podmínka)*. Není-li podmínka splněna, dojde k tzv. „uspání“ procesu v monitoru. Aby nedošlo k zablokování monitoru, vyjímá se operace wait(podmínka) ze vzájemného vyloučení. Tj. v monitoru může být k procesů, ale $k-1$ jich musí být uspáno v monitoru příkazem wait(podmínka).

- „Probuzení“ procesů uspaných v monitoru se provádí pomocí operace *send(podmínka)*. Tato operace není vyjmuta ze vzájemného vyloučení, neboť se předpokládá, že je použita jako poslední operace monitorové procedury.

- Formálně můžeme monitor popsat jako trojici $M = \{D, Op, W\}$, kde D je množina dat, Op množina operací nad daty (procedur) a W množina procesů uspaných v monitoru.

Monitory se uplatňují při rozdělování paralelně proveditelných funkcí programu mezi moduly. Realizují se jimi ty služby, které jsou využívány více procesy.

programové moduly využívané k výpočtu lze dělit na :

- dominantní (aktivní), též client
  implementují algoritmus (zajišťují činnost), mají programovou podobu paralelních procesů

- služební (pasivní), též server
  neimplementují samotný algoritmus. Na požádání poskytují služby aktivním procesům. Dělí se na moduly využívané jedním procesem (obálky, envelope) a na využívané více procesy (monitory).

Na uplatnění monitorů má zásluhu především jazyk **MODULA**. V dalším výkladu se proto stručně zmíníme o základních vlastnostech tohoto jazyka, s důrazem na jeho schopnosti vyjadřovat paralelní zpracování.

Stručný přehled programovacího jazyka MODULA

Uváděný přehled se týká **MODUL**y–2, která je z rodiny jazyků **MODULA** dosud nejpopulárnější. Je jazykem pascalského typu, dominantní konstrukcí je modul. Deklarace modulu má syntaktický tvar:

```
MODULE jméno_modulu [priorita];
  FROM jméno IMPORT seznam_jmen; (* seznam jmen importovaných
       objektů z modulu jméno. Část from lze užít násobně. *)
  EXPORT seznam_jmen; (* seznam jmen exportovaných objektů *)
  CONST ...
  TYPE ...
  VAR ...
  PROCEDURE ...
  MODULE ... (* viditelnost lze řídit i vnořováním modulů *)
BEGIN
  příkazy;
END jméno_modulu;
```

Část *příkazy* je provedena při zpracování deklarace modulu a využívá se k inicializačním akcím.

Seznam `EXPORT` může obsahovat:

- jména konstant,
- jména procedur,
- jména proměnných,
- jména typů,
- jména modulů.

Hlavní program má rovněž formu modulu. Jeho příkazy jsou mezi `BEGIN` a `END`. Pochopitelně neobsahuje seznam `EXPORT`.

Důležitá je možnost moduly překládat separátně a vytvářet si tak knihovnu s potřebnými službami.

Prostředky MODULY pro paralelní programování

Kromě konstrukcí již zmíněných v předchozím textu, je k dispozici knihovní modul `Processes`. Ten exportuje:

- Typ `SIGNAL`,
- proceduru `StartProcess`(*BezparamProcedura; CeléNezápornéČíslo*) (startuje proces popsaný bezparametrovou procedurou a přiděluje mu paměťový prostor o velikosti určené celým číslem,

- proceduru `SEND(VAR s: SIGNAL);` jeden proces čekající na signál `s` je aktivován,
- proceduru `WAIT(VAR s: SIGNAL);` proces přejde do stavu čekající na signál `s`,
- proceduru `Awaited(s: SIGNAL): BOOLEAN;` zjistí, zda někdo čeká na signál `s`,
- proceduru `Init(VAR s: SIGNAL);` inicializace fronty na `s`.

Uvedené prostředky nyní použijeme k vytvoření vyrovnávací paměti (realizované frontou), do které mohou ukládat a z ní vybírat různé procesy. Bude mít podobu modulu – monitoru. Tím zamezíme možnosti kolize při současném čtení i zápisu z několika procesů.

Příklad 2.2

```
MODULE VyrovnavaciPamet [1];
     EXPORT Uloz, Vyber;
     FROM Processes IMPORT SIGNAL,SEND,WAIT,Init;
     FROM NejakyModul IMPORT ElementType;
     CONST N = 256;      (* delka fronty *)
     VAR   n: [0..N];    (* pocet vlozenych elementu *)
           neplny:   SIGNAL;    (* pro n < N *)
           neprazdny:SIGNAL;    (* pro n > 0 *)
           in, out: [0..N-1];   (* pristupove indexy *)
           buf: ARRAY[0..N-1] OF ElementType;

     PROCEDURE Uloz(x: ElementType);
     BEGIN
           IF n = N THEN WAIT(neplny) END;
           n := n + 1;
           buf[in] := x; in := (in + 1) MOD N;
           SEND(neprazdny)
     END Uloz;

     PROCEDURE Vyber(VAR x: ElementType);
     BEGIN
```

```
            IF n = 0 THEN WAIT(neprazdny) END;
            n := n - 1;
            x := buf[out]; out := (out + 1) MOD N;
            SEND(neplny);
     END Vyber;

BEGIN       (* inicializace *)
     n := 0; in := 0; out := 0;
     Init(neplny); Init(neprazdny)
END VyrovnavaciPamet
```

Vzhledem k vysoké prioritě procedur tohoto monitoru (zajištěno číslem 1 v jeho záhlaví), je zaručena nepřerušitelnost procedur. Procesy, které monitor využívají, nemohou proto po přerušení (např. od časovače) převzít procesor před dokončením monitorové procedury.

2.3.2 Rendezvous

V české terminologii se užívá také názvu souběh či schůzka. Oproti monitoru, kterým jsme schopni zvládat problémy paralelismu na jednoprocesorovém systému, či na multiprocesoru se sdílenou pamětí, uplatňuje se rendezvous i v případě paralelních procesů probíhajících v distribuovaném prostředí (volně vázané multiprocesorové systémy).

Podstatou rendezvous je společné provedení určitého úseku programu dvěma procesy

Historie rendezvous sahá do r.1978, kdy Brinch Hansen zavedl jazyk pro distribuované procesy (dále DP) a současně Hoare jazyk pro komunikující sekvenční procesy (dále **CSP**). Oba návrhy vychází z myšlenky, že jak předávání dat mezi procesy, tak i jejich synchronizace jsou neoddělitelné aktivity. Princip, který navrhli, předpokládá:

Jestliže proces A chce předat zprávu procesu B, pak oba procesy musí dát najevo přání komunikovat. Pokud proces A provede požadavek předání jako prvý, je pozastaven až do té doby, kdy proces B je ochoten ji přijmout (a naopak). Jakmile jsou procesy synchronizovány, provede se komunikační akce a procesy pokračují dále již samostatně ve výpočtu.

Takový způsob komunikace se nazývá rendezvous

Rozdíl mezi DP a **CSP** je ve způsobu, kterým procesy komunikují. V **CSP** zasílá proces A zprávu procesu B provedením příkazu tvaru:

$$B \ ! \ akce(x)$$

Proces B přijímá zprávu příkazem:

$$A \ ? \ akce(y)$$

Identifikátor akce určuje typ zprávy. V uvedeném příkladu zpráva obsahuje jeden údaj, jehož hodnota je poskytována procesem A prostřednictvím proměnné x a přijímána procesem B v proměnné y. Proces B musí tudíž uvést jak proces, z něhož je ochoten přijímat zprávu, tak i typ zprávy ($akci$). Způsob komunikace je z hlediska obou procesů symetrický.

V DP je navržena přijatelnější forma komunikace pomocí procedury. Proces A volá proceduru P v procesu B příkazem např.:

call B.P(parametry)

V procesu B je pak uvedena pouze deklarace procedury P, takže proces B neuvádí jméno spolupracujícího procesu. Tento způsob komunikace je tedy asymetrický. Důsledkem je, že dovoluje použití služebních (knihovních) procesů, pro něž při vytváření jejich programového modulu není ještě znám komunikující partner. Proto byl také asymetrický způsob akceptován v jazyce **ADA**.

Rozdíl mezi **CSP** a DP je také ve způsobu provedení synchronizace. V obou řešeních musí jeden proces čekat na příslušný příkaz druhého procesu. V případě **CSP**, jakmile oba procesy si předají parametry, pokračují ve výpočtu paralelně. V řešení DP je proces volající proceduru P pozastaven, dokud druhý proces neprovede volanou proceduru. Teprve pak pokračují oba procesy v paralelním výpočtu.

K řízení nedeterminismu (volnosti ve výběru jednoho z několika možných příkazů) využívá **CSP**, DP i **ADA** tzv. strážených příkazů. Ty umožňují, aby proces vybral jeden z několika možných příkazů, jejichž proveditelnost je obvykle vázána na splnění podmínek vyjádřených předponami příkazů (viz příkaz select v kap.3).

Protože v případě distribuovaných procesů nemohou mít procesy společné proměnné, je nutné provádět jejich interakci prostřednictvím lokální proměnné pomocného (pro tento účel navrženého) procesu. Příklady použití rendezvous záměrně v této části textu neuvádíme. Budou presentovány později, v souvislosti s výkladem jazyka **ADA**.

2.4 Modely paralelních výpočtů

Modely paralelních výpočtů umožňují prostřednictvím matematického aparátu či simulačními nástroji zkoumat vlastnosti a chování výpočetního systému tvořeného paralelně prováděnými procesy.

Na základě primárního použití modelů můžeme rozlišit:

- modely pro analýzu uvíznutí (mrtvých stavů),
- modely pro analýzu výkonostních ukazatelů,
- modely umožňující důkaz správnosti chování paralelního programu.

Precedenční graf

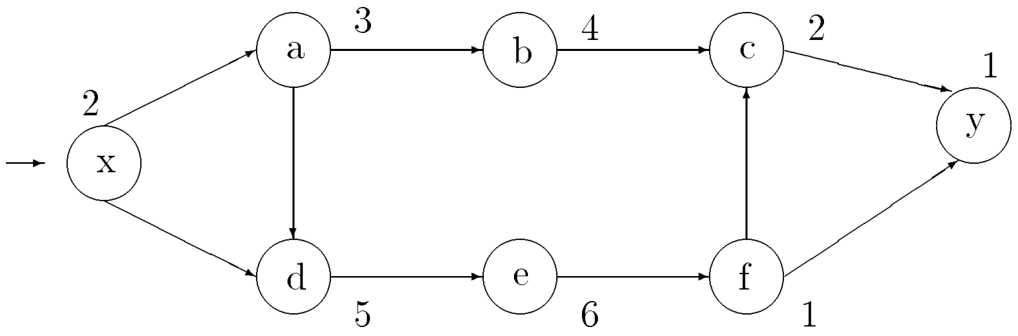
Precedenční grafy jsou nejjednodušší formou vyjádření návazností paralelně prováděných aktivit. Uzly grafu reprezentují jednotlivé procesy a hrany představují svou orientací vzájemnou precedenci procesů. Proces je v tomto případě nedělitelná aktivita, komunikující se sousedními procesy pouze na začátku a na konci své činnosti. Hrany proto reprezentují i komunikaci mezi procesy např. formou předávání výsledků. Z vlastností precedenčního grafu plyne:

1. Pokud graf neobsahuje cykly, nemůže dojít k uvíznutí programu.
2. Pokud ohodnotíme každý z uzlů dobou potřebnou k jeho exekuci, můžeme analyzovat zrychlení výpočtu programu způsobené jeho paralelizací.

Uvažujme precedenční graf výpočtu uvedený na obr.2.5.

Uzel x představuje začátek výpočtu, uzel y představuje konec výpočtu. Pokud výpočet bude prováděn jednoprocesorovým systémem, bude potřebná doba exekuce součtem dob jednotlivých procesů, tj.

$$2 + 3 + 4 + 2 + 5 + 6 + 1 + 1 = 24$$



Obr. 2.5: Precedenční graf výpočtu

Bude-li výpočet prováděn na víceprocesorovém systému, bude nejdelší cesta v grafu určovat nejmenší možnou dobu výpočtu (pokud bude k dispozici potřebný počet procesorů). V uvedeném příkladu je nejdelší cestou

$$2 + 3 + 5 + 6 + 1 + 2 + 1 = 20$$

Výpočet může současně využít dvou procesorů takto:

$$\begin{array}{llllllll}
\text{procesor 1} & \rightarrow & a \rightarrow & b & \longrightarrow & & c \rightarrow & y \\
 & & & \searrow & & \nearrow & & \\
\text{procesor 2} & & & d \rightarrow & e \rightarrow & & &
\end{array}$$

Obecně můžeme vyjádřit maximálně možné urychlení výpočtu při paralelním zpracování vztahem

$$\frac{\sum \text{časů všech procesů}}{\sum \text{časů nejdelší cesty}}$$

rozsáhlé paralelní programy mohou být konstruovány propojením menších paralelních programů. Platí, že nejkratší možný čas exekuce rozsáhlého paralelního programu je při neomezeném počtu procesorů vždy menší než součet časů výpočtu jednotlivých částí tohoto programu. Precedenční grafy jsou základem metod CPM a PERT, které jsou používány v operační analýze.

Petriho sítě

Petriho sítě jsou prostředkem pro popis a analýzu toku informace a řízení v asynchronních paralelních systémech. Dovolují modelovat události a podmínky na kterých mohou události nastat. Petriho síť je grafem se dvěma typy uzlů:

- místa (označované kroužky),
- přechody (označované úsečkami).

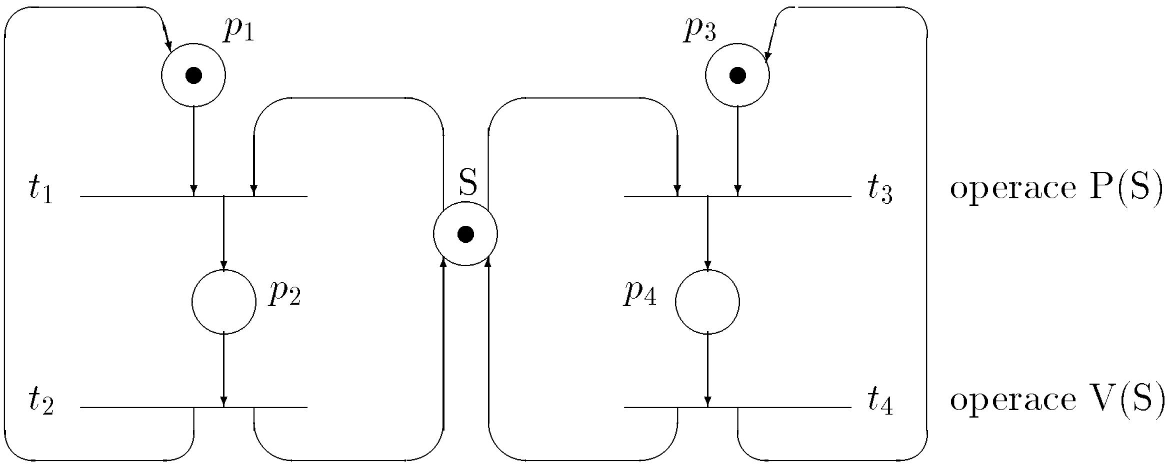
Stav modelovaného systému je popsán tzv. značením míst. Značení místa je celočíselná nezáporná hodnota, která je graficky znázorněna počtem teček (značek) obsažených v příslušném místě.

Místa a přechody jsou propojeny orientovanými hranami z míst do přechodů a z přechodů do míst. Aby mohla proběhnout událost modelovaná určitým přechodem, musí všechna místa z nichž vede hrana k přechodu (vstupní místa) obsahovat alespoň jednu značku. Provedením události (přechodu) je odebráno po jedné značce ze všech vstupních míst a současně do všech míst k nimž vede z přechodu hrana je po jedné značce přidáno. Vznik a zánik značek v místech při provádění přechodů reprezentuje různé možné stavy modelovaného systému.

V nejjednodušším případě tzv. C/E Petriho sítě (Condition/Event) představují místa logické (dvouhodnotové) podmínky. Platí-li podmínka, pak místo, které ji modeluje, obsahuje značku. Neplatí-li podmínka, je odpovídající místo prázdné. Obecnější použití mají Petriho sítě, které připouštějí větší počet značek v místech. Lze je výhodně použít např. při modelování systému s omezenými zdroji, kdy počet značek v místě reprezentuje počet zdrojů (např. délku fronty či hodnotu semaforu).

Obr. 2.6. ilustruje modelování vzájemného vyloučení pomocí binárního semaforu. Operace V přidává značku do S a operace P z něj značku odstraňuje. Místa p_1 a p_2 představují výpočtové akce jednoho procesu, místa p_3 a p_4 představují výpočtové akce druhého procesu. Jejich kritické sekce, prováděné v režimu vzájemného vyloučení, jsou reprezentovány místy p_2 a p_4. Z označené sítě plyne, že může být proveden pouze jeden z přechodů reprezentujících operaci $P(S)$.

Petriho síť je vhodným nástrojem pro zjišťování možného uvíznutí systému. V případě, že v systému může nastat uvíznutí, bude v jemu odpovídající Petriho síti existovat stav, ve kterém není možné provedení žádného z přechodů. Takový stav je nazýván „neživý“. Opakem je „živá“ síť, ve které je alespoň jeden přechod proveditelný. Pro analýzu možných stavů sítě



Obr. 2.6: Petriho síť modelující vzájemné vyloučení pomocí binárního semaforu

je použitelný tzv. strom dosažitelných značení. V případě konečné množiny možných stavů sítě, je jeho konstrukce zřejmá.

Protože listy stromu mají shodné značení s kořenem, má každý ze stavů sítě svého následníka a v modelovaném systému nemůže nastat uvíznutí. Uvedená Petriho síť je živá. Problém živosti sítě je ekvivalentní problému dosažitelnosti z teorie grafů.

Je-li stavový prostor Petriho sítě nekonečný, t.j. některá složka vektoru značení roste nade všechny meze, pak tuto složku označíme ω a příslušný vektor reprezentuje nekonečnou množinu značení. Strom dosažitelných značení je vlastně abstrakcí přechodové funkce Petriho sítě.

Ve své základní podobě nepracují Petriho sítě s časem, přechody jsou prováděny okamžitě. Zavedení časové míry dovolují tzv. stochastické Petriho sítě viz [PN90].

Axiomatická specifikace paralelního výpočtu

Axiomatická specifikace je použitelná k formálnímu důkazu správnosti programu. Je založena na tvrzeních o proměnných programu na začátku výpočtu, v jeho průběhu a po jeho skončení. Ověření správnosti programu je prováděno dokazováním teorémů typu:

$$< P > \ S \ \ < Q >$$

kde P a Q jsou tvrzení a S je příkaz (či příkazy). Interpretace teorému je tato:

Je-li P pravdivé před provedením S a S skončí, pak Q je pravdivé po provedení S. P nazýváme počáteční podmínkou a Q koncovou podmínkou teorému.

Použijeme axiomatický přístup pro důkaz správnosti paralelního programu zapsaného v jazyce **CSP**, kde paralelní provádění sekvenčních procesů $p_1, p_2, \ldots p_n$ označíme

$$[p_1 \,||\, p_2 \,||\, \ldots\, p_n].$$

Např. Předpokládejme, že chceme dokázat platnost teorému:

$$< true > \;\; [p_1 \,||\, p_2 \,||\, p_3] \;\; < x = u >$$

kde	p_1	provádí	$p_2 \;!\; x$	
	p_2	provádí	$p_1 \;?\; y$	; $\quad p_3 \;!\; y$
	p_3	provádí	$p_2 \;?\; u$	

Význam uvedených zápisů je tento:

$p_i \;!\; x$ znamená zápis do proměnné x,

$p_i \;?\; x$ znamená čtení do proměnné x.

Pro jednotlivé procesy můžeme dokázat tyto vlastnosti:

$$\begin{array}{lll} < x = z > & p_1 & < x = z > \\ < true > & p_2 & < y = z > \\ < true > & p_3 & < u = z > \end{array}$$

Poněvadž platí $x = z$ a také $u = z$, pak platí i $x = u$. Dokázali jsme, že pro libovolnou počáteční podmínku platí koncová podmínka $x = u$ pro paralelní program $[p_1 \,||\, p_2 \,||\, p_3]$.

Pro důkaz správnosti komunikujících procesů existují dva možné postupy:

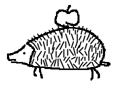
1. Rozdělit důkaz správnosti na dvě části. V prvé provést sekvenční ověření správnosti jednotlivých procesů, při kterém jsou vytvářeny předpoklady o účinku komunikačních příkazů. Ve druhé části je dokazována legitimita předpokladů.

2. Dokázat vlastnosti jednotlivých procesů pomocí axiomů a odvozovacích pravidel, která jsou aplikovatelná na příkazy jednotlivých procesů. Axiomy a pravidla jsou navrženy tak, že není nutné v sekvenčním ověření procesu uvažovat chování ostatních procesů. Ověřené vlastnosti jsou pak použity k důkazu správnosti celého programu.

Pro hlubší seznámení viz např. [Isip93].

Kapitola 3

Paralelní procesy v jednoprocesorovém systému



V jednoprocesorovém systému je nutné logicky paralelní program zpracovávat pseudoparalelně. Je-li program zpracováván pseudoparalelním způsobem, nelze kalkulovat s urychlením výpočtu. Z praktického hlediska má takový způsob zpracování smysl v následujících třech případech:

1. **Simulační výpočty orientované na procesy (Simula)**
   Modeluje se reálný svět, členěný do určitých souběžných činností.

2. **Zvýšení propustnosti systému a vyvážené zatížení zdrojů**
   Situaci ilustrujme příkladem:
   Uvažujme dva programy a jim odpovídající procesy, kde:

   $P1$ má charakter vědecko-technického výpočtu
   (přečte data, hodinu počítá a vytiskne tabulku výsledků na tiskárně),

   $P2$ provádí výpočet mezd
   (probíhá hodinu, z toho 50 min se převíjí mg. páska.)

   Alternativy zpracování jsou tyto:

   (a) *Sekvenční*

   – výpočet trvá dvě hodiny. Zdroje systému jsou využívány nevyváženým způsobem.

   (b) *Pseudoparalelní* – procesy se střídají o procesor

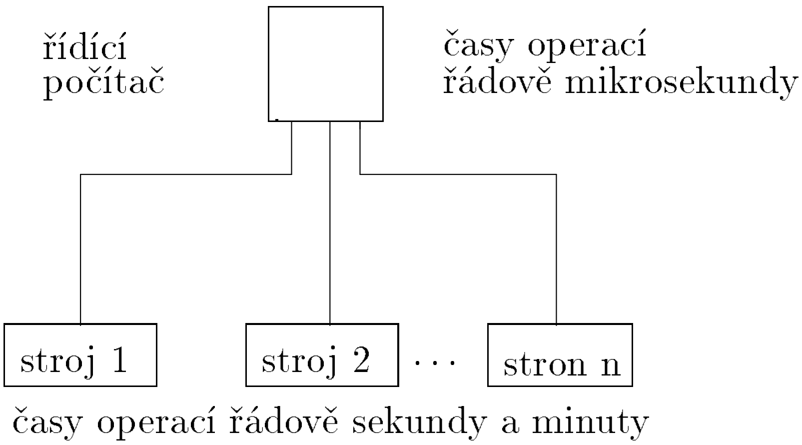
   – výpočet trvá něco málo přes hodinu, zatížení zdrojů je vyvážené. V tomto příkladě jsou oba procesy nezávislé, nekomunikují spolu. Takový způsob zpracování je označován jako multiprogramování.

Multiprogramováním (angl. *multiprogramming*) se rozumí pseudoparalelní zpracování zakázek různých uživatelů (na rozdíl od multiprocessingu, kterým je míněno členění jednoho aplikačního programu na paralelně vykonatelné části)

Poznámky k multiprogramování:

- Procesem se míní sekvenční výpočet jednoho programu. Podle programu probíhá jen jeden proces.
- Počet současně zpracovávaných programů se označuje jako *stupeň multiprogramování.*

3. **Systémy reálného času** (angl. *real-time systems*)
Příkladem je řízení průmyslových procesů, viz obr.3.1.



Obr. 3.1: Průmyslový systém reálného času

Pro řízení každého stroje (části technologie) je zřízen speciální proces. Předpokládejme, že zpracování jednoho procesu zatěžuje počítač relativně málo. Pak je možné přepínat procesor mezi jednotlivé procesy (možnosti jsou probírány v předmětu OS). Procesy spolupracují (viz výše uvedené formy) na společném cíli, kterým je řízení technologického procesu.

Programové vybavení členěné na procesy je logicky přehledné a odpovídá svojí strukturou reálnému světu aplikace.

3.1 Korutiny

Název *korutiny* (angl. coroutines) je zkrácením pojmu kooperující rutiny = spolupracující procedury. Jejich název vznikl v době, kdy rozlišení pojmů program a proces nebylo ještě aktuální. Korutiny jsou takové procedury, které si navzájem předávají procesor v monoprocesorovém prostředí. Každá z korutin je ve skutečnosti procesem. Je vyvořena specifikací příslušné procedury a paměťové oblasti. Po vytvoření je korutina proveditelná, zůstává však ve stavu připravenosti (t.j. neaktivní), dokud ji jiná korutina nevyvolá. Hlavní program je rovněž korutinou.

Korutiny alternují mezi stavy:

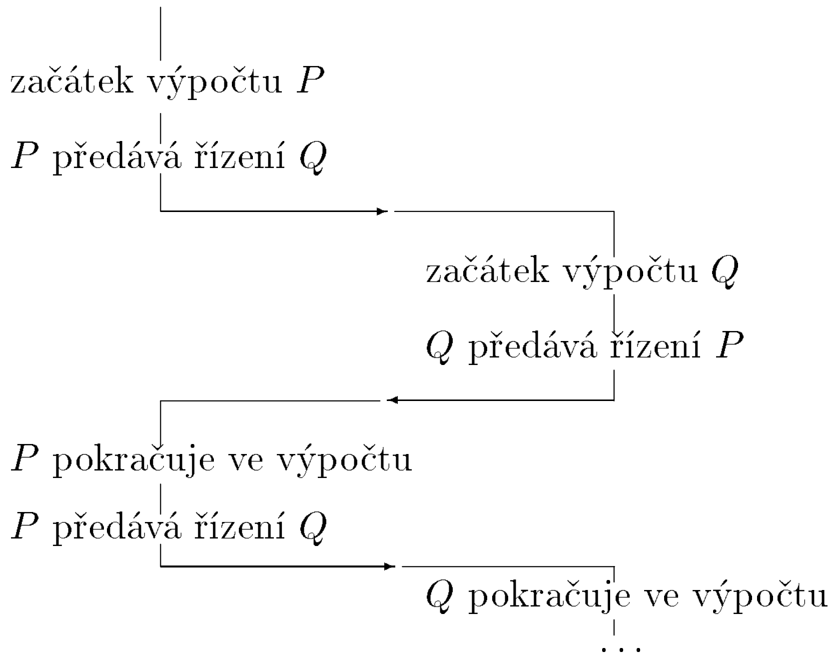
- běžící (t.j. aktivní), v tomto stavu může být vždy jen jedna korutina a dokud nevyvolá sama jinou korutinu, zůstává stále ve stavu běžící,
- připravený, do něj přechází volající korutina. Připravených korutin může být současně několik,

Korutiny se liší od procedur v tom, že:

- volání nemohou být rekurzivní,
- neimplikují návrat do místa volání. Výpočet může skončit, aniž by došlo k návratu do místa volání korutiny,
- nevyjadřují podřízenost volané vůči volající. Volaná korutina pokračuje ve své činnosti z toho místa, kde předala při svém posledním provádění řízení jiné korutině,
- jejich činnost není obecně ukončována průchodem koncem programového textu, jako je tomu u procedur. Obvykle mají podobu nekonečného cyklu, obsahujícího jedno či několik volání korutin,
- výpočet končí, dosáhne-li některá korutina konec těla svého programového popisu. Ukončí se tím současně i všechny ostatní korutiny. Korutina, která dospěla na konec svého programu vlastně ani nemá komu předat řízení.

Princip vzájemné součinnosti dvou korutin P, Q znázorňuje obr. 3.2.
Korutiny jsou užitečným prostředkem pro:

- konstrukci programových celků sestávajících z navzájem si nepodřízených komponent, pracujících nad společnými daty,



Obr. 3.2: Spolupráce korutin P a Q

- vytváření multiprogramových OS,
- konstrukci programů, které zpracovávají odezvy na události, ať vnitřní (např. časový tik), či vnější (např.stisk klávesy).
- programy simulující diskrétní systémy. Každá aktivita je modelována jako korutina. Předávání řízení mezi korutinami vytváří dojem jejich postupu v čase.

Některé jazyky, jako např. Simula, Concurrent Pascal a Modula mají vestavěné prostředky pro použití korutin. Naznačme řešení v jazyce Modula-2. Prostředky pro práci s korutinami poskytuje modul `SYSTEM`, exportující prostředky pro vytváření korutin a jejich volání.

Vytvoření korutiny a její ustavení do připraveného stavu zajišťuje procedura

```
NEWPROCESS(BezparametrováProceduraUrčujícíProgramNovéKorutiny;
           AdresaPracovníOblastiKorutiny;
           RozsahPracovníOblastiKorutiny;
           OdkazNaVytvářenouKorutinu);
```

Předání řízení mezi korutinami provádí procedura

```
TRANSFER(OdkazNaVolajícíKorutinu; OdkazNaVolanouKorutinu);
```

Spolupráci korutin vypisujících text „Ha“, „Lo“ uvádí příklad 3.1

Příklad 3.1

```
MODULE WriteHaLo;
      FROM SYSTEM IMPORT NEWPROCESS,TRANSFER;
      FROM IO IMPORT WrCard,WrLn,WrStr;
      CONST wkspSize = 1000;
            maxHaLo = 10;
      VAR wksp1, wksp2: ARRAY[1..wkspSize] OF BYTE;
            i:CARDINAL;
          main, cor1, cor2: FarADDRESS; (* deklarace procesů *)
      PROCEDURE WriteHa;           (* procedura procesu cor1 *)
      BEGIN
        LOOP
            WrStr("Ha");
            TRANSFER(cor1, cor2);    (* předání řízení *)
        END;
      END WriteHa;
      PROCEDURE WriteLo;          (* procedura procesu cor2 *)
      BEGIN
          LOOP
              WrStr("Lo");  INC(i);
                IF i > maxHaLo THEN
                     WrLn; i:=0;
                END;
                TRANSFER(cor2, cor1) (* předání řízení cor2 *)
          END;
      END WriteLo;
BEGIN
      i:=0;                              (* vytvoření procesů *)
      NEWPROCESS(WriteHa, FarADR(wksp1), SIZE(wksp1), cor1);
      NEWPROCESS(WriteLo, FarADR(wksp2), SIZE(wksp2), cor2);
      TRANSFER(main, cor1)           (* spuštění procesu cor1 *)
END WriteHaLo.
```

Konkrétní překladače Moduly bývají vybaveny ještě dalšími prostředky pro komunikaci procesů. Možný způsob jejich vytvoření právě prostřednictvím korutin lze nalézt ve [Wirth82]. O některých z nich ještě pojednáme v další kapitole.

3.2 Procesy v systému Unix

Základní možností při programové realizaci paralelních procesů je využítí libovolného programovacího jazyka, spolu se základními operacemi jádra OS (volaných formou procedur).

Tento přístup budeme demonstrovat na případu dvojice

jazyk C a operační systém **Unix**

Uvedeme zde jen základní informace týkající se procesů v OS **Unix** (verze BSD). Podrobnější popis lze nalézt v lit. [Brod89].

Identifikace procesů

Proces je jednoznačně identifikován celým číslem *pid* (process identification description)

3.2.1 Procesní operace jádra

Jádro při inicializaci systému zřídí proces číslo 0, který má dále funkci dispečera pro ostatní procesy. Dispečer vytvoří proces číslo 1 pro ovládání terminálů. Proces č.1 vytvoří podle tabulky připojených terminálů (v souboru /etc/ttys) pro každý připojený terminál základní proces (s čísly 2 až n+1). Základní proces terminálu je po většinu doby své existence řízen programem SHELL (v překladu „skořápka“), což je interpret příkazového jazyka. Příkaz může vést i k založení nového procesu (procesů). Uvedené údaje se mohou lišit (číslování základních procesů, jména systémových souborů) pro různé verze **Unix**u.

pid procesu lze zjistit voláním jádra OS, funkcí:

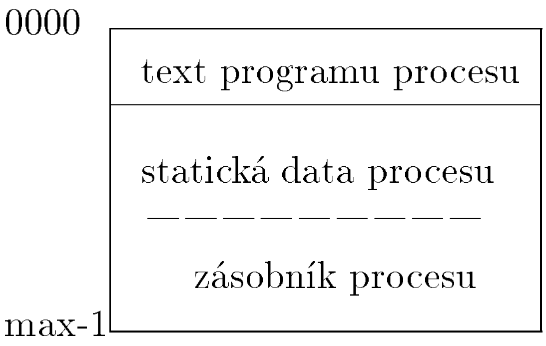
```
int getpid(void);
```

pid rodičovského procesu (ppid=parent *pid*) lze zjistit voláním jádra:

```
int getppid(void);
```

Adresový prostor procesu

Základní způsob strukturování paměti příslušící procesu ilustruje obr.3.3. Obsahuje oblast programu a datové oblasti: statickou a dynamickou.



Obr. 3.3: Struktura adresového prostoru procesu

Vznik a zánik procesů

Proces (potomek) vznikne voláním jádra funkcí s prototypem

```
int fork (void);
```

Tato funkce je volána v rodičovském procesu. Vytvořený proces je úplnou kopií svého rodiče. To znamená, že:

- jeho adresový prostor je kopií adresového prostoru rodiče,
- potomek se provádí podle stejného programu jako jeho rodič, má stejnou historii (data a zásobník), včetně místa kam výpočet dospěl (tj. za volání funkce `fork()`),
- má stejná práva jako jeho rodič.

Aby bylo možné rozlišit chování obou procesů v programu, který je společný pro rodiče i potomka, vrací `fork()` hodnotu *pid* :

- $pid = 0$ do procesu potomka,

- *pid* potomka do procesu rodiče.

Potomek končí exekuci voláním jádra:

`void exit (int status);`

kde `status` je číslo od 1 do 255.

Podle hodnoty `status` rozliší rodič způsob ukončení procesu potomka. Rodičovský proces se synchronizuje na ukončení procesu potomka voláním jádra

`int wait (int*p_status);`

kde `p_status` je ukazatel na místo, kam má funkce `wait` vrátit stav ukončení procesu potomka.

Funkce `wait()` vrací *pid* ukončeného potomka. Je třeba si však uvědomit, že funkce `wait()` čeká na ukončení procesu jednoho potomka. Jestliže rodičovský proces založil více potomků a chce vyčkat ukončení všech, musí volat funkci `wait()` tolikrát, kolik je založených potomků (např. v cyklu). Vrácený status je určen podle následující tabulky a jeho obsahem je:

vyšší byte ... koncový stav,
nižší byte ... kód příčiny.

způsob ukončení potomka	koncový stav	kód příčiny
exit(k)	k	0
„signálem“	0	kód signálu

Příklad 3.2 Vytvoření a zánik potomka

```
main()
{
  int   pid, who, status;

  switch(pid = fork()) {
    case 0:
      printf("Ja jsem potomek s cislem %d, zalozeny otcem
                     %d\n", getpid(), getppid());
      exit(1);
    case -1:
```

```
        printf("chyba pri vytvareni potomka \n");
        exit(2);
      default:
        printf("Ja jsem rodic %d potomka %d\n", getpid(), pid);
        who = wait(&status);
        printf("Ukoncen potomek %d se stavem %d\n", who, status);
    }
  }
```

Změna programu procesu

Proces může prostřednictvím volání jádra OS zaměnit program podle kterého je prováděn. Umožňuje to funkce:

```
int execve(char *path, char *argv[], char *envp[]);
```

kde

- `path` je cesta vedoucí k souboru s daným programem,
- `argv[]` je pole argumentů (řetězců) programu,
- `envp[]` je pole ukazatelů na řetězce prostředí (prostředím je míněna množina proměnných shellu, např. `PS = $` .... prompt).

Stejného účinku lze docílit také voláním dalších funkcí z rodiny *exec()*.

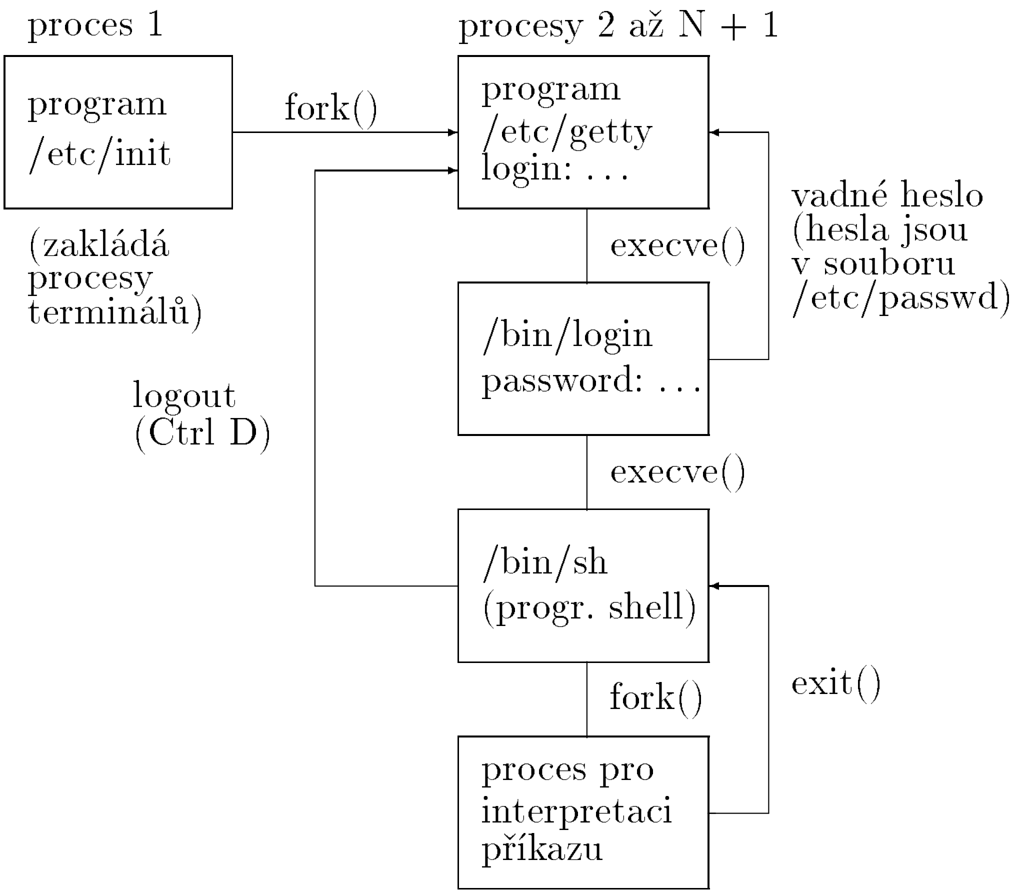
Základní proces terminálu

Základní proces terminálu v průběhu své existence probíhá podle různých programů, uchovává si ale svůj *pid*. Postupně je řízen programy

`shell, getty a login.`

Programy getty a login zajišťují přihlášení uživatele do systému, program shell zajišťuje interpretaci příkazů. Jeho činnost zhruba vyjadřuje cyklus:

```
while (1) {
  čti_příkaz(parametry);
  if (fork() != 0) {
    wait(status);
    /* analýza stavu */
  }
  else execve(parametry);
}
```



Obr. 3.4: Vytvoření a činnost základního procesu terminálu

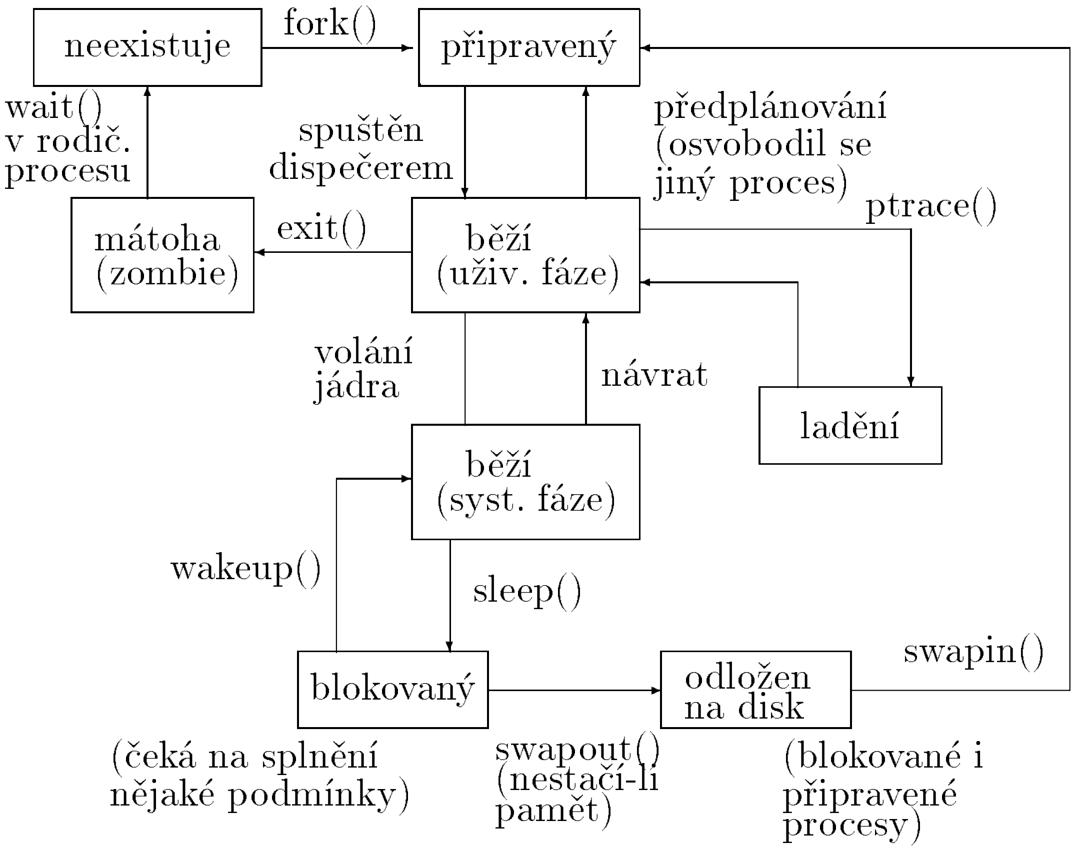
Stavy procesu

Základní stavy, kterými proces prochází, byly obecně uvedeny v kap.2. Grafické znázornění stavů a přechodů v případě procesů v BSD-**Unix**u ilustruje obr.3.5

Poznámky k obr.3.5:

- Funkce `sleep()`, `wakeup()`, `swapin()` a `swapout()` jsou interní funkce jádra a nemohou být volány uživatelem.

- Je-li zapotřebí paměťový prostor, získá jej OS odložením vybraného procesu na disk (pouze datové části jeho adresového prostoru). Výběr se provádí z množiny blokovaných procesů a je závislý na době, po kterou byl proces blokován.



Obr. 3.5: Přechodový diagram stavů procesu v OS BSD-**Unix**

Plánování a priorita procesů

Jádro přidělí procesům tzv. dispečerskou prioritu. Priorita je určena celým číslem. Nižší hodnota čísla znamená vyšší prioritu. Je dána výrazem:

proc.pri = 50 + proc.nice - 20 + proc.cpu/16

PUSER

implicitně 20
proces ji může změnit
voláním jádra nice(n),
čímž zvýší nice o n

zvyšuje se načítáním
časových přerušení
(např. po 20 ms)

- Procesy v uživ. fázi mají zřejmě priority větší než 50.
- Procesy v syst. fázi si mohou voláním `sleep(` *parametr* `)` prioritu zvětšit, tj. nastavit v intervalu <-32678, 49>.
- Plánovacím algoritmem je time-sharing, ve kterém se procesy se stejnou prioritou cyklicky střídají s časovým kvantem stanoveným při konfiguraci.

Zpracování signálů

Signály umožňují reakce procesu na asynchronní události. Událostí je zavedeno celkem 15 a jsou označovány symbolickými jmény. Uveďme alespoň jejich hlavní typy, zahrnující signály:

- způsobené chybou ve výpočtu ... `SIGFPE`,
- od časovače ... `SIGALARM`,
- způsobené klávesnicí ... `SIGINT`,
- ladicí přerušení ... `SIGTRAP`,
- přerušení z jiného procesu ... `SIGKILL,SIGUSR1,SIGUSR2`

Poslední dva signály jsou využitelné uživatelem.

Proces využívá signálů prostřednictvím:

- Volání jádra s prototypem
  `int signal(int sig, int (*func)())`

umožňuje procesu specifikovat uživatelsky definovanou obsluhu signálu `sig` funkcí `func`;
Dosazením nuly (patřičně přetypované) na místo parametru `func` se signál přenechává k obsloužení jádru. Dosazením jedničky se ignoruje.
Pro tento účel můžeme definovat
```
# define SIG_DFL (int(*)()) 0
# define SIG_IGN (int(*)()) 1
```

- Volání jádra s prototypem
  `int pause()`

  Proces pak čeká (ve stavu blokovaný) na libovolný signál.
- Volání jádra s prototypem
  `int kill(int pid, int sig)`

  Způsobí se tím zaslání signálu `sig` procesu `pid`.
- Volání jádra s prototypem
  `unsigned alarm(unsigned sec)`

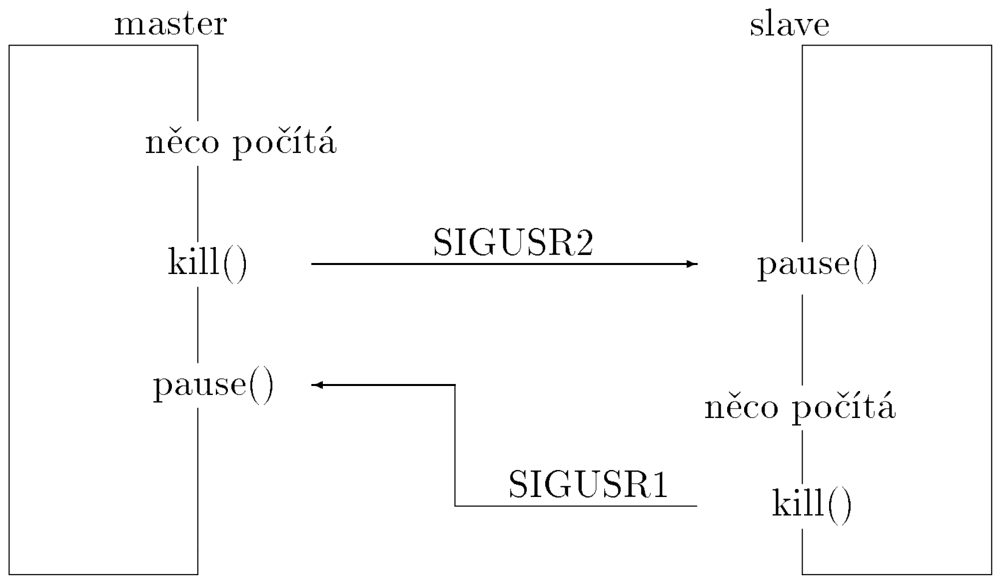
  Příkaz umožňuje procesu nastavit budík a probíhat dál, nezastaví-li se použitím `pause()`.

Příklad 3.3. Využití signálů ke komunikaci procesů

Uvažujme dva procesy (`master` a `slave`), střídající se ve výpočtu. Nejdříve počítá `master`, pak `slave`. Navzájem se synchronizují zasíláním signálů, viz obr.3.6.

Programové vyjádření komunikace naznačuje následující text:

```
#include <signal.h>
int mpid, spid;

obsluha_1(){
  signal(SIGUSR1, obsluha_1);
}
```



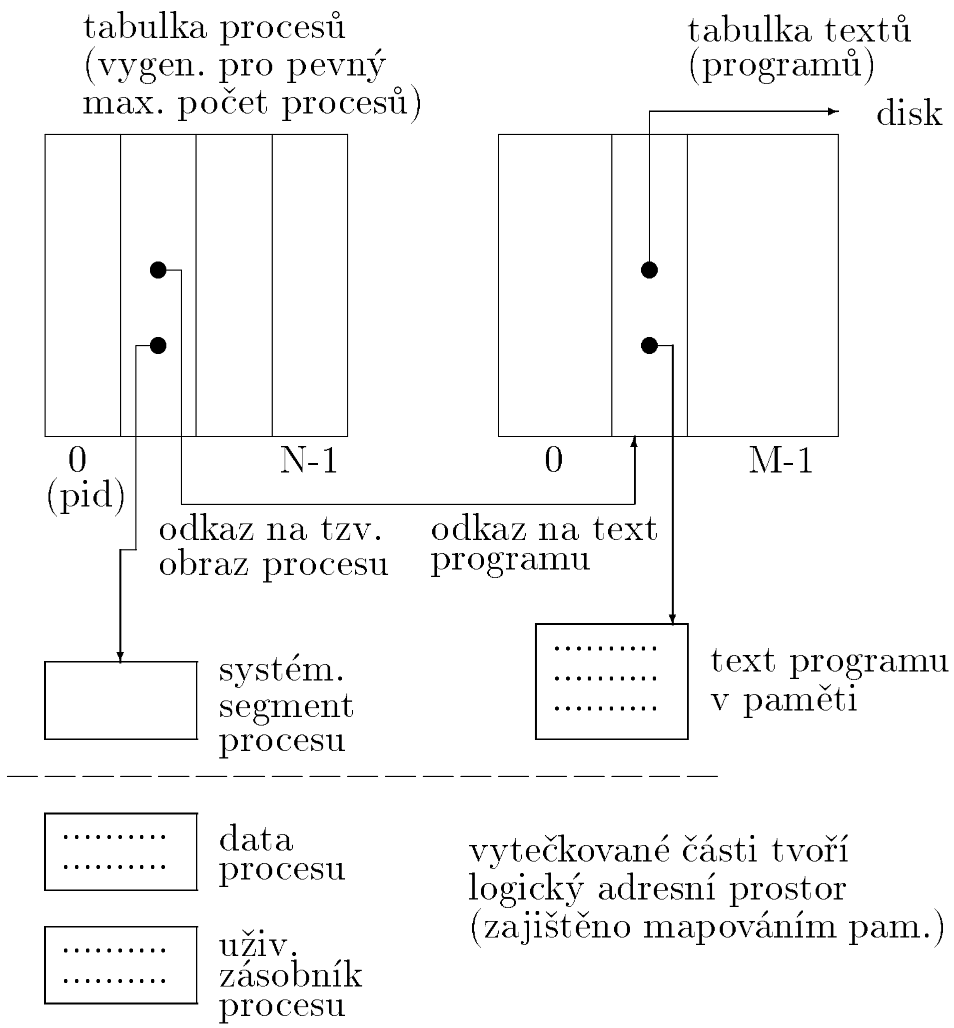
Obr. 3.6: Procesy komunikující pomocí signálů

```
obsluha_2(){
  signal(SIGUSR2, obsluha_2);
}

void main(){
  int i;

  signal(SIGUSR1, obsluha_1);
  signal(SIGUSR2, obsluha_2);
  mpid = getpid();
  spid = fork();
  if (spid == 0)      /* slave */
   for(;;) {
    pause();
    /* neco pocita */
    kill(mpid, SIGUSR1);
   }
  else {              /* master */
   for(i = 0;i < 100;i++) {
    /* neco pocita */
    kill(spid, SIGUSR2);
```

```
    pause();
   }
   kill(spid, SIGKILL);
  }
 }
```

Datové struktury jádra

Zmíníme se pouze o těch datových strukturách, které mají přímou souvislost s procesy. Jsou uvedeny na obr.3.7.



Obr. 3.7: Datové struktury jádra systému **Unix**-BSD

Poznámky k obr.3.7:

- Záznam procesu je stále v paměti, obsahuje údaje: priorita, pid, ppid, stav, adresu a rozměr obrazu v paměti (viz lit. [Brod89]).
- Systémový segment je rozšířením záznamu procesu, využívá jen jádro. Rozměrnější údaje jsou odkládány na disk (swapout) spolu s uživatelskými daty procesu.
- Texty programů jsou stále v paměti, neodkládají se na disk s obrazem procesu. Důsledkem je, že podle jednoho reentrantního programu (např. shell) může probíhat větší počet procesů.
- Záznam programu obsahuje také adresu umístění programu v paměti a na disku a počet procesů, které podle programu aktuálně probíhají. Pokud klesne jejich počet na nulu, může se místo pro program uvolnit.

Závěr:

Prostředky pro paralelní programování poskytované **Unix**em jsou účelově zaměřené na psaní jednoduchých systémových programů. Zde uvedený přehled prostředků odpovídá **Unix**u BSD v nižších verzích a měl by poskytnout pouze základní orientaci v problematice procesů v OS **Unix**.

Vyšší verze (**Unix** 4.2 BSD) obsahuje prostředky pro komunikaci procesů v síti (TCP/IP) založené na konstruktu označovaném socket.

Unix Systém V obsahuje knihovnu IPC, která poskytuje prostředky pro práci se semafory, se sdílenou pamětí a s frontami zpráv. Podrobnější informace lze nalézt v [Šmrha94], [Bach86].

3.2.2 Prostředky IPC

Prostředky meziprocesové komunikace (Interprocess Communication) umožňují procesům vyměňovat si data a synchronizovat svoji činnost. Původním prostředkem pro komunikaci procesů v **Unix**u jsou roury (pipes). Dovolují procesům navzájem komunikovat prostřednictvím proudu znaků. Existují dva typy rour:

- nepojmenované roury
- pojmenované roury

Nepojmenované roury je možné používat pouze pro vzájemnou komunikaci mezi procesem, který rouru vytvořil a jeho potomky. Pojmenované roury dovolují komunikaci i mezi procesy, mezi kterými není příbuzenský vztah.

K vytvoření nepojmenované roury slouží příkaz: `pipe(fdptr);` kde `fdptr` je ukazatel na celočíselné pole, ve kterém budou obsaženy dva popisovače souboru, jeden pro čtení a druhý pro zápis. Jejich vytvoření zajistí systém, procesy ani nerozeznají, zda zapisují, čtou soubor nebo rouru. K vytvoření pojmenované roury je použit příkaz `open`, obdobně jako v případě otevření normálního souboru. Taková roura má své jméno v systému souborů.

Pro práci s rourami jsou používány stejné příkazy jako pro manipulaci se soubory – `read, write, close`. Procesy zapisují na jednu stranu roury a čtou z její druhé strany. V případě, že proces chce číst z prázdné roury, či zapisovat do roury plné, je uspán. K jeho probuzení dojde až tehdy, když obsah roury dovolí mu v činnosti pokračovat.

Další meziprocesové komunikační prosředky se v jednotlivých typech unixovských systémů liší. **Unix** Systém V poskytuje 3 možnosti: zprávy, semafory a sdílenou paměť. Před jejich použitím je třeba alokovat frontu zpráv, pole semaforů či oblast sdílení paměti a zrušit je po jejich použití. Zprávy umožňují zasílat libovolným procesům formátované posloupnosti dat. Pro zprávy existují 4 systémové služby:

`msgget` – vytváří a vrací popisovač určující frontu, ve které bude zpráva uložena,

`msgctl` – dovoluje zjistit stav popisovače zprávy, změnit jeho stav, odstranit popisovač,

`msgsnd` – zaslání zprávy,

`msgrcv` – příjem zprávy.

Procesy mohou komunikovat přímo sdílením části svých virtuálních adresních prostorů. Systémová volání pro manipulaci se sdílenou pamětí jsou podobná voláním pro práci se zprávami:

`shmget` – vytváří novou oblast sdílení paměti, nebo (rozlišeno parametrem) vrací existující, `shmat` – přiřazení oblasti sdílené paměti k virtuálnímu adresovému prostoru procesu,

`shmdt` – odpojení oblasti sdílené paměti od virtuálního adresového prostoru procesu,

`shmctl` – zjišťování stavu a nastavení parametrů oblasti sdílené paměti.

K synchronizaci běhu mohou procesy použít semaforů. Semafor v **Unix** Systém V sestává z těchto informací:

- hodnota semaforu,
- *pid* procesu, který naposledy pracoval se semaforem,
- počet procesů čekajících na zvýšení hodnoty semaforu,
- počet procesů čekajících na hodnotu semaforu 0.

Příkazy, kterými lze ovládat semafory jsou:

`semget` – vytvoření a zpřístupnění pole semaforů,

`semctl` – řídicí operace nad semafory,

`semop` – manipulace se semafory (P a V operace).

Systémy BSD poskytují obecnější mechanismus meziprocesorové komunikace tzv. schránky (*sockets*). Je použitelný pro komunikace mezi lokálními procesy, ale i mezi procesy prováděnými v různých uzlech počítačové sítě. Jelikož sítě používají různé konvence pro komunikaci mezi uzly, jsou schránky sdílející komunikační vlastnosti sdruženy do tzv. komunikačních domén. Systém obsahuje protokol pro každou z kombinací komunikačních domén. Procesy komunikují způsobem client-server.

Mechanismus schránek je využitelný prostřednictvím několika příkazů volání systému. Schránka je vytvořena příkazem:

```
popisovač_schránky = socket (komunikační_doména,
            typ_komunikace, protokol);
```

Přiřazení schránky adrese provede příkaz

```
bind (popisovač_schránky, adresa, délka_adresy);
```

Přiřazení provádí proces server a inzeruje schránku ke komunikaci klientům.

Vytvoření spojení s existující schránkou provádí příkaz:

```
connect(popisovač_schránky, adresa, délka_adresy);
```

V případě, že proces server akceptuje spojení prostřednictvím virtuálního kruhu, musí jádro vkládat příchozí požadavky do fronty. Délka fronty je specifikována prostřednictvím příkazu:

```
listen (popisovač_schránky, délka_fronty);
```

Přijmutí požadavku na spojení zajišťuje příkaz:

```
nový_popisovač_schránky = accept (popisovač_schránky,
                          adresa, délka_pole);
```

kde:

- adresa ukazuje na datové pole uživatele, které systém doplní o návratovou adresu připojujícího se klienta,
- `délka_adresy` určuje délku uživatelova pole,
- hodnotou `accept` je nový popisovač schránky, který je pak použit pro vlastní komunikaci.

Pro přenos dat prostřednictvím schránky jsou pak použitelné kromě příkazů `read` a `write` i příkazy `recv` a `send`.

3.3 Programovací jazyk ADA

ADA může být provozována jak na jednoprocesorovém, tak i na na víceprocesorovém počítači. Zda výpočet bude probíhat pseudoparalelně či paralelně, závisí na systému a programátor má pouze k dispozici prostředky, jejichž pomocí vyjadřuje potenciálně možný paralelní běh programu.

3.3.1 Základní rysy

Jazyk je určen k programování rozsáhlých úloh, zejména z oblasti vestavěných RT systémů (pracujících v reálném čase). Je preferovaným jazykem DOD (ministerstvo obrany) USA. Základními hledisky, která **ADA** respektuje jsou:

- Bezpečnost vytvářených programů (zajišťovaná zejména silným typovým systémem),
- možnosti pro týmovou práci programátorů (modulární struktura programu, dovolující separátní kompilaci modulů),
- přenositelnost programů (překladače jsou testovány na soulad se standardem jazyka, tzv. validovány).

Další významné schopnosti zahrnují:

- Vyjadřování paralelně proveditelných akcí,
- možnost programového ošetření výjimečných situací (chybových stavů),
- prostředky pro vytváření datových abstrakcí (tzv. generické programové jednotky – šablony pro vygenerování programů).

Nevýhodou ADY je zejména její rozsáhlost. V roce 1995 došlo k rozšíření jazyka o prostředky OOP (třídy typů a dědičnost) na podobném principu jaký používají i ostatní imperativní jazyky. Poněvadž nemáme k dispozici kompletní podklady ani překladač jazyka ADA95, poznamenejme pouze, že pomocí slova "tagged" je možné označit takový typ, jehož vlastnosti mohou dědit z něj odvozované typy.

Způsob zápisu programu zjevně odráží vliv **Pascal**u. Identifikátory jsou tvořeny posloupností písmen, číslic a znaku podtržení, velikost písmen se nerozlišuje. Klíčová slova jsou vyhražena. V programech je zvykem je psát malými a identifikátory velkými písmeny. Předdefinované identifikátory lze předeklarovat.

K zápisu čísla je možné použít i jiné než desítkové soustavy uzávorkováním čísla znaky #, s předchozím uvedením základu. Pro zlepšení čitelnosti mohou také čísla obsahovat znak podtržení. Např. 2#1001_0110#

Řetězce jsou uzavírány mezi uvozovky, znakové konstanty mezi apostrofy. Poznámky začínají dvěma pomlčkami a pokračují až do konce řádku.

I/O operace nejsou součástí jazyka, ale jsou exportovány z knihovních modulů, případně tyto moduly exportují prostředky pro jejich vygenerování ze šablon (důsledek typového systému).

Program sestává z jedné, příp. z několika tzv. kompilačních jednotek. Kompilační jednotkou bývá modul (nazývaný package) nebo podprogram. Kompilační jednotku předchází tzv. kontextová specifikace. Ta uvádí jména jednotek, jejichž služby kompilační jednotka používá. Hlavní program má podobu bezparametrové procedury. Jeho implicitním vnějším prostředím je modul `STANDARD`, zavádějící předdefinované typy a jejich operace. V kontextové části jednotek se `STANDARD` necituje. Příkladem triviálního programu je následující text, kterým je vypsán obrázek totemu se zadaným počtem tváří.

Příklad 3.4.

```
with TEXT_IO; -- specifikace prostředí (modulu pro I/O textů)
use  TEXT_IO; -- zviditelnění prostředků modulu TEXT_IO
procedure OBRAZEK is
               --deklarační část
   TOTEM:INTEGER:=1;
   package INT_IO is new INTEGER_IO(INTEGER);
                                             --generování modulu
   --pro I/O operace nad typem INTEGER ze šablony INTEGER_IO
   use INT_IO; --zviditelnění prostředků z modulu INT_IO
begin
               --příkazová část
   PUT_LINE("kolik tváří má totem?");
   GET(TOTEM);
   NEW_LINE;
   for F in 1..TOTEM loop
      PUT_LINE("(((||)))");
      PUT_LINE("(*)++(*)");
      PUT_LINE("   ..   ");
      PUT_LINE(" ------");
   end loop;
end OBRAZEK;
```

Program `OBRAZEK` je tvořen jedinou kompilační jednotkou. Používá typ `INTEGER`, který je deklarován v modulu `STANDARD`.

Klauzule `with` zpřístupňuje knihovní modul `TEXT_IO` s prostředky pro práci s textovými soubory. Klauzule `use` dovoluje v dalším textu používat jmen z uvedeného modulu bez kvalifikace jménem modulu, tj. psát např. `PUT_LINE` namísto `TEXT_IO.PUT_LINE`.

`INTEGER_IO` je generický modul definovaný v modulu `TEXT_IO`, obsahující prostředky pro I/O operace s celými čísly. Uživatel má možnost vytvořit si svůj vlastní celočíselný typ požadovaného rozsahu a vygenerovat si pro něj množinu periferních operací.

Každý příkaz končí středníkem, a proto nelze vynechat středník před **end** (jak je tomu v **Pascal**u).

3.3.2 Klasické výrazové prostředky

V této části textu stručně probereme ty jazykové konstrukce, které **ADA** s určitými úpravami převzala od svých předchůdců. Protože tyto konstrukce nepodporují výrazně vytváření rozsáhlých programových celků, bývají označovány souhrnně jako prostředky pro programování v malém (angl. *programming in small*).

ADA používá silný typový systém s tzv. jmennou ekvivalencí. Rozumí se tím, že každá definice typu zavádí typ nový, odlišný od ostatních i v tom případě, kdy mají stejnou strukturu. Nepřipouští se mixovat ve výrazech objekty různých typů bez explicitně vyjádřené smysluplné konverze. Deklarace typu sdružuje jméno typu spolu s jeho definicí.

Proměnným je dovoleno přiřadit při deklaraci inicializační hodnotu, která nemusí být statická, musí být však vyhodnotitelná v okamžiku zpracování deklarace při výpočtu.

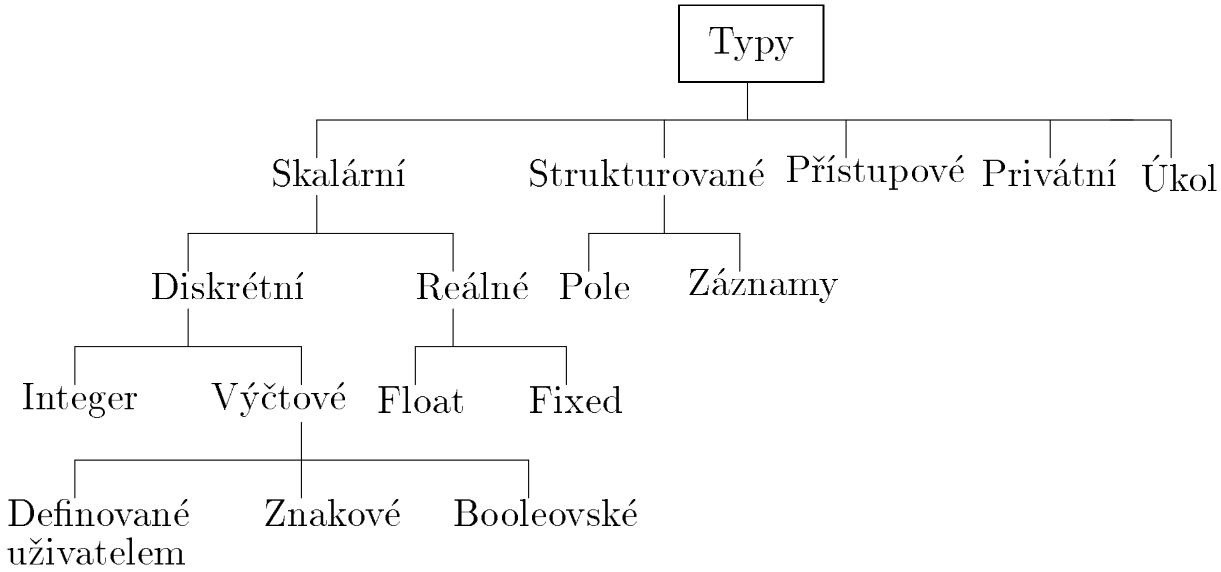
Číselné konstanty mohou být zaváděny bez specifikování jejich typu. Např. bude-li deklarován `type SMER is (NAHORU, DOLU);` pak:

```
PROM_SMER: SMER := NAHORU; -- deklaruje inicializovanou proměnnou
KONST_SMER:constant SMER := NAHORU; -- definuje konstantu typu SMER
PI: constant:=3.141592; -- zavádí konstantu univerzálního typu REAL
DESET: constant := 10; -- zavádí konstantu univ. typu INTEGER
```

Univerzální typ je kompatibilní s libovolným typem odvozeným ze stejné číselné báze.

Údaje o vlastnostech typů v konkrétní implementaci lze získat pomocí atributů. Pomocí atributů je možné ověřovat přenositelnost programů mezi počítači s různou délkou slova (zjištěním mezních hodnot daného typu), určit sousední hodnotu diskrétního typu, rozsah indexu pole apod.

Typový systém jazyka **ADA** je značně rozsáhlý. Klasifikaci typů uvádí obr.3.8.



Obr. 3.8: Klasifikace typů v jazyce ADA

Skalární datové typy

Typ INTEGER

Typ `INTEGER` je předdefinován v modulu `STANDARD`. Spolu s ním může implementace nabízet i `SHORT_INTEGER` či `LONG_INTEGER`.

Z předdefinovaných celočíselných typů lze odvozovat další celočiselné typy, obvykle s rozsahovým omezením hodnot:

```
type INT  is new INTEGER;
type DNY  is range 1..366;
type ROKY is range 0..2000;
```

Tím, že uživatel neuvede, ze kterého předdefinovaného typu odvozuje svůj typ, dává překladači možnost vybrat nejvhodnější, který zaručí požadovanou přesnost.
Deklarujeme-li pak proměnné:

```
CISLO_DNE: DNY;
CISLO_ROKU:ROKY;
```

můžeme je použít v příkazech:

```
CISLO_DNE:=CISLO_DNE+1;
CISLO_ROKU:=ROKY(CISLO_DNE);-- explicitní konverze
```

příkaz `CISLO_ROKU:=CISLO_DNE;` je však chybný a nepřeloží se důsledkem silné typové kontroly.

Z atributů použitelných pro celočíselné typy jmenujme alespoň dva:

`JMENO_TYPU'FIRST`	udává nejmenší hodnotu udaného typu,
`JMENO_TYPU'LAST`	udává největší hodnotu udaného typu.

Oba atributy jsou použitelné pro libovolný diskrétní typ.

Výčtový typ

Způsob zavedení je obdobný **Pascal**u s tím rozdílem, že **ADA** dovoluje přetěžování (vícevýznamové použití) literálů. Je tedy možné deklarovat:

```
type BARVA is (bila, zluta, cervena, modra, cerna);
type SEMAFOR is (zelena, oranzova, cervena);
```

Při použití literálu je však nutné buď z kontextu, nebo na základě tzv. kvalifikace jeho typ determinovat. Kvalifikace má tvar:

```
SEMAFOR'(cervena)
BARVA'(cervena)
```

Obecně lze kvalifikovat výraz zápisem `JMENO_TYPU'(výraz)`.

Z atributů, které jsou užitečné pro práci s výčtovým typem (použitelné však pro libovolný diskrétní typ) uveďme:

`PRED(X)`	předchůdce `X`,
`SUCC(X)`	následník `X`,
`POS(X)`	poziční číslo `X`,
`VAL(I)`	hodnota typu určená pozičním číslem `I`.

Např.:	`BARVA'VAL(0)`	má hodnotu bílá
	`BARVA'POS(BARVA'FIRST)`	má hodnotu 0

Typy `BOOLEAN` a `CHARACTER` jsou předdefinovanými výčtovými typy v modulu `STANDARD`.

```
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is (nul, ...,‘A‘,...,‘z‘,...,del);
```

Netisknutelné hodnoty typu `CHARACTER` jsou pojmenovány literály, takže tento typ obsahuje všech 128 ASCII znaků.

Typ FLOAT

Je předdefinovaným typem pro hodnoty v pohyblivé řádové čárce. Implementace může opět obsahovat jeho kratší (méně přesnou), či delší (přesnější) alternativu.

Uživatel má možnost definovat své vlastní typy v pohyblivé čárce. Přesnost uvede počtem míst mantisy (digits). Ten určuje dolní limit přesnosti. Implementací je vybrána nejbližší vyhovující přesnost. Přenositelnost programů je tím zaručena. Počet míst mantisy je zadáván statickým výrazem. Součástí definice může být i uvedení přípustného rozsahu hodnot (range). Např.:

```
type REALNE is digits 4; – neuvedený rozsah hodnot
type PRESNE is digits 10 range 0.0 .. 5.0E3;
```

Předpokládejme existenci typu `F` v pohyblivé čárce. Pak jeho charakteristiky můžeme zjišťovat pomocí atributů:

`F‘DIGITS`	–	počet desítkových číslic mantisy typu `F`,
`F‘MANTISA`	–	počet bitů mantisy typu `F`,
`F‘EPSILON`	–	jemnost zobrazení typu `F`,
`F‘EMAX`	–	nejvyšší hodnota exponentu,
`F‘SMALL`	–	nejmenší zobrazitelné kladné číslo typu `F`,
`F‘LARGE`	–	největší zobrazitelné kladné číslo typu `F`.

Typ Fixed

Není předdefinovaným typem. Uživateli je umožněno definovat si reálný typ v pevné řádové čárce uvedením jeho absolutní přesnosti a rozsahu hodnot. Např.:

```
type VYPLATA is delta 0.01 range 0.00 .. 100_000.00;
```

Hodnoty typu `VYPLATA` budou zobrazovány s přesností jedné setiny. Přesnost typu v pevné řádové čárce umožňuje zjistit atribut `DELTA`.

Komplikace, které způsobuje zvětšení počtu cifer výsledku aritmetických operací nad typy v pevné řádové čárce, je nutno řešit explicitně vyjádřenou konverzí typu. Např. při deklaraci:

```
type P is delta 0.1 range 0.0 .. 1000.0;
A,B,C: P range 0.0 .. 100.0 := 1.5;
```

je nutné konvertovat hodnotu vzniklou násobením `A` s `B` na původní přesnost, chceme-li ji přiřadit `C`.

```
C := P(A*B);
```

Záměr konverze je obecně (pro všechny druhy typů) vyjádřen formou:

```
JMENO_TYPU (KONVERTOVANY_VYRAZ),
```

přičemž převod reálného typu na celočíselný se provádí zaokrouhlením.

Operátory skalárních typů

Předdefinované operátory jsou sumarizovány v tab. 3.1, 3.2 a 3.3. Jejich vlastnosti jsou konvenční s několika výjimkami:

- Pro vyjádření zkráceného výpočtu jsou zavedeny operátory `and then` a `or else`. Takže např. ve výrazu

  ```
  A > B and then C >= 1
  ```

  bude druhá část podmínky vyhodnocována pouze v případě, že prvá část je pravdivá.

- Operátory `rem` a `mod` vytváří zbytek po celočíselném dělení. Jsou-li oba operandy těchto operátorů stejného znaménka, jejich výsledek se neliší. Platí, že výsledek rem má znaménko levého operandu a výsledek mod má znaménko pravého operandu. Např.:

  ```
  2  rem -5 = 2,  2 mod -5 = -3, -1 rem 5 = -1, -1 mod 5 = 4
  ```

- Operátor dělení s celočíselnými operandy dává celočíselný výsledek, takže např. $7/4 = 1$, neboť frakční část je zanedbávána.

- Není dovoleno používat záporný exponent s celým číslem, takže výraz $2**(-5)$ způsobí runtime error.

operátor	operand	výsledek
`+`	numerický	stejný
`-`	numerický	stejný
`not`	boolean	stejný
`abs`	numerický	stejný

Tab.3.1.Unární operátory

operátor	operandy		výsledek
and, or, xor **and then,** **or else**	boolean		stejný
`=, /=`	libovolné		boolean
`<, <=, >, >=`	skalární diskr. jednor. pole		boolean
in,	hodnota interval		boolean
not in	hodnota podtyp		boolean
`+ -`	numerické		stejný
&	řetězce či znaky		řetězec
*	integer	integer	stejný
	integer	fixed	stejný fixed
	float	float	stejný float
/	integer	integer	stejný
	float	float	stejný,
	real	integer	real
mod rem	integer	integer	stejný
**	integer	nezáp. integer	stejný integer,
	float	integer	stejný float

Tab.3.2. Binární operátory

`and`	`or`	`xor`	`and`	`then`	`or`	`else`		
`=`	`/=`	`<`	`<=`	`>`	`>=`	**in**	**not**	**in**
bin.	`+`	`-`						
un.	`+`	`-`						
`*`	`/`							
`**`	`abs`	`not`						

Tab.3.3. Priorita operátorů

Podtypy a odvozené typy

Podtyp reprezentuje podmnožinu nějakého již existujícího, tzv. bázového typu. Oproti oboru hodnot bázového typu bývá jeho obor hodnot zúžen. Užitečnost podtypu spočívá v možnosti mixovat ve výrazech dle něj deklarované proměnné s proměnnými bázového typu bez uvedení konverze. Deklarace podtypu nezavádí nový typ. Kromě omezení má podtyp stejné vlastnosti jako typ bázový. Např.:

```
subtype PRIROZENA_CISLA is INTEGER range 1 .. INTEGER'LAST;
subtype PRACOVNI_DNY is DNY_V_TYDNU range PONDELI .. PATEK;
```

Vlastnosti bázového typu lze zjišťovat prostřednictvím atributu `BASE`. Např. `PRACOVNI_DNY'BASE'LAST` bude mít hodnotu posledního dne v týdnu.

Odvozený typ je podobně jako podtyp získán z již existujícího typu. Zavádí se jím však nový typ, s vlastnostmi zděděnými od bázového typu. Proto ve výrazech nemohou být mixovány objekty bázového typu s objekty odvozeného typu bez explicitně vyjádřené konverze. Tvar jeho deklarace ilustrují příklady:

```
type NOVE_INTEGER is new INTEGER;
type NOVE_FLOAT is new FLOAT digits 5;
```

Druhý příklad má však nevýhodu, neboť nutí překladač odvodit typ z typu `FLOAT`, i když by mohl být např. použit úspornější typ `SHORT_FLOAT`. Z tohoto důvodu je výhodnější dříve zavedená konstrukce

```
type NOVE_FLOAT is digits 5;
```

která překladač neomezuje. I ta však zavádí odvozený typ, neboť dle definice je ekvivalentní deklaracím:

```
type ANONYMNI is new PREDDEFINOVANY_FLOAT;
subtype NOVE_FLOAT is PREDDEFINOVANY_FLOAT digits 5;
```

kde `ANONYMNI` je fiktivním typem a `PREDDEFINOVANY_FLOAT` je některý z implicitně deklarovaných float typů, vybraný překladačem. Z obdobných důvodů jsou zkracovány i konstrukce deklarací odvozených typů z typů celočíselných a reálných v pevné řádové čárce. Např.

```
type VEK is range 0 .. 200;
```

je odvozeným typem z vhodného celočíselného předdefinovaného typu.

Strukturované datové typy

Typ pole

Deklarace typu pole má obecně tvar:
type *JMÉNO_TYPU_POLE* **is array** *(SPECIFIKACE_INDEXŮ)* **of** *TYP_PRVKŮ;*

Je-li v deklaraci typu určen rozsah indexů, nazýváme typ pole **determinovaný** (angl. constrained), není-li určen, nazýváme takový typ pole **nedeterminovaný** (angl. unconstrained).

```
type VEKTOR_10 is array (INTEGER range 1..10) of FLOAT;
                                               -- nebo
type VEKTOR_10 is array (1..10) of FLOAT;--determinovaný typ
type DIÁŘ is array (PRACOVNI_DNY, 1..24) of CO_DELAT;-- " "


type VEKTOR is array (INTEGER range <>) of FLOAT;
                                     -- nedeterminovaný typ
type MATICE is array (INTEGER range <>,INTEGER range <>)
                      of FLOAT;      -- nedeterminovaný typ
```

Pro specifikaci indexů lze použít libovolný diskrétní typ. Nedeterminované typy pole používáme při konstrukci algoritmů zpracovávajících pole s různými rozsahy indexů. Určení rozsahu je odloženo až do okamžiku použití nedeterminovaného typu k deklaraci proměnné, konstanty či podtypu. Např.:

```
subtype VEKTOR_100 is VEKTOR(1..100);
V1, V2: VEKTOR_100;
KONSTANTA: VEKTOR := (0.0, 0.1, 0.4, 10.0, 12.55);
M1: MATICE(1..5, 10..20);
```

Z příkladů je patrné, že v případě konstantních polí je jejich rozměr určen zadáním přiřazované hodnoty.

!! Použijeme-li anonymního typu k deklaraci proměnných, např.

```
P, Q: array(1..2, 1..5);
```

budou `P` a `Q` odlišného typu, neboť uvedený zápis je dovoleným zkrácením dvou deklarací s anonymními typy.

Meze polí nemusí mít vždy statický charakter, mohou být dány výrazem vyhodnotitelným až v době výpočtu. Taková pole jsou nazývána *dynamická*. Dovolují ekonomicky využívat paměť v těch případech, kdy předem rozměry pole neznáme.

Vlastnosti polí je možné zjišťovat v programu pomocí atributů. Pro objekt `A` typu pole, či typ `A` s určenými mezemi udávají:

`A'FIRST`	spodní mez prvního indexu,
`A'LAST`	horní mez prvního indexu,
`A'LENGTH`	počet hodnot prvního indexu,
`A'RANGE`	interval `A'FIRST..A'LAST`

Pro zjištění charakteristik n-tého rozměru jsou použitelné zápisy

```
A'FIRST(n), A'LAST(n), A'LENGTH(n), A'RANGE(n).
```

Řetězce jsou považovány za nedeterminovaná jednorozměrná pole znaků. V modulu `STANDARD` je pro ně předdefinován typ `STRING`.

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

Operátor zřetězení `&` je aplikovatelný na libovolná jednorozměrná pole. Dolní mez výsledného pole je dána dolním indexem levého operandu.

Pole stejného rozsahu a typu prvků lze přiřazovat jako celek, což platí i pro řetězce. Řetězce s proměnnou délkou předdefinovány nejsou.

V případě jednorozměrných polí lze přiřazovat i jejich řezy. Řezem se rozumí souvislá část původního pole. Pro výše deklarované proměnné je např. přípustný zápis:

```
V1 := V2;
V1(10..25) := V2(41..56);
```

Polím je možné přiřazovat složenou hodnotu, tzv. **agregát**. Agregáty mohou být **poziční** nebo **jmenné**. V prvém případě jsou jednotlivé hodnoty uváděny postupně (viz výše uvedená `KONSTANTA`), ve druhém případě jsou hodnoty prvku sdruženy s hodnotou indexu.
Např.

```
M: MATICE(1..2,1..2) := ((1.1, 1.2), (2.1, 2.2)); -- poziční
M := (1=>(1=>1.1, 2=>1.2),
      2=>(1=>2.1, 2=>2.2));    -- jmenný agregát
```

Obě formy je povoleno mixovat, avšak ne v jednom rozměru. Další možnosti zápisu agregátů uvedeme formou příkladů.

```
M := (1..2=>(1..2=>0.0));
MUJ_DIAŘ:=(PONDELI|PATEK=>(1..8=>SPÁT,others=>CESTOVAT),
          others=>(1..6=>SPAT,7..22=>PRACOVAT,others=>SPAT));
```

Pokud je použita alternativa `others`, musí být uvedena v daném rozměru jako poslední. Při jejím použití musí být rozsah pole patrný z kontextu, což lze zajistit např. kvalifikací jménem typu

```
JMENO_TYPU‘ agregát
```

Na jednorozměrná booleovská pole jsou aplikovatelné operátory

not **or** **xor**

Jednorozměrná diskrétní pole lze porovnávat relačními operátory, což je nejčastěji použitelné pro řetězce. Platí např.

```
"AA" < "B"      "A " > "A"     " Z" < "A"
```

Budeme-li deklarovat typ pole tak, že typ jeho prvků je opět typem pole, vzniká pole polí. Např. matici lze vytvořit jako vektor vektorů. Na prvky takového pole pak budeme odkazovat zápisem

```
JMENO_POLE (INDEX1) (INDEX2) .
```

Tento zápis není zaměnitelný s `JMENO_POLE (INDEX1, INDEX2)` při odkazu na vícerozměrné pole.

Záznamy

Pro práci se záznamy jsou použity konvence známé z **Pascal**u či **C** jazyka. Deklarace záznamu bez variantní části má např. tvar:

```
type DATUM is
      record
            DEN: INTEGER range 1..31;
            MESIC: JMENO_MESICE;
            ROK: INTEGER range 1900..2000;
      end record;
type KOMPLEXNI is
      record
            RE: FLOAT := 0.0;
            IM: FLOAT := 0.0;
      end record;
```

Těchto typů pak lze využít k deklaraci proměnných:

```
VCERA, DNES, ZITRA: DATUM;
X, Y: KOMPLEXNI;
```

Proměnné `X` a `Y` budou inicializovány počáteční hodnotou, uvedenou v deklaraci typu. Inicializaci lze provést i při deklaraci proměnných prostřednictvím pozičního či jmenného agregátu.

```
Z: KOMPLEXNI := (1.0, 5.0);            -- poziční notace
NAROZEN: DATUM := (ROK=>1993, DEN=>25, MESIC=>LISTOPAD);
                                       --jmenná notace
```

Na komponenty záznamu se odkazuje pomocí tečkové notace. Např.

```
NAROZEN.ROK := 1989;
```

Předdefinovanými operacemi pro záznamy jsou přiřazení a testy na rovnost a nerovnost.

Takové typy záznamů, které mají pevně zadaný počet, typ a velikost komponent, jsou nazývány *typy záznamů bez diskriminantů*. Oproti tomu tzv. *typy záznamů s diskriminanty* dovolují vytvářet záznamy:

a) s komponentami pole různé délky,

b) s variantní částí.

ad a)

Je-li komponentou záznamu nedeterminovaný typ pole, lze pomocí parametru (diskriminantu) určit velikost pole. Např.

```
type VYROVNAVACI_PAMET (ROZSAH: CELOCISELNY_TYP := 100) is
      record
            PAMET: STRING(1..ROZSAH);
      end record;
```

Poněvadž při deklaraci typu `VYROVNAVACI_PAMET` jsme uvedli předběžnou hodnotu diskriminantu `100`, nemusíme při deklaraci proměnné tohoto typu hodnotu diskriminantu již uvádět. Pokud jej uvedeme, bude tato nová hodnota akceptována.

```
VPAMET1: VYROVNAVACI_PAMET; --rezervované místo pro 100 znaků
VPAMET2: VYROVNAVACI_PAMET(50);--  "      "     "    50   "
VPAMET3: VYROVNAVACI_PAMET(ROZSAH=>10);   "     "    10   "
```

Pokud by v deklaraci typu nebyla předběžná hodnota uvedena, je uvedení hodnoty diskriminantu povinné při deklaraci záznamu. Diskriminant je ve skutečnosti jednou z položek záznamu. Nelze mu však samostatně přiřazovat hodnotu. Těm záznamům, které jsou deklarovány bez uvedení hodnoty diskriminantu, je možné v rámci přiřazení celému záznamu přiřadit i hodnotu diskriminantu. Např.

```
     VPAMET1 := (400,"ahoj");
```

Takové záznamy jsou nazývány nedeterminované.

!! Opatrně používat, řada překladačů si zjednodušuje práci tím, že rezervuje paměť pro největší možný záznam, který může vzniknout (v našem příkladu s polem o rozsahu `1..CELOCISELNY_TYP'LAST`).

Záznamy, které jsou deklarovány s uvedením hodnoty diskriminantu se označují **determinované záznamy**.

Pro zjištění, zda záznam je či není determinovaný je k dispozici atribut `CONSTRAINED`, nabývající pro

VPAMET1'CONSTRAINED hodnotu FALSE a pro
VPAMET2'CONSTRAINED hodnotu TRUE.

Diskriminantů, kterými je záznam parametrizován může být libovolný počet (např. je-li komponentou záznamu vícerozměrné pole, lze jimi parametrizovat libovolný počet dimenzí).

ad b)

Variantní záznamy dovolují sdružit několik alternativních záznamů do jednoho celku. Mohou být rovněž determinované či nedeterminované, na základě určení či neurčení hodnoty diskriminantu při deklaraci záznamu. Variantní část musí být jediná, uvedená v záznamu jako poslední. Necitované alternativy variantní části je možné zahrnout do části označené **others**. Např.

```
type TYP_STAVU is (SVOBODNY, ZENATY);
type OSOBA (STAV: TYP_STAVU := SVOBODNY) is
      record
            NAROZEN: DATUM;
            VAHA, VYSKA: FLOAT;
            case  STAV is
                  when SVOBODNY => null; --prázdná položka
                  when ZENATY   => DATUM_SNATKU: DATUM;
                                   POCET_DETI: INTEGER;
            end case;
      end record;
```

Pak je možné deklarovat záznamy:

```
OSOBA1: OSOBA;        --nedeterminovaný záznam
OSOBA2: OSOBA(ZENATY);--determinovaný záznam
OSOBA3: OSOBA:=(SVOBODNY,(17,LISTOPAD,1989),1.55,38.5);
        --nedeterminovaný, inicializovaný pozičním agregátem
```

Stejně jako v případě a) je možné přiřazovat determinovaným záznamům jako celku pouze takové agregáty, které mají souhlasnou hodnotu diskriminantu a nedeterminovaným záznamům i agregáty, které hodnotu diskriminantu mění. Diskriminant může být použit i pro deklaraci podtypu. Např.

```
subtype MLADENEC is OSOBA(SVOBODNY);
```

Dle něj deklarované záznamy budou pak determinované. Důvodem výše uváděných konstrukcí je zajištění bezpečnosti konstrukcí se záznamy. Nepromyšlená změna samotného diskriminantu (pokud ji jazyky připouští – viz **Pascal**) způsobí nesmyslnost celého obsahu variantní části.

Typ ukazatel

Deklarace typu ukazatel má v jazyku **ADA** tvar:

```
type JMENO_TYPU_UKAZATEL is access JMENO_TYPU_NA_KTERY_UKAZUJE;
```

Např.
```
type I_UKAZ is access INTEGER;
UK1, UK2: I_UKAZ; -- deklarace ukazatelových proměnných
```

Pro vytvoření paměťového místa zpřístupněného ukazatelem je nutno použít alokátoru **new**.

Např.
```
UK1 := new INTEGER'(10); -- UK1 ukazuje na hodnotu 10
UK2 := new INTEGER; -- UK2 ukazuje na místo pro INTEGER
```

Existence paměťového místa trvá, pokud výpočet neopustí rozsahovou jednotku, ve které je deklarován ukazatel, který místo zpřístupňuje. Poté může čistič paměti provést jeho zrušení. Brání se tím opomenutí uvolnění paměti programátorem. Pokud však by chtěl sám řídit uvolňování paměti, může si vytvořit z knihovního podprogramu uvolňovací proceduru příkazem:

```
procedure FREE is new UNCHECKED_DEALLOCATION
      (JMENO_UVOLNOVANEHO_TYPU, JMENO_UKAZATELOVEHO_TYPU);
```

Provedením příkazu `FREE(UK);` způsobí pak dosazení prázdné hodnoty do `UK` a uvolnění paměti kam `UK` ukazoval.

Hodnota proměnné, na kterou ukazatel odkazuje, je dosažitelná kvalifikací **all**.

```
UK2.all := UK1.all+1;--přiřadí do místa kam ukazuje UK2 hodnotu 11
UK2 := UK1; --          UK2 ukazuje do stejného místa jako UK1
```

Nejčastějším použitím ukazatelů jsou odkazy na záznamy. Je-li komponentou záznamu opět ukazatel, mohou vznikat rekurzivní struktury. Jelikož je požadováno, aby všechny typy byly deklarovány před použitím, je takový typ uveden formou neúplné deklarace, a jeho úplná deklarace musí být provedena v téže deklarační části.

```
type UDAJ;  --neúplná deklarace
type SPOJ is access UDAJ;
type UDAJ is
      record
            HODNOTA: INTEGER;
            PREDCHUDCE,NASLEDNIK: SPOJ;
      end record;
```

Literálem **null** se označuje prázdná hodnota ukazatele. Budou-li např. deklarovány proměnné

```
ZACATEK, DALSI: SPOJ := null;
```

provede následující sekvence příkazů zařazení dalšího prvku na začátek seznamu:

```
ZACATEK := new UDAJ'(22, null, null);
DALSI   := new UDAJ'(55, null, ZACATEK);
ZACATEK.PREDCHUDCE := DALSI; --dereference neni expl.označena
ZACATEK := DALSI;
```

Z příkladu vyplývá, že **ADA** nepoužívá explicitně vyjádřenou dereferenci (viz **Pascal**), kromě případu odkazování na celý obsah zpřístupňované struktury pomocí kvalifikace `all`, jako např. v příkazu

```
DALSI.all := ZACATEK.all;
```

Základní příkazové struktury

Přiřazovací příkaz

Při zpracování přiřazovacího příkazu je kontrolován jak soulad typů levé a pravé strany, tak i splnění rozsahových omezení. V případech, které mají smysl, je možné zajistit kompatibilitu explicitně vyjádřenou konverzí hodnoty pravé strany. Existuje i možnost provedení konverze ve smyslu bitových posloupností, za pomoci speciální knihovní funkce. Její používání by však mělo být výjimečné.

Přiřazovat je možné i strukturám a jejich částem, jak bylo již dříve uvedeno.

Podmíněný příkaz

Obecný tvar podmíněného příkazu je:

```
if podmímka1 then
    posloupnost_příkazů1;
elsif podmínka2 then
    posloupnost_příkazů2;
elsif ...
...
else posloupnost_příkazůN;
end if;
```

Části elsif a else jsou nepovinné. Význam zápisu je zřejmý.

Příkaz case

Formou je podobný variantním záznamům. Obecně má tvar:

```
case výraz is
     when výběr1 => posloupnost_příkazů1;
     when výběr2 => posloupnost_příkazů2;
     ...
end case;
```

Výraz musí nabýt hodnotu diskrétního typu. Každý z výběrů představuje jednu nebo více statických hodnot tohoto typu. Např.

```
case DNES is
     when PONDELI => ODJEZD_NA_PRACOVISTE; PRACUJ;
     when UTERY..CTVRTEK => PRACUJ;
     when PATEK => PRACUJ; ODJEZD_DOMU;
     when SOBOTA | NEDELE => null;
end case;
```

Obsahuje-li výběr několik nesouvislých hodnot, jsou tyto odděleny znakem |. Prázdný příkaz je ve všech konstrukcích označen `null`. Neuvedené alternativy lze shrnout pod výběr `others`, který může být uveden pouze jako poslední. Poněvadž výběry musí pokrýt plně obor hodnot nabývaných výrazem, je konstrukce `when others` pro některé případy dokonce povinná.

Příkazy cyklu

Kromě nekonečného cyklu je zaveden i cykl **while** a cykl **for**. Jejich podobu ilustrují následující příklady.

```
loop        --nekonečný cykl
      VSTUP(DATA);   -- za loop následuje
      ZPRACUJ(DATA); --   posloupnost
      VYSTUP(DATA);  --   příkazů
end loop;

while DALSI /= null  loop
      SUM := DALSI.HODNOTA;
      DALSI := DALSI.NASLEDNIK;
end loop;

for I in A'RANGE  loop
      A(I) := B(I) + C(I);
      B(I) := 0; C(I) := 0;
end loop;
```

Poznámky k cyklům:

- Výstup z nekonečného cyklu je možný buď pomocí příkazu skoku, nebo příkazem `exit` . Ten rovněž tvrdě ukončí cykl, jeho účinek je však možné vázat na splnění podmínky, zapíše-li se ve tvaru:

  ```
  exit when {\em podmínka;}
  ```

  Příkazem `exit` lze ukončit provádění i ostatních druhů příkazu cyklu. Za normálních okolností ukončuje vždy nejvnitřnější cykl. Pokud však cykl označíme jménem, které pak uvedeme v příkazu `exit`, dojde k ukončení pojmenovaného cyklu.

  ```
  HLAVNI_CYKL: loop ...
                   for ... loop
                         ...
                         exit HLAVNI_CYKL when PODMINKA;
                         ...
                   end loop;   ...
               end loop HLAVNI_CYKL;
  ```

- Parametr cyklu **for** je implicitně zaveden překladačem, nedeklaruje se. Mimo cykl neexistuje. Jeho typ si překladač zjistí z kontextu. Parametru cyklu nelze programem změnit hodnotu. Pokud je rozsah provádění cyklu prázdný, považuje se provádění za ukončené. Rozsah, určující počet průchodů je vyhodnocován jen jednou, před začátkem provádění cyklu. Je určen libovolným diskrétním rozsahem, například intervalem -1 ... 99. Pro reverzní provádění cyklu je zavedena konstrukce:

  `for` *parametr* `in reverse` *diskrétní_rozsah* `loop` ...

Příkaz skoku

Příkaz skoku je zaveden v obvyklé formě

`goto` *jméno_návěští*;

Není přípustné provádět skok dovnitř jazykových konstrukcí, tedy ani do strukturovaných příkazů. Skok ven z konstrukcí povolen je, kromě speciálních případů (z programové jednotky, z kritické sekce, ze sekvence zpracování výjimečné situace). Označení cílového místa je prováděno zápisem

`<<TO_JE_NAVESTI>>` *příkaz;* ...

3.3.3 Bloky, procedury, funkce

Bloky, procedury i funkce představují rozsahové jednotky, tj. takové části programu, které sestávají z deklarační části a z příkazové části. Později zavedeme ještě jejich nepovinnou část pro zpracování výjimečných situací.

Deklarační část obsahuje deklarace:

- typů a podtypů,
- proměnných,
- konstant,
- procedur a funkcí,
- modulů (viz později),

- úkolů (viz později).

Příkazová část musí explicitně obsahovat alespoň jeden příkaz. Může jím být i prázdný příkaz **null.**

Bloky

Na rozdíl od procedur a funkcí, které jsou umístěny mezi deklaracemi, jsou bloky součástí příkazů. Není dovolen jejich separátní překlad. Slouží k zavedení lokálních objektů nebo k vyjádření zpracování výjimek. Blok má obecně tvar:

```
declare
   deklarace
 begin
   příkazy
 end;
```

Deklarační část může být včetně slova declare vynechána. Jeho použití by pak mělo smysl jen při uvedení části s reakcemi na výjimky.

Blok může být pojmenován (obdobně jako cykl). Motivem je umožnit zviditelnění globálních jmen, majících shodné jméno s lokálními jmény. Např.

```
VNEJSI: declare
             X: INTEGER;
         begin
             declare
                   X: TYP_T;
             begin
                   X:= ...      --vnitřní X
                   VNEJSI.X:=...--vnější X
             end;
         end VNEJSI;
```

Procedury a funkce

Spolu s moduly, úkoly a generickými jednotkami (budou probrány později) patří procedury a funkce mezi tzv. kompilační jednotky, které mohou být samostatně překládány. Základní vlastnosti procedur i funkcí se shodují s běžnými zvyklostmi. Řada vlastností je ale nekonvenční, v jiných jazycích nezavedená.

Základní tvar procedury a funkce vyplývá z následujících dvou příkladů.

```
procedure RAZENI(X: in out VEKTOR) is
      POM: FLOAT;
begin
      for I in X'FIRST+1..X'LAST loop
            for J in reverse I..X'LAST loop
                  if X(J-1) > X(J) then
                        POM:=X(J);
                        X(J):=X(J-1); X(J-1):=POM;
                  end if;
            end loop;
      end loop;
end RAZENI; --pro čitelnost se doporučuje zopakovat jméno
            --zároveň se tím zvyšuje bezpečnost

function SKALARNI_SOUCIN(A,B: VEKTOR) return FLOAT is
      SUMA: FLOAT := 0.0;
begin
      for I in A'RANGE loop
            SUMA := SUMA + A(I) * B(I);
      end loop;
      return SUMA;
end SKALARNI_SOUCIN;
```

Příklady předpokládají deklarovaný typ

```
type VEKTOR is array (INTEGER range <>) of FLOAT;
```

což umožní použít podprogramy pro pole s různými počty prvků.

Příkaz return

Příkaz `return` následovaný výrazem slouží k výstupu z funkce. Je však použitelný i k výstupu z procedury v podobě prostého zápisu `return`. Častější je ukončování procedury průchodem výpočtu koncovým `end`.

Parametry

Parametry mohou být vstupní, výstupní, vstupně-výstupní (`in`, `out`, `in out`). Není-li druh parametru explicitně uveden, předpokládá se `in`.

Všechny parametry funkcí musí být `in`. Brání se tím používání vedlejších efektů funkcí.

Skutečné parametry druhu `out` a `in out` musí být proměnné. Typy korespondujících si formálních a skutečných parametrů musí být stejné.

Při volání jsou hodnoty skutečných parametrů druhu `in` a `in out` kopírovány do formálních parametrů. Při výstupu jsou hodnoty formálních parametrů druhu `out` a `in out` kopírovány do skutečných parametrů.

Při volání je možné uvést parametry nejen v obvyklé poziční notaci, ale také ve jmenné notaci, či v jejich kombinaci (pak musí poziční předcházet jmenné). Např.

```
...SKALARNI_SOUCIN(B => SK_P_2, A => SK_P_1);
...SKALARNI_SOUCIN(SK_P_1, B => SK_P_2);
```

Parametr může mít libovolný typ. Bezparametrová funkce se zapisuje bez závorek.

Parametry `in` lze inicializovat předběžnou hodnotou. Skutečný parametr pak nemusí být uveden. Je-li uveden, předběžná hodnota se neuplatní. Takový postup nelze doporučit, snižuje přehlednost.

Separace

Pro dosažení větší volnosti v uspořádání programového textu je možné oddělit záhlaví podprogramu. To je nazýváno specifikační částí, poněvadž specifikuje všechny vnější znaky podprogramu. Oddělení specifikační části od těla podprogramu je používáno při separátním překladu podprogramu a k přehlednějšímu použití rekurzivních podprogramů. Pro případ funkce znázorňuje separaci příklad:

```
function F(X: TYPX) return TYPF; -- specifikační část funkce

function F(X: TYPX) return TYPF is --tělo funkce
      --lokální deklarace
begin
      --příkazy
end;
```

Separací specifikace od těla zabráníme hlubokému vnořování rekurzivních podprogramů obdobně jako příkazem `FORWARD` v **Pascal**u.

Dalším přínosem separace je možnost použití tzv. podjednotek (angl. *subunits*). Ty se uplatní zejména v případě rozsáhlého textu podprogramu. Důsledkem dlouhé deklarace je jednak nepřehlednost (podprogram je volán z místa textově vzdáleného jeho deklaraci), ale i zdlouhavý překlad a nemožnost program dokončit při neznámých podrobnostech podprogramu. Podjednotka je tvořena separátně překládaným tělem podprogramu, který je v místě, kde by měl být normálně deklarován, nahražen tzv. zbytkem tvaru:

specifikace_podprogramu `is separate;`

Podjednotka má podobu samostatného souboru, tvořeného tělem podprogramu, kterému předchází klauzule

`separate` *(jméno_rodičovské_jednotky)*

Např.

```
procedure MAIN;         --samostatný soubor
       --deklarace konstant, typů a proměnných

       procedure P_SEPAROVANA is separate;
       function F_SEPAROVANA return TYP is separate;

begin
       --prikazy MAIN
end MAIN;
---------------------------------------------------------

separate (MAIN)         --samostatný soubor
procedure P_SEPAROVANA is
       --deklarace konstant, typů a proměnných procedury
begin
       --příkazy procedury
end P_SEPAROVANA;
----------------------------------------------------------

separate (MAIN)         --samostatný soubor
function F_SEPAROVANA is
       --deklarace konstant, typů a proměnných funkce
```

```
begin
       --příkazy funkce
end F_SEPAROVANA;
```

Poznamenejme, že separovaný podprogram může rovněž ve svých deklaracích obsahovat další separovaný podprogram. Ten by pak měl na svém začátku klauzuli tvaru např.:

```
separate(MAIN.P_SEPAROVANA)
```

Pro oddělený překlad podprogramů existuje ještě další možnost. Vytvořit z podprogramu samostatnou kompilační jednotku – soubor a v té jednotce, kde je používán, jej uvést v klauzuli `with`. V tomto případě je nutné překládat podprogram před překladem jednotky, která jej používá. Při použití konstrukce `separate` je tomu naopak.

Funkční hodnoty

Funkce mohou vracet i hodnoty, které jsou polem nebo záznamem. Např.

```
procedure P is
      type VE is array (INTEGER range <>) of INTEGER;
      R4, V4: VE(1 ..4);
      R6, V6: VE(1 ..6);
      function ARRAY_ABS(V: VE) return VE is
            R: VE(V‘RANGE);
      begin
            for J in V‘RANGE loop
                  R(J) := abs V(J);
            end loop;
            return R;
      end ARRAY_ABS;
begin
      V4 := (-1, 9, 5, 0);
      V6 := (1, 5, 21, 4, -81, 7);
      R4 := ARRAY_ABS(V4);
      R6 := ARRAY_ABS(V6);
end P;
```

Přetěžování

Přetěžováním podprogramů se míní použití stejného jména pro různé podprogramy. Důvodem je umožnit, aby logicky stejné funkce a procedury mohly mít stejné jméno. Např.:

```
function  PRIDEJ(F: FRONTA; P: PRVEK) return FRONTA is ...
procedure PRIDEJ(A: in out POLE; H: HODNOTA;KAM: INDEX) is...
procedure PRIDEJ(S: in out SEZNAM; P: PRVEK;KAM: MISTO) is...
```

Při volání těchto podprogramů je z kontextu (dle typů a počtu parametrů či typu funkční hodnoty) rozhodnutelné, který z podprogramů je volán. K odlišení je používáno i pořadí parametrů (při poziční notaci), či jmen formálních parametrů (při jmenné notaci).

Podobně jako podprogramy je dovoleno přetěžovat i operátory. Programátor si tak může zavést operátory pro své vlastní typy. Musí však dodržet zásadu, že přetěžovat lze jen ty operátory, které jsou v jazyku **ADA** zavedeny a jejich aritu musí zachovat. Přetížení je vyjádřeno deklarováním funkce, jejímž jménem je operátor.
Např.

```
function "+"(A,B: VEKTOR) return VEKTOR is
      VYSLEDEK: VEKTOR;
begin
      for I in A‘RANGE loop
            VYSLEDEK(I) := A(I) + B(I);
      end loop;
      return VYSLEDEK;
end "+";
```

Takto zavedenou funkci pak můžeme použít ke sčítání vektorů buď ve funkční notaci:

```
VEKTOR1 := "+"(VEKTOR2, VEKTOR3);
```

nebo v názornější notaci operátorové:

```
VEKTOR1 := VEKTOR2 + VEKTOR3;
```

3.3.4 Moduly

Moduly jsou prostředkem pro vytváření rozsáhlých programových celků. Umožňují sdružovat programové a datové prvky (procedury, funkce, typy, konstanty ...) do samostatně přeložitelných knihovních souborů. **ADA** je nazývá **package**. Již v dřívějším výkladu jsme se setkali např. s modulem `TEXT_IO`. Nemá sice podobu souboru, neboť je součástí prostředí jazyka **ADA**. Jeho používání i formální popis jsou však stejné jako těch modulů, které si programátor vytváří sám. Moduly mohou být také použity jako prostředek řízení viditelnosti jmen a vnořovány do jiných rozsahových jednotek. Tento způsob využití je však méně častý.

Základní vlastnosti modulů

Stejně jako procedury a funkce, patří i moduly mezi programové jednotky. Všechny programové jednotky mohou mít specifikační část a tělo. Specifikační část modulu uvádí prostředky, které modul zveřejňuje pro použití svým klientům (exportuje). Tělo obsahuje nezveřejňované implementační detaily exportovaných služeb, s nimi související lokální deklarace a případně i inicializační část. Inicializační část je umístěna před závěrečným end modulu a je provedena v okamžiku zpracování modulu v době výpočtu.

Příklad 3.5 Modul exportující zásobník

```
package STACK1 is            --specifikační část modulu
  subtype STACK_ITEM is INTEGER range -10_000..10_000;
  procedure PUSH(ITEM:STACK_ITEM);
  function POP return STACK_ITEM;
end STACK1;                  --konec specifikační části

package body STACK1 is        --tělo modulu
  STACK_SIZE: constant INTEGER := 1_000;
  STACK_ARRAY: array(1..STACK_SIZE) of STACK_ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(ITEM: STACK_ITEM) is
  begin
    TOP := TOP + 1;
```

```
    STACK_ARRAY(TOP) := ITEM;
  end PUSH;
  function POP return STACK_ITEM is
  begin
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin                                  --inicializační část
  TOP := 0;
end STACK1;                            --konec těla modulu
```

Klientem, který využívá služeb modulu `STACK1`, by mohl být např. hlavní program ve tvaru:

```
with STACK1,TEXT_IO; use STACK1,TEXT_IO;
procedure STACK1H is
  package STACK_IO is new INTEGER_IO(STACK_ITEM);
  use STACK_IO;
  X,Y: STACK_ITEM;

begin
  PUSH(5);
  X := 4;
  PUSH(X);
  PUT(POP);  --výstup 4
  Y := POP;  --výstup 5
  PUT(Y);    --výstup 5
  NEW_LINE;
end STACK1H;
```

Služby poskytované modulem `STACK1` jsou nedokonalé. Exportuje pouze jeden zásobník, neošetřuje překročení jeho rozsahu, jeho velikost je statická, typ prvků v zásobníku je neměnný. Možnosti jeho zlepšení uvedeme jako příklady dále probíraných jazykových konstrukcí.

Specifikační část modulu může být buď překládána s tělem společně (v jednom souboru), nebo odděleně. Pokud je překládána odděleně, musí její překlad předcházet překladu modulu klienta. Tělo modulu neovlivňuje propojení na uživatele, může být překládáno kdykoliv později a změna těla nevyžaduje rekompilaci modulů využívajících jeho služeb.

Specifikace kontextu

Moduly mohou ke své činnosti používat služeb jiných modulů. Ty, příp. jejich specifikační části, pak musí být v procesu překladu zpracovány dříve. V tomto smyslu existuje nad kompilačními jednotkami částečné uspořádání.

Jména modulů, ze kterých jsou importovány do modulu zdroje, musí být uvedena na začátku modulu formou klauzule.

`with` *jména_kompilačních_jednotek*;

Ta způsobí vyhledání příslušných jednotek v knihovně. Obvykle je následována use klauzulí:

`use` *jména_kompilačních_jednotek*;

která zviditelní importované zdroje a umožní jejich používání bez kvalifikace jménem modulu z něhož pochází. V případě, že některá jména nebudou tímto způsobem jednoznačně určena, je možné i nadále používat kvalifikace tečka-notací (*jméno_modulu.jméno_zdroje*).

Klauzule **with** nemá tranzitivní účinek.

Privátní typy

Pokusme se modifikovat program z př. 3.5 tak, aby modul exportoval typ zásobník a dovolil tím uživateli pracovat s více než jedním zásobníkem. Specifikační část by pak navíc obsahovala popis

```
type STACK_TYPE is
      record
            STK: array(1..1000) of STACK_ITEM;
            TOP: INTEGER range 0..1000;
      end record;
```

Uvedení této struktury ve specifikační části má však za následek nejen možnost deklarovat ve všech modulech exportujících `STACK1` libovolný počet zásobníků, ale i zpřístupnění vnitřní struktury zásobníku, což je nežádoucí. Zásobník by měl být chráněn proti destrukci ukrytím jeho interní podoby. Jediné uživateli povolené operace s ním by měly být ty, které pro něj modul exportuje.

ADA dovoluje znepřístupnit interní detaily typu protřednictvím tzv. privátních typů. Deklarace privátního typu má tvar:

a) `type` *jméno_typu* `is private;`

b) `type` *jméno_typu* `is limited private;`

Skutečná podoba privátního typu je popsána v privátní části, která je uváděna na konci specifikace modulu a od veřejné části je oddělena slovem **private**. Její přítomnost ve specifikační části je potřebná pro možný překlad uživatelů modulu dříve než těla modulu.

Nad privátními typy má uživatel možnost provádět modulem exportované operace a v případě a) ještě:

- přiřazování,
- test na rovnost,
- test na nerovnost.

Použití privátního typu ukazuje příklad 3.6. Exportovaný typ zásobníku je typu záznam s diskriminantem, který dovoluje určit velikost zásobníku. Uživateli jeho struktura přístupná není, je označen **limited private**, a proto všechny s ním manipulující operace jsou exportovány modulem.

Příklad 3.6 Modul exportující typ pro zásobník

```
package STACK2 is                  --specifikační část
  type STACK_TYPE(STACK_SIZE: NATURAL) is limited private;
  function IS_EMPTY(S:in STACK_TYPE) return BOOLEAN;
  function IS_FULL(S:in STACK_TYPE) return BOOLEAN;
  procedure PUSH(S: in out STACK_TYPE; ITEM:in INTEGER);
  procedure POP(S: in out STACK_TYPE; ITEM:out INTEGER);
  function "="(LEFT,RIGHT:in STACK_TYPE) return BOOLEAN;
  procedure COPY(FROM:in STACK_TYPE; TO:out STACK_TYPE);
  procedure CLEAR(S:in out STACK_TYPE);

private         --pro uživatele nepřístupná privátní část
```

```
  type INTEGER_ARRAY is array(NATURAL range<>)of INTEGER;
  type STACK_TYPE(STACK_SIZE: NATURAL) is
    record
      STK: INTEGER_ARRAY(1..STACK_SIZE) := (others=>0 );
      TOP: NATURAL := 0;
    end record;
end STACK2;                  --konec specifikační části
------------------------------------------------------------

with TEXT_IO; use TEXT_IO;      --tělo modulu, samostatně
                                --přeložitelné
package body STACK2 is
  function IS_EMPTY(S:in STACK_TYPE) return BOOLEAN is
  begin
    return S.TOP = 0;
  end IS_EMPTY;
  function IS_FULL(S:in STACK_TYPE) return BOOLEAN is
  begin return S.TOP = S.STACK_SIZE;
  end IS_FULL;

  procedure PUSH(S:in out STACK_TYPE; ITEM: in INTEGER) is
  begin
    if IS_FULL(S) then
       PUT_LINE("ERROR-OVERFLOW");
     else
       S.TOP := S.TOP + 1;
       S.STK(S.TOP) := ITEM;
     end if;
  end PUSH;

  procedure POP(S:in out STACK_TYPE;ITEM:out INTEGER) is
  begin
    if IS_EMPTY(S) then
       PUT_LINE("ERROR-UNDERFLOW");
    else
       ITEM := S.STK(S.TOP);
       S.TOP := S.TOP - 1;
```

```
    end if;
  end POP;

  function "="(LEFT,RIGHT:in STACK_TYPE) return BOOLEAN is
  begin
    return LEFT.TOP = RIGHT.TOP and then
        (LEFT.STK(1..LEFT.TOP)=RIGHT.STK(1..RIGHT.TOP));
  end "=";

  procedure COPY(FROM:in STACK_TYPE; TO:out STACK_TYPE) is
  begin            --kontrola zda je dostatek mista pro copy
    if FROM.TOP > TO.STACK_SIZE then
       PUT_LINE("ERROR-OVERFLOW");
    else                 --kopirovani
       TO.TOP := FROM.TOP;
       TO.STK(1..FROM.TOP) := FROM.STK(1..FROM.TOP);
    end if;
  end COPY;

  procedure CLEAR(S: in out STACK_TYPE) is
  begin
    S.TOP := 0;                --stačí nastavit TOP na nulu
  end CLEAR;
end STACK2;                    --konec těla modulu
```

S takto vytvořeným zásobníkem pak může uživatel pracovat ve svém programu např. způsobem:

```
with STACK2, ...; use STACK2, ...;
procedure STACK2H is
  S1,S2:   STACK_TYPE(20);  --zde deklaruje své vlastní
                                                  zásobníky
  I,J,K:INTEGER;
begin
  PUSH(S1,1);PUSH(S1,2);PUSH(S1,3);
  COPY(S1,S2);
  POP(S2,I);POP(S2,J);POP(S2,K);
  PUT(I);PUT(J);PUT(K);NEW_LINE;
end STACK2H;
```

3.3.5 Zpracování výjimečných situací

K zajištění co největší bezpečnosti programů **ADA** poskytuje mechanismus zpracování nenormálních a chybových stavů výpočtu – tzv. výjimečných situací (angl. *exception*). Uživatel má možnost popsat jejich zpracování v závěru příkazových částí programových konstrukcí. Výjimečné situace (krátce *výjimky*) **ADA** rozděluje na standardní (předdefinované) a definované uživatelem.

Výjimky jsou mimořádné situace při výpočtu programu, vedoucí k run-time chybám.

Standardní výjimečné situace

V modulu `STANDARD` jsou předdefinovány výjimečné situace:

1. `CONSTRAINT_ERROR`
2. `NUMERIC_ERROR`
3. `STORAGE_ERROR`
4. `TASKING_ERROR`
5. `PROGRAM_ERROR`

`CONSTRAINT_ERROR` je důsledkem překročení rozsahu přípustných hodnot. Vzniká např.:

- při hodnotě indexu mimo meze pole,
- je-li přiřazována hodnota překračující povolený rozsah pro levostrannou proměnnou,
- je-li předávaný parametr podprogramu mimo rozsah formálního parametru.

`NUMERIC_ERROR` je způsobována takovými aritmetickými operacemi, jejichž výsledek je ilegální, nebo operace je neproveditelná. Např. přetečení, podtečení, dělení nulou apod.

`STORAGE_ERROR` je důsledkem vyčerpání datové oblasti (zásobníku, či hromady). Nastává při vstupu do bloku, podprogramu, při zpracování deklarace, příp. alokace dynamické proměnné. Poněvadž samotné zpracování výjimky nemá paměťové nároky, je možné je provést.

`TASKING_ERROR` vzniká při pokusu o komunikaci s paralelně probíhajícím procesem, který není schopen komunikovat (viz později).

`PROGRAM_ERROR` zahrnuje ostatní druhy chybových situací. Častou příčinou je nedefinovaná hodnota funkce (výstup přes závěrečné end).

Ošetření výjimečných situací naznačuje příklad:

```
...
     begin
         ...  --posloupnost příkazů
     exception
         when NUMERIC_ERROR => příkazy záchranné akce1
         when CONSTRAINT_ERROR => příkazy záchranné akce2
     end;
...
```

Dojde-li v *posloupnosti příkazů* k některé z citovaných chyb, je výpočet přenesen na *příkazy záchranné akce*. Po jejich provedení je rozsahová jednotka (např. blok nebo podprogram), ve které došlo k chybě opuštěna, výpočet se do místa vzniku chyby nevrací. Forma popisu výjimek je obdobná příkazu **case**. Připouští i použití others alternativy, pro zahrnutí všech ostatních (explicitně necitovaných výjimečných situací. Oproti **case** příkazu nemusí však **exception** část obsahovat vyčerpávající výčet výjimek.

Standardně předdefinované výjimečné situace mohou nastat i při zpracování deklarací. V takovém případě je zpracování deklarací opuštěno a výjimka předána do dynamicky nadřazené konstrukce.

Uživatelem zavedené výjimečné situace

Uživateli je umožněno definovat své vlastní výjimky. Způsobí je pak příkazem

`raise` *jméno_výjimečné_situace*;

Tímto příkazem je ostatně možné vyvolat i předdefinované výjimky. Deklarace výjimky je podobná deklaraci proměnné, nemá však její vlastnosti (není dovoleno ji přiřazovat ani předávat jako parametr).

Časté je exportování výjimečných situací modulem. Jsou vlastně také jednou ze služeb, které může modul poskytovat, neboť představují nenormální situaci při obsluze uživatele modulu. Jak ale postupovat při nestandardní situaci, to musí určit uživatel modulu. Demonstrujme použití výjimek na příkladu modulu zásobníku. Pro úspornost zápisu použijme jako výchozí text modul STACK1.

Příklad 3.7 Modul exportující výjimky

```
package STACK3 is              --specifikační část modulu
  subtype STACK_ITEM is INTEGER range -10_000..10_000;
  procedure PUSH(ITEM:STACK_ITEM);
  function POP return STACK_ITEM;
   FULL, EMPTY: exception;     -- deklarace výjimek
end STACK3;                --konec specifikační části

package body STACK3 is     --tělo modulu
  STACK_SIZE: constant INTEGER := 1_000;
  STACK_ARRAY: array(1..STACK_SIZE) of STACK_ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(ITEM: STACK_ITEM) is
  begin
 if TOP = STACK_SIZE then raise FULL; end if; -- způsobí
                                                 výjimku
    TOP := TOP + 1;
    STACK_ARRAY(TOP) := ITEM;
  end PUSH;

  function POP return STACK_ITEM is
  begin
 if TOP = 0 then raise EMPTY; end if;   -- způsobí výjimku
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin                          --inicializační část
  TOP := 0;
end STACK3;                    --konec těla modulu
```

Uživatel modulu pak může programovat akce prováděné při překročení rozsahu zásobníku.

```
with STACK3,...; use STACK3,...;
   ...
  begin ...
     PUSH(NĚCO);  ...
     NĚCO := POP; ...
     ...
    exception
      when FULL   => PUT("zásobník je plný, hodnota"
                                       & "neuložena");
      when EMPTY  => PUT("zásobník je prázdný, pracuj"
                             & "s náhradní hodnotou");
                     NĚCO := NÁHRADNÍ_HODNOTA;
      when others => PUT("jina chyba");
   end;
   ...
```

Propagace výjimečných situací

Zpracování výjimky je dynamickým procesem. Pokud její zpracování (ovladač výjimky) není popsáno v rozsahové jednotce kde vznikla, je tato jednotka opuštěna a výpočet přenesen do dynamicky nadřazené jednotky. Buď do místa vyvolání v případě vzniku výjimky v podprogramu, či za koncové end v případě jejího vzniku v bloku. Ovladač výjimky je pak hledán v této nadřazené jednotce. Dojde-li se při této propagaci (šíření) výjimky až do hlavního programu a ani tam není ovladač uveden, výpočet je ukončen.

Vzhledem k dynamickému šíření výjimek mohou nastávat situace, kdy výjimku je třeba částečně ošetřit v místě, ve kterém její jméno není viditelné. K tomuto účelu lze využít alternativy **others**, spolu s příkazem **raise** bez udaného jména výjimky. Tuto podobu může mít příkaz **raise** pouze v exception části. Důsledkem pak je propagování té výjimky, která provádění exception části způsobila. Například:

```
exception
  when MOJE_CHYBA | TVOJE_CHYBA => PUT("naše chyba");
  when others   => PUT("jiná chyba");    --částečné ošetření
                   raise; --propagování výše k dokončení ošetření
end;
```

Provádění kontrol při výpočtu může být neúnosně časově náročné. Proto se připouští i možnost jejich potlačení pomocí direktiv překladači (nazývaných *pragma*). Např.

```
pragma SUPPRESS(INDEX_CHECK);
```

potlačí kontroly rozsahu indexů pole. Potlačení se může týkat také jen určité proměnné či typu, např.

```
pragma SUPPRESS(RANGE_CHECK, INTEGER);
```

Použití by se mělo pečlivě zvažovat, neboť snižuje bezpečnost programu. Je pouze doporučením pro implementaci, které nemusí být překladačem plně realizováno.

3.3.6 Generické programové jednotky

Generické programové jednotky jsou předlohami, které reprezentují abstraktní datové struktury

Umožňují programátorovi aplikovat logicky stejné funkce na různé datové objekty. Jejich konkrétní, proveditelné exempláře jsou vytvářeny příkazem generického ustavení - instalace (viz generování modulu `INT_IO` v příkladu 3.4.). V programovém textu jsou generické jednotky odlišeny jim předřazeným symbolem **generic**. Za ním následují generické formální parametry, které při generickém ustavení jsou nahrazovány skutečnými parametry a pak teprve je uveden obvyklý tvar programové jednotky. Rozdělují se na:

- generické funkce,
- generické procedury,
- generické moduly.

Generické jednotky nejsou přímo proveditelné, jsou vzorem pro vytvoření proveditelné programové jednotky

Generické procedury

Generické procedury mají obecný tvar:

> **generic**
> *generické formální parametry*
> **procedure JMENO**(*formální parametry*);
>
> generická specifikace

> **procedure JMENO**(*formální parametry*) **is**
> *deklarace*
> **begin**
> *příkazy*
> **end JMENO;**
>
> tělo

Práci s generickou procedurou ilustruje program `PLANETY`, který používá generický podprogram `VYJMENOVANY_TYP` pro tisk hodnot různých vyjmenovaných typů. Samotná procedura `VYJMENOVANY_TYP` je vytvořena s pomocí generického modulu pro I/O operace s výčtovými typy.

Příklad 3.8

```
with TEXT_IO; use TEXT_IO;
procedure PLANETY  is
   type SLUNECNI_SOUSTAVA is (MERKUR,VENUSE,ZEME,MARS,
                          JUPITER,SATURN,URAN,NEPTUN,PLUTO);
   type TYP_HUDBY is (JAZZ,FOLK,SONATA,METAL);

   generic                            -- generická procedura
      type VYJMENOVANY_TYP is (<>); -- gen.parametr je diskr.
                                      -- typu
   procedure TISK_VYJMENOVANY_TYP;
   procedure TISK_VYJMENOVANY_TYP is
       package VYJMENOVANY_TYP_IO is new
                           ENUMERATION_IO(VYJMENOVANY_TYP);
       use VYJMENOVANY_TYP_IO;
   begin for I in VYJMENOVANY_TYP loop
         PUT(I); NEW_LINE;
       end loop;
   end TISK_VYJMENOVANY_TYP;
```

```
   procedure TISK_PLANETY is new
       TISK_VYJMENOVANY_TYP(VYJMENOVANY_TYP =>
                                           SLUNECNI_SOUSTAVA);
   procedure TISK_HUDBY is new
       TISK_VYJMENOVANY_TYP(VYJMENOVANY_TYP => TYP_HUDBY);

begin -- hlavni program
   TISK_PLANETY;
   TISK_HUDBY;
end PLANETY;
```

Generické funkce

Generické funkce mají obecný tvar:

generic
        generické formální parametry
function *JMENO (formální parametry)* **return** *TYP*;

generická specifikace

function *JMENO (formální parametry)* **return** *TYP* **is**
        deklarace
begin
        příkazy
end *JMENO;*

tělo

Způsob práce s generickými funkcemi je obdobný práci s generickými procedurami.

Generické formální parametry

Generický formální parametr může udávat:

- typ,
- hodnotu (lze označit **in**) nebo proměnnou (musí být označen **in out**),
- podprogram.

Záměna generických formálních parametrů je prováděna v době překladu, při ustavení konkrétního exempláře programové jednotky.

Nejčastěji jsou používány generické formální typy. Jejich jednotlivé možnosti uvádí následující souhrn:

`type OBECNÝ is limited private;` skutečným parametrem může být libovolný typ, veškeré operace s ním jsou zadány generickými formálními podprogramy

`type PRIVÁTNÍ is private;` skutečný parametr je typem, který oproti předchozímu dovoluje navíc operace `:=, =, /=`

`type PŘÍSTUPOVÝ is access KAM;` skut. par. může být ukazatel zpřístupňující typ `KAM`, který bývá také generický

`type DISKRÉTNÍ is (<>);` lze jej nahradit diskrétním typem

`type INTEGER_TYP is range <>;` lze jej nahradit libovolným celočíselným typem

`type FIXED_TYP is delta <>;` lze jej nahradit libovolným typem v pevné řádové čárce

`type FLOAT_TYP is digits <>;` může být nahrazen libovolným typem v pohyblivé řádové čárce

`type DETERMINOVANÉ_POLE is array (INDEXY) of ELEMENTY;`
je přípustný pro libovolné determinované pole stejných dimenzí, typů indexů a typu komponent

`type NEDETERMINOVANÉ_POLE is array (INDEXY range <>) of ELEMENTY;`
je přípustný pro libovolné nedeterminované pole stejného počtu rozměrů, typů indexů a typů komponent

Parametrizace generickými hodnotami a proměnnými je formálně zapisována stejně jako deklarace proměnných. Pokud není uveden druh, předpokládá se implicitně **in**. Např.

```
generic
       MAX: INTEGER;
       DELKA: INTEGER := 60;          gener. hodnotové
       SIRKA: in INTEGER := 80;       parametry
procedure STRANKY ...
```

Parametrizaci generickými proměnnými nelze doporučit, neboť zpřístupňuje globální proměnné.

Generickými formálními parametry mohou být také podprogramy. Jejich použití je motivováno zejména potřebou zavést operace na generických typech. Proto se nejčastěji setkáváme s formálními generickými funkcemi. Způsob jejich použití uvedeme příkladem.

Příklad 3.9 Generická funkce vyhodnocující sumu z prvků vektoru. Předpokládejme obecnost rozsahu pole i typu prvků. V takovém případě musí uživatel funkce vyjádřit smysl operace sčítání pro příslušný typ prvků (čísla, matice, barvy ...).

```
generic
     type H is private;
     type P is array(INTEGER range <>) of H;
     NULA: H;

with function "+"(U,V: H) return H; --form. gener. funkce

function SUMA(X: P) return H is
     S: H := NULA;
begin
     for I in P'RANGE loop
          S := S + X(I);
     end loop;
     return S;
end SUMA;
```

Programový segment, který funkci používá, pak může např. mít tvar:

```
... declare

function MOJE_SUMA is new SUMA(FLOAT, MUJ_VEKTOR, 0.0,"+");

      V1: MUJ_VEKTOR;
      CELKEM: FLOAT;
      ...
    begin
          ...  CELKEM := MOJE_SUMA(V1);  ...
    end;  ...
```

Generické moduly

Generické prostředky jsou užitečné především pro vytváření programových knihoven. Také většina I/O operací jazyka **ADA** je poskytována z předdefinovaných generických modulů.

Tvar generického modulu se od normálního modulu liší pouze předřazením slova **generic** a generickými formálními parametry před jeho specifikační část.

Příklad, který dále uvádíme implementuje zásobník pro hodnoty libovolného typu, který připouští přiřazení a test na rovnost a nerovnost. Pro úspornost použijeme za výchozí text modulu STACK1 z př. 3.5.

Příklad 3.10 Generický zásobník

```
generic
  STACK_SIZE: POSITIVE; --generický parametr - velikost
                                              -- zásobníku
  type ITEM is private; --generický parametr - typ prvků
package STACK4 is
  procedure PUSH(A: ITEM);
  function POP return ITEM;
end STACK4;             --konec specifikační části
```

```
package body STACK4 is
  --deklaracni cast
  STACK_ARRAY: array(1..STACK_SIZE) of ITEM;
  TOP: INTEGER range 0..STACK_SIZE;

  procedure PUSH(A: ITEM) is
  begin
    TOP := TOP + 1;
    STACK_ARRAY(TOP) := A;
  end PUSH;
  function POP return ITEM is
  begin
    TOP := TOP - 1;
    return STACK_ARRAY(TOP + 1);
  end POP;

begin          --inicializační část
  TOP := 0;
end STACK4;
-------------------------------------------------------

--hlavní program, využívá kompilační jednotku STACK4
with STACK4,TEXT_IO; use TEXT_IO;
procedure STACK4H is
  package INTSTACK_IO is new INTEGER_IO(INTEGER);
  type TCIGARET is (PETRA, SPARTA, START, CAMEL);
  package ENUMSTACK_IO is new ENUMERATION_IO(TCIGARET);

    package STACK_100 is new STACK4(100, INTEGER);
    package STACK_100 is new STACK4(60, TRIGARET);

  use INTSTACK_IO, STACK_100, STACK_60;
  X,Y: INTEGER;
begin
  PUSH(5);  PUSH(SPARTA);
  X := 4;
  PUSH(PETRA);  PUSH(X);
  INTSTACK_IO.PUT(POP);  --výstup 4, kvalifikace to rozliší
```

```
  Y := POP;              --výstup 5
  PUT(Y);
  ENUMSTACK_IO.PUT(POP);  --vyžaduje kvalifikaci
  ENUMSTACK_IO.PUT(POP);  --vyžaduje kvalifikaci
  NEW_LINE;
end STACK4H;
```

3.3.7 Prostředky pro paralelní výpočty

Paralelně proveditelná aktivita (proces) je nazývána *task* (překl. úkol). Forma jeho zavedení je obdobná modulu. Jeho deklarace může být součástí deklarační části modulu, podprogramu, bloku nebo jiného úkolu.

Konstrukce **task** představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.

Podle tohoto programu je automaticky založen proces se stejným jménem, tj. nerozlišuje se program a podle něj probíhající proces (pokud není použita konstrukce typ úkolu, kterou probereme později). Příkaz aktivace není zaveden explicitně. Všechny procesy–úkoly jsou potomky hlavního programu, který lze rovněž chápat jako úkol, i když syntaktickou podobu úkolu nemá. Úkoly jsou aktivovány v okamžiku, kdy výpočet úkolu, který zpracovává jejich deklarace dosáhne **begin** za deklarační částí.

Úkol je složen ze specifikační části a z těla.

task *JMÉNO* **is**
  deklarace jmen komunikačních vstupů
end *JMÉNO*;
task body *JMÉNO* **is**
  lokální deklarace a příkazy
end *JMÉNO*;

– Úkoly mohou sdílet jeden procesor (multiprogramový režim), přepínání procesoru mezi úkoly zajišťuje podpůrný systém jazyka bez nutnosti jazykového vyjádření.

– Úkoly mohou sdílet více procesorů (multiprocesorový režim), jsou pak vykonávány fyzicky paralelně.

– Úkoly mohou být prováděny na více počítačích (distribuovaný systém). Jazykem **ADA** se to rovněž nevyjadřuje.

Oproti jiným programovým jednotkám platí pro úkoly jistá omezení:

a) Úkol nemůže být generickou jednotkou.

b) Úkol nemůže být knihovní jednotkou, tzn. jeho deklarace musí být součástí podprogramu nebo modulu. Jeho tělo však může být v rodičovské jednotce nahraženo zbytkem

   `task body` *JMÉNO* `is separate;`

   a překládáno jako podjednotka.

c) Textově lze umístit specifikaci úkolu stejným způsobem jako specifikaci modulu, s výjimkou uvedenou v b).

K normálnímu ukončení práce úkolu dojde průchodem end na konci jeho těla. Nadřazený úkol může skončit činnost až v okamžiku, kdy skončily všechny jím vytvořené procesy. K násilnému ukončení z jiného úkolu je možné použít příkazu

`abort` *jméno ukončovaného úkolu;*

Jeho uplatnění by mělo být výjimečné. Úkol je v jazyku **ADA** synonymem procesu. Ukončení procesu obecně implikuje ukončení všech jím vytvořených procesů. Pozastavené procesy je nutné před ukončením vyřadit z front, ve kterých čekají. V případě ukončování komunikujícího procesu je nutné uvědomit jeho partnery. Z těchto důvodů se použitím příkazu **abort** neplýtvá.

Rendezvous v ADě

Pro interakci procesů používá **ADA** principu asymetrického rendezvous (viz čl. 2.3), kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje také elegantní konstrukci monitorů.

Místům, ve kterých úkol akceptuje rendezvous se říká **entry** (tj. vstup). Popis vstupu je syntakticky stejný jako deklarace procedury. Příkazy, kterými jsou volány služby jiného úkolu, jsou nazývány volání vstupu (angl. call entry). Jedná se o příkaz k vlastnímu provedení rendezvous. Na úkol lze zevně pohlížet jako na množinu jeho vstupů. Specifikace úkolu pak zřejmě obsahuje specifikaci všech jeho vstupů.

Příkaz volání vstupu je obdobou volání procedury. Má tvar:

JMÉNO_ÚKOLU.JMÉNO_VSTUPU (parametry);

Pro každý vstup musí volaný úkol obsahovat alespoň jedno místo, ve kterém tento vstup akceptuje. Místo, kde je vstup akceptován má tvar příkazu:

`accept` *JMÉNO_VSTUPU (formální_parametry)* `do`
    seznam příkazů
`end` *JMÉNO_VSTUPU;*

Seznam příkazů (tělo **accept** příkazu) je prováděn v režimu vzájemného vyloučení, je kritickou sekcí.

Příkaz **accept** musí být uveden v základní úrovni těla úkolu, nelze jej vnořovat. Je však možné použít stejného jména příkazu **accept** v jednom úkolu několikrát.

Následující příklad představuje úkol, který realizuje paměťovou schránku pro jednu celočíselnou proměnnou. Úkoly používající tuto schránku mohou do ní zapsat je-li prázdná a číst z ní je-li plná. Mechanismem rendezvous se zabraňuje kolizi ve čtení a zápisu.

Příklad 3.11 Paměťová schránka

```
task SCHRANKA is
      entry PUT(X: in INTEGER);
      entry GET(X: out INTEGER);
end;
```

specifikační část

```
task body SCHRANKA is
      V:  INTEGER
begin
      loop
            accept PUT(X: in INTEGER) do
                  V := X;
            end PUT;
            accept GET(X: out INTEGER) do
                  X := V;
            end GET;
      end loop;
end SCHRANKA;
```

Uspořádáním příkazů `accept` v nekonečné smyčce je zajištěno, že číst bude možné až po zapsání údaje do schránky. Po aktivaci úkol čeká nejprve na volání vstupu `PUT` z jiného úkolu, které může mít tvar:

```
SCHRANKA.PUT(22);
```

Pak teprve může akceptovat vstup `GET`, např. tvaru:

```
SCHRANKA.GET(J);
```

V případě, že jeden vstup je volán z více úkolů, jsou volání frontována. Je zřejmé, že úkol může být jen v jedné frontě. Dokud není obsloužen, je pozastaven. Pomocí atributu `COUNT` je možné zjišťovat počet úkolů, které čekají na daný vstup. Např.

```
SCHRANKA.PUT'COUNT
```

Atributem `TERMINATED` je zjistitelné, zda úkol včetně svých potomků byl ukončen. Pro úkoly není použitelná klauzule **use**.

Vznikne-li uvnitř úkolu výjimečná situace, zpracuje se normálním způsobem. Singulární je však případ, kdy vznikne v průběhu komunikace úkolů, tj. při zpracování příkazu **accept**. Pokud tento příkaz neobsahuje lokální zpracování výjimky, je příkaz **accept** opuštěn a výjimka se propaguje jak za příkaz **accept**, tak i do volajícího úkolu. Při volání vstupu úkolu, který již skončil svoji činnost nebo byl ukončen příkazem **abort**, je ve volajícím úkolu způsobena výjimečná situace `TASKING_ERROR`.

Příkaz select – zavedení nedeterminismu

Dosud bylo uvažováno sekvenční zpracování programu úkolu. Pokud výpočet došel k příkazu **accept**, úkol byl nucen vyčkat volání vstupu právě tohoto příkazu **accept**. Úkol však často poskytuje množinu služeb, které mohou být volány v libovolném pořadí. Takový způsob zpracování umožňuje příkaz **select**.

Forma příkazu **select** je:

```
select
  accept  něco( parametry)  do
    příkazy;      --vlastní rendezvous
  end;
  příkazy;        --již mimo rendezvous. Např. úpravy
                  --předaných dat. Volající úkol pokračuje
                  --dál, ale volaný ještě neakceptuje další
                  --volání vstupu
or
  accept  něco_jiného( parametry) do
    ...
  end;
or
    ...
end select;
```

V příkazu **select** se akceptuje volání kteréhokoliv ze vstupů. Při souběhu volání několika vstupů z úkolů se stejnou prioritou *není determinováno v jakém pořadí se budou provádět.*

Alternativ **accept** v příkazu select může být libovolný počet a může jim předcházet zpřístupňující klauzule (tzv. *hlídka* – ang. *guard*) ve tvaru:

`when` *podmínka* =>

Ta zabrání provedení přislušného **accept** příkazu není-li podmínka splněna.

select
  when *podmínka* =>
    accept
      příkazy kritické sekce
    end;
    příkazy mimo kritickou sekci
or
  ...
end select;

Další možnosti alternativ v příkazu **select**:

a) Časově omezený příkaz **select**

> **select**
>   **accept** ...
> **or**
>   **delay** *nějaká_doba;*
> **end select;**

Parametr *nějaká_doba* je speciálního datového typu **duration** (trvání), předdeklarovaného v modulu `STANDARD`. Způsobí pozastavení úkolu na udanou dobu, nemůže-li se uplatnit žádný z příkazů **accept**. Je použitelný i mimo konstrukci **select**.

b) Alternativa terminate

> **select**
>   **accept** ...
> **or**
>   **terminate;**
> **end select;**

Úkoly, které čekají v příkazu **select** s alternativou **terminate** na volání některého ze svých vstupů, jsou potenciálně ukončitelné. Skončí svoji činnost při ukončení činnosti své nadřízené jednotky (programové jednotky v níž jsou deklarovány). Tou může být blok, podprogram, úkol či knihovní modul. Nadřízená jednotka (ang. *master*) může skončit činnost (např. průchodem **end**), pokud všechny v ní aktivované procesy již skončily činnost a nebo připouštějí své ukončení alternativou **terminate**. Alternativa **terminate** je potřebná zejména tehdy, je-li úkol realizován nekonečným cyklem. Současné použití příkazu `delay` se ovšem vylučuje.

c) Alternativa else

> **select**
>   **accept** ...
> **or**
>   ...
> **else**
>   *příkazy prováděné v případě neuplatnění žádného accept*
> **end select;**

Kombinace alternativ **delay, terminate** a `else` není smysluplná a proto se ani nedovoluje.

Příklad 3.12

Abstraktní datový typ `PAMĚŤOVÁ_SCHRÁNKA` dovolující libovolný počet čtení ze sdílené proměnné současně.

```
generic
     type PRVEK is private;
package SDILENA_PROMENNA is
     procedure PUT(X: in PRVEK);
     procedure GET(X: out PRVEK);
end SDILENA_PROMENNA;

package body SDILENA_PROMENNA is
     V: PRVEK;

     task RIZENI is
           entry START; --je možný i vstup bez parametrů
           entry STOP;
           entry PIS;
     end RIZENI;

     task body RIZENI is
           POCET: INTEGER := 0;
     begin
        accept PIS(X: in PRVEK) do
           V:=X;   --zajistí naplnění proměnné první
                   --hodnotou
        end PIS;
        loop
           select
             when PIS‘COUNT = 0 =>
              accept START; --"do příkazy" je nepovinné
              POCET:=POCET+1;--je mimo accept, probíhá
                             --paralelně
           or
```

```
               accept STOP;
               POCET:=POCET-1;
            or
              when POCET = 0 =>
               accept PIS(X: in PRVEK) do
                     V := X;
               end PIS;
            end select;
         end loop;
      end RIZENI;

      procedure GET(X: out PRVEK) is
      begin RIZENI.START;  X := V; RIZENI.STOP;
      end GET;

      procedure PUT(X: in PRVEK) is
      begin RIZENI.PIS(X);
      end PUT;

end SDILENA_PROMENNA;
```

Úkol `RIZENI` je aktivován po zpracování deklarací v těle modulu, vygenerovaného ze šablony `SDILENA_PROMENNA`. Vstupy `START` a `STOP` mají funkci signálů. Počet žádostí o čtení je obsažen v proměnné `POCET`. Čtení ze sdílené proměnné je dovoleno zahájit pouze tehdy, nechce–li žádný proces do ní zapisovat. Zapisovat je možné v tom případě, kdy nikdo právě její obsah nečte. Uvedené podmínky jsou zajišťovány hlídkami.

Příklad 3.13. ADT zásobník dovolující paralelní použití

Předpokládejme, že provádět zápis a výběr mohou různé procesy. Z toho plyne, že zásobník musí být koncipován jako monitor.

```
generic
SIZE: POSITIVE;
type  POLOZKA is private;
package STACK5 is
```

```
    task ZASOBNIK is
      entry VLOZ( X :  in POLOZKA);
      entry UBER( X : out POLOZKA);
    end ZASOBNIK;
  end STACK5;

  package body STACK5  is          -- tělo
    task body ZASOBNIK is
    S  : array  (1..SIZE)  of POLOZKA;
    SP : INTEGER range 0..SIZE;
    begin
      SP := 0;
      loop
        select
          when SP < SIZE =>
            accept VLOZ( X : in POLOZKA) do
              SP := SP + 1;
              S(SP) := X;
            end VLOZ;
        or
          when SP > 0 =>
            accept UBER( X : out POLOZKA) do
              X  := S(SP);
              SP := SP - 1;
            end UBER;
        or
          terminate;
        end select;
      end loop;
    end ZASOBNIK;
  end STACK5;
```

Příklad použití v hlavním programu:

```
with STACK5, TEXT_IO;  use TEXT_IO;
procedure MAIN is
   package STACKI is new STACK5(20, INTEGER);
   package STACKC is new STACK5(100, CHARACTER);
```

```
   C: CHARACTER := ‘A‘;
   I: INTEGER := 10;
begin ...
   STACKI.ZASOBNIK.VLOZ(I);
   STACKC.ZASOBNIK.VLOZ(C);
   STACKI.ZASOBNIK.VLOZ(5);
   STACKC.ZASOBNIK.VLOZ(‘B‘);
   STACKI.ZASOBNIK.UBER(I);
   STACKC.ZASOBNIK.UBER(C);
   PUT(C); NEW_LINE;...
end MAIN;
```

Vše, co bylo doposud řečeno o příkazu **select**, se týká jeho použití v obsluhujících (volaných) úkolech. Možnosti podmíněného a časově omezeného volání vstupu pomocí příkazu **select** má ale i aktivní (tj. volající) úkol.

d) Podmíněné volání vstupu
(řeší situaci není-li obsluha připravena)
select
jméno_úkolu.jméno_obsluhy (aktuální parametry);
případné další příkazy po provedení rendezvous
else
příkazy alternativního výpočtu není-li rendezvous ihned proveditelné
end select;

e) Časované volání vstupu (volání s timeoutem)
Volající úkol tím vyjadřuje, že akceptovat jeho volání vstupu lze v zadaném čase.
select
jméno_úkolu.jméno_obsluhy (aktuální parametry);
případné další příkazy po provedení rendezvous
or
delay *nějaká_doba;*
případné další příkazy při neuskutečnění rendezvous v zadané době
end select;

Typ úkol

Konstrukce typového úkolu představuje program, podle kterého může paralelně probíhat libovolný počet úkolů-procesů (tzn. je šablonou pro vytváření konkrétních úkolů). Úkoly-procesy musí být explicitně vytvořeny (a pojmenovány), a to buď staticky (deklarací úkolu příslušného typu), nebo dynamicky (pomocí new, kdy se k pojmenování používají ukazatelové proměnné). Platí, že typ úkol a proces-úkol dle něj probíhající, jsou ve stejném vztahu jako datový typ a proměnná tohoto typu.

Příklad 3.14. Typ zásobníku pro paralelní přístup

```
with EXPORTER_TYPU_POLOZKA;
use  EXPORTER_TYPU_POLOZKA;

 package STACK6 is
  task type ZASOBNIK is
     entry VLOZ( X: in POLOZKA);
     entry UBER( X: out POLOZKA);
  end ZASOBNIK;
  type REF_ZASOBNIK is access ZASOBNIK;
end STACK6;

package body STACK6 is
   task body ZASOBNIK is
     SIZE: constant INTEGER := 200;
     -- ostatní, počínaje deklarací S: array ...
     -- je stejné jako v předchozím příkladě 3.13.
end STACK6;
```

Použití v programu:

```
with STACK6, EXPORTER_TYPU_POLOZKA;
use  STACK6, EXPORTER_TYPU_POLOZKA;

procedure MAIN is
   ...
  Z1, Z2 : ZASOBNIK;
  Z3 : REF_ZASOBNIK := new ZASOBNIK;
```

```
    ...
  begin
    ....
    Z1.VLOZ(NĚCO);
    ...
  end MAIN;
```

Poznámka:
Vytvořené úkoly `Z1, Z2, Z3` jsou paralelně spuštěny ihned za příkazem **begin** v hlavním programu.
Nevýhodou současného tvaru příkladu 3.14 oproti generické konstrukci příkladu 3.13 je:

- potřeba importovat typ `POLOZKA` (nelze jej bohužel importovat z hlavního programu, do kterého je `STACK6` importován – viz kompilační závislosti),
- statický rozsah zásobníku.

Druhou z nevýhod je možno eliminovat prostřednictvím předání parametrů úkolu. Předání parametrů lze jednoduše zařídit doplněním speciálních vstupů, které slouží jen pro přenos parametrů, a přes které musí úkol přejít dříve než zahájí vlastní činnost. Upravme v tomto smyslu příklad 3.14.

```
with EXPORTER_TYPU_POLOZKA;use EXPORTER_TYPU_POLOZKA;
package STACK7 is
  task type ZASOBNIK is
    entry VLOZ( X :  in POLOZKA);

     entry PARAMETR(SI: in POSITIVE);

    entry UBER( X : out POLOZKA);
  end ZASOBNIK;
end STACK7;

package body STACK7  is          -- tělo
 task body ZASOBNIK is
  SIZE: POSITIVE;
 begin
```

```
accept PARAMETR(SI: in POSITIVE) do SIZE := SI;
```

```
      end PARAMETR;
      declare
       S  : array  (1..SIZE)  of POLOZKA;
       SP : INTEGER range 0..SIZE;
      begin
      ...
     end ZASOBNIK;
    end STACK7;
```

Hlavní program pak využívá typ `ZASOBNIK` způsobem např.

```
with STACK7, TEXT_IO;  use TEXT_IO, STACK7;
procedure MAIN is
   C: CHARACTER := 'A';
   I: INTEGER := 10;
   STACKC: ZASOBNIK;
begin
     ...
   STACKC.PARAMETR(100);
   STACKC.VLOZ(C);
   STACKC.VLOZ('B');
   STACKC.UBER(C);
   PUT(C); NEW_LINE;
     ...
end MAIN;
```

Příklad 3.15.
Na závěr výkladu o paralelním programování v jazyce **ADA** uvedeme ještě jeden rozsáhlejší příklad. Jedná se o generický modul – typový monitor, který umožní asynchronní komunikaci úkolů zasíláním zpráv přes kruhovou vyrovnávací paměť. Typ zasílaných zpráv může být libovolný. Vyrovnávací paměť je implementována jako pole.

Program je rozdělen do tří kompilačních jednotek:

- Soubor `BUF_S.ADA` obsahuje specifikaci univerzálního knihovního modulu (generic package `G_BUFFER`) pro monitor – kruhový buffer,

- soubor BUF_B.ADA obsahuje tělo modulu,
- soubor TESTBUF.ADA obsahuje proceduru TESTBUF – hlavní program pro demonstrační ověření správné funkce knihovního modulu.

```
    -- Soubor BUF_A.ADA (první kompilační jednotka) --

generic
  -- PARAMETRY

  type MESSAGE is private;          -- typ přenášené zprávy
  LBUFF: positive;                  -- délka bufferu v počtu
                                    -- zpráv
  with procedure NULL_MES (THE_MES: out MESSAGE);
                                    -- procedura pro nulování
                                    -- záznamu zprávy

package G_BUFFER is

  task type SERVER is               -- úkol pro obsluhu
                                    -- bufferu
    entry RESET;                    -- reset bufferu
    entry DESTROY;                  -- likvidace bufferu
    entry ADD (NEW_MES: in MESSAGE);-- zápis zprávy
    entry READ(THE_MES: out MESSAGE);-- poskytuje nejstarší
                                      -- zprávu
    entry STATE (FREE_NUM: out NATURAL); -- poskytuje stav
                                          -- bufferu
  end;

  type BUF_SERVER is access SERVER;
                      -- typ ukazatel na úkol typu SERVER

end G_BUFFER;
-----------------------------------------------------------
```

```
      -- Soubor BUF_B.ADA (druhá kompil. jednotka) --

package body G_BUFFER is

  task body SERVER is

    THE_BUFF: array(INTEGER range 1..LBUFF) of MESSAGE;
                                      -- vlastní buffer
    TAIL :INTEGER range 1..LBUFF+1;   -- index na první
                                      -- volné místo

    procedure INIT is          -- inicializační procedura
    begin
      for I in 1..LBUFF loop
        NULL_MES(THE_BUFF(I));
      end loop;
      TAIL := 1;
    end INIT;

    begin                             -- tělo úkolu
      INIT;
      loop
        select
          accept RESET;
            INIT;
        or
          accept DESTROY;
            exit;
        or
          when TAIL < LBUFF+1 =>
            accept ADD(NEW_MES: in MESSAGE) do
              THE_BUFF(TAIL) := NEW_MES;
              TAIL:=TAIL+1;
            end;
        or
          when TAIL > 1 =>
            accept READ(THE_MES: out MESSAGE) do
              THE_MES := THE_BUFF(1);
```

```
              TAIL := TAIL - 1;
              THE_BUFF(1..(TAIL-1)) := THE_BUFF (2..TAIL);
              NULL_MES(THE_BUFF(TAIL));
            end;
        or
          accept STATE (FREE_NUM: out NATURAL) do
            FREE_NUM := LBUFF - TAIL + 1;
          end;
        or
          terminate;
        end select;
      end loop;
    end SERVER;

end G_BUFFER;
------------------------------------------------------------

    -- Soubor TESTBUF.ADA (třetí kompilační jednotka) --

with TEXT_IO; use TEXT_IO;
with G_BUFFER;

procedure TESTBUF is                        -- hlavní program

  L_STRING : constant POSITIVE := 40;   -- délka zprávy
  L_BUFF : constant POSITIVE := 4;      -- délka bufferu

  subtype MESSAGE is STRING(1..L_STRING); -- typ zprávy
  KBD_MES: MESSAGE;                  -- zpráva z klávesnice

  procedure NULL_MES (THE_MES: out MESSAGE); -- specif.
                         -- procedury pro nulování zprávy
  package B is new G_BUFFER (MESSAGE, L_BUFF, NULL_MES);
    -- konkrétní verze generického modulu G_BUFFER
  use B;
  BUF: SERVER;    -- deklarace bufferu
                  -- (tj. vytvoření obslužného úkolu)
```

```
task type CONSUMER is  -- typový úkol pro čtení zprávy
                       -- z bufferu
                       -- nemá žádné entry
end;

task body CONSUMER is      -- tělo typového úkolu
  READ_MES : MESSAGE;      -- lok. proměnná pro přečtenou
                           -- zprávu
begin
  delay 0.5;          -- aby se stihlo vypsat ohlášení
                      -- hlavního programu
  PUT_LINE("Task CONSUMER pripraven");
  loop
    NULL_MES(READ_MES);   -- čistění lokální proměnné
    BUF.READ(READ_MES);   -- !!! rendezvous s úkolem BUF
    if READ_MES(1..6) = "zhebni" then
      PUT_LINE("Task CONSUMER se s Vami louci s pozdravem"
               & "DDT ZDAR !");
      exit;
    else
      PUT("Task CONSUMER :");
      PUT_LINE(READ_MES); NEW_LINE;
    end if;
  end loop;
end CONSUMER;

procedure NULL_MES (THE_MES: out MESSAGE)is
  -- tělo procedury - těla procedur a úkolů musí být až
  -- na konci deklarační části (po dekl. typů a prom.),
  -- proto byla na začátku specif. část pro využití
  -- v generickém modulu G_BUFFER
begin
  for I in 1..L_STRING loop
    THE_MES(I) := ' ';
  end loop;
end NULL_MES;
```

```
begin
  declare
    CONS1: CONSUMER; -- deklarace (vytvoření) úkolu CONS1
  begin
    PUT_LINE("Demonstracni program - komunikace"
                                  & "zasilanim zprav");
    delay 1.0;         -- aby se stihlo vypsat ohlášení
                       -- úkolu CONS1
    loop
      declare
        NUM_CHAR : NATURAL;
      begin
        PUT_LINE("Zadej nejaky text !");
        NULL_MES(KBD_MES);
        GET_LINE(KBD_MES, NUM_CHAR);
        BUF.ADD(KBD_MES); -- !!! rendezvous hlav. programu
                          -- s úkolem BUF
        delay 1.0;        -- aby úkol CONS1 stačil vypsat
                          -- zprávu
        if KBD_MES(1..6) = "zhebni" then
          exit;
        end if;
      end;
    end loop;
    PUT_LINE("Hlavni program - konec komunikace");
  end;                 -- zde dojde k likvidaci úkolu CONS1
end TESTBUF;           -- až zde dojde k likvidaci úkolu BUF
```

3.3.8 Kompilační závislosti

V předchozím výkladu byl intuitivně zaveden a používán pojem kompilační jednotka jako úsek programu mající podobu souboru, který je schopný samostatného překladu. Vlastnosti a formy kompilačních jednotek nyní probereme podrobněji.

Kompilační jednotkou může být:

- specifikační část modulu,
- tělo modulu,

- specifikační část generického modulu,
- tělo generického modulu,
- specifikační část podprogramu,
- tělo podprogramu,
- instalace generického podprogramu,
- instalace generického modulu.

Výhody použití kompilačních jednotek zahrnují:

- rekompilaci jen části programu v případě jeho modifikace,
- průhlednější modulární strukturu celého programu,
- umožnění týmové práce.

Kompilační jednotky se dělí ─┬─ primární (dále značeno 1.)
└─ sekundární (dále značeno 2.)

Překládaný soubor (kompilace) může být tvořen jedinou kompilační jednotkou nebo může zahrnovat několik kompilačních jednotek. Možnosti rozčlenění na kompilační jednotky:

a) V případě procedur:

   1. specifikace procedury
   2. tělo procedury

   nebo

   1. tělo procedury se specifikací (neliší se od těla)
   2. neexistuje

b) V případě funkcí:

   1. specifikace funkce
   2. tělo funkce

   nebo

   1. tělo funkce se specifikací (neliší se od těla)
   2. neexistuje

c) V případě modulů:

   1. specifikace modulu
   2. tělo modulu

   nebo neobsahuje–li modul příkazy

   1. specifikace modulu
   2. neexistuje

d) V případě generických jednotek vyžaduje řada implementací, aby primární jednotky (generické specifikace) byly překládány se sekundárními (těly) v jedné kompilaci t.j. v jednom souboru.

e) V případě podjednotek:

   1. neexistuje (je součástí rodiče a tak ji nelze samostatně překládat)
   2. separátně překládaná podjednotka.

Zavedením podjednotek je podporován způsob vytváření programů *shora dolů*, používání samostatného překladu ostatních kompilačních jednotek podporuje metodologii *zdola nahoru*.

Pro realizovatelnost překladu musí na kompilačních jednotkách existovat částečné uspořádání, vyjadřující jejich závislost. Závislosti, které platí mezi kompilačními jednotkami jsou tyto:

i. Podjednotky jsou závislé na svém rodiči. Rodičem může být jak primární tak i sekundární jednotka.

ii. Existuje-li primární i sekundární jednotka, pak sekundární jednotka (tělo) závisí na primární jednotce (specifikaci).

iii. Použitím klauzule **with** vzniká závislost klienta na primární jednotce citované v klauzuli **with**. Tuto klauzuli může používat jak primární, tak i sekundární jednotka. Použije-li ji primární jednotka, bude i odpovídající sekundární jednotka závislá na jednotkách citovaných ve **with** klauzuli.

Závislé kompilační jednotky lze překládat až po přeložení všech kompilačních jednotek, na kterých závisí.

Závislosti určují rozsah rekompilace při modifikaci kompilační jednotky. Překladač udržuje knihovnu v tomto smyslu konzistentní. Zda automaticky nebo upozorněním uživatele, to již závisí na implementaci.

3.3.9 Programování vstupních a výstupních operací

Podobně jako v jazyce **C** či **MODULA**, nejsou periferní operace ani v ADě součástí samotného jazyka. Jsou poskytovány prostřednictvím předdefinovaných modulů `TEXT_IO`, `SEQUENTIAL_IO`, `DIRECT_IO` a `IO_EXCEPTIONS`. Specifikační část těchto modulů je standardizována formou dodatků referenčního manuálu, popisujících tvary použitelných procedur, funkcí a výjimek pro periferní operace. Považujeme za účelné nepopisovat v tomto textu uvedené moduly vyčerpávajícím způsobem, zvídavější odkázat na manuál a soustředit se pouze na principy a nejčastěji potřebné operace.

`SEQUENTIAL_IO` je generický modul, dovolující vygenerovat pro zadaný datový typ periferní operace nad sekvenčními soubory. Exportovanými podprogramy jsou např. `CREATE, OPEN, CLOSE, DELETE, IS_OPEN, READ, WRITE, END_OF_FILE` atd.

`DIRECT_IO` je generický modul poskytující periferní operace pro práci se soubory s přímým přístupem. Pojmenování operací je shodné s předchozími. Je použitelný i pro sekvenční zpracování, navíc však dovoluje indexový přístup k položkám. Jeho procedury `READ` a `WRITE` proto mohou kromě parametrů udávajících soubor, proměnnou (příp. hodnotu), obsahovat ještě třetí parametr – index. Položky v souboru jsou indexovány od 1 výše.

V obou uvedených případech nemají zpracovávané soubory čitelný (v textovém smyslu) tvar. Položky souborů však mohou být libovolného privátního typu, je proto možné číst a zapisovat např. celé záznamy.

Pro práci s textovými soubory je k dispozici modul `TEXT_IO`. Byl použit v předchozích příkladech, některé jím poskytované služby již známe. Exportuje operace pro práci s řetězci a se znaky a čtyři generické moduly. Jeho strukturu znázorňuje obr.3.9.

Jsou k dispozici přetížené procedury pro I/O na standardní zařízení a do textových souborů. Doplňují je procedury pro formátování textu a pro manipulaci se soubory.

I/O operace pro znaky a řetězce

Generický modul INTEGER_IO
I/O operace s celými čísly

Generický modul FLOAT_IO
I/O operace s čísly v poh. řád. čárce

Generický modul FIXED_IO
I/O operace s čísly v pevné ř. čárce

Generický modul ENUMERATION_IO
I/O operace s výčtovými hodnotami

Obr. 3.9: Struktura modulu TEXT_IO

Negenerické prostředky modulu TEXT_IO

Pokud pracujeme se standardním I/O zařízením, máme k dispozici procedury:

```
procedure PUT(ITEM: in CHARACTER);
procedure PUT(ITEM: in STRING);
procedure PUT_LINE(ITEM: in STRING);
procedure GET(ITEM: out CHARACTER);
procedure GET(ITEM: out STRING);
procedure GET_LINE(ITEM: out STRING; LAST: out NATURAL);
```

U posledně jmenované je v druhém parametru vrácena délka přečteného řetězce. Dále jsou použitelné:

`SET_COL` (*číslo_sloupce)*; – nelze se ale vracet k nižším číslům

`SET_LINE` (*číslo_řádky*); – nelze se ale vracet k nižším číslům

`SET_LINE_LENGTH` (*délka_řádky*);

`SET_PAGE_LENGTH` (*délka_stránky*);

`NEW_LINE` (*počet_řádků*) – nemá-li parametr, pak odřádkuje o 1 ř.

`NEW_PAGE;`

`SKIP_PAGE;`

Zjištění současných hodnot je možné funkcemi:

`LINE_LENGTH`	–	bezparametrová, vrací počet sloupců řádky,
`PAGE_LENGTH`	–	bezparametrová, vrací počet řádků stránky,
`END_OF_LINE`	–	vrací pravdivostní hodnotu,
`END_OF_PAGE`	–	vrací pravdivostní hodnotu,
`COL`	–	bezparametrová, vrací číslo aktuálního sloupce,
`LINE`	–	bezparametrová, vrací číslo aktuálního řádku,
`PAGE`	–	bezparametrová, vrací číslo aktuální stránky.

V případě, že pracujeme s textovým souborem, jsou použitelné všechny výše citované operace. Mají pouze navíc jeden in parametr, který je v poziční notaci uváděn jako prvý a určuje soubor, na nějž se operace vztahuje. Např. specifikace `PUT` pro znak má tvar:

```
procedure PUT(FILE: in FILE_TYPE; ITEM: in CHARACTER);
```

Navíc jsou k dispozici procedury:

```
procedure CREATE(FILE: in out FILE_TYPE;--interní jméno soub.
                 MODE: in FILE_MODE:=OUT_FILE;
                 NAME: in STRING:="";--vnější jméno souboru
                 FORM: in STRING:="");--speciální informace
```

Poznámka:

Typy `FILE_TYPE` a `FILE_MODE` jsou exportovány modulem `TEXT_IO`. Parametr `FILE` určuje jméno souboru používané v programu, např. v příkazu `PUT`. Parametr `NAME` je jménem tohoto souboru ve vnějším systému (např. `DATA.TXT`). Neuvede-li se, je soubor považován za dočasný a při skončení programu je zrušen. Parametr `FORM` vyžadují některé implementace. Obsahuje např. údaj o maxim. velikosti souboru, password a pod. Třetí a čtvrtý parametr mají iniciované hodnoty a není proto nutné je uvádět. Jakmile soubor je vytvořen a má vnější jméno, je možné se na něj v programech obracet prostřednictvím i dalších uvedených procedur.

```
procedure OPEN(FILE: in out FILE_TYPE;
               MODE: in FILE_MODE;--OUT_FILE nebo IN_FILE
               NAME: in STRING;
               FORM: in STRING:="");
procedure CLOSE(FILE: in out FILE_TYPE);
procedure DELETE(FILE: in out FILE_NAME);
procedure RESET(FILE: in out FILE_NAME; MODE: in FILE_MODE);
procedure RESET(FILE: in out FILE_TYPE);--zachová hodn. MODE
function MODE(FILE: in FILE_TYPE) return FILE_MODE;
function NAME(FILE: in FILE_TYPE) return STRING;
function FORM(FILE: in FILE_TYPE) return STRING;
function IS_OPEN(FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_FILE(FILE: in FILE_TYPE) return BOOLEAN;
```

Poslední funkce je použitelná i v bezparametrovém tvaru, testuje-li se konec souboru na standardním zařízení. Je-li toto zařízení interaktivní, bývá použití `END_OF_FILE` problémové.

Pro zjištění a určení přednastaveného I/O zařízení a zjištění standardního I/O zařízení jsou k dispozici podprogramy:

```
procedure SET_INPUT(FILE: in FILE_TYPE);
procedure SET_OUTPUT(FILE: in FILE_TYPE);
function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;
function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;
```

Příklad 3.16. Vytvoření souboru

```
with TEXT_IO; use TEXT_IO;

procedure ZAPISUJ is
      SOUBOR: FILE_TYPE;    --deklaruje soubor
      LINE: STRING(1..0);
      LINE_LENGTH: NATURAL;
begin
      CREATE(SOUBOR, OUT_FILE,"VYSTUP.TXT");--vytvoří soubor
      PUT_LINE("napis nejakou moudrost (max. 80 znaku)");
```

```
      PUT_LINE("napis ""DOST"" kdyz jiz zadnou nevis");
      GET_LINE( LINE, LINE_LENGTH);
      while LINE(1..LINE_LENGTH) /= "DOST" loop
            PUT_LINE(SOUBOR, LINE(1..LINE_LENGTH); --zapíše
            GET_LINE(LINE, LINE_LENGTH); --čte z terminálu
      end loop;
      CLOSE(SOUBOR);      --uzavře soubor
 end ZAPISUJ;
```

Generické prostředky modulu TEXT_IO

Doplňují možnosti práce s textovými soubory o přetížené procedury `PUT` a `GET` pro číselné typy a výčtové typy. Dovolují při I/O operacích pracovat s implicitním I/O zařízením, s explicitně uvedeným souborem a i s interní řetězcovou proměnnou. Poslední z možností zde pouze připomínáme, ve výkladu není rozvedena, bude ale ilustrována příkladem.

Modul `INTEGER_IO` poskytuje tyto tvary procedur `GET` a `PUT`:

```
procedure GET(ITEM: in NUM;       --NUM je generickým
                                  --parametrem
              WIDTH: in FIELD:=0);--WIDTH udává počet
                                  --cifer čísla, pro 0
                                  --je volný formát, což
                                  --je nejčastější případ
procedure PUT(ITEM: in NUM;
              WIDTH: in FIELD:=NUM‘WIDTH;
              BASE: in NUMBER_BASE:=10);--volitelný základ
                                        --čísla
```

V uvedeném tvaru procedury pracují s implicitním I/O zařízením. Jejich mutaci (zde pro stručnost neuváděnou), kde prvým parametrem je zadáno jméno souboru, je snadné si představit. Totéž platí i pro případ ostatních generických modulů z `TEXT_IO`.

Modul `FLOAT_IO` dává k dispozici tytéž služby, ovšem pro typy v pohyblivé řádové čárce.

```
procedure GET(ITEM: out NUM; WIDTH: in FIELD := 0);
```

Je-li zadán parametr `WIDTH`, očekává se, že přečtený údaj s tímto počtem znaků je číslem v pohyblivé řádové čárce.

```
procedure PUT(ITEM: in NUM;
              FORE: in FIELD := 2;        --míst před tečkou
              AFT: in FIELD := NUM'DIGITS;--míst za tečkou
              EXP: in FIELD := 3);        --míst exponentu
```

Z modulu ENUMERATION_IO jsou exportovány procedury tvaru:

```
procedure GET(ITEM: ENUM); --ENUM je generickým parametrem

procedure PUT(ITEM: in ENUM;
              WIDTH: in FIELD := 0 --0 znamená, že literál
                                   --bude vypsán jen na
                                   --potřebný počet míst
              SET: in TYPE_SET:=UPPER_CASE);
                                   --altern.je LOWER_CASE
```

Příklad 3.17

```
with TEXT_IO; use TEXT_IO;
procedure VYSTUPY is
     package INT_IO is new INTEGER_IO(INTEGER);
     package FLT_IO is new FLOAT_IO(FLOAT);
     package BOOL_IO is new ENUMERATION_IO(BOOLEAN);
     use INT_IO, FLT_IO, BOOL_IO;
     VYSTUP: STRING(1..8);
begin
     PUT(TO=>VYSTUP,ITEM=>22, BASE=>2);--zapíše cel.čís.
                                       --do řetězce
     PUT(VYSTUP); --2#10110# vypíše na terminál
     PUT(TO=>VYSTUP,ITEM=>3.141,AFT=>1,EXP=>0);
                                     --float p. číslo
     PUT(VYSTUP):--     3.4  vypíše na terminál
     PUT(TO=>VYSTUP,ITEM=>TRUE,SET=>LOWER_CASE);--literál
     PUT(VYSTUP);--true a čtyři mezery vypíše na terminál
end VYSTUPY;
```

Kapitola 4

Programování na symetrickém multiprocesoru

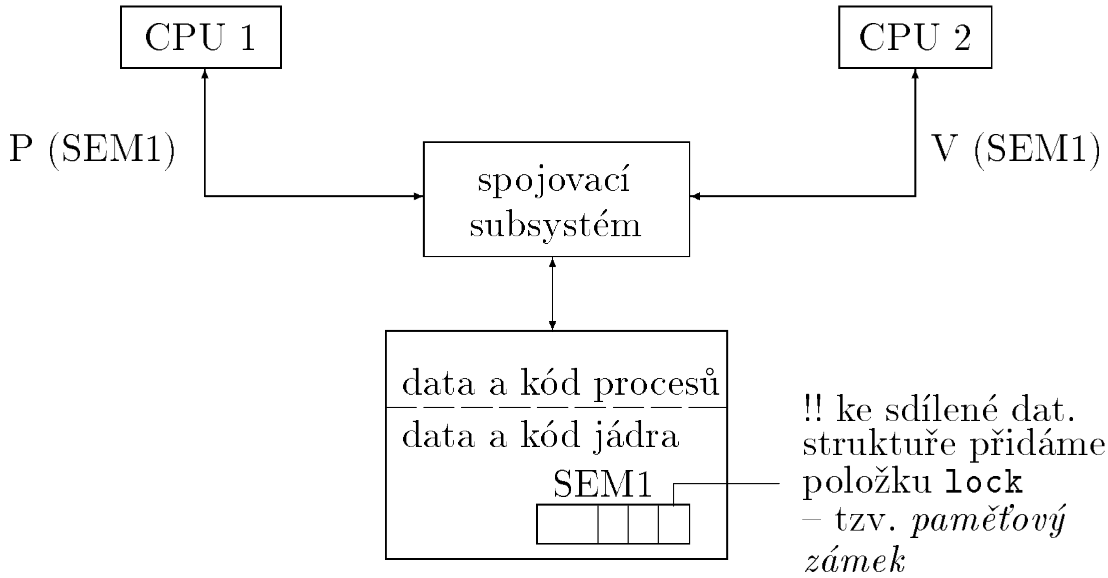
4.1 Nízkoúrovňové prostředky pro realizaci paralelního výpočtu

Systémová charakteristika symetrického multiprocesorového počítače již byla uvedena v první kapitole. Nejvýznamnějším rysem je uložení veškeré informace (kód OS, programy, data) v jedné sdílené paměti a náhodný (prostřídaný) dostup do sdílené paměti ze všech N procesorů systému. Probíhající paralelní výpočet organizuje *jádro OS*, které udržuje frontu procesů (vláken) připravených k výpočtu a dále fronty procesů čekajících na nějakou událost (např. přerušení z časovače, uvolnění semaforu ap.). *Jádro OS zřejmě může být vyvoláno (externím či programovým přerušením) na několika procesorech systému současně.* Tedy dříve než přistoupíme k problematice paralelizace aplikačního výpočtu, musíme se vyrovnat se zajištěním paralelní činnosti samotného operačního systému.

> Při paralelní činnosti jádra na několika procesorech vzniká problém konzistence sdílených datových struktur jádra. Musíme například řešit situaci, kdy v prvním procesoru je voláno jádro operací P a v druhém současně operací V, a to nad stejným semaforem.

Situace je schematicky znázorněna na obrázku 4.1.

Zřejmě je zapotřebí nějaký elektronicky podporovaný elementární synchronizační objekt. Nejčastěji se využívá technika tzv. *paměťových zámků.* Na paměťový zámek můžeme nahlížet jako na binární semafor s jednobitovou pamětí (typicky: hodnota 1 ... uzamčená data, 0 ... přístupná data)



Obr. 4.1: Zabezpečení paralelní činnosti jádra OS

s aktivním čekáním (tj. nikoliv ve frontě, ale v programové smyčce – angl. *busy wait*) [1].

Nad zámkem jsou definovány dvě operace:

lock (zámek); – uzamknutí (data nepřístupná)
unlock (zámek); – odemknutí (data přístupná)

Zajištění nedělitelnosti (atomicity) operace nad paměťovým zámkem (zejména *lock*) vyžaduje technickou podporu – klasická je operace označovaná jako `TEST_AND_SET` a realizovaná jedinou strojovou instrukcí. Tato operace v jednom přístupu do spojovacího subsystému (typicky v jednom paměťovém cyklu) nedělitelně uzamkne zámek (tj. zapíše do příslušné paměťové buňky hodnotu 1) a vrátí jeho starou hodnotu.

[1]Pro procesy–běhy jádra je aktivní čekání akceptovatelné, protože čekání nad zámkem by nemělo dlouho trvat. Navíc *pasivní čekání nemá kdo (superjádro?) organizovat.* Aby čekání netrvalo dlouho, je lepší použít větší počet zámků (např. pro každý semafor) než jeden globální zámek pro všechna data jádra (takový zámek by zabránil paralelnímu běhu jádra na více procesorech).

Paměťové zámky lze použít například i k zajištění kritických sekcí aplikačních procesů (viz dále PPL pro Sequent), ale jen v případě, že průměrná doba aktivního čekání na zámek je srovnatelná s dobou přepnutí kontextu procesů – využití semaforu s pasivním čekáním sice vyžaduje přepnutí kontextu (tj. zdržení), ale jádro může využít procesor spuštěním jiného připraveného procesu.

Zapsáno v syntaxi C jazyka:

```
int test_and_set (char *p_zamek)
{
    ....
    (* zamyká zámek a~vrací
    jeho starou hodnotu *)
}
```

Operace **lock()** je pak pomocí `test_and_set()` implementována například následovně:

```
void lock (char *p_zamek) {
  while (test_and_set(p_zamek) == 1) {
    ; (* prázdný příkaz *)
  }
}
```

Operace jádra (např. $P()$, $V()$ pro semafor) jsou pak realizovány zhruba podle následujícího vzoru [2]:

```
void P (SEMAFOR *p_sem) {
  disable ();       (* zákaz ext. přerušení *)
  lock (&(p_sem->zamek));
    ...
    (* vlastní operace P *)
  p_sem->zamek = 0; (* operace unlock, stačí obyč. přiřazení)
  enable ();        (* povolení přerušení *)
}
```

Příklad 4.1

Uvedeme funkci `lock()` využívanou pro počítače Sequent Symmetry. Místo instrukce `TEST_AND_SET` se využívá instrukce `XCHG`, která vymění obsah registru a paměťové buňky zámku. Atomicita operace je zajištěna zablokováním systémové sběrnice. U některých dalších instrukcí lze v assembleru předepsat atomicitu operace prefixem `LOCK`.

Funkce `lock()` je zapsána v celkem průhledné podobě v assembleru pro počítače Sequent.

[2] Zákaz externích přerušení může být realizován automaticky, pokud např. jsou všechna volání jádra realizována jako programová přerušení.

```
asm void LOCK (lockadd) {
  %reg lockadd; lab loop, spin, done;

  loop: movb  $LOCK, %dl    /* hodnota "zamčeno" do reg. dl */
        xchgb %dl, (lockadd)  /* provedení "test & set" */
        cmpb  $UNLOCK, %dl  /* test zda byl zámek zamčený */
        je    done          /* nebyl (a~už je)  */

        /* dyn. čekání nad kopií zámku v~cache paměti */
  spin: cmpb   $UNLOCK, (lockadd)
        je    loop          /* zámek odemknut - změna cache */
        jmp   spin
  done:

}
```

Při prvním čtení hodnoty zámku je zavedena příslušná stránka sdílené paměti do vyrovnávací (angl. *cache*) paměti procesoru. Smyčka `spin` potom probíhá nad kopií hodnoty zámku a *nezatěžuje spojovací subsystém.* Při odemknutí zámku (jiným procesem) se nejprve změní obsah ve sdílené paměti, což vyvolá elektronicky zajištěnou aktualizaci vyrovnávacích pamětí procesorů. Teprve potom je proveden skok `je loop`.

4.2 Programová implementace modelu SPMD

V předchozích kapitolách jsme vesměs uvažovali logicky paralelní výpočet (tj. i s možnou pseudoparalelní realizací) s uvažovanou aplikací: simulace, real-time a operační systémy. V tomto případě byla dekompozice výpočtu na paralelní procesy přímočará (dílčí málo závislé činnosti v aplikaci) a vedla na využití programového modelu MPMD. Nezdůrazňovali jsme výkonnost, resp. výkonnostní aspekt paralelního přístupu mohl být vítaný, ale nebyl podstatný pro strukturování programů [3].

Využívání modelu MPMD na symetrickém multiprocesorovém počítači nevyžaduje žádné specifické programovací techniky ve srovnání s dosud

[3] Například program v jazyku **ADA** odladěný na jednoprocesorovém počítači (např. viz Digital **ADA** v dodatku) je beze změn přenositelný na symetrický multiprocesor (např. viz **ADA** na počítači Sequent, dodatek).

uvedenými [4]. O alokaci procesů na procesory není třeba se starat, u symetrických multiprocesorů může běžet každý proces na jakémkoliv procesoru a rozvržení provádí jádro dynamicky za běhu výpočtu.

V symetrické multiprocesorové architektuře přichází v úvahu rovněž využití modelu SPMD, tj. paralelní dekompozice podle dat.

> Všechny výpočty, které lze urychlit pomocí dekompozice podle dat, nějak souvisí s paralelizováním cyklů. Jinak řečeno: řešíme-li problém sekvenčním algoritmem, musíme použít cykl přes všechny části dat. Jde tedy o paralelní provedení tohoto cyklu.

4.2.1 Paralelizace cyklů

> Základní myšlenka: Svěřit každou *iteraci* cyklu (popřípadě několik iterací) k vykonání jinému procesu. Jinak řečeno: snažit se vykonat co nejvíce iterací cyklu paralelně.

Dříve než budeme tuto myšlenku programově realizovat, je třeba analyzovat všechna data výpočtu (tj. všechny proměnné využívané v cyklu) s cílem zařadit je do určitých kategorií, podle kterých s nimi budeme nakládat. Nejprve určujeme:

a) *která data mohou být sdílena mezi paralelními procesy iterací a která musí být lokální v každém procesu,*

> Proměnná může být *lokální* (privátní), jestliže je iniciována v každé iteraci cyklu. Všechny ostatní proměnné jsou *sdílené* (tj. jejich hodnoty se přenáší mezi jednotlivými iteracemi cyklu).

b) *které sdílené proměnné způsobují závislost výsledku na pořadí vykonávání iterací nebo musí být ochráněny kritickou sekcí.*

> Sdílené proměnné dělíme na *nezávislé* a *závislé*. Nezávislé jsou tehdy, jestliže:
>
> - jsou využívány pouze pro čtení,
> - jedná se o pole, přičemž v jedné iteraci je využíván jen jeden prvek pole (pro čtení, pro zápis, popřípadě pro čtení i pro zápis).

[4] Lze využívat paměťové zámky podobným způsobem jako binární semafory.

S lokálními a nezávislými proměnnými nejsou při paralelizaci cyklu žádné potíže. Totéž se nedá říci o zbývajících kategoriích dat, které takto vymezujeme jako *závislé.*

Závislé proměnné dělíme do tří kategorií:

1. *redukční proměnná* (angl. *reduction variable*)

   - V průběhu iterace je využita jen v jedné asociativní komutativní operaci, tj. $+$, $*$, `OR, AND, XOR`.
   - příkaz má obecně formu:
     var := var oper výraz;

     Tedy redukční proměnná je v rámci přiřazovacího příkazu nejprve čtena, pak zapsána. Příklad cyklu:
     `for(i = 0; i < n; i++) suma = suma + a[i];`

zde:	`suma`	je redukční proměnná,
	`a`	je nezávislá sdílená proměnná.

2. *uzamykaná proměnná* (angl. *locked variable*)

   - může být čtena a zapisována ve více než jedné iteraci cyklu (a na více místech iterace),
   - jestliže budou iterace cyklu vypočítávány postupně (tj. sekvenčně) v náhodném pořadí, operace využívající uvažovanou proměnnou budou bez dalších opatření produkovat správné výsledky.

Příklad 4.2

Dáno pole bodů `x[n]`, `y[n]`. Hledáme bod s nejmenší vzdáleností od středu souřadné soustavy.

```
min = 1.0e10;
for(i = 0; i< n; i++) {
  dist = sqrt (sq (x[i]) + sq (y[i]));
  if (dist < min) { min = dist; min_i = i;}
}
```

tedy:	`dist`	je lokální proměnná,
	`x, y`	jsou nezávislé sdílené proměnné,
	`min, min_i`	jsou sdílené uzamykané.

3. *uspořádaná proměnná* (angl. *ordered variable*)

   Skalární proměnná nebo proměnná typu pole, která má následující vlastnost: *Operace s proměnnou dávají správné výsledky tehdy, jsou-li prováděny postupně ve stanoveném pořadí (daném předpisem cyklu).*

   Příklad 4.3

   V poli `x[n]` jsou uloženy hodnoty prvků posloupnosti X_i pro $i = 0, \ldots, n-1$. Chceme určit první diferenci (ukládáme do pole $d1[n-1]$ pro $i = 0, \ldots, n-2$) a druhou diferenci (pole $d2[n-2]$, $i = 0, \ldots, n-3$).

```
d1[0] = x[1] - x[0];
for (i = 1; i < n-1; i++) {
  d1[i] = x[i+1] - x[i];
  d2[i-1] = d1[i] - d1[i-1];
}
```

zde:	`x`	je sdílené nezávislé pole
	`d2`	je sdílené nezávislé pole (v i-té iteraci se používá jen i-tý prvek)
	`d1`	je uspořádaná proměnná.

   V i-té iteraci se používá hodnota $d1[i-1]$ vypočítaná v předchozí iteraci. Tedy i-tá iterace může být provedena až po ukončení iterace $i-1$.

Jak se při paralelizaci cyklu zachází s jednotlivými typy proměnných?

a) *Řídicí proměnná cyklu.* Její kopii mají všechny procesy jako lokální proměnnou. Po vytvoření procesu si ji každý proces nastaví na číslo první iterace, kterou má vykonat a dále ji zvyšuje sám (viz dále statický a dynamický způsob přidělování práce procesům, příklady).

b) *Sdílené a lokální proměnné* se musí rozlišit již při deklaraci. Sdílené proměnné přijdou do sdílené části paměti, lokální do části přístupné jen příslušnému procesu (např. do zásobníku procesu).
   Operace se sdílenými a lokálními proměnnými dále nevyžadují žádná speciální programová opatření.

c) *Uzamykané proměnné* se chrání kritickou sekcí (jako prostředky slouží semafory, zámky, atd. – viz dále PPL – příklady).

d) *Redukční proměnná.* Zavede se jako sdílená `RP_GLOB`. Dále se zavede lokální proměnná `RP_LOC` „zastupující“ redukční proměnnou (každý proces má svou kopii), inicializuje se a pracuje se s ní jako s lokální proměnnou. Po provedení všech iterací cyklu se v kritické sekci provede aktualizace `RP_GLOB` s využitím vypočítané lokální hodnoty `RP_LOC` (kritická sekce je krátká).

Příklad 4.4

Provedeme paralelní součet všech prvků pole `a[n]`, uvažujeme indexování od 0 do $n-1$.
Sekvenční podoba uvažovaného algoritmu je:

```
suma = 0;
for (i = 0; i < n; i++)  suma = suma + a[i];
```

Pokud je pole dlouhé, vyplatí se vytvořit $k, (k << n)$ procesů (číslovaných od nuly), každý z nich sečte k-tý díl pole a svůj výsledek „přidá“ ke „kolektivnímu“ součtu. Zavedeme proměnné:

`suma`	...	bude sdílená redukční proměnná
`a`	...	nezávislá sdílená proměnná
`loc_suma`	...	lokální náhrada sdílené redukční proměnné

V inicializační části výpočtu (nutno provést sekvenčně jedním procesem – připomínáme symbol f v Amdahlově zákonu, kap. 1) se provede nastavení `suma = 0`. Každý proces podílející se na výpočtu výsledného součtu pak bude probíhat podle následujícího programu:

```
loc_suma = 0;
for (i = číslo_procesu; i < n; i = i + k)
  loc_suma = loc_suma + a[i];
kritická_sekce_begin();
  suma = suma + loc_suma;
kritická_sekce_end();
```

e) **Uspořádaná proměnná.**
Sekce kódu v těle cyklu, která manipuluje s uspořádanou proměnnou, musí být v jednotlivých procesech počítána ve správném pořadí. Pro každou uspořádanou proměnnou a jí odpovídající sekci kódu musí být opakován následující postup:

- v programu základního procesu (tj. mimo paralelizovaný cyklus) se deklaruje globální celočíselný čítač, který udržuje číslo aktuální iterace cyklu (je inicializován na výchozí hodnotu řídicí proměnné cyklu),
- v programu procesu iterace cyklu se na začátek příslušné sekce kódu zařadí podmíněný příkaz, který realizuje dynamické čekání až čítač dosáhne hodnotu předvolené iterace,
- po provedení iterace se zařadí inkrementace globálního čítače aktuální iterace.

Příklad 4.5:

Využijeme příklad 4.3 – výpočet druhé diference. Zavedeme „hlídač" čísla aktuální iterace `i_guard`. V základním procesu provedeme inicializace `i_guard = 0; d1[0] = x[1] - x[0];` Vytvoříme k procesů (číslovaných od nuly), každý z nich by měl vypočítat k-tý díl výsledného pole druhých diferencí.

Společný program spolupracujících procesů:

```
for (i = číslo_procesu+1; i < n-1 ; i = i + k) {
  while (i_guard != i) continue;  \*  čekání *\
  d1 [i] = x [i+1] - x [i];
  d2 [i-1] = d1 [i]  - d1 [i-1];
  i_guard = i_guard + 1;
}
```

Z takovéto „paralelizace" ovšem nebudeme mít žádný užitek, protože z k vytvořených procesů bude v určitém čase v činnosti pouze jeden.

Uspořádané proměnné jsou (na rozdíl od ostatních uvažovaných kategorií) výraznou překážkou při paralelizaci cyklu.

4.2.2 Problém statického a dynamického plánování

Jedná se o problém specifický pro paralelní dekompozici podle dat a zahrnuje následující složky:

- Kolik procesů máme vytvořit?
- Jak jim rozdělit práci?
- Jak poznat, že už je všechna práce hotová?

Přitom předpokládáme že:

- Výpočet probíhá na symetrickém multiprocesoru.
  Je tedy vyloučeno pevné, překladačem dané přidělení procesů procesorům (a nemá ani smysl, protože procesory jsou rovnocenné).
- Počítač nemáme sami pro sebe (multiuživatelské prostředí).
  Dostaneme tedy jen část procesorů (předem nespecifikovanou) nebo se o dostupné procesory musíme střídat (*time sharing*) s ostatními uživateli počítače.

> Nemá valný smysl vytvářet více procesů, než je k dispozici procesorů [a]. Administrativní režie spojená se zakládáním (rušením) procesů by zmenšila urychlení S odpovídající počtu dostupných procesorů.
>
> [a] Snad jedině v případě, kdy na symetrickém multiprocesoru realizujeme algoritmus vyžadující konkrétní topologii (třeba 2D mřížku) procesorové sítě, tj. když na sym. multiprocesoru emulujeme multiprocesor s distribuovanou pamětí.

Při „dělení práce“ máme dvě základní možnosti:

Statické plánování (angl. *static scheduling*)

- Před zahájením paralelního výpočtu rozdělíme práci na části, počet částí k odpovídá počtu použitelných procesorů N.
- Vytvoříme proces pro každou část počítaného úkolu.
- Každý vytvořený proces se chová následovně:
  - zjistí, jakou část práce má dělat,
  - vykoná ji,
  - čeká na všechny ostatní procesy až ukončí svoji práci.

Dynamické plánování (angl. *dynamic scheduling*)

- Vytvoříme seznam prací, které se mají udělat (nemusí vypadat explicitně jako seznam, stačí např. čítač – viz dále příklad 4.7). Je–li to jednodnoduše možné zařídit, *objemnější části práce by měly být na začátku seznamu.*
- Vytvoříme takový počet procesů, *jaký umožní aktuální stav výpočetního systému (tj. kolik je procesorů k dispozici).*
- Každý proces se chová následovně:
  Čeká, až se vyskytne v seznamu úkol, odstraní jej ze seznamu a vykoná požadovanou práci. Potom testuje seznam, zda neobsahuje další neprovedený úkol a buď úkol opět zpracuje, nebo čeká dále.
- Při dělení dat na paralelně zpracovávané části nevytváříme příliš „malé úkoly" do seznamu prací s ohledem na administrativní režii spojenou s převzetím příslušného úkolu (viz dále volání `m_next()` ve funkci `find_dist()` v příkladu 4.7). Na druhé straně vytvoření malého počtu „velkých úkolů" (tj. počet úkolů je srovnatelný s počtem dostupných procesorů) zhoršuje ukazatel S urychlení výpočtu (výpočet posledního úkolu probíhá zčásti sekvenčně).

Výhody a nevýhody obou způsobů:

<table>
<tr><th>Statické plánování</th><th>Dynamické plánování</th></tr>
<tr><td>– vyplácí se jen když všechny úkoly trvají stejně dlouho a je k dispozici procesor pro všechny procesy (zakládání procesů je časově náročné a zakládat procesy „do zásoby" se nevyplatí)
– není vhodné pro multiuživatelský režim práce počítače</td><td>– počet úkolů nijak nesouvisí s počtem dostupných procesorů
– vede k rovnoměrnému zatížení procesorů i pro nestejně velké úkoly (musí jich ale být větší počet)
– teoretickou nevýhodou je dodatečné zpoždění vzniklé testováním seznamu úkolů</td></tr>
<tr><td colspan="2">V obou případech je možné paralelní algoritmus programově realizovat tak, že může využít libovolný počet dostupných procesorů (angl. označováno jako scalable computation).</td></tr>
</table>

4.3 Parallel Programming Library

Jedná se o knihovnu funkcí pro podporu paralelního výpočtu (dále PPL) využívanou pod systémem DYNIX (paralelní verze **Unix**u) na počítačích

Sequent Symmetry 2000

Zde uvádíme jako příklad knihovny, která podporuje využití zejména programového modelu SPMD (s využitím dalších knihoven i MPMD a MPSD) v běžném programovacím jazyce [PPL89].

PPL jde použít v programovacích jazycích **Fortran**, **Pascal**, **C**. Dále se budeme orientovat na využití jazyka **C**. Jazyk se liší od ANSI standardu používáním klíčových slov **shared** a **private** umožňujících rozlišit při definici sdílenou a lokální proměnnou. Rozlišení má smysl jen u proměnných v paměťové třídě *extern* nebo *static*, automatické a registrové proměnné v podprogramech jsou vždy privátní.

Knihovna je založena na metodě označované jako **microtasking**. Jedná se o paralelní výpočet mnoha malých úkolů – *mikrotasků* v rámci jednoho makro–úkolu.

Princip metody lze stručně shrnout do následujících bodů:

- Paralelní procesy *sdílí část dat* a vytváří si vlastní privátní kopie ostatních dat.
- Rozdělení výpočetního zatížení na dostupné procesory je zajištěno automaticky na úrovni jádra OS a aplikační programátor se o tuto záležitost nemusí nijak starat.
- Každý cykl určený pro paralelní výpočet je naprogramován formou podprogramu (viz dále `func()`).
- V hlavním programu (program základního – otcovského procesu) jsou prostřídány sekvenční části (provádí jen základní proces) a paralelně proveditelná větvení (paralelizované cykly). Rozvětvení výpočtu na paralelní procesy (potomky) je provedeno speciální operací `m_fork()`, která má jako parametr příslušný podprogram `func()`. Pro každý proces je vytvořena jeho kopie privátních dat.
- Jestliže jednotlivé iterace v cyklu nejsou úplně nezávislé, musí podprogram `func()` zajistit synchronizaci procesů (např. kritické sekce s využitím paměťových zámků).

- Po ukončení operace `m_fork()` zůstávají procesy–potomci v čekací smyčce a proces–otec probíhající podle hlavního programu je může buď zrušit, nechat běžet, nebo "uspat" a později znovu využít (s ohledem na časově náročné vytváření „nových potomků").

Knihovna PPL se dělí na tři části, označované v originálu [PPL89] jako:

- *mikrotasking library* (hlavičkový soubor *microtask.h*),
- data–partitioning routines (hlavičkový soubor *parallel.h*),
- memory allocation routines (hlavičkový soubor *parallel.h*).

Tyto části budou popsány v následujících odstavcích.

4.3.1 Mikrotasking library

Základní funkcí v knihovně je `m_fork()`. Ostatní funkce mají k uvedené nějaký vztah a nemá smysl je používat samostatně (bez souvislosti s předchozím nebo následujícím voláním `m_fork()`).

Idea:	Vytvoří se podprogram `func()` pro jednu nebo několik iterací cyklu a voláním `m_fork()` se rozběhne **najednou několik** paralelních procesů probíhajících podle tohoto podprogramu.

Hlavní proces (tj. ten, co vyvolal `m_fork()`) po volání nezahálí, ale převezme jeden díl práce, podobně jako založené děti (založení procesu je v DYNIXu časově náročné a tímto se jedno ušetří – pouze se přepne program procesu – způsob již byl popsán v kap.3 o procesech v **Unix**u).

Všechny procesy vytvořené voláním `m_fork()` mohou využívat pro koordinaci svojí činnosti jeden (nepojmenovaný) *zámek*, jeden (nepojmenovaný) *čítač* a jednu tzv. *bariéru*. Bariéra je synchronizační objekt, určený pro synchronizaci všech procesů (počkají na sebe). Synchronizační bod je ve společném programu procesů realizován voláním `sync()`.

Vybrané funkce z PPL mikrotasking library:

`int m_set_procs (int nprocs);`

Nastavuje globální proměnnou `m_numprocs` na počet procesů `nprocs`, které si přejeme vytvořit následným voláním `m_fork()`. *Dosazovaná* *hodnota nesmí být větší než konstanta MAXPROC (počet procesorů v konfiguraci) a rovněž nesmí být větší než počet dostupných procesorů* (viz dále `cpus_online()`) minus 1. Vrací 0 jako příznak úspěchu nebo -1 jako neúspěch.

`void m_fork (void func(), seznam_argumentů_pro_func);`

Voláním se vytvoří procesy (nebo využijí už vytvořené předchozím voláním `m_fork()`) s programem činnosti `func()` a přenesou se do nich argumenty. Pokud předem nebylo určeno (voláním `m_set_procs()`), kolik procesů se má vytvořit, vytvoří se jich `cpus_online/2`. Za vyvoláním `m_fork()` (implicitní synchronizace) pokračuje výpočet všech procesů (tj. otce i dětí) podle podprogramu `func()`, až všechny doběhnou do konce podprogramu. Procesy dětí se ale automaticky nelikvidují (s ohledem na časově náročné zakládání) – leda explicitním voláním `m_kill()` z otcovského procesu, ale čekají *v programové čekací smyčce* na další práci. Doporučovaný postup je „uspat" procesy-děti voláním `m_park_procs()` (uvolní se procesory) a před opakovaným voláním `m_fork()` (pochopitelně s jiným programem činnosti `func()`) je pouze „vzbudit" voláním `m_rele_procs()` s relativně malou časovou režií. Využívá se unixovská idea s přepínáním programu procesu (viz např. realizace *shellu*, kap.3).

`int m_get_myid(void);`

Vrací identifikační číslo (id) procesu

0 ... do hlavního procesu (otce),
1 ... do prvního procesu-dítěte ... atd.

`void m_lock(void); void m_unlock();`

Voláním se zamyká a odemyká globální zámek – využití pro sdílené redukční a uzamykané proměnné – viz metodika paralelizace cyklů.

`int m_next(void);`

Nedělitelně inkrementuje globální čítač a vrací jeho (novou) hodnotu. Volání `m_fork()` nebo `sync()` čítač nuluje, první následující volání `m_next()` vrací 1 atd.

`void m_park_procs(); void m_rele_procs();`

Využívají se pouze v hlavním procesu, tj. *nikoliv uvnitř podprogramu paralelní činnosti* `func()`. Suspendují a znovu aktivují (do čekací smyčky) procesy–děti vytvořené v předchozím volání `m_fork()`.

`void m_kill_procs();`

Likviduje všechny procesy–děti vytvořené posledním voláním `m_fork()`.

`void m_single(); void m_multi()`

Používají se pouze uvnitř podprogramu paralelní činnosti `func()`, a to pouze v uvedeném pořadí. Volání `m_single()` představuje bariéru, přes kterou projde pouze otcovský proces (proces s *id = 0*). Procesy–děti čekají na bariéře a jsou „propuštěny" dále až proces–otec zavolá `m_multi()`. Procesy–děti „přeskočí" operace mezi `m_single()` a `m_multi()`. Je zřejmé, že nelze připustit „vhnízděné" použití uvedené dvojice operací, rovněž je nelze beztrestně kombinovat s dvojicí `m_lock()`, `m_unlock()` s ohledem na riziko zablokování výpočtu.

`void m_sync(void);`

Volání této funkce označuje ve všech kooperujících procesech místo, kde „se sejdou a počkají na sebe" a teprve potom mohou pokračovat dále; toto volání nesmí být v kritické sekci (tj. mezi `m_lock()` a `m_unlock()` (vede na zablokování) a mezi `m_single()` a `m_multi()`. Nuluje se globální čítač. Místo čekání se označuje jako bariéra (angl. *barrier*) – zde jen jedna (globální nepojmenovaná), podobně jako jeden zámek a jeden čítač.

Pokusíme se o shrnutí:

V obecném modelu výpočtu jsou prostřídány sekvenční části (realizuje „hlavní" proces) a paralelně vykonatelná větvení realizovaná voláním `m_fork()`. Pro *i*-tou paralelní část se vytvoří podprogram `func_i()`, podle kterého probíhají všechny paralelní „větve". Pokud možno použijeme dynamické plánování.

V programu základního procesu (hlavní program):

- nejprve zjistíme, kolik je k dispozici procesorů voláním `cpus_online()`,

- prvním voláním `m_fork()` založíme tolik procesů, kolik je k dispozici procesorů minus 1 (jeden ponecháme systému),
- za `m_fork()` „parkujeme“ procesy-děti voláním `m_park_procs()`,
- před dalším voláním `m_fork()` „vzbudíme“ procesy–děti voláním funkce `m_rele_procs()`.

V podprogramech pro paralelní větve (tj. procesy–děti):

- v případě potřeby chráníme sdílená data voláním `m_lock(), m_unlock()`,
- v případě potřeby zajistíme sekvenční vykonávání části operací hlavním procesem s využitím `m_single()` a `m_multi()`.

I/O operace

Dosud jsme se nezmiňovali o problematice I/O operací v paralelním programu. Potíže způsobuje zejména:

- současné využití jednoho I/O zdroje (periferie, soubor) několika procesy,
- možné zdržení (neproduktivní čekání) procesů na ukončení I/O operace.

Pro uživatele PPL a operačního systému DYNIX lze poskytnout následující doporučení k překlenutí výše uvedených komplikací:

- Pokud možno vykonávat I/O operace v základním procesu (typicky čtení před paralelním větvením a zápisy za větvením). Uvnitř paralelního výpočtu lze vynutit samostatný výpočet základního procesu voláním `m_single()`, provést I/O a pomocí `m_multi()` opět obnovit paralelní výpočet.
- Pokud by uvedený přístup znamenal přílišnou degradaci urychlení paralelního výpočtu, lze v prostředí několika paralelních procesů využít ještě funkční paralelismus, vyčlenit jeden proces jako I/O server, s kterým ostatní komunikují zasíláním zpráv (pokud možno asynchronně přes náležitě dimenzovanou vyrovnávací paměť).
- Pokud využívají paralelní procesy soubory, snažíme se v první fázi, aby měl každý proces svůj soubor (soubory).

- Jestliže musí zapisovat několik procesů do téhohož sekvenčního souboru, lze pro bezkonfliktní zápis (vždy na konec) využít volbu `FAPPEND` v systémové operaci `fcntl()`.
- Jestliže několik procesů využívá tentýž soubor s přímým přístupem, je třeba volání systému `lseek()` a následující `read()` nebo `write()` zajistit jako kritickou sekci.

4.3.2 Data-partitioning routines

Tato část PPL dává k dispozici mj. prostředky, které umožňují práci s libovolným počtem paměťových zámků. Zámek se definuje například:

```
shared slock_t zamek_1;
```

a pro operace nad ním jdou použít funkce:

`s_init_lock(slock_t *p_zamek);`

Inicializuje uvedený zámek na „otevřeno“.

`s_lock(slock_t *p_zamek);`

Zamyká uvedený zámek (uvnitř je programová čekací smyčka).

`s_nulock(slock_t *p_zamek);`

Odemyká uvedený zámek.

Dále je možné pracovat s libovolným počtem synchronizačních bodů (bariér). Bariéra se zavede takto:

```
shared sbarrier_t bariera_1;
```

a jdou s ní provádět následující operace:

`s_init_barrier(sbarrier_t *p_bariera, int nprocs);`

Inicializace bariéry je provedena pro souběh `nprocs` procesů.

`s_wait_barrier(sbarrier_t *p_bariera);`

Procesy čekají v programové smyčce, dokud se jich „nesejde“ `nprocs`. Potom (všechny současně) pokračují ve výpočtu. Jednou inicializovaná bariéra se může využít vícekrát bez opakování inicializace. Nová inicializace nesmí být provedena, pokud na bariéře čekají nějaké procesy.

Dále je možné zjistit počet procesorů v konfiguraci voláním funkce:

```
int cpus_online(void);
```

4.3.3 Prostředky pro realizaci modelu MPMD

Všechny dosud uvedené funkce poskytované knihovnou PPL jsou vhodné i pro paralelní dekompozici algoritmu podle funkce. Paralelní procesy potřebné pro realizaci jednotlivých (zde různých) činností lze opět založit voláním `m_fork()`. Uvnitř společného programu činnosti procesů (označován jako `func()`, parametr v `m_fork()`) se využije přepínač podle identifikačního čísla procesu a každý proces vykonává specifickou činnost popsanou sekvencí příkazů v příslušné pozici přepínače [5]. K synchronizaci procesů lze využít paměťové zámky nebo bariéry, informační výměny mezi procesy je možné realizovat *sdílením dat*, *kritické sekce* lze realizovat pomocí *paměťových zámků.*

Je rovněž možné využít standardní operace **Unix**u pro práci s procesy (viz dříve kap. 3), které jsou implementovány i v DYNIXu pro fyzicky paralelní procesy. Dále je možné využít i další, dosud nezmiňované prostředky. DYNIX implementuje volání jádra z AT&T **Unix**u System V , jmenovitě:

- `semop(), semget(), semctl(),`
  pro práci se semafory,

- `msgsnd(), msgrcv(), msget(), msgctl(),`
  pro komunikaci zasíláním zpráv mezi procesy (*message queues*).

DYNIX dále implementuje prostředky IPC (Interprocess Communication subsystem) zavedené v **Unix**u 4.2bsd. Jedná se o jednosměrné „potrubí“ (angl. *pipe*) mezi procesy, zápis do potrubí se provede voláním `write()`, čtení pomocí `read()`. Dále jsou to prostředky pro obousměrné komunikační kanály mezi procesy a prostředky pro vysílání (*broadcast*) paketů dat skupině spolupracujících procesů. Využití těchto prostředků ale není

[5] Jsme ale omezeni počtem dostupných procesorů (víc procesů nejde pomocí `m_fork()` založit) a využití procesorů může být velmi nízké (např. jedna dlouhá činnost a ostatní krátké), protože všechny procesy čekají na ukončení `m_fork()` v programové čekací smyčce.

typické v počítači se sdílenou pamětí, kde lze sdílením dat obvykle implementovat interakci procesů s větší efektivností [6].

Knihovna PPL je vhodná i pro paralelní dekompozici způsobem *proudového zpracování* dat (dříve označeno jako model MPSD). Postupujeme zhruba podle následující šablony:

- Vytvoří se procesy pro jednotlivé operace nad proudem dat. Pro menší počet procesů lze využít `m_fork()`, činnosti procesů budou různé (přepínač ve `func()`).
- Každý zúčastněný proces vykoná příslušnou operaci nad prvkem datového proudu, zapíše výsledek do sdílené paměti a případně „uvědomí“ následující proces, který prvek převezme.
- Ukončení činnosti může být provedeno například pro vybranou „koncovou“ hodnotu prvku datového proudu.

Je pochopitelně třeba zajistit synchronizaci činnosti procesů při „předávání“ dat. Synchronizace je nutná s ohledem na různou dobu trvání operace, kterou procesy vykonávají.

Rovněž lze bez větších potíží realizovat výpočetní model *processor farm* (viz dále v podkapitole o PVM).

4.3.4 Příklady použití PPL

Příklad 4.6

Násobení matic s využitím statického plánování.

jeden díl práce = výpočet několika řádek výsledné matice

```
/* PPR1X.C - příklad na násobeni dvou matic. */
#include<stdio.h>
#include <parallel/microtask.h>
#define  SIZE  10              /* rozměr matic */

/* sdílená data */
```

[6] Z hlediska přenositelnosti paralelního programu je naopak lepší se orientovat na komunikaci procesů zasíláním zpráv.

```
shared float  a[SIZE][SIZE];
shared float  b[SIZE][SIZE];
shared float  c[SIZE][SIZE];

/* funkce pro paralelní inicializaci matic: */
void init_mat(float a[][SIZE], float b[][SIZE]) {
  int  i,j, nprocs;
  nprocs = m_get_numprocs();
  m_lock();      /* ukázka I/O - kritická sekce */
    printf(" init_mat - proces c. %d.\n",m_get_myid());
  m_unlock();
  for (i=m_get_myid(); i<SIZE; i+=nprocs) {
    for (j=0; j<SIZE; ++j) {
      a[i][j] = (float)i+j;  b[i][j] = (float)i-j;
    }
  }
}

/* funkce pro paralelní násobení matic: */
void nas_mat (float a[][SIZE], float b[][SIZE],
              float c[][SIZE]) {
  int i,j,k,nprocs;
  nprocs = m_get_numprocs();
  m_lock();
    printf(" nas_mat - proces c. %d.\n",m_get_myid());
  m_unlock();
  for (i=m_get_myid(); i<SIZE; i+=nprocs) {
    for (j=0; j<SIZE; ++j)
      for (k=0; k<SIZE; ++k)
        c[i][k] += a[i][j]*b[j][k];
  }
}

/* funkce pro tisk výsledků: */
void tisk_mat (float a[][SIZE], float b[][SIZE],
 float c[][SIZE])
{ /* sekvenčně provedený tisk výsledné matice */};
```

```
/* hlavní program */
void main(void) {
  int  nprocs;            /* počet paralelních procesů */
  printf("Zadej pocet procesu: ");   /* sekvenční část */
  scanf("%d",&nprocs);
  m_set_procs(nprocs);    /* nastavení počtu procesů */
  printf("Fork - init_mat().\n");
  m_fork(init_mat, a, b, c);  /* paralelní inicializace */
  m_park_procs();         /* sekvenční část */
  printf("Vlastni nasobeni \n");
  m_rele_procs();         /* "vzbuzení" dětí */
  m_fork(nas_mat, a, b, c);   /* paralelní násobení */
  m_kill_procs();         /* likvidace dětí */
  tisk_mat(a, b, c);      /* tisk výsledků */
}
```

Poznámka:

Dynamický způsob přidělování práce procesům by v tomto případě vypadal tak, že jedním dílem práce by byl výpočet jedné řádky výsledné matice. Procesy by soupeřily o získání první dosud nezpracované řádky výsledku. Cykl v `nas_mat()` by mohl mít tuto formu:

```
  while ((i = m_next() - 1) < SIZE) {
    for (j=0; j<SIZE; ++j)
      for (k=0; k<SIZE; ++k)
        c[i][k] += a[i][j]*b[j][k];
  }
```

Příklad 4.8

Prohledávání tabulky záznamů s využitím dynamického plánování.

Záznamy jsou typu {`jmémo_města, X_souř, Y_souř`}. Cílem je získat město s minimální vzdáleností od daného bodu.

Jeden díl práce je prohlédnutí jednoho záznamu.

```
/* PPR2.C - příklad na prohledávání tabulky */
#include <stdio.h>
#include <math.h>
#include <parallel/microtask.h>

#define  N_MEST   10   /* počet měst */
#define  BITE     1    /* kus práce pro "hungry puppy" */
   /*  hladový proces => představa, že procesy při dyn.
       způsobu plánování "užírají" zbývající práci     */

/* globální sdílená data - definice */
shared float nejblize; /* vzdálenost k~nejbližšímu městu */
shared int   index;    /* index nejbližšího města        */

struct location {char *name;  float x, y;};

shared struct location mesta[N_MEST]  = {
    { "PLZEN",    0.  , 100.  },
    { "PRAHA",    500., -550. },
    { "KLATOVY",  150., 100.  },
    /* atd. */
};

shared struct location lochotin = {
  "LOCHOTIN", 0., 0. };

/* Podprogram pro paralelní hledání */
void find_dis(mesta)
  struct location mesta[];{
  int i, base, top;  /* index, počát. a~konc. hodnota */
  float xsqdis, ysqdis, dist;
  while ((base = BITE*(m_next() - 1)) < N_MEST) {
    top = base + BITE;      /* práce pro proces */
    if (top >= N_MEST)
      top = N_MEST - 1;
    for (i~= base; i~<= top; i++) {
      xsqdis = pow(fabs(lochotin.x - mesta[i].x), 2.);
```

```
      ysqdis = pow(fabs(lochotin.y - mesta[i].y_, 2.);
      dist   = sqrt(xsqdis, ysqdis);
      /* změna sdílených proměnných - kritická sekce */
      m_lock();                         /* začátek */
        if (dist < nejblize) {
          index = i;  nejblize = dist;
        }
      unlock();                         /* konec */
    }
  }
}

/* Hlavní program */
main () {
  void get_cities(), find_dis(), m_fork();
  shortest = 999999999;
  m_fork(find_dis, mesta);
    /* využije se polovina dostupných procesorů */
  printf("%s je nejblize k Lochotínu", mesta[index].name);
}
```

4.4 Paralelizující kompilátory

V předchozím textu byla popsána tvorba programů pro prostředí SMP pomocí jednoduchých rozšíření obvyklého (sekvenčního) programovacího jazyka a poskytnutím knihovny procedur pro řízení paralelního běhu více menších úkolů v rámci obvyklého rodičovského procesu. Jde o možnost využívající prostředky systému na trochu nižší úrovni, než je u aplikačního programování obvyklé. Z toho vyplývá delší, obtížnější vývoj programů a nutnost provedení mnohdy značných zásahů při přenosu již existujících programů. Všechny tyto okolnosti nakonec neblaze ovlivňují konečnou přenositelnost výsledných programů. I přes následnou standardizaci v této oblasti (POSIX 1003.1c pthreads interface) je tento způsob pro aplikačního programátora dost obtížný.

4.4.1 Využití paralelizmu v SMP

V této části se budeme věnovat další možnosti pro vývoj programů pro architekturu SMP, kterou představuje plně nebo částečně automatická paralelizace již existujících programů pomocí preprocesorů a kompilátorů detekujících paralelizmus. Paralelizující kompilátor provede nejdříve analýzu zdrojového programu v obvyklém programovacím jazyku. Tato analýza je obvykle zaměřena na detekci možností paralelizace výpočtu paralelním běhu vláken procesu. Podle výsledků analýzy provede preprocesor následně úpravy zdrojového programu zajišťující vlastní využití paralelizmu. Pro konečný překlad se pak použije běžný kompilátor dodaný se systémem.

Automatická detekce paralelizmu je založena převážně na lokální paralelizaci cyklů s velkým počtem iterací. Při tomto tzv. jemnozrnném paralelizmu, je každému zúčastněnému procesoru svěřeno k provedení několik iterací cyklu. S ohledem na lokalitu všech datových referencí v architektuře SMP, rozhoduje o možnosti paralelizace cyklu pouze bezkonfliktnost přístupu k proměnným a výsledná efektivita. Tento způsob umožňuje snažší a přenositelnější využití nových možností SMP. Pro jeho náročnost však lze někdy pochybovat o jeho efektivitě. Ideálně by měl paralelizující preprocesor najít a následně vyjádřit paralelní algoritmus podle mnohdy velmi subjektivního popisu problému některým obvyklým programovacím jazykem. Vývoj těchto prostředků je provázen dost skeptickými názory na možnost automatické paralelizace vůbec, nicméně jde o předmět zájmu velmi aktivního výzkumu. Není divu, že problém automatické detekce zejména globálního paralelizmu byl zařazen do seznamu tzv. „Grand Challenges for HPC“[7] ([Blume94]), tedy do skupiny problémů, k jejichž vyřešení jsou zapotřebí především metody dovolující využití vysokého stupně paralelizace.

4.4.2 Standard ANSI X3H5

Díky historickému vývoji pokročily metody automatické paralelizace pro architektury SMP nejdále u programovacího jazyka Fortran, pro jazyk C jsou obdobné prostředky vyvíjeny s menším časovým odstupem. Zhruba asi před deseti lety byli totiž tvůrci fortranských programů, mnohdy zároveň vědci z nepočítačových oborů, prvními, kteří netrpělivě čekali na

[7] z angl. High Performance Computing, tj. vysoce intenzívní výpočty

možnost využití tehdy zpřístupněných paralelních počítačů. Samotná implementace SMP do jádra OS z hlediska vývoje paralelních programů nic nového nepřinesla a bylo tedy nutné co nejrychleji najít doplňující vývojové prostředky. Vzhledem k velkému fondu převážně numericky zaměřených programů v jazyce Fortran byly nejdříve vyvinuty a standardizovány právě prostředky na automatickou paralelizaci těchto programů. Díky práci skupiny Parallel Computing Forum (PCF) byla standardizována rozšíření jazyka Fortran pro architektury s globálně sdílenou pamětí, která se stala později základem pro standard ANSI X3H5. Je dlužno dodat, že díky výhradnímu zaměření na sdílenou paměť není již tento standard předmětem dalšího zdokonalování.

4.4.3 Preprocesor KAP Fortran Optimizer

Mezi typické a nejrozšířenější softwarové prostředky pro automatickou paralelizaci patří optimalizující a paralelizující preprocesor `KAP Fortran Optimizer` (dále jen **KAPF**) od společnosti Kuck and Associates, Incorporated[8]. Jde o produkt, který je dopracován pro využití na téměř všech běžně dostupných počítačových platformách, na ZČU je nainstalován na počítačích řady Alpha firmy Digital[9] a na pracovních stanicích firmy Silicon Graphics. Obdobně dosažitelný je i produkt KAP C Optimizer, který má, zejména pro absenci obdoby standardu ANSI X3H5 pro jazyk C, poněkud menší spektrum možností. Výsledný efekt paralelizace lze samozřejmě pozorovat pouze na serveru Kirke[10], na ostatních počítačích lze využít pouze skalární optimalizační funkce tohoto preprocesoru.

Z výše uvedeného důvodu se pro větší přehlednost dále zaměříme na implementaci preprocesoru **KAPF** v rámci OS Digital UNIX a na paralelizaci zdrojových programů v jazyce FORTRAN 77 pro architekturu SMP. Vedle toho provádí **KAPF** ještě skalární optimalizaci zdrojového programu vzhledem k vlastnostem cílového mikroprocesoru (většinou typu RISC) a optimalizaci využívání cache–paměti.

V prostředí OS Digital UNIX jsou možnosti SMP realizovány mechanismem vícevláknových (multithread) procesů a zpřístupněny pomocí procedur z knihovny DECthreads vyhovující standardu POSIX. Paralelizaci

[8] KAI, viz `http://www.kai.com`
[9] Viz popis použití v dodatku A.5
[10] 8 x CPU Alpha 21164/300MHz

programu je tedy možné chápat jako dekompozici programu na paralelní úseky a následné zabezpečení paralelního běhu takto zjištěných úseků prostředky knihovny DECthreads. Při obtížnosti daného problému je samozřejmé, že vedle plně automatické paralelizace preprocesorem je nutné umožnit i režim dovolující návody nebo přímé zásahy programátora.

4.4.4 Automatická paralelizace

V tomto případě provede preprocesor **KAPF** restrukturalizaci zdrojového programu postupně v těchto etapách:

1. analýza programu

2. detekce časově náročných cyklů, vhodných pro uplatnění paralelizace

3. konečné rozhodnutí o možnosti paralelizace vybraných cyklů. Je zvažováno, zda paralelní provedení cyklu nedává pro případné datové závislosti jiné výsledky než provedení sériové.

4. změna programu v místech uplatnění paralelizmu, vytvoření a řízení běhu vláken. **KAPF** přesune příkazy v paralelním úseku do samostatné procedury a zajistí správné vyvolání této procedury prostředky podpůrné knihovny. Některé proměnné jsou podle zjištěných závislostí převedeny na lokální proměnné nově vzniklé procedury.

5. zajištění synchronizace vláken na potřebných místech programu

Klíčovým místem pro detekci paralelizmu je analýza datových závislostí. Případný dovolený paralelizmus je implementován zcela automaticky, v případě opačném je o zjištěných závislostech a konfliktech podána zpráva v diagnostickém výpisu.

Pro analýzu možností používá **KAPF** graf datových závislostí, který zachycuje vznik a použití (čtení, zápis) proměnných v rámci paralelizovaného cyklu. Graf datových závislostí je analyzován obvyklými prostředky pro zpracování grafů s ohledem na výskyt cyklů, které mohou znemožňovat paralelizaci. Před konečným rozhodnutím o nemožnosti správné paralelizace je ještě vyzkoušeno, zda nelze závadné cykly závislostí odstranit nějakou invariantní transformací výpočtů.

4.4.5 Řízená automatická paralelizace

Pro nedostatečné informace v čase kompilace a mnohdy z důvodu složité struktury programu nelze některé smyčky zevrubně analyzovat a zařadit do procesu automatické paralelizace. Programátor, seznámený s analyzovaným programem, potřebuje pro zvýšení efektivity výsledného programu dodatečné možnosti jednak pro řízení chodu preprocesoru **KAPF**, jednak pro předávání doplňujících informací.

Programátorem doplňovaná informace je dvojího druhu: jedna slouží k potlačení *předpokládaných* (tj. v čase analýzy a kompilace neprokazatelných) datových závislostí a druhá jen jako doporučení pro proces plně automatické paralelizace. Pokud tedy nebudou během chodu programu zajištěny přislíbené předpoklady, dojde k ovlivnění správnosti programu.

Dodatečné informace je nutné vložit do zdrojového programu před jeho kompilací ve formě pseudopříkazů. Aby nebyla narušena syntaktická správnost doplňovaného programu, mají všechny direktivy formu poznámky podle syntaxe pro jazyk FORTRAN 77: na začátku řádku musí obsahovat znak `c` (nebo `C`). Dalších několik znaků, pro direktivy **KAPF** je to řetězec `*$*`, pak blíže označuje druh pseudopříkazu. Nakonec následuje zápis příkazu[11].

Řídící direktivy

Tyto direktivy slouží k řízení (zastavování nebo spouštění) funkcí preprocesoru a neovlivňují, až na jednu výjimku, správnost programu. Globální uplatnění těchto direktiv lze zajistit při spuštění preprocesoru **KAPF** uvedením vhodných (mnohdy téměř stejnojmenných) přepínačů v příkazové řádce.

Platnost řídících direktiv se vztahuje na celou kompilační jednotku (soubor s několika programovými jednotkami) pokud jsou uvedeny na jejím počátku, při uvedení v programové jednotce je platnost direktiv omezena výskytem dalších direktiv s opačným významem, nejvýše však na rozsah programové jednotky. Lokální význam direktiv překrývá v rámci programové jednotky význam globální. Většinou jde o funkce pro řízení skalární optimalizace, pro řízení paralelizace jsou k dispozici tyto tři

[11] V dalším popisu není dvojice znaků [ ] součástí syntaktických pravidel, nýbrž omezuje volitelnou část zápisu.

možnosti:

`C*$* [NO]CONCURRENTIZE`

Tato direktiva povoluje resp. zastavuje paralelní dekompozici programu. Direktiva slouží především k přesnému vymezení automatické paralelizace ve fázi ladění a k jejímu zamezení v částech programu s převážně krátkými smyčkami. Globálně lze chod preprocesoru ovlivnit přepínači `-concurrentize` resp. `-noconcurentize`.

`C*$* MINCONCURRENT(`n`)`

Direktiva slouží k nastavení práhové hodnoty počtu iterací pro paralelní provádění smyček. Práhová hodnota je určena parametrem n, který musí být celé číslo v rozsahu 0 – 999999. Výsledný zdrojový program je upraven tak, že v případě kdy nelze zjistit počet iterací v čase kompilace, obsahuje sériovou i paralelní verzi smyčky, o způsobu provedení smyčky bude rozhodnuto až za běhu programu podle aktuálního počtu iterací. Přednastavená prahová hodnota je 1000. Globálně lze hodnotu práhu nastavit při spuštění preprocesoru přepínačem `-mc=`n.

`C*$* [NO]ASSERTIONS`

Tyto dvě direktivy mají velmi všeobecnou platnost, neboť v rámci působnosti povolují resp. zastavují platnost všech asertivních direktiv (viz dále) – tím mohou nepřímo ovlivnit správnost výsledného programu. Zápornou variantu direktivy lze globálně nastavit přepínačem `-nodirectives`.

Asertivní direktivy

Tyto direktivy slouží především pro sdělení doprovodných informací, které nejsou známy v době analýzy datových závislostí nebo umožní podrobnější analýzu grafu závislostí během automatické paralelizace.

Je nutné uvést, že možnosti vyplývající z doporučení podle direktiv nemusí být preprocesorem akceptovány. Ze syntaktického hlediska mají stejný tvar jako řídící direktivy, pokud nebude uvedeno omezení platnosti na následující smyčku, platí pro rozsah jejich platnosti stejná pravidla jako pro řídící direktivy.

Pro paralelizaci jsou důležité tyto direktivy:

`C*$* ASSERT [NO] ARGUMENT ALIASING`

Funkce a subroutiny definované v rozsahu platnosti direktivy s `NO` budou vždy volány s různými skutečnými parametry na místě rozdílných formálních parametrů. Rovněž musí být zaručeno, že jako skutečné parametry nebudou použity žádné globální proměnné, tj. proměnné v blocích `COMMON`, které jsou referovány v těle procedury.

`C*$* ASSERT [NO] BOUNDS VIOLATIONS`

Direktiva bez `NO` naznačuje, že indexy polí mohou při běhu programu nabývat hodnot mimo deklarované meze. Takový způsob práce s poli byl mnohdy z úsporných důvodů používán, jindy umožňoval obcházet potíže, vzniklé pro nemožnost dynamické alokace paměti.

`C*$* ASSERT [NO] EQUIVALENCE HAZARD`

Varianta direktivy s `NO` vyjadřuje, že ve smyčkách nejsou používány proměnné z příkazu `EQUIVALENCE`, které při rozdílném jménu odkazují na stejnou adresu paměti.

`C*$* ASSERT [NO] LAST VALUE NEEDED (`*jméno*`)`

Při variantě s `NO` není po ukončení smyček nutné dodržet koncovou hodnotu vyjmenované proměnné, kterou by jinak nabyla při sériovém výpočtu.

`C*$* ASSERT PERMUTATION (`*jméno*`)`

Vyjmenované pole celočíselného typu je prostou funkcí podle indexu, tj. nenabývá stejnou hodnotu pro různé indexy. Tato vlastnost dovoluje paralelizaci při nepřímé adresaci pole, tak jak je naznačeno v příkladu:

```
C*$* ASSERT PERMUTATION (IP)
     DO 100 I = 1, N
       A(IP(I)) = A(IP(I)) + B(I)
100  CONTINUE
```

`C*$* ASSERT [NO] RECURRENCE (`*jméno*`)`

Z grafu závislostí pro následný cykl budou vyjmuty všechny konflikty vzniké přístupem k uvedené proměnné.

`C*$* ASSERT RELATION (`*relace*`)`

Při analýze grafu závislostí následujícího cyklu lze uplatnit pravdivost uvedené relace. Relace vyjadřuje vztah dvou proměnných nebo proměnné a konstanty (konstanta musí být uvedena na pravé straně relace) pomocí relačních operátorů jazyka FORTRAN 77 (`.LE.`, `.GT.` apod.). Příklad:

```
C*$* ASSERT RELATION (M .GE. N)
      DO 100 I = 1, N
        A(I) = A(I + M) + B(I)
100   CONTINUE
```

`C*$* ASSERT NO SYNC`

Direktiva slouží k odstranění nadbytečné synchronizace např. v důsledku rozkladu smyček s větším počtem příkazů v jedné iteraci na více smyček jednodušších.

`C*$* ASSERT CONCURRENT CALL`

Vyjadřuje možnost paralelního provedení procedur v následující smyčce, zejména není nutné přihlížet k předpokládaným datovým závislostem parametrů procedur.

`C*$* ASSERT DO (CONCURRENT)`

Je navrhována paralelizace následující smyčky bez ohledu na předpokládané datové závislosti.

`C*$* ASSERT DO (SERIAL)`

Uvedení direktivy zabrání paralelizaci následující smyčky a případně všech smyček nadřazených.

`C*$* ASSERT DO PREFER (CONCURRENT)`

Direktiva navrhuje v případě možnosti upřednostnit paralelizaci následující smyčky. Slouží jako návrh v případě více možností, při zjištění datových závislostí není direktiva akceptována.

`C*$* ASSERT DO PREFER (SERIAL)`

Uvedením direktivy je navrhováno seriové provedení následující smyčky, posouzení paralelizace případných nadřazených smyček není ovlivněno. Slouží opět spíše k usnadnění při výběru z více možností.

4.4.6 Přímo řízená paralelizace

V tomto případě je ponecháno, na rozdíl od předchozího, vyznačení paralelních úseků a zajištění potřebné synchronizace zcela na rozhodnutí programátora, preprocesor pouze zařídí implementaci paralelizmu zařazením volání potřebných procedur z podpůrné knihovny. Není prováděna analýza závislosti dat ani nejsou použity žádné další prostředky pro vyloučení nesprávného použití paralelizace. Na straně druhé lze provádět paralelizaci na více úrovních než pouze jemnozrnné a při detailní znalosti programu (algoritmu) lze dosáhnout větší účinnosti paralelizace. Přímo řízenou paralelizaci lze uplatnit již ve fázi návrhu a vývoje nového programu.

Pro vyjádření přímo řízeného paralelizmu je preprocesorem **KAPF** akceptována podmnožina již zmíněných rozšíření jazyka FORTRAN 77 podle doporučení ANSI X3H5. Tato rozšíření jsou opět implementována ve formě doplňujících komentářových řádků, tentokrát jsou uvozena řetězcem `C*KAP*`, a neovlivňují tedy použití originálního kompilátoru. Díky standardizaci je zvýšena přenositelnost programů.

Při přímo řízené paralelizaci lze vymezovat tyto úseky: paralelní úsek, paralelní cykl, jednoprocesorovou sekci a sekci kritickou. V těchto úsecích neprovádí preprocesor **KAPF** skalární optimalizaci.

Paralelní úsek

Výpočet v paralelním úseku je prováděn všemi vlákny procesu. Z důvodu časové náročnosti vytváření vláken jsou jednou vytvořená vlákna opětně využívána v dalších paralelních úsecích.

Konstrukt pro vymezení paralelního úseku má tento zápis[12]:

```
C*KAP* PARALLEL REGION
C*KAP*& [IF(logický_výraz)]
C*KAP*& [SHARED(jméno, ...)]
C*KAP*& [LOCAL(jméno, ...)]
        příkazy programu
C*KAP* END PARALLEL REGION
```

Doplňující části `IF`, `SHARED` a `LOCAL` jsou volitelné a určují po řadě podmínku pro paralelní vykonání úseku, seznam proměnných, které budou

[12]Pro názornost jsou použity pokračovací řádky direktivy, u kterých je úvodní řetězec prodloužen o znak &.

zúčastněnými vlákny sdíleny a seznam proměnných, které budou vláknům dostupné v lokálních instancích.

Paralelní cykl

Direktiva pro paralelní cykl uvozuje smyčku, která může být vykonávána po částech různými vlákny. Direktiva má obecně tvar:

C*KAP* PARALLEL DO
C*KAP*& [LAST LOCAL[(*jméno*, ...)]]
C*KAP*& [BLOCKED [(*konstantní_výraz*)]]
    smyčka DO[13]

U proměnných ve volitelném seznamu LAST LOCAL bude v nadřazeném kontextu (vně PARALEL DO) zachována správná hodnota odpovídající sériovému provedení iterací (tj. hodnota těsně před ukončením činnosti vlákna, provádějícího iteraci pro koncovou hodnotu iterované proměnné). Volitelným parametrem BLOCKED lze určit počet iterací, které budou vykonány jedním během vlákna. Pokud není tato část direktivy uvedena, je počet iterací mezi vlákna rozdělen rovnoměrně.

Prostředky pro synchronizaci

V jednoprocesorové sekci vymezené dvojicí:

C*KAP* ONE PROCESSOR SECTION
    příkazy programu
C*KAP* END ONE PROCESSOR SECTION

je aktivní pouze jedno vlákno, ostatní procesory jsou dočasně neaktivní.

V kritické sekci určené dvojicí:

C*KAP* CRITICAL SECTION
    příkazy programu
C*KAP* END CRITICAL SECTION

je paralelní běh aktivních vláken dočasně převeden na sériový.

4.4.7 Některá doporučení pro práci s KAPF

Použití preprocesoru **KAPF** pro paralelizaci složitějších programů není jednoduchou záležitostí a vyžaduje poměrně velkou zkušenost, kterou lze

[13]Tato direktiva není výjimečně ukončena klauzulí END PARALLEL DO.

získat jedině soustavnějším používáním. Zejména nelze dost dobře uvažovat o jednorázovém použití plně automatické paralelizace uplatněné globálně na všechny kompilační jednotky. K úspěšnému použití preprocesoru je, vedle dokonalé znalosti paralelizovaného programu, potřebná i detailní představa o práci preprocesoru, kterou je však nutné získat spíše empiricky, neboť dokumentace od výrobce neposkytuje vhodný „teoretický“ základ (např. popis analýzy datových závislostí je dost nepřesný a nedostatečný). Zběhlost v používání je dále vynucena skutečností, že konzervativním, opatrným přístupem nelze většinou dosáhnout uspokojivé zvýšení výkonnosti, při volnějším přístupu dochází často k narušení správnosti programu a malé změny pak mohou vyvodit velké následky. Ostatně, podle manuálů není úplně korektní ani skalární optimalizace.

Naprosto podstatné je nalezení výpočetně náročných míst původního programu některým prostředkem pro sledování výpočtu (`gprof` apod.) a tato místa nějakým způsobem izolovat, např. umístěním do samostatné procedury. Vedle soustředění úsilí na zdokonalení podstatných úseků dojde i ke zlepšení přehlednosti. Během prací na paralelizaci programu je totiž nutné pracovat s rozsáhlými diagnostickými výpisy, které mohou být pro větší programové jednoky velmi nepřehledné.

Kapitola 5

Programování volně vázaných systémů

Systémová charakteristika volně vázáných multiprocesorových systémů byla uvedena již v kapitole 1. Rekapitulujme pouze, že procesory systému jsou v podstatě samostatné počítače (s architekturou von Neumannova typu) spojené vzájemně rychlými sériovými linkami. Od počítačových sítí se liší zejména geometrií (vzdálenosti procesorů řádu cm nebo desítek cm) jednoduchým komunikačním protokolem a vysokým stupněm integrace řízení (jeden OS, sdílené periferie).

Programovací prostředky pro paralelní programování na aplikační úrovni mohou být opět rozděleny podle dvou základních přístupů:

- Specializovaný programovací jazyk, doplněný nějakým vývojovým prostředím. Tento případ je dále reprezentován programovacím jazykem **occam2** popsaným v podkapitole 5.1.

- Univerzální programovací jazyk využívající paralelní prostředky doplněné formou knihovny externích funkcí. Tento případ je dále rozebírán v podkapitolách 5.2 a 5.3 pro programové prostředí PVM a programový systém počítačů nCUBE.

Teoretické výhody jsou na straně prvního přístupu (zejména možnost využívání strukturovaných paralelních konstruktů, vyšší spolehlivost programů s ohledem na kontroly realizované překladačem, nezávislost na operačním systému ap.).

Praxe ale spíše preferuje druhý případ, pravděpodobně s ohledem na využívání standardních programových prostředků (jazyk **C** a nějaká modifikace **Unix**u) doplněných nějakou nadstavbou pro paralelní programování. Do standardního prostředí (nebo blízkého standardu) se lépe přenáší už hotové programy a různé mocné ladicí prostředky částečně eliminují „nebezpečnost“ využívání primitivních konstruktů pro paralelizaci.

5.1 Programovací jazyk occam2

5.1.1 Charakteristika

Je primárně určen jako programovací prostředek pro sítě *transputerů*. Transputer [a] je (zjednodušeně) rychlý 32-bitový jednočipový mikropočítač s vlastní pamětí, spojený s okolím několika rychlými sériovými linkami.
occam bývá označován jako „assembler“ pro transputery, svojí koncepcí je ale univerzálním programovacím jazykem (nižší úrovně) pro volně vázané architektury. Umožňuje realizovat jemnou *granularitu* paralelního výpočtu (na úrovni příkazů konvenčního programovacího jazyka).

[a] Název *transputer* vznikl kompozicí zkrácených názvů *transistor computer*, což mělo symbolizovat předpokládané masové využití těchto prvků v architekturách počítačů.

Autoři: C. Hoare (Oxford) a D. May (INMOS Ltd.) C. Hoare je jeden z nejvýznačnějších teoretiků Computer Science – viz např. koncept monitorů (*Hoareovy monitory*). Anglická firma INMOS Ltd. vyrábí transputery.

Verze jazyka: occam2 z roku 1982, inovace z roku 1988, literatura: The **occam2** reference manual, Prentice Hall 1988.

Komunikace je *synchronní*, tj. oba účastníci na sebe musí počkat (srovnej s rendez-vous v Adě). Účastníci komunikace se neznají (znají jen jméno kanálu po kterém komunikují – srovnej s jazykem **Ada**). Komunikační protokol může být jednoduchý a není třeba vyrovnávací paměť (angl. *buffer*). **Komunikační kanály** (logické) jsou v jazyce zavedeny jako speciální typ datových objektů (**chan**). Logické kanály v **occam**u jsou jednosměrné (tj. používají se pro přenos zpráv jedním směrem), na rozdíl od *fyzických sériových kanálů* transputeru, které jsou obousměrné (tj. dají se využít jako dva logické kanály).

Datové typy. Jazyk **occam** má jen jednoduché datové typy [1] a pole jednoduchých typů. **occam** má *silnou typovou kontrolu a vůbec nepoužívá ukazatele* (podobné důvody jako v jazyku **Ada**).

[1] Datové struktury jsou *pro přenos po kanálech* nahraženy prostřednictvím tzv. *protokolů*.

Statický jazyk. Všechny prostředky výpočtu je nutno znát už při překladu. *Nepracuje s dynamickou pamětí.*

Vývoj programů. Program napsaný v **occam**u může běžet a být odladěn na jednom procesoru a poté (jen s malými změnami ve zdrojovém textu) převeden do konkrétního prostředí procesorové sítě.

> *Paralelismus* je základní filosofií jazyka. Implicitně se předpokládá výpočet jednotlivých příkazů (elementárních procesů) v nedefinovaném pořadí [a] a *sekvenční provádění příkazů se musí předepsat* (viz dále konstrukt `SEQ`).
>
> [a]Pseudo–paralelní výpočet více procesů na jednom transputeru je podporován technicky (např. speciální registry pro frontu připravených procesů, spec. instrukce). Doba přepnutí kontextu je velmi krátká.

5.1.2 Typy dat, deklarace, operátory

Bez nároků na úplnost uvedeme údaje potřebné k pochopení principů jazyka (zdůrazníme odlišnost od jazyků pascalského typu) a k psaní jednoduchých programů:

- Psaní programu má *pevný formát* (jedna řádka je jeden příkaz), není třeba viditelný oddělovač příkazů.
- Neexistují *blokové závorky*, používá se odsazení (angl. *indentation*) o 2 mezery.
- Rozlišují se malá/velká písmena u *identifikátorů*. Identifikátory mohou obsahovat tečku (obdoba podtržítka - *underline* v **C**).
- *Klíčová slova* se píší velkými písmeny.
- *Komentáře* mají stejný tvar jako v jazyku **Ada** (tj. `--`).
- Při `deklaraci` je nutno uvést před proměnnou její typ, např. `INT x:`

  > *Dvojtečka* připojuje deklaraci k následujícímu procesu, ve kterém je proměnná lokální. Všechny deklarace před procesem (i procedury) jsou zřetězeny dvojtečkou.

- *Standardní datové typy:*

`INT` – celé číslo (*integer*)
`BYTE` – celé číslo v rozsahu 0 - 255
`BOOL` – logická proměnná

- *Implementačně závislé datové typy* (číslo znamená pč. bitů): `INT16`, `INT32`, `INT64`, `REAL32`, `REAL64`.

- *Komunikační kanály* jsou typu `CHAN` nebo `CHAN OF` *typ* .

- *Konstanty* jsou deklarovány jako:
  VAL *typ jméno* IS *hodnota*:
  například:
  ```
  VAL INT hodiny IS 24, minuty IS 60:
  ```
  Typ může být vynechán, pokud je zřejmý z hodnoty – v případě dvojznačnosti lze psát například:
  ```
  VAL pi IS 3.14159 (REAL64):
  ```

- *Proměnné* nejsou před prvním použitím nijak inicializovány a po ukončení procesu mají nedefinovanou hodnotu.

- *Přehled operátorů:*

  aritmetické: `+`, `-`, `*`, `/`, `REM` nebo `\` (zbytek),

  modulo operátory (pro typ `INT`): `PLUS, MINUS, TIMES` (modul $= N/2$, kde $N = 2^n$, kde n je počet bitů pro zobrazení čísel typu `INT`),

  relační: `=, <>, >, <, >=, <=`,

  logické: `AND, OR, NOT`,

  bitové - pro práci nad celými čísly: `/\` (log. součin), `\/` (log. součet), `><` (xor), (negace všech bitů),

  bitové posuvy: `<<`, `>>`

- *Operátory nemají definované priority* a je třeba používat závorky.

- *Zkratky* (angl. *abbreviations*) např.:
  ```
  VAL seconds IS 60*((60*hours) + minutes) :
  ```
  Dvojtečka znamená připojení k následujícímu procesu, `hours` a `minutes` jsou externí proměnné deklarované ve vyšší úrovni (neměly by se v rozsahu viditelnosti `seconds` měnit, tj. `seconds` je ve „svém" procesu konstanta).

- *Typová konverze.* Jazyk **occam** má silnou typovou kontrolu. V případě nutnosti je možné provádět typové konverze, například:

```
INT  number :
BYTE digit :
number := number + (INT digit)
```

- *Konverze typu* `BOOL` :
  `INT TRUE` nebo `BYTE TRUE` je `1`,
  `INT FALSE` nebo `BYTE FALSE` je `0`,
  `BOOL 1` je `TRUE` a `BOOL 0` je `FALSE`.

- *Konverze typu* mezi `INT` a `REAL` – příklady:
  `REAL32 ROUND x` – `x` typu `INT` na `REAL32` se *zaokrouhlením*,
  `REAL32 TRUNC x` – dtto s *oříznutím*,
  `INT ROUND x` – `x` typu `REAL32` na `INT` se *zaokrouhlením*,
  `INT TRUNC x` – dtto s *oříznutím*.

5.1.3 Primitivní procesy

Odpovídají základním příkazům v sekvenčním programovacím jazyce. Příkaz klasického programovacího jazyka (přesněji jeho provedení) je zde chápán jako elementární výpočetní proces schopný paralelní koexistence s jinými procesy.

proces přiřazení

má obvyklou syntaxi, například: `var := 6 + var2`

vstupní proces

má formu: `chan1 ? xvar`

kde `chan1` je jméno kanálu a `xvar` jméno proměnné, do které bude hodnota z kanálu načtena. Použití vstupního procesu zahrnuje *časově neohraničené čekání na příchod zprávy.*

výstupní proces

má formu: `chan2 ! yvar`

Zde `yvar` je jméno proměnné, jejíž hodnota bude po kanálu `chan2` vyslána. Použití výstupního procesu znamená *časově neohraničené čekání* na připravenost příjemce převzít zprávu (tj. hodnotu `yvar`).

V obou případech se jedná o *synchronní komunikaci* ⇒ oba procesy (ten, který posílá, i ten, který čeká na hodnotu proměnné) se musí „dobrovolně sejít“ – tj. počkat jakoukoliv dobu na druhého účastníka komunikace.

Dále je možné deklarovat fiktivní kanál typu `TIMER` (pouze pro čtení), který vrací hodnotu času (typ `INT`), odvozenou od generátoru hodin procesoru [2]. Například:

```
TIMER clock :
clock ? time -- naplní proměnnou time
clock ? AFTER time
```

Poslední operace znamená čekání po stanovenou dobu (zde `time`) na čtení (fiktivní – nic se nepřečte) z kanálu `clock`. Konstrukce `AFTER time` se může použít i v konstruktu `ALT` (viz dále *konstrukty*) jako strážní podmínka *alternativy*.

procesy SKIP a STOP

Prázdné procesy (využitelné například ve stadiu ladění programu), přičemž:

- `SKIP` je odstartován, nic neudělá a *skončí*,
- `STOP` je odstartován, nic neudělá a *neskončí*.

5.1.4 Konstrukty

Umožňují kombinovat primitivní procesy do složitějších procesů, popřípadě jakékoliv procesy do větších celků. Jsou opět chápány jako procesy a při implementaci pro jejich výpočet také skutečně musí být proces vytvořen.

SEQ

Předpisuje sekvenční provádění procesů – složek, například :

```
SEQ
  chan1 ? var1
  var2 := var1 + 1
  chan2 ! var2
```

[2] Tento čas je cyklicky nulován při přetečení rozsahu `INT` – operace s časem se provádí „modulo“ a složitější časová struktura se musí zařídit zvláštním procesem.

Jedná se o rozdíl v přístupu proti jiným popisovaným prostředkům pro paralelní programování. *Implicitně se předpokládá paralelní vykonávání příkazů, sekvence se musí předepsat* !

Program procesu může mít formu procedury (klíčové slovo `PROC`) a tato může mít formální parametry, například:

```
PROC add.one (CHAN chan1, chan2)
  INT var1, var2 :
  SEQ
    chan1 ? var1
    var2 := var1 + 1
    chan2 ! var2
:
```

Dvojtečka připojující proceduru k následujícímu procesu (nebo deklaraci) musí ležet na stejné úrovni jako `PROC`. Proměnné deklarované uvnitř programu procesu jsou lokální a platnost deklarací (a viditelnost proměnných) je určena odsazením.

WHILE

Umožňuje opakovaný výpočet procesu, forma je:
`WHILE` *boolovský výraz*

Jako příklad uvedeme program procesu obsluhujícího jednoprvkovou vyrovnávací paměť [3]:

```
PROC buffer (CHAN buffer.in, buffer.out)
  WHILE TRUE
    INT xvar :
    SEQ
      buffer.in  ? xvar
      buffer.out ! xvar
:
```

PAR

[3] Tečky jsou součástí jmen (jako v **C** podtržítko.

Předepisuje paralelní výpočet všech svých částí (ukončení je provedeno až po doběhnutí nejpomalejšího procesu – obdoba klasického paralelního rozvětvení *fork–join*).

Jako příklad vytvoříme dvoustupňovou vyrovnávací paměť s využitím předchozího programu procesu:

```
CHAN comms, b.in, b.out :
PAR
  buffer (b.in, comms)
  buffer (comms, b.out)
```

Je možné zavést priority procesů předepsáním `PRI PAR`, které jednotlivým procesům (podle pořadí zápisu) přidělí priority. Toto má smysl jen u procesů lokalizovaných fyzicky na jednom procesoru.

Omezení pro procesy–složky `PAR`:

- musí být nezávislé, tj. nemohou sdílet globální proměnné nebo komunikační kanály (hlídáno překladačem),
- mohou si vyměňovat data jen prostřednictvím komunikačních kanálů,
- pracují jen se svými lokálními daty,
- pouze dva procesy–složky `PAR` mohou využívat konkrétní komunikační kanál, jeden pro čtení, druhý pro zápis.

Uvedená omezení umožní lokalizovat procesy–složky `PAR` na různé procesory v síti (viz dále `placed PAR`).

IF

Umožňuje podmíněné provedení jednoho procesu. Jedná se o proces–přepínač sekvenčně testující podmínky a startující proces spojený s první splněnou podmínkou. Není-li splněna žádná podmínka, proces `IF` se zablokuje. Příklad:

```
IF
  var = 1          -- podmínka
    chan1 ! x      -- výstupní proces (primitivní)
```

```
  var = 2
    IF              -- vnořený proces IF (konstrukt)
      x = 6
        chan2 ! x
      TRUE
        chan3 ! x
  TRUE              -- ošetření případu var <> 1,2
    SKIP            -- prázdná akce
```

ALT

Zavádí indeterminismus při vykonávání I/O akcí na kanálech (obdoba `select` v jazyku **Ada**).

Z alternativ následujících za `ALT` se vykoná ta, *která je připravena k vykonání*. Žádná jiná se neprovede. Před každou z alternativ je uveden jako strážní podmínka (*guard* – viz **Ada**) I/O proces, dále může být doplněna dodatečná blokující podmínkou vázaná pomocí operátoru `&`.

Pokud je v `ALT` připraveno k výpočtu více alternativ, je vybrána nějaká *náhodně* (resp. nejde předem určit která, opět jako `select`). Lze ale zavést priority připravených alternativ pomocí `PRI ALT`. Priorita je pak dána textovým pořadím, tj. z několika připravených alternativ se provede ta, která je v textu „nejvýše“. Předvedeme na příkladu:

```
CHAN chan1, chan2 :
INT var :
ALT
  chan1 ? var
    -- text procesu--alternativy  1
  (ext.var < 0) & chan2 ? var
    -- text procesu--alternativy  2
```

Dále uvedeme příklad, který může sloužit jako ukázková „kostra“ typické aplikace, pro které bylo využití transputerů a **occamu** původně zamýšleno.

Proces (lokalizovaný např. na jednom transputeru):

- přebírá nějaká data (zde `xvar`) z kanálu, popř. z několika vstupních proudů (zde jen jeden – `buffer.in`),
- provede na nich nějakou transformaci (zpracování) – zde vynecháno,
- odešle zpracovaná data někam jinam (zde do `buffer.out`),
- toto všechno v nekonečném cyklu (zde ukázka přerušení cyklu).

```
BOOL running :
SEQ
  running := TRUE
  WHILE running
    INT xvar, any :
    ALT
      buffer.in ? xvar     -- podmínka
        buffer.out ! xvar  -- alternativa
      interrupt ? any      -- podmínka
        running := FALSE   -- alternativa
```

5.1.5 Pole

Všechny základní datové typy zavedené v **occam**u (včetně kanálů) mohou být použity jako prvky pole.

Příklady deklarací polí:

```
[4] INT vector:          -- 4-prvkové pole integer
[8][8] BYTE chessboard:  -- 2-rozměrné pole
[3] CHAN interface:      -- 3 kanály
```

Vlastnosti polí:

- Počet dimenzí pole není v **occam**u omezen.
- Rozměry pole musí být vyhodnotitelné při překladu.

- Pole je bráno jako proměnná a tudíž s ním lze provést přiřazení nebo vstupně–výstupní operaci (viz příslušné primitivní procesy). Jinak s polem jako celkem nejdou dělat žádné operace.
- Pole je indexováno od nuly.
- Konkrétní prvek pole se indexuje v syntaxi například:
  `chessboard[0][3]`
- Lze používat konstanty typu pole, například:
  `VAL days IS [1, 2, 3, 4, 5, 6, 7]:`
- Lze používat i tzv. *segmenty* polí, například:

```
[20] INT sklad :
  ... [sklad FROM 8 FOR 6] ...
      -- segment od sklad[8] do sklad[13]
```

- Ke zjednodušování indexových výrazů lze využívat *zkratky*, například:

```
[5] INT cache :
s27 IS [sklad FROM 2 FOR 5] :
s27 := cache
```

5.1.6 Protokoly

Pro každý kanál může být v jeho deklaraci zadán *protokol* (angl. *protocol*), který specifikuje typ dat přenášených přes kanál [4]. Uvedeme několik typických příkladů protokolu:

```
CHAN OF INT::[][3] INT vectors:
```

Přes takto deklarovaný kanál lze přenášet libovolný počet vektorů obsahujících tři prvky typu `INT`. Například zápis dvou vektorů do výše zavedeného kanálu by mohl být proveden takto:

```
vectors ! 2::[[1,1,1], [2,2,2]]
```

[4] Podobně jako se typ uložené informace stanoví v deklaraci proměnné. Přes kanál s nespecifikovaným protokolem se může přenášet proměnná libovolného typu (tj. i pole), což není v souladu s principem silné typové kontroly.

Protokol může být označen jménem a toto jméno lze použít při deklaraci (jako typ) kanálu, například:

```
PROTOCOL STRING IS INT[]::BYTE
CHAN OF STRING file:
```

Kromě protokolu realizujícího přenos daného množství prvků pole *counted array* lze zavést také tzv. *sekvenční* protokol (popřípadě kombinaci obou), například:

```
PROTOCOL COMPLEX IS REAL64; REAL64
CHAN OF COMPLEX complex:
complex ? 1.0; 2.0
```

Další možnosti poskytují protokoly s diskriminantem typu.

5.1.7 Repliky konstruktů

Replika znamená založení několika procesů pracujících podle stejného programu, každý s jinou hodnotou tzv. *indexu* (srovnej `m_fork()` v kombinaci s `m_get_id()` u PPL pro Sequent). Replikovat je možné konstrukty `SEQ, PAR, IF, ALT`. Obecná forma je:

X index `=` *base* `FOR` *count*

kde *X* je název konstruktu, *base* počáteční hodnota indexu a *count* počet kopií. Symbol *index* znamená číslo repliky, lze jej využít zejména k označení dat repliky – viz dále příklady. Na místě *index* může být libovolné jméno (nemusí být deklarováno jinde) za `base` a `count` musí být dosazeny celočíselné konstanty. Dále uvedeme příklady na repliky jednotlivých konstruktů:

SEQ

```
INT var :
SEQ index = 0 FOR 5
   chan1.out ! var + index     -- syntaktická forma, jak
                               -- říci, že každá replika
                               -- pracuje se svou var
```

PAR

```
 [10] CHAN link :              -- tichá pošta
 PAR num = 0 FOR 9
   WHILE TRUE
     INT vzkaz :
     SEQ
       link[num] ? vzkaz
       link[num + 1] ! vzkaz
```

Replikovaný `SEQ` a `PAR` jdou využít jako náhrada počítaného cyklu (angl. *counted loop*), u `SEQ` je dodrženo pořadí, u `PAR` jsou všechny obrátky cyklu počítány najednou – například paralelně realizovaný součet vektorů lze s využitím replikovaného `PAR` zapsat takto:

```
PROC suma( [] INT a, [] INT b, [] INT c)
  PAR i = 0 FOR (SIZE c)
    c[i] := a[i] + b[i]
:
```

IF

```
[5] INT sklad :
  ...
IF                           -- obyčejný IF
  IF polozka = 0 FOR 5       -- vnořený replikovaný IF
    sklad[polozka] = 0       -- podmínka
      sklad[polozka] := 1     -- akce
TRUE                         -- pro případ, že žádná z
  SKIP                       -- podmínek repliky není splněna
```

Provádí se sekvenční testování pěti podmínek a nahrazení první nalezené nuly v poli `sklad` jedničkou.

ALT

U `ALT` je smysl jasný – jednotlivé repliky sledují připravenost alternativ a provádí se jen jedna alternativa (náhodně vybraná z připravených). Jako příklad ukážeme multiplexování zprávy z několika vstupních na jeden výstupní kanál.

```
PROC multiplex( [] CHAN inputs, CHAN output, interrupt)
  INT any, signal :
  BOOL running :
  SEQ
    running := TRUE
    WHILE running
      ALT
        ALT i = 0 FOR (SIZE inputs)
            -- operace SIZE určí velikost pole
          inputs[i] ? signal
            output ! signal
          interrupt ? any
            running := FALSE
:
```

5.1.8 Procedury

Procedury již byly použity v předchozích příkladech. Zde uvedeme doplňující informace:

- *Procedura* je chápána jako *program procesu*, tj. každé její vyvolání představuje samostatný proces (tzv. *instanci* procedury).
- Procedura může mít *libovolný počet formálních parametrů*, vyvolání (přesněji založení instance procesu) se provádí standardním způsobem:

  jméno_procedury (seznam_skutečných_parametrů)

- Parametry jsou přenášeny *pouze jménem*. Pokud není parametr měněn uvnitř procedury, musí být označen jako `VAL`, například:

  ```
  PROC x (VAL INT step)
  ```

  V tomto případě lze při vyvolání dosadit na místo formálního parametru `step` jako aktuální parametr kromě jména proměnné také výraz.

- Pokud je parametrem procedury pole, nezáleží na jeho rozměru a parametr se specifikuje například:

  ```
  PROC determinant ([][] INT matice, INT výsledek)
  ```

- Funkce jsou v **occamu** zavedeny zejména jako zobecnění hodnoty proměnné, například:

  ```
  REAL64 FUNCTION square (VAL REAL64 a) IS a * a:
  ```

  Proces–instance funkce nemůže používat hodnoty nelokálních proměnných a nemůže komunikovat [5].

[5]Tím je zaručena absence tzv. *postranních účinků* (angl. *side effect*) volání funkce.

5.2 PVM - Parallel Virtual Machine

5.2.1 Charakteristika PVM

PVM je v nejobecnějším pohledu *univerzální výpočetní model* nebo jinak řečeno *norma pro paralelní programování* (viz dále *uživatelské rozhraní*), která má dobrou naději stát se mezinárodním standardem. Tato norma je implementována pro paralelní počítače různého typu (tj. z pohledu programátora se jedná o *programovací prostředek*) a jejím dodržením při návrhu a implementaci aplikace lze zajistit dobrou přenositelnost vytvořeného programu. Označení PVM se ovšem využívá i pro konkrétní emulovaný (virtuální) multiprocesor – „spuštění PVM“ znamená start procesů–démonů PVMD na vybrané množině (viz dále *hostfile*) počítačů, tj. vytvoření virtuálního stroje připraveného řešit aplikaci uživatele, který jej vytvořil.

Dále budeme pod pojmem PVM rozumět programovací nástroj **PVM3**,[6] určený zejména k využití v heterogenní síti konvenčních pracovních stanic pracujících pod operačním systémem **Unix**. Prostřednictvím PVM jsou tyto stanice *softwarově integrovány do jednoho paralelního počítače* - multiprocesoru s distribuovanou pamětí [7]. PVM umožňuje realizaci všech dříve zmiňovaných modelů paralelní dekompozice (viz kap 1). Jako způsob interakce procesů je využívána asynchronní komunikace s přímým adresováním procesů.

Klíčovou částí prostředí PVM je proces PVMD běžící na pozadí (tj. jako *démon* v terminologii **Unix**u) na každém počítači, který je zařazen do virtuálního multiprocesoru vytvořeného pro konkrétního uživatele. Počítače použité v konfiguraci virtuálního multiprocesoru jsou v rámci PVM *identifikovány svým síťovým jménem.* PVMD je tedy programová vrstva, která *na konkrétním stroji pro konkrétního uživatele* realizuje jednotlivé komunikační a synchronizační funkce [8], které vytváří *aplikační rozhraní.* Toto rozhraní je k dispozici ve formě knihoven využitelných v programovacích jazycích **C** nebo **FORTRAN**.

Vytvoření virtuálního multiprocesoru se provede startem démonů PV-

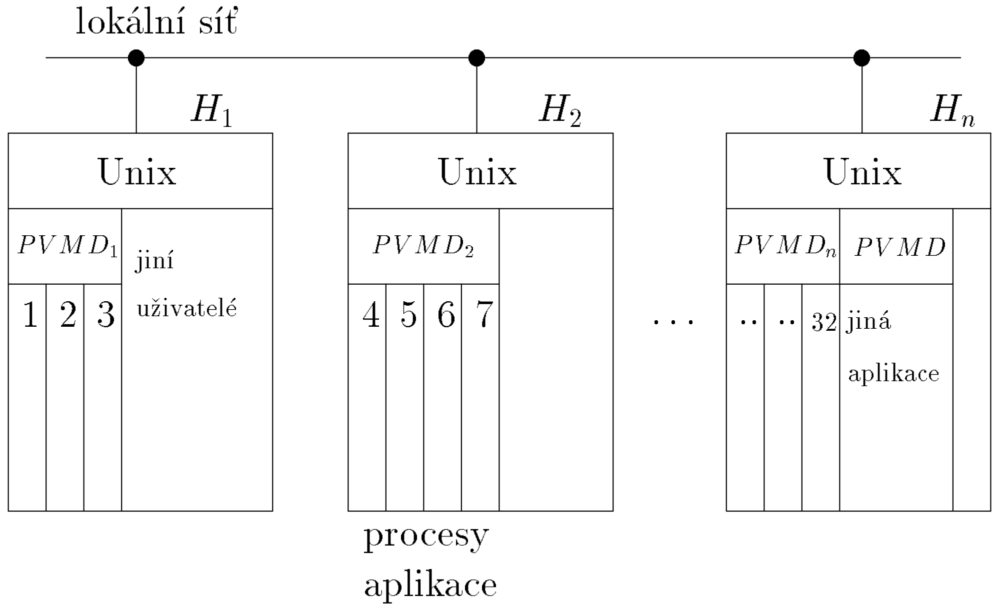
[6] Program **PVM3** byl vytvořen v USA jako public–domain produkt, který je volně dostupný na Internetu. Základní literaturou je PVM 3 User's Guide [PVM94]

[7] Existují také implementace pro symetrické multiprocesory nebo clustery sym. multiprocesorů - např. Digital Alpha, kde je zasílání zpráv mezi procesy alokovanými na témže stroji efektivně emulováno prostřednictvím sdílené paměti.

[8] K implementaci se využívají BSD *sockety* z OS **Unix**.

MD na každém využívaném počítači a navázáním komunikace mezi těmito démony [9]. PVMD, který byl odstartován jako první (tj. „ručně“ - příkazem `pvmd` z klávesnice) je vždy nadřízený a jeho PVMD pak provede start ostatních (podřízených) PVMD pomocí síťových služeb **Unix**u (`rsh`, `rexec`).

Na takto připraveném virtuálním paralelním počítači lze spustit aplikační program, jehož běh je členěn na spolupracující paralelní procesy. Jedná se o standardní unixovské procesy. Na jednom stroji (dále též *host*, *počítač*) může být alokováno několik procesů počítané aplikace [10]. Alokaci procesů na jednotlivé stroje lze provést buď explicitním přiřazením nebo ponechat „na vůli“ PVM (viz dále `spawn()`). Představu vytvořeného virtuálního stroje usnadní obrázek 5.4. Symboly H_1, H_2, H_n jsou označeny stroje (*host*) použité v konfiguraci.



Obr. 5.1: Virtuální multiprocesor PVM

[9] *Pro každého uživatele se vytváří jiný virtuální multiprocesor* (s jinou – uživatelem předepsanou – konfigurací), takže démonů PVMD může v téže době běžet na jednom stroji několik.

[10] Je tedy možné i s (relativně) malým počtem fyzických stojů v konfiguraci emulovat výpočet na velkém počtu *logických* strojů nějak (např. v 2D mřížce) uspořádaných, nebo jinak řečeno – pod PVM lze psát *scalable* aplikace nezávisle na konkrétním počtu strojů dostupných v lokální síti. V tomto případě každý aplikační proces zároveň emuluje procesor – prvek masivně paralelního systému (na kterém by běžel, kdyby byl k dispozici).

5.2.2 Instalace PVM a vytvoření aplikace

Instalace PVM

Zdrojový text PVM je možné vyžádat via e-mail zasláním zprávy s obsahem `send index from pvm3` na adresu `netlib@ornl.gov`. Uživatel lokalizuje získaný soubor ve svém adresáři `$HOME/pvm3` [11] a provede jeho dekódování a rozvinutí do příslušné adresářové struktury (`tar`). Kromě zdrojových souborů programů jsou k dispozici i soubory typu *makefile* pro množinu podporovaných počítačových architektur. Proto stačí dále v instalačním adresáři (označíme jako `PVM_ROOT` - viz dále proměnné prostředí) provést příkaz `make`, načež je automaticky rozpoznána architektura hostitelského počítače a jsou vytvořeny knihovny a spustitelné soubory (překlad zdrojových textů) do podadresáře `PVM_ROOT/lib/PVM_ARCH`, kde `PVM_ARCH` je symbolické označení architektury (například `SUN4` pro Sun 4 SPARC Station nebo `SGI5` pro Silicon Graphics s OS IRIX 5.1).

Běžný uživatel PVM výše popsanou instalační fázi obvykle nerealizuje, musí však před zahájením vývoje své aplikace provést akce, které jsou blíže popsány v dodatku A.4.

Konzola PVM

„Manuální ovládání“ virtuálního multiprocesoru umožňuje proces označovaný jako *konzola PVM* (angl. *console*). Tento proces je po provedení instalace PVM možné jednoduše spustit příkazem `pvm`. Běžný postup použití konzoly ke spuštění a ovládání paralelního výpočtu je zhruba tento:

- Příkazem `add` *host_name*, kde *host_name* je síťové jméno stroje (nebo více jmen) se přidává jmenovaný stroj(e) do konfigurace virtuálního multiprocesoru.
- Příkazem `spawn` se spouští výpočet připravené aplikace.
- Stav výpočtu je možné zjistit zejména pomocí příkazů:

  `conf` vypíše konfiguraci virtuálního multiprocesoru (jména a typ strojů, identifikátory procesů PVMD a další údaje),

[11] Pro více uživatelů se PVM instaluje například do adresáře `usr/local/pvm3` a uživatelé si provedou připojení (link) tohoto adresáře.

`mstat` *host_name* vypíše stav specifikovaného stroje,

`pstat` *process_id* vypíše stav specifikovaného procesu,

`ps -a` vypíše všechny procesy běžící aplikace, jejich alokaci (na stroj) a identifikátory.

- Ovládat probíhající výpočet je možné (pokud je pro to aplikace připravena) zasíláním signálů procesům pomocí příkazu `sig` *signal_num process_id*. Dále může být použit příkaz `kill` *process_id* k ukončení libovolného procesu aplikace.
- Výpočet konzoly se končí příkazem `quit` (procesy PVMD a aplikace běží dál). Konzolu je pak možné znovu spustit (i současně na více strojích) k dalšímu monitorování aplikace. Výpočet PVM se z konzoly dá ukončit příkazem `halt`, tímto příkazem jsou kromě konzoly likvidovány též běžící procesy PVMD a procesy aplikace.

Překlad a sestavení aplikace

Zdrojovým jazykem aplikace může být buď **C**, nebo **Fortran**. Vytvořené programy se přeloží příslušným překladačem a sestaví se s knihovnou `libpvm3.a` (pro jazyk **C**) nebo `libfpvm3.a` (**Fortran**) [12].

V některých speciálních případech je třeba připojit další knihovny. Inspirativní příklady využití PVM lze nalézt v adresáři `pvm3/examples`. V tomto adresáři je zároveň soubor `Readme`, který popisuje jak příklady sestavit a spustit. Jako součást PVM je distribuován univerzální (na typu stroje nezávislý) příkazový soubor `pvm3/lib/aimk`, který po spuštění automaticky rozpozná typ stroje a připojí patřičné knihovny. K sestavení příkladu se použije příkaz: `aimk` *example_name*.

Spuštění aplikace

První možný způsob spuštění připravené aplikace (tj. spustitelné soubory aplikace připravené ve správných `ARCH` adresářích) prostřednictvím konzoly PVM již byl popsán. Druhý možný způsob je startovat na libovolném stroji uvažované konfigurace proces–démon PVMD příkazem

[12] Je třeba zdůraznit, že programy procesů určených k samostatnému běhu na různých strojích se překládají a sestavují do samostatných spustitelných souborů.

`pvmd3` *hostfile* &

kde *hostfile* je jméno textového souboru obsahujícího v nejjednodušším případě [13] pouze síťová jména všech strojů, které si přejeme zapojit do konfigurace virtuálního multiprocesoru [14]. Výsledkem provedení příkazu je „spuštění PVM", přesněji start démonů PVMD na strojích uvedených v *hostfile*. *Voláním příkazu* `pvmd3` *se vytvoří (zatím nečinný) virtuální multiprocesor.* Aplikaci spustíme voláním příkazu *master*, kde *master* je jméno programu realizujícího hlavní (řídicí) proces aplikace [15].

5.2.3 Uživatelské rozhraní

Uživatelské rozhraní PVM je tvořeno množinou funkcí, které lze volat z programů aplikace. Funkce jsou obsaženy v knihovně `libpvm3.a` pro případ volání z **C** nebo `libfpvm3.a` pro volání z programů psaných ve **Fortran**u. Knihovny se vytvoří překladem (pro všechny používané typy počítačů) zdrojových textů v rámci instalace PVM.

Veškeré „objekty", které PVM definuje a s nimiž dále popsané funkce manipulují, jsou v rámci aplikace označeny identifikátory s typem *kladné celé číslo*. Například každý uživatelský proces (v terminologii PVM *task*) dostane při svém spuštění číslo *tid* (z angl. *task identifier*) jednoznačné v rámci celého virtuálního počítače a každý počítač–*host* má jednoznačné číslo *hi_tid* (je součástí *tid*). Vyrovnávací paměti pro komunikaci mají kladné číslo (*bufid*), jednoznačné v rámci jednoho PVMD.

Stručně popíšeme základní funkce z knihovny pro jazyk **C** [16] zhruba v pořadí, ve kterém se použijí při psaní programu. Většina funkcí vrací hodnotu typu `int`. Pokud v popisu funkce není řečeno jinak, *vrácená záporná*

[13]Soubor *hostfile* může kromě jmen podřízených počítačů obsahovat také parametry ovlivňující způsob startu PVMD na jednotlivých počítačích. Pomocí těchto parametrů je možné řešit situace, kdy daný uživatel má na použitých počítačích různá uživatelská jména, nebo není-li povoleno vzdálené přihlášení bez hesla (pomocí souboru `.rhosts`) na daný počítač, nebo pokud nefungují síťové služby `rexec` a `rlogin`. V takovém případě požádá PVMD uživatele o "ruční" přihlášení se na vzdálený počítač, spuštění PVMD na tomto počítači a sdělení odpovědi spuštěného démona.

[14]Pokud není v příslušné síti realizována jednotná registrace uživatelů, musí mít konkrétní uživatel přidělená konta na všech strojích, jejichž jména jsou v *hostfile* uvedena.

[15]Pokud PVM běží, můžeme aplikaci spustit z libovolného stroje aktuální konfigurace.

[16]Knihovna pro **Fortran** se až na několik málo chybějících funkcí liší pouze syntaxí jednotlivých volání.

hodnota znamená chybový kód (a selhání volání funkce). Pro chybové kódy jsou (viz [PVM94]) zavedeny symbolické hodnoty, například `PvmBadParam` znamená chybu při dosazení aktuálního parametru funkce.

Vytváření konfigurace

Zde poznamenáme, že zpravidla se konfigurace vytvoří staticky prostřednictvím seznamu *hostfile* zadaného jako parametr při startu PVM (viz dříve). Dále uvedené funkce umožňují změny konfigurace například ze základního procesu spuštěné aplikace.

`int pvm_addhosts (char **hosts, int nhost, int *infos);`

Přidává do konfigurace (spustí démony) na `nhost` strojích, síťová jména strojů jsou uvedena v poli `hosts`. V poli `infos` se vrací kód stavu jmenovaných strojů, hodnota menší než nula indikuje chybu.

`int pvm_delhosts (char **hosts, int nhost, int *infos);`

Inverzní funkce k `pvm_addhosts`, vyjme z konfigurace stroje, jejichž adresy jsou v poli `hosts`.

Vytváření procesů, skupiny procesů

```
int pvm_spawn (char *task, char **argv, int flag,
              char *where, int ntask, int *tids);
```

Vytvoří a spustí `ntask` procesů podle programu `task` a předá jim parametry uložené v `argv` (obvykle `ntask=1`). Řetězec `task` obsahuje jméno souboru se spustitelným programem, soubor musí být dosažitelný z počítače, na kterém je proces startován, implicitní poloha souboru je `$HOME/pvm3/bin/$PVM_ARCH/task`. Pokud parametr `flag` obsahuje hodnotu `PvmTaskDefault`, je nový proces spuštěn na náhodně vybraném počítači, naopak hodnota `PvmTaskHost` znamená, že parametr `where` obsahuje jméno počítače, na němž má být proces spuštěn [17]. V parametru (poli) `tids` se vrací číselný identifikátor vytvořeného procesu (procesů).

[17] Volbou hodnoty `flag` se zpravidla musí vyřešit konflikt *přenositelnost* versus *výkonnost*. Je zřejmé, že uváženou alokací vybraných procesů na vybrané počítače lze lépe využít (větší) výkon některých strojů v systému a dosáhnout tak větší *urychlení* výpočtu aplikace. Na druhé straně pak ale nemůžeme program přenést na jiný systém (lokální síť, paralelní

`int pvm_joingroup (char* group)`

Volání funkce připojí proces ke skupině `group` a vrátí číslo procesu ve skupině. Prvním voláním funkce s určitým jménem skupiny je tato vytvořena. Je možné zaslat zprávu celé skupině procesů voláním `pvm_bcast()` (viz dále).

`int pvm_lvgroup (char* group)`

Inverzní funkce k `pvm_joingroup()`, tj. vypuštění volajícího procesu ze skupiny. Pokud není volající proces členem skupiny, vrátí volání chybový kód.

Interakce procesů

Interakce procesů v PVM je řešena jako asynchronní komunikace – tj. procesy komunikují prostřednictvím zpráv zasílaných přes vyrovnávací paměti (angl. *buffer*). Součástí vytvoření virtuálního multiprocesoru je shodné nastavení tabulky konfigurace (*host table* – mapování všech *h_tid* na odpovídající IP adresu) ve všech vytvořených PVMD. Bezprostředním důsledkem je možnost přímého adresování každého počítače z každého jiného. PVMD tedy nemusí realizovat žádné směrování (angl. *routing*) zpráv. Přímé adresování [18] lze použít i pro komunikaci mezi procesy, protože *tid* je složeno z *h_tid* a čísla procesu v rámci jeho PVMD, který udržuje mapovací tabulku (*task table*) mezi čísly „svých“ procesů a jejich unixovským *pid*. Díky tomu je možné v aplikaci zvolit libovolnou komunikační strukturu mezi procesy (tj. například programově emulovat libovolnou topologii propojení virtuálního multiprocesoru).

Vyrovnávací paměti se vytváří a ruší dynamicky a jsou v rámci jednoho PVMD unikátně identifikovány celým číslem (*bufid*). Každý proces má jednu *aktivní* vyrovnávací paměť pro vysílání a jednu pro čtení. Standardně se vysílací vyrovnávací paměť vytváří (a aktuální ruší) voláním `pvm_initsend()` a paměť pro čtení zprávy se vytváří (a dosavadní aktivní ruší) voláním funkcí `pvm_recv()` nebo `pvm_nrecv()`. V tomto případě není

počítač) beze změn ve zdrojovém textu programu.

Symbolické hodnoty parametru `flag` jsou číselně kodovány tak, že je lze (pokud to dá smysl) sčítat, například `PvmTaskDefault + PvmTaskDebug + PvmTaskTrace`.

[18] V operacích *send()* a *receive()* se využívá adresa (tj. *tid*) procesu, nikoliv adresa (*bufid*) vyrovnávací paměti – ta je dána implicitně (viz aktivní vyrovnávací paměť).

třeba se o číslování vyrovnávacích pamětí nijak zvlášť starat, funkce pro naplnění a rozbalení zprávy pracují vždy nad aktivní pamětí.

Pro složitější komunikaci lze další vyrovnávací paměti pro vysílání vytvářet voláním `pvm_mkbuf()` a přepínat na aktivní voláním `pvm_setsbuf()`. Přijímací vyrovnávací paměť lze přepínat na aktivní (typicky před použitím `pvm_recv()`) voláním `pvm_setrbuf()`. Nepotřebné paměti lze explicitně rušit voláním `pvm_freebuf()`.

Do aktivní vyrovnávací paměti je nejprve postupně uložena celá informace (viz dále množina funkcí `pvm_pk()`), kterou si přejeme odeslat a poté je zpráva odeslána. Strukturování uložené informace je dáno celočíselným typem (kódem) zprávy, který si volí programátor [19]. Z přijaté zprávy je možné inverzním postupem k ukládání (podle typu zprávy) přečíst uloženou informaci (viz množina funkcí `pvm_upk()`).

Vytvoření a odeslání zprávy

`int pvm_initsend (int encoding);`

Volání ruší starou a inicializuje novou aktivní vysílací vyrovnávací paměť a určuje způsob kódování. Vrací identifikátor *bufid* vytvořené paměti. Je třeba provést znovu před uložením další zprávy do aktivní paměti. Jednou uloženou zprávu lze odeslat vícekrát. Parametr `encoding` může mít hodnoty:

`PvmDataDefault` – data ukládaná do vysílací paměti se kódují (XDR) a jdou tudíž poslat na libovolný stroj v heterogenní síti.

`PvmDataRaw` – data se nekódují, ušetří se čas, ale jdou poslat jen na počítač, který má stejné binární formáty dat.

`PvmDataInPlace` – data se nekopírují (další ušetření času), tudíž ani nekódují, do vysílací paměti se ukládají jen ukazatele a rozměry datových polí (v PVM verze 3.2 neimplementováno).

[19] Tedy ve fázi analýzy aplikace je třeba určit všechny typy zpráv, se kterými budou procesy aplikace pracovat a přidělit jim unikátní celočíselné kódy. *Chaotické číslování typů zpráv je častým zdrojem chyb (zablokování výpočtu - čeká se na jiný typ zprávy než byl odeslán).*

`int pvm_pk`*type* `(...);`

Množina funkcí pro uložení datového prvku s primitivním typem (nebo pole prvků primitivního typu) do *aktivní* vysílací vyrovnávací paměti. Typ dat (například `int`, `double`) [20] je v konkrétní verzi funkce dosazen na místo *type*. *Naplnění vyrovnávací paměti složitěji strukturovanou zprávou se provede postupným voláním funkcí* `pvm_pk`*xxx*`()` *příslušných k typu položek ukládané struktury* [21].

Například funkce pro uložení pole prvků typu `int` do vysílací vyrovnávací paměti má prototyp:

`int pvm_pkint(int* ip, int nitem, int stride);`

Zde *ip* je ukazatel na první prvek pole (jako akt. parametr lze v **C** dosadit jméno pole), `nitem` je počet prvků pole, které se kopírují do zprávy a parametr `stride` umožňuje „krokovat" v poli (například pro dosazenou hodnotu `2` se ukládají jen prvky pole s indexem `0`, `2`, `4` atd.).

`int pvm_send (int tid, int msgtag);`

Odeslání obsahu *aktivní* vysílací vyrovnávací paměti procesu–příjemci `tid` s označením (kódem typu) zprávy `msgtag`.

`int pvm_mcast( int *tids, int ntask, int msgtag);`

Odeslání obsahu *aktivní* vysílací vyrovnávací paměti s kódem `msgtag` skupině procesů, jejichž *tid* jsou uloženy v poli `tids`.

[20] Pro typ `char` je přípona `byte`, pro komplexní čísla v základní a dvojité přesnosti jsou přípony `cplx` a `dcplx`.

[21] Časově úspornější vyslání dat bez postupného „balení" položek lze realizovat pro jednorozměrné pole prvků s primitivním typem (zavedena symb. jména typů - např. `PVM_INT`) pomocí funkcí `pvm_psend()` a `pvm_precv()`. Při volání se udává ukazatel na první prvek a počet prvků, neprovádí se kopírování do vyrovnávací paměti, neovlivňuje se aktivní vysílací paměť nastavená `pvm_initsend()`. Datovou strukturu s položkami různého typu lze úsporně odeslat v **C** stylu následujícím způsobem:

- Zjistí se rozměr struktury *kolik_byte* pomocí `sizeof()`.
- Zpráva se odesílá jako pole bytů voláním: `pvm_psend (`*tid*`, `*msg_typ*`, `*poin_odkud*`, `*kolik_byte*`, PVM_BYTE);`

Tento způsob ovšem není zcela korektní z hlediska přenositelnosti programu, protože na cílovém počítači (potenciálně jiný typ) může být jiná binární reprezentace dat.

`int pvm_bcast(char* group, int msgtag)`

Odeslání zprávy – obsahu *aktivní* vysílací vyrovnávací paměti s kódem `msgtag` skupině procesů se jménem `group`.

Příjem a čtení zprávy

`int pvm_probe (int tid, int msgtag);`

Test přítomnosti zprávy. Volání vrací číslo vyrovnávací paměti *bufid* je–li k dispozici zpráva s typem `msgtag` od procesu–odesilatele s identifikátorem `tid`. Není–li k dispozici zpráva, vrátí se hodnota 0 (vrácené záporné číslo je chybový kód).

`int pvm_recv (int tid, int msgtag);`

Blokující (tj. volající proces čeká) příjem zprávy typu `msgtag` od procesu–odesilatele `tid`. Hodnota `tid=-1` znamená „libovolný odesilatel". Volání vrací identifikátor *bufid* (nové) *aktivní* přijímací vyrovnávací paměti obsahující zprávu nebo chybový kód (záporné číslo). Stará aktivní přijímací paměť je zrušena.

`int pvm_nrecv (int tid, int msgtag);`

Neblokující varianta k `pvm_recv()`. Nebyla–li požadovaná zpráva dosud přijata, volání funkce (bez čekání) vrátí hodnotu 0. Je–li zpráva k dispozici, vrací volání *bufid* (nové) *aktivní* přijímací vyrovnávací paměti nebo záporný chybový kód. Stará aktivní přijímací paměť je zrušena.

`int pvm_upkint (int* ip, int nitem, int stride);`

Čtení dat ze zprávy *v aktivní přijímací paměti* je realizováno množinou funkcí `pvm_upk()` inverzních k `pvm_pk()`.

Například funkce `pvm_upkint()` kopíruje z aktuálního místa v aktivní zprávě `nitem` čísel typu `int` do pole reprezentovaného ukazatelem `ip`. Parametr `stride` je celočíselný krok indexu při ukládání do cílového pole.

Synchronizace procesů

`int pvm_barrier (char *group, int count);`

Synchronizace skupiny `group` procesů na bariéře. Zablokuje volající proces do doby, dokud `count` procesů patřících do skupiny `group` nezavolá tutéž funkci. Všechny procesy musí volat funkci se stejnou hodnotou `count` (normálně počet procesů ve skupině) – tím je ošetřena možnost, že rozměr skupiny se změní v době, kdy procesy čekají na bariéře. Jakmile je bariéra úspěšně překročena, dojde k její inicializaci a a lze ji znovu použít (další volání `pvm_barrier()`).

`int pvm_sendsig (int tid, int signum);`

Volání zasílá signál číslo `signum` procesu `tid`.

Informace o konfiguraci a procesech

```
int pvm_config (int *nhost, int *narch,
                struct hostinfo **hostp);
```

Volání vrátí informaci o aktuálním počtu `nhost` počítačů v konfiguraci, počtu `narch` různých architektur a ve struktuře `hostp` bližší popis jednotlivých strojů (typ, síťové jméno, ap.)

`int pvm_tasks (int where,int *ntask,struct taskinfo **taskp);`

Volání vrací informace o aplikačních procesech běžících na celém multiprocesoru. Dosazenou hodnotou `where` určíme, zajímá-li nás celý multiprocesor, nebo pouze určitý stroj (0 – všechny procesy, *tid* – konkrétní proces, *pvmd_tid* – všechny procesy pod daným PVMD). Ukazatel `ntask` odkazuje na (voláním vyplněný) aktuální počet procesů aplikace, ukazatel `taskp` odkazuje na (voláním vyplněné) pole struktur popisujících jednotlivé procesy.

`int pvm_mytid (void);`

Volání vrátí jednoznačný kladný číselný identifikátor *tid*, který je v rámci celého multiprocesoru přidělen volajícímu procesu. Pokud nebyl proces dosud zařazen pod (běžící) PVM (například spuštěný hlavní proces aplikace), dojde k jeho zařazení a přidělení *tid*.

`int pvm_parent (void);`

Volání vrátí jednoznačný kladný číselný identifikátor *tid* rodičovského procesu (tj. toho, který vytvořil volající proces voláním `pvm_spawn()`).

`int pvm_pstat (int tid);`

Test stavu procesu `tid`. Volání vrátí hodnotu `PvmOk`, jestliže proces běží nebo `PvmNoTask` jestliže neběží.

`int pvm_mstat (char* host);`

Test stavu počítače `host`. Volání vrátí hodnotu `PvmOk`, jestliže počítač běží nebo `PvmHostFail` jestliže je nedosažitelný (typicky vadný) nebo `PvmNoHost` jestliže počítač není v konfiguraci [22].

Ukončení výpočtu

`int pvm_kill (int tid);`

Volání likviduje proces s identifikátorem `tid`. Volající proces pokračuje dál ve výpočtu.

`int pvm_exit(void);`

Tímto voláním proces opouští PVM (jeho PVMD jej vyřadí z registrace), může ale pokračovat dále ve výpočtu jako normální unixovský proces. Pokud je toto volání použito jako poslední příkaz v programu, proces je po vyřazení z PVM operačním systémem likvidován.

5.2.4 Virtuální počítač typu *processor farm*

Jak již bylo řečeno, prostřednictvím PVM lze emulovat multiprocesor s distribuovanou pamětí a libovolnou (virtuální) topologií komunikačního propojení prvků (nebo jinak řečeno s libovolně zavedenou komunikační sousedností procesů) [23].

[22]Poslední dvě uvedené funkce slouží k implementaci nějaké úrovně spolehlivosti virtuálního multiprocesoru. Například ve výpočetním modelu *processor farm* lze příkaz nesplněný procesem–dělníkem na „spadlém“ stroji zadat jinému procesu. Dále lze k uvedenému účelu využít funkci `pvm_notify()`, jejímž voláním lze vyžádat zaslání zprávy volajícímu procesu, pokud dojde ke změně v konfiguraci virtuálního multiprocesoru (přidání počítače, ubrání či výpadek počítače, ukončení (jiného) procesu).

[23]Například virtuální počítač s topologií 2D mřížky s rozměrem n x n by se realizoval takto:

Pro většinu instalací PVM bude ale fyzickou topologií *sběrnice*. Problémem je pak *propustnost* jednoho sdíleného komunikačního kanálu, na kterém může být v jednom čase přenášena jen jedna zpráva. *Proto úspěšné využití PVM lze očekávat zejména v případech paralelní dekompozice aplikace na relativně samostatné procesy s velkým objemem výpočtu a nízkou komunikační režií.*

Pro takovéto aplikace realizujeme prostřednictvím PVM virtuální multiprocesor označovaný jako *processor farm*. Přirozený výpočetní model (označovaný angl. jako *demand driven*) pro tento počítač lze charakterizovat zhruba následovně:

- jeden procesor v konfiguraci je řídicí (angl. *farmer*) a vykonává řídicí aplikační proces – jeho program označíme jako *PF*,
- ostatní procesory v konfiguraci (počet označíme n) jsou podřízené (angl. *workers*) a vykonávají všechny stejný aplikační proces – příslušný program označíme *PW*,
- řídicí proces zadává požadavky na výpočet (zasílá zprávy–*příkazy*) podřízeným procesům, tyto po vykonání požadavku vrací nazpět řídicímu procesu zprávu–*výsledek*,
- řídicí procesor sbírá výsledky (čtení zpráv), provádí případně jejich průběžné zobrazování, komunikuje s obsluhou, zahajuje a ukončuje celý výpočet.

- Spustí se řídicí proces realizující celkovou koordinaci výpočtu.
- Řídicí proces vytvoří n x n podřízených procesů *pracujících podle stejného programu* (model SPMD). Identifikátory vytvořených procesů se poznamenají do pole `int tids [n] [n]`.
- Řídicí proces rozešle pole `tids` všem vytvořeným procesům voláním `pvm_bcast()`.
- Každý podřízený proces nejprve zjistí voláním `pvm_mytid()` své `tid`. Dále po přijetí zprávy od řídicího procesu (zároveň jeho otec) najde své místo v mřížce a dosadí si do svých lokálních proměnných (např. `soused_vlevo`, `soused_vpravo`, atd.) identifikaci sousedů.
- Dále již sousední procesy komunikují vzájemně a realizují příslušný výpočet v 2D mřížce.

Virtuální topologii lze zřejmě v případě potřeby relativně snadno měnit pro jednotlivé fáze paralelního výpočtu.

Uvedený – relativně primitivní – výpočetní model je vhodný pochopitelně pouze pro omezené spektrum paralelních algoritmů (nenáročných na synchronizaci procesů a objem komunikace). Ideální je situace, kdy zprávy jsou datově úsporné (krátké) a objem výpočtu potřebného pro vykonání příkazu je velký. Jako příklad můžeme uvést paralelizaci úlohy *sledování paprsku* (angl. *ray–tracing*) z počítačové grafiky. Zobrazuje se scéna, jejíž (konstantní) datový popis je na začátku výpočtu zaveden spolu se (stejným) programovým kódem podřízených procesů do n procesorů virtuálního multiprocesoru. Další procesor v konfiguraci realizuje řídicí proces, který zadává podřízeným procesům výpočet jednoho prvku obrazovky (pixelu, popřípadě řádku či sloupce - viz dále problém dynamického vyvažování zatížení). Zprávy jsou krátké (*příkaz* obsahuje jen polohu prvku, *výsledek* jen zakódovanou barvu a jas prvku) a výpočet šíření paprsku ve scéně (tj. jeden úkol zadaný podřízenému procesu) je relativně časově náročný.

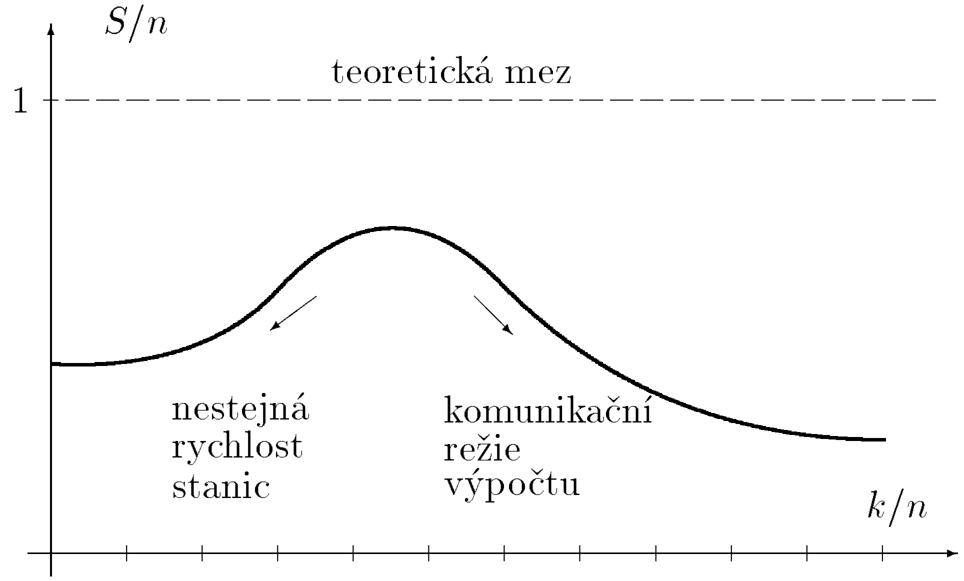
Využitelnost PVM významně ovlivňuje skutečnost, že jak procesy PVMD, tak řízené ovládané aplikační procesy, jsou zpravidla vykonávány v operačním systému **Unix** příslušné stanice na pozadí. To znamená, že neomezují významně proces „standardního“ uživatele stanice probíhající v popředí. Na druhé straně ovšem, vzhledem k apriori neznámému zatížení stanic (prvků virtuálního multiprocesoru), lze jen obtížně predikovat rychlost výpočtu paralelizované aplikace. Proto při používání modelu *processor farm* preferujeme dynamické rozvrhování výpočtu na podřízené procesy s cílem dosáhnout automatickou adaptaci aplikace na aktuální zatížení procesorů virtuálního počítače (angl. *dynamic load balancing*) a tím i největší možné reálné urychlení aplikace.

Uvažujme dále aplikaci zahrnující velký objem (relativně jednoduché a homogenní) výpočetní činnosti. Tuto činnost lze rozdělit na k dílčích činností probíhajících podle stejného programu PW a nevyžadujících ani synchronizaci ani výměny dat. Řídicí proces (má program PF) zadává dílčí činnosti k provedení podřízeným procesům (počet n, program PW), sbírá dílčí výsledky a zpracuje je do celkového výsledku výpočtu.

Problémem zůstává vhodné dělení celkového objemu výpočetní činnosti na dílčí činnosti. Nepochybně nejjednodušší je statické rozdělení podle počtu dostupných procesorů, tedy $k = n$. V tomto případě zpracuje každý podřízený proces stejný díl práce, výsledek je ovšem k dispozici až po ukončení činnosti nejpomalejšího procesoru.

Vhodnější je vytvořit větší počet dílčích činností ($k >> n$) a dynamicky přidělovat tyto dílčí činnosti podřízeným procesům [24]. V tomto případě „rychlé procesory" zpracují větší objem práce než „pomalé" (*dynamic load balancing*) a odpadá tak závislost celkové doby výpočtu na nejpomalejším procesoru. Navíc algoritmus (program PF) využívající dynamické přidělování práce podřízeným procesům není závislý – nebo jen nepatrně – na počtu dostupných procesorů (vlastnost *scalability*).

Na druhé straně ale s narůstajícím poměrem k/n vzrůstá komunikační režie výpočtu. Lze očekávat závislost reálného urychlení výpočtu S na poměru k/n podle obrázku 5.5.



Obr. 5.2: Reálné urychlení výpočtu

Křivka naznačuje existenci optimálního poměru k/n, pro který je urychlení v důsledku paralelizace výpočtu největší. Určit přesně tento poměr pro konkrétní aplikaci PVM ale není jednoduché s ohledem na heterogennost výpočetní sítě a proměnlivost zatížení jejích prvků. V konkrétních případech vystačíme s hrubým experimentálním ověřením vhodného poměru k/n.

[24]Pokud je to možné jednoduše zařídit, měly by být v seznamu dílčích činností časově náročnější činnosti na začátku.

Dynamický způsob přidělování práce má nezanedbatelný vliv i na *spolehlivost*, se kterou v reálné době získáme výsledky výpočtu. Uživatel PVM typicky spouští řídicí proces na „své" pracovní stanici – ideálně na nejrychlejší dostupné stanici. Pokud některá z „podřízených" stanic nedodá výsledek včas (např. náhodné vypnutí „cizí" stanice jejím uživatelem), je možné testovat stav procesu (funkce `pvm_pstat()` vrátí `PvmSysErr`) a příslušný díl práce lze zadat ke zpracování jinému procesu.

5.2.5 Ladění programů

Ladění paralelních programů je obecně mnohem komplikovanější než ladění konvenčních programů zejména s ohledem na možnost *zablokování výpočtu* a možnosti vzniku *časově závislých chyb*.

Základní možnost diagnostiky průběhu výpočtu poskytují chybové kódy, které navrací volání jednotlivých funkcí PVM. Tyto chybové kódy je jednak možné v programu testovat a dále PVM implicitně realizuje výpis příslušného diagnostického hlášení (možno vypnout prostřednictvím volání funkce `pvm_setopt()`). U procesů startovaných „ručně" (typicky řídicí proces aplikace) jsou diagnostická hlášení směrována na standardní výstup *stdout* (tj. do okna, z kterého byl proces spuštěn). Pokud řídicí proces vytvoří (`pvm_spawn ()`) procesy–děti, může jejich diagnostická hlášení PVM i programově realizované výpisy přesměrovat na vlastní *stdout* (nebo do souboru) voláním `pvm_catchout()`. Ručně startované procesy mohou být spuštěny pod standardním (sériovým) ladicím programem (angl. *debugger*). Rovněž procesy spuštěné voláním `pvm_spawn()` mohou být spuštěny pod ladicím programem [25].

Při využití PVM lze doporučit následující tři fáze ladění paralelního programu:

- Každý proces je nejprve laděn samostatně, tj. je spuštěn pod libovolným ladicím programem. Pokud má několik procesů stejný program, je tato fáze pro všechny společná. Všechna volání PVM musí být v

[25] V tomto případě je třeba nastavit volbu `PvmTaskDebug` v parametru `flag` ve volání `spawn()`. V tomto případě PVM (implicitně) spustí příkazový soubor, který otevře nové *xterm* okno na počítači, kde běží proces–otec a spustí založený proces–syn pod ladicím programem na počítači určeném v parametru `flag`. Ladicí výpisy jsou pak směrovány do vytvořeného okna.

této fázi vypuštěna nebo nějak nahražena. Cílem je odstranit běžné chyby (indexování, logika ap.) v programu.

- Doplní se volání PVM a program se spustí s malým počtem (2–4) procesů [26] na jednom počítači. Cílem je odladit syntaxi a logiku komunikace (např. zjistit nekonzistence v kódování zpráv). Spuštění procesů na jednom stroji (každý pod ladicím programem) umožňuje plně řídit výpočet z terminálu (krokování, breakpointy).

- Paralelní program odladěný v předchozí fázi (malý počet procesů) se spustí s procesy alokovanými na různé počítače v konfiguraci. Cílem je ověřit *rychlostní nezávislost výpočtu*, tj. eliminovat synchronizační chyby způsobené *časovým zpožděním* přenosu zpráv mezi prvky virtuálního multiprocesoru a necitlivost algoritmu na *pořadí příchodu zpráv*. V této fázi už není dobré využívat ladicí program s ohledem na změnu časových poměrů výpočtu způsobenou jeho použitím.

Je rovněž možné si představit i jiný způsob postupného vytváření a ladění aplikace – nejprve se odladí „prázdná“ komunikační struktura (ve zprávách záleží jen na jejich kódu – informační obsah je prázdný, výpočet se nerealizuje nebo se emuluje jeho časová náročnost programovou smyčkou) a potom se teprve doplňují „sekvenční“ sekce výpočtu (postupně do jednoho nebo zároveň do několika procesů). Vliv každého „přidání“ lze bezprostředně testovat.

5.2.6 Příklad použití PVM

Použití funkcí uživatelského rozhraní PVM dále předvedeme na jednoduchém příkladu virtuálního počítače výše diskutovaného typu *processor farm*.

Příklad 5.1

Nejprve uvedeme program hlavního procesu *šéf* (viz výše označení *PF*), který pouze vytvoří zadaný počet dělníků (bez bližšího určení stroje, na kterém mají běžet), vyšle jim zprávu (broadcast) se svým identifikátorem a přijme jejich zpětné hlášení o připravenosti k činnosti.

[26] Paralelní programy jsou normálně psány jako *scalable*, tj. jako nezávislé na počtu dostupných procesorů (a odpovídajícím počtu spolupracujících procesů).

```
#include "pvm3.h"
main() {
  int mytid;      /*  identifikátor procesu šéf  */
  int tids[32];   /*  identifikátory procesů dělníci */
  int nproc;      /*  počet dělníků */
  int i, tid;     /*  pomocné proměnné */

  mytid = pvm_mytid();  /* přihlášení do PVM */

  /* vytvoření dělníků */
  printf("Zadej počet dělníků (1 až 32):");
  scanf("%d", &nproc);
  pvm_spawn("worker", (char**)0, 0, "", nproc, tids);

  /* ukázka komunikace šéfa s dělníky */
  pvm_initsend(PvmDataDefault);   /* inicializace bufferu */
  pvm_pkint(&mytid, 1, 1);     /* uložení identifikace */
  pvm_mcast(tids, nproc, 4);   /* broadcast zprávy s kódem 4 */
                               /*              všem dělníkům */
  /* čekání na odezvu od dělníků */
  for(i=0; i<nproc; i++) {
    pvm_recv(-1, 5);          /* 5 je kód zprávy */
    pvm_upkint(&tid, 1, 1);  /* číslo dělníka */
    printf("Delnik cislo %d k praci pripraven! \n ", tid);
  }
  /* ukončení činnosti */
  pvm_exit();
}
```

Dělníci se nejprve registrují pod PVM voláním `pvm_mytid()`, dále čekají na zprávu od šéfa, kontrolují identitu šéfa a hlásí svoji připravenost k činnosti. Program procesu dělníka (v popisu modelu *processor farm* označovaný jako *PW*) je tedy:

```
#include "pvm3.h"
main() {
  int mytid;      /*  identifikátor procesu dělník  */
```

```
  int chief_tid;  /*  identifikátor procesu šéf  */
  int i;          /*  pomocná proměnná */

  mytid = pvm_mytid();  /* přihlášení do PVM */

  /* čekání na zprávu od šéfa */
  pvm_recv(-1, 4);               /* 4 je kód zprávy */
  pvm_upkint(&chief_tid, 1, 1);  /* zjistí číslo šéfa */

  /* kontrola zda je šéf ten pravý - totožný s otcem */
  if (chief_tid == pvm_parent()) {
     /* odpověď šéfovi */
     pvm_initsend(PvmDataDefault);/* inic. bufferu */
     pvm_pkint(&mytid, 1, 1);    /* uložení identifikace */
     pvm_send(chief_tid, 5);     /* kód odpovědi 5 */
  }

  /* ukončení činnosti */
  pvm_exit();
}
```

5.3 Programování na počítačích nCUBE2

5.3.1 Systémová charakteristika nCUBE2

Počítače nCUBE2 patří mezi komerčně úspěšné superpočítače v kategorii multiprocesorových systémů s distribuovanou pamětí. Procesorové pole je propojeno rychlými sériovými komunikačními kanály do *n-rozměrné krychle* a může obsahovat až 8192 prvků. Rozšiřitelný I/O subsystém je založen jako extenze procesorového pole s cílem dovolit „masivně paralelní" I/O. Konfigurace je doplněna „organizačním" počítačem *host system*, který mj. zajišťuje standardní uživatelské rozhraní (**Unix**) a připojení k počítačovým sítím. Pro vývoj programů je k dispozici programové vybavení nazývané Parallel Software Environment, které je z části určeno k exekuci ve vlastním procesorovém poli a z části je využíváno v hostitelské pracovní stanici pro vývoj programů (křížové prostředky), k jejich zavádění do procesorového pole a k ovládání výpočtu.

Dále uvedeme základní technické informace.

nCUBE VLSI procesor

Speciální zákaznicky navržený integrovaný obvod, obsahující:

- 64-bitovou aritmeticko-logickou jednotku (25 MHz, 15 MIPS),
- 64-bitovou jednotku pro operace v pohyblivé řádové čárce s výkonem 3 MFLOPS,
- řízení dyn. RAM paměti se samoopravným kódem,
- komunikační jednotku pro směrování zpráv,
- 14 obousměrných DMA komunikačních kanálů.

Operační paměť

je přidána k procesorovému obvodu formou několika DRAM čipů. Maximální velikost paměti je 64 Mbyte v jednom *procesorovém modulu* (destička 2.5 krát 8.5 cm).

Komunikační kanály

Z celkového počtu 14 DMA kanálů je 13 využíváno pro spojení s ostatními prvky sítě, přesněji řečeno se sousedními prvky v krychli. Maximální rozměr krychle je tedy $n = 13$, čemuž odpovídá maximální počet prvků $2^{13} = 8192$. Jeden kanál je vyhražen pro účely

I/O. Kanály pracují s přenosovou rychlostí 2.75 Mbyte/s. Komunikace mezi prvky sítě je možná pouze zasíláním zpráv po komunikačních kanálech. Komunikační jednotka procesorového čipu „odlehčuje" procesor od činnosti spojené s přijímáním a vysíláním zpráv. Zprávy, které pouze prochází do jiného prvku sítě, nezpůsobí přerušení činnosti procesoru.

Procesorové pole

Maximálně 64 procesorových modulů (procesorový čip + paměťové čipy) je umístěno na jedné *procesorové desce*. V jedné skříni může být maximálně 16 procesorových desek. Skříně mohou být dále spojovány. Firma nCUBE dodává dvě řady počítačů: nCUBE 2E s počtem procesorů 8-128 a nCUBE 2S s 64-8192 procesory.
Uživatel může postupně zvyšovat výpočetní i I/O výkon doplňováním dalších prvků sítě.
Procesory jsou propojeny komunikačními kanály do n-rozměrné krychle, každý procesor má tedy n přímých sousedů a maximální vzdálenost procesorů v krychli je n.
Používané adresy procesorů v síti jsou založeny na Grayově kódu, adresy sousedních procesorů se liší jen v jedné pozici kódu.

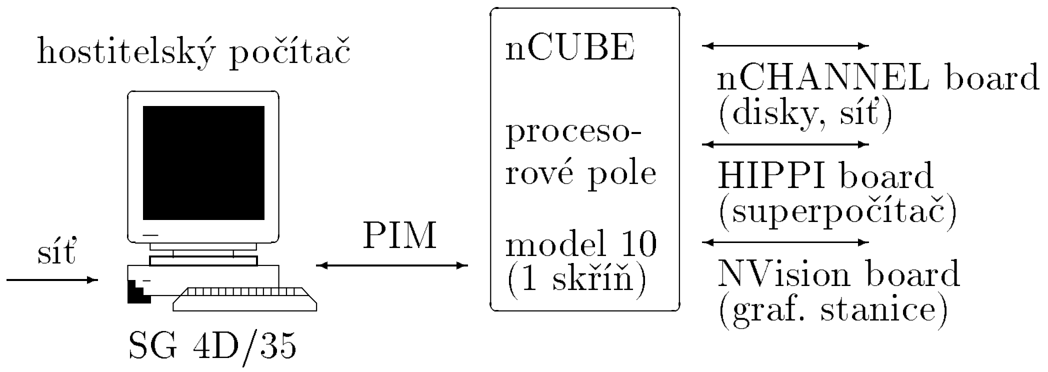
I/O subsystém

Je navržen s cílem dosáhnout (libovolně) vysoké přenosové rychlosti vstupních/výstupních datových proudů. Jediná možnost jak nevytvořit „úzký profil", kterým musí projít všechna I/O data je distribuovat datové „proudy" přímo do procesorové sítě (tj. data mohou vstupovat/vystupovat paralelně ve více proudech).
Do skříně počítače (základní model 10, 1024 procesorů) lze doplnit až 8 I/O desek (*nCUBE nCHANNEL board*), na každé desce je umístěno 16 sériových kanálů (modulů) umožňujících připojení různých periferií (SCSI, Ethernet, A/D a D/A převodníky, video rozhraní a digitální vstupy). Každý kanál je obsluhován jedním nCUBE procesorem (stejný jako prvek procesorové sítě) a umožňuje přenos dat rychlostí max. 20 Mbyte/s. Přes komunikační kanály procesoru může být I/O datový proud „rozveden" až na 8 míst v síti.
Pro spojení s hostitelskou pracovní stanicí je určen speciální vstupní kanál (PIM – Platform Interface Module) s přenosovou rychlostí 16 Mbyte/sec (v obou směrech), jehož modul musí být umístěn na ně-

které I/O desce (má dvojnásobnou šířku, takže na desku lze dát už jen 14 I/O kanálů).
Kromě základní I/O desky (*nCUBE nCHANNEL board*) lze ještě využít *nCUBE HIPPI board* pro přímé spojení s jiným superpočítačem. Další využitelnou I/O deskou je *nVision Graphic Subsystem* pro špičkové grafické operace (obsahuje mj. 16 nCUBE procesorů, 16+2 Mbyte video paměť, rozhraní pro myš, grafický tablet a barevný monitor). Podpora grafiky je opět „zaintegrována“ do procesorové sítě a umožňuje velmi rychlé "on line" zobrazení výsledků výpočtu.

Hostitelský počítač

Jako standardní hostitelský počítač je využívána pracovní stanice Silicon Graphics 4D/35. Může být využita i pracovní stanice Sun. Pro pracovní stanice spojené v lokální síti může být kterákoliv využita jako *host* pro procesorové pole nCUBE. Konfigurace celého systému je schematicky znázorněna na obr. 5.6.



Obr. 5.3: Systém nCUBE

5.3.2 Programové prostředí

Programové prostředí počítačů nCUBE je tvořeno souborem prostředků označovaných jako nCUBE PARALLEL SOFTWARE ENVIRONMENT (dále zkráceně PSE). Přesto, že studenti ZČU nemají přímou možnost využívat počítače nCUBE, bude PSE v následujících odstavcích přehledově popsáno jako ukázka členění programového vybavení komerčního multiprocesoru s distribuovanou pamětí. PSE obsahuje zhruba následující složky:

Operační systém nCX

Jedná se o jednoduché a účinné jádro operačního systému (128 kbyte) *rezidující ve všech prvcích procesorové sítě.* Volání systému splňuje normu rozhraní **Unix** System V, Release 4. Totéž jádro je využíváno i v I/O uzlech sítě (viz výše I/O) s tím rozdílem, že je sestaveno s potřebnými I/O drivery.

Host Interface Driver

Jedná se o program rezidující v I/O uzlu PIM (Platform Interface Driver) a jeho úkolem je zajišťovat veškerou komunikaci mezi *host* počítačem a procesorovým polem včetně zavádění programů do prvků procesorového pole a komunikace mezi procesy v poli a procesem v *host* počítači.

Křížové vývojové prostředky

Jsou určeny pro psaní programů, překlad, ladění a zvyšování efektivnosti (angl. *profiling*), zavádění programů do procesorového pole a jejich ovládání za běhu. Tyto prostředky (označované jako *utility*) jsou vesměs implementovány pod operačním systémem hostitelské pracovní stanice (**Unix**) jako samostatné programy.
Jsou k dispozici překladače programovacích jazyků **FORTRAN**, **C** a **C++**. Ovládání křížových překladačů je sjednoceno v systémové utilitě `ncc`.

Knihovny funkcí

Jedná se o knihovny funkcí využitelných v programech psaných ve výše uvedených programovacích jazycích pro podporu zvoleného modelu paralelního výpočtu (viz dále). Zejména jde o knihovny:

- nCUBE Extended Run-time library,
- nCUBE Parallelization library.

První uvedená knihovna obsahuje základní funkce nutné pro podporu všech využitelných modelů paralelního výpočtu, druhá knihovna rozšiřuje možnosti modelu SPMD. Funkce z uvedených knihoven *jsou volány z programů určených k výpočtu v procesorové síti* nCUBE. Další významnou knihovnou určenou ale *k využití v programech hostitelské pracovní stanice* je

- nCUBE Host Interface Library.

Příkazy Unixu určené k výpočtu v procesorovém poli nCUBE

Jedná se o množinu obvyklých příkazů **Unix**u (např. `ls, cat, cp, date`, ...) implementovaných jako samostatné programy schopné zavedení a exekuce v libovolném prvku procesorového pole (operační systém nCX má rozhraní **Unix**u). Tyto programy se buď zavádí do procesorového pole stejným způsobem jako aplikační programy (příkaz `xnc`), nebo je možné spustit interpret příkazů (*shell*) v příslušném prvku sítě a ten si dokáže sám zavést příslušný program (např. `ls`).

Do kategorie prostředků zařazených pod operační systém hostitelské pracovní stanice dále spadají utility pro systémovou administraci, monitorování procesorové sítě a řazení zakázek (angl. *job*) pro dávkové zpracování.

Je k dispozici množina vzorových programů a on-line dokumentace dostupná obvyklým způsobem (help).

Při programování aplikace máme následující možnosti rozdělení práce mezi hostitelskou pracovní stanici a pole procesorů nCUBE:

- Programy $\{P_i\}$ pro prvky procesorového pole jsou vytvořeny v hostitelské pracovní stanici pomocí příslušných křížových prostředků a zavedeny do procesorového pole prostřednictvím *shellu* hostitelské pracovní stanice (příkaz `xnc`).

- Program H napsaný pro hostitelskou pracovní stanici (a přeložený např. C-kompilátorem pro SUN) běží na pracovní stanici, zavádí programy $\{P_i\}$ (vytvořené křížovými prostředky) do prvků sítě a zajišťuje s nimi komunikaci (například převezme a zobrazí výsledky výpočtu).

Programy pro prvky sítě (nezávisle na výše uvedených způsobech jejich zavedení) jsou založeny na jednom ze dvou základních modelů již dříve uváděných, tj. buď MPMD (*funkční paralelismus*), nebo SPMD (*datový paralelismus*):

- **SPMD** (Single Program, Multiple Data),
  Tentýž program je zaveden do množiny prvků procesorové sítě. Každý výpočetní proces probíhající v prvku sítě podle zavedeného programu má vlastní „kopii“ dat. Globální data výpočtu mohou být rozdělena na části alokované do paměti jednotlivých prvků participujících na výpočtu [a].

  Výpočet může probíhat různými větvemi na různých procesorech (v závislosti na konkrétních hodnotách dat). Procesy si mohou vyměňovat informaci zasíláním zpráv po komunikačních kanálech. Komunikace se typicky provádí při výměně „mezivýsledků“ nebo při kompozici celkového výsledku z dílčích výsledků vypočítaných lokálně v prvcích sítě.

- **MPMD** (Multiple Program, Multiple data),
  Množina různých programů je zavedena do různých prvků sítě. Každý program má svoje data (strukturálně odlišná od ostatních). Procesy probíhající podle zavedených programů si vyměňují informaci zasíláním zpráv.

  Pod tento model výpočtu lze zahrnout i řetězové zpracování (MPSD), při kterém si procesy probíhající podle různých programů „předávají“ ke zpracování datové záznamy.

[a] Při využití paralelizace způsobem SPMD je na nCUBE přirozenější statický způsob přidělování práce procesům. Dynamický způsob (viz sym. multiprocesory a model *processor farm*) vyžaduje, aby kterýkoliv proces mohl převzít kterýkoliv díl práce (tj. příslušná data ke zpracování). To samozřejmě není jednoduše možné, pokud jsou data (tj. díly práce) distribuována do lokálních pamětí procesorů. Počet dílů, na které dělíme práci je zde dán počtem procesorů alokované podkrychle.

5.3.3 Překlad, zavedení a spuštění programu

Pro výpočet programu se zpravidla přiděluje „podkrychle“ ze sítě procesorů, tj. počet přidělených procesorů je mocninou dvou. Může ale být přidělen i vybraný procesor (krychle s rozměrem 0). Podkrychle vždy obsahuje „nejbližší“ procesory (připomínáme, že největší vzdálenost je n, kde n je rozměr podkrychle). Tímto způsobem je výrazně zmenšeno komunikační zpoždění mezi procesy zúčastněnými na výpočtu. Uživatel se nemusí (pokud nechce) starat o to, které procesory mu budou přiděleny (jsou ostatně funkčně rovnocenné, mohou se lišit pouze velikostí dedikované paměti). Hlavní charakteristiky použitého způsobu přidělování procesorů pro aplikační výpočet rozvedeme podrobněji v následujících bodech:

- Rozměr přidělené krychle je typicky požadován při překladu programu (volba $-d$ v příkazu `ncc`). Lze jej také určit (změnit) při zavádění programu (příkaz `xnc`). *Program založený na modelu SPMD by měl být napsán tak, aby mohl běžet na krychli s libovolným rozměrem* (tzv. *scalable computation*) a v ideálním případě by navíc měl vykazovat (téměř) lineární závislost urychlení výpočtu S na počtu přidělených procesorů.

  Pokud nebyl explicitně uveden požadavek na velikost přidělené krychle, přidělí se krychle rozměru 0, tj. jeden procesor.

- *Program může odkazovat na procesory přidělené podkrychle pomocí logických čísel procesorů*, která jsou v rozsahu 0 až $2^n - 1$, kde n je rozměr přidělené podkrychle. Logická čísla jsou vždy „relativní" v podkrychli, tj. nezávisí na skutečné poloze procesorového prvku v poli procesorů. Fyzicky sousední procesory dostanou sousední logická čísla. Pokud je využit model SPMD, lze logická čísla procesorů využít i jako logická čísla procesů na nich počítaných a „čísla částí práce", která se má vykonat. *Run-time library dává k dispozici funkce* `whoami()` *a* `npid()`*, pomocí kterých může proces zjistit logické číslo procesoru, na kterém běží.*

- Každá přidělená podkrychle je identifikovatelná pomocí PID procesu hostitelské pracovní stanice, který krychli alokoval a případně řídí výpočet probíhající v krychli.

- *Jednou přidělená podkrychle nemůže být odebrána před ukončením výpočtu.* Je to samozřejmé s ohledem na poměrně pracné zavádění programů do prvků krychle (a navíc nemožnost zaznamenat a jednoduše obnovit kontext výpočtu).

  Do přidělené krychle ale může být zaveden paralelní program jiného uživatele (OS v prvcích sítě je **Unix**, který umožňuje multitasking). Pokud si uživatel přeje mít přidělenou podkrychli pouze pro svůj výpočet (a docílit tak maximální možné urychlení), má možnost zakázat zavedení dalších programů do „své" krychle volbou $-x$ v příkazu `xnc`.

Dále ukážeme jednoduché základní uživatelské ovládání složitého superpočítače. Vytvoříme následující program v jazyce **C**:

```
/* Soubor hello.c */
#include <stdio.h>
main {
  printf ("DDT ZDAR!!!\n");
}
```

Přeložíme a sestavíme program překladačem **C** pro hostitelskou pracovní stanici a spustíme jeho výpočet (% je ohlášení iterpretu příkazů (*shell*)).

```
% cc hello.c
% a.out
```

Vypíše se: `DDT ZDAR!!!` na standardní výstupní zařízení. Dále přeložíme (a zároveň sestavíme) tentýž program křížovým překladačem pro prvky procesorové sítě nCUBE.

```
% ncc -d 2 hello.c
```

Volba $-d2$ znamená požadavek na přidělení krychle s rozměrem $n = 2$. Výpočet spustíme stejným způsobem jako v předchozím případě.

```
% a.out
```

Interpret příkazů pozná, že se jedná o spustitelnou formu programu určenou k zavedení do krychle, program zavede a spustí. Vypíše se:

```
DDT ZDAR!!!
DDT ZDAR!!!
DDT ZDAR!!!
DDT ZDAR!!!
```

Tentýž program byl zaveden do $2^2 = 4$ prvků procesorové sítě a každý ze čtyř vytvořených procesů vypsal na standardní výstupní zařízení tentýž text. Pořadí výpisů z jednotlivých prvků je náhodné (zde se nepozná, protože vypisovaný text je stejný).

Pokud chceme, aby program běžel na větší krychli, nemusíme jej znovu překládat, ale zařídíme jeho zavedení a spuštění příkazem `xnc`:

```
% xnc -d 5 a.out
```

a dostaneme 32 pozdravných výpisů na standardní výstupní zařízení.

5.3.4 Extended Run-time library

Jedná se o knihovnu standardních run-time funkcí volaných z programu v jazyce **C**, **C++** nebo **FORTRAN** v prostředí OS **Unix**. Dále budeme sledovat pouze využití v **C**. Rozšíření knihovny spočívá v doplnění funkcí umožňujících programovou podporu základních modelů paralelního výpočtu.

Není možné (a ani účelné) popisovat detailně jednotlivé funkce knihovny. Dále uvedeme jen základní charakteristiky funkcí tak, aby čtenář získal ucelenější představu o stylu paralelního programování na nCUBE.

Funkce pro získání informace o prostředí procesu

`whoami (int* pid, int* proc, int* host, int* dims)`

Volání funkce vrací následující informaci: `pid` je pozice procesoru v přidělené podkrychli (tj. logické číslo procesoru lokální v podkrychli), `proc` je *pid* volajícího procesu (v rámci **Unix**u na procesoru - prvku sítě), `host` je *pid* procesu hostitelské pracovní stanice (v rámci **Unix**u stanice), který alokoval krychli, zavedl paralelní program a řídí paralelní výpočet, `dims` je dimenze podkrychle.

`int npid(void)`

Volání vrací pozici procesoru v podkrychli (logické číslo).

`int ncubesize(void)`

Volání vrací dimenzi přidělené podkrychle.

Funkce pro komunikaci mezi procesory

Uvažovaný model komunikace je asynchronní (tj. odeslání zprávy není blokující). V operacích pro odesílání zpráv *se adresuje logickým číslem cílový procesor* (nikoliv cílový proces, protože nelze předem určit, jaké číslo *pid* dostane vzniklý proces od **Unix**u v příslušném prvku sítě). Každá zpráva má jako prvek *celočíselný typ zprávy*, který může mj. sloužit jako *číslo logického kanálu* určeného pro komunikaci mezi dvěma procesy (toto číslo lze určit předem při vytváření paralelního programu, takže programový popis komunikace nezávisí na alokaci komunikujících procesů). Proces může v operaci `nread()` reagovat na zprávy s typem v intervalu nastaveném pomocí `nrange()`. Implicitně je nastaven interval od 0 do 32 767, takže se čtou všechny přijaté zprávy. Ještě před operací čtení je možné testovat přítomnost zprávy pomocí `ntest()`.

Pomocí zde uvedených komunikačních primitivů jsou dále implementovány „mocnější" prostředky pro paralelizaci (viz dále v čl. 5.2.6. Dále uvedeme základní komunikační funkce určené pro komunikaci mezi aplikačními procesy alokovanými v prvcích krychle [27]:

```
int nwrite (char* buff, int nbytes, int dest, int type,
                                               int* flag)
```

Neblokující funkce pro odeslání zprávy, kde:
`buff` je adresa vyr. paměti, z které se odesílá zpráva,
`nbytes` je délka zprávy v počtu byte,
`dest` je logické číslo cílového procesoru,
`type` je typ zprávy,
a testováním `flag` lze zjistit, zda byla zpráva úspěšně odeslána.

```
int nread (char* buff, int nbytes, int* src, int* type,
                                               int* flag)
```

Funkce pro čekání na zprávu (tj. blokující) a čtení zprávy, kde:
`buff` je adresa vyr. paměti, do které je zpráva přečtena,
`nbytes` je délka přijaté zprávy v počtu byte,
`src` je logické číslo odesílajícího procesoru,
`type` je adresa pro uložení typu přijaté zprávy,
a testem `flag` lze zjistit, zda byla zpráva úspěšně přijata.

```
int ntest (int* src, int* type)
```

Funkce, kterou proces může testovat přítomnost očekávané zprávy typu `type` z procesoru `src`. Vrací 1, pokud je zpráva přítomna. Použitím `ntest()` se může proces vyhnout čekání s nepredikovatelnou dobou (a eventuelnímu zablokování).

```
void nrange (int low, int high)
```

[27] Analogické komunikační funkce jsou využitelné i v programu procesu řídícího z hostitelské stanice výpočet v procesorovém poli nCUBE. Tyto funkce jsou součástí Host interface Library (lehce se liší v parametrech).

Volání funkce umožňuje nastavit rozsah typů zpráv akceptovaných v operaci `nread()`.

Funkce pro nastavení modu I/O operací

Jsou možné dva způsoby zajištění I/O. *Lokální mod* znamená, že I/O operace procesoru probíhá nezávisle na ostatních procesorech. Lokální mod je nastaven implicitně. *Globální mod* vyžaduje *synchronní* provádění stejných I/O operací (se stejnými daty) ve všech procesech. Typicky je globální mod užitečný v modelu SPMD při vstupu parametrů výpočtu (všechny procesy přečtou stejnou informaci). Při výstupu v globálním modu *se výstup provede jen z procesoru s logickým číslem 0* (stejný efekt lze ovšem dosáhnout testem čísla procesoru a výstupem podmíněným výsledkem testu). Přepínání I/O modu lze provést voláním funkcí:

```
int nglobal();
int nlocal();
```

Funkce vrací „starý“ mod (0 - lokální, 1 - globální).

Příklad 5.2

Jako příklad využití funkcí z popisované knihovny uvedeme paralelní program, podle kterého probíhají procesy ve všech prvcích přidělené podkrychle (s libovolným rozměrem). Každý proces pošle zprávu „nejvzdálenějšímu“ procesu v podkrychli a přečte jeho zprávu. K získání logického čísla nejvzdálenělšího procesoru musí proces provést negaci všech významových bitů (tj. n bitů) logického čísla vlastního procesoru. *Knihovna je v rámci činnosti* `ncc` *připojena k programu automaticky* (tj. není třeba direktiva `#include` v programu).

```
#define L_LINE     80
#define MES_CODE   333

main () {
 int my_node, my_pid, my_host_pid, cube_dim;
 int flags, dest_node, ret_val, type;
 char buf_out[] = "DDT ZDAR !!!";
 char buf_in [L_LINE];
```

```
  /* identifikace "okolí" procesu */
  whoami (&my_node, &my_pid, &my_host_pid, &cube_dim);

  /* výpočet čísla nejvzdálenějšího procesoru */
  dest_node = my_node^((1 << cube_dim) - 1);

  /* odeslání pozdravu */
  ret_val = nwrite (buf_out, sizeof(buf_out), dest_node,
    MES_CODE, &flags);
  if (ret_val == -1)
    perror ("Chyba v~nwrite");
  printf ("Procesor %d odeslal zprávu do %d \n", my_node,
    dest_node);

  /* příjem pozdravu */
  ret_val = nread (buf_in, L_LINE, &dest_node, &type, &flags);
  if (ret_val == -1)
    perror ("Chyba v~nread");
  printf ("Procesor %d přijal zprávu z %d \n", my_node,
    dest_node);
}
```

Program lze přeložit pomocí `ncc` pro libovolný rozměr krychle. Výpisy z programu by byly provedeny na standardní výstupní zařízení *v náhodném pořadí*. Uspořádání výpisů by bylo možné provést využitím funkcí `seqstart()` a `seqend()` – viz dále.

5.3.5 Host Interface Library

Jedná se o knihovnu funkcí volaných z programu H počítaného v prostředí OS **Unix** hostitelské pracovní stanice. Program je překládán **C**-kompilátorem pro příslušnou stanici. Funkce v knihovně umožňují *organizovat výpočet v procesorovém poli nCUBE* z hostitelské pracovní stanice. Hlavičkový soubor knihovny je `nhost.h` [28].

[28]Stejné funkce jako uvedené v tomto odstavci jsou zahrnuty i v Extended Run-time library (popis dříve) pro využití z procesů počítaných v uzlech sítě nCUBE. V programech těchto procesů (překládaných křížově pomocí `ncc`) se musí provést `#include <unistd.h>`.

Předpokládejme, že paralelní program $\{P_i\}$ je připraven (tj. přeložen kompilátorem `ncc`, bez udání rozměrů požadované podkrychle procesorového pole). „Organizační" program H pak typicky obsahuje následující části:

- volání `nopen()` jako požadavek na přidělení podkrychle,
- volání `nodeset()`, kterým specifikuje využívanou podmnožinu procesorů z přidělené krychle,
- volání `rexec()`, kterým se zavádí specifikovaný program z P_i do procesoru i a spouští jeho výpočet,
- případnou komunikaci se založenými procesy pomocí funkcí `nwrite()` a `nread()`, například odeslání parametrů a převzetí výsledků,
- uvolnění přidělené krychle po ukončení výpočtu voláním `nclose()`.

Základní funkce knihovny:

`int nopen (int dimension)`

Přiděluje krychli s rozměrem `dimension` a vrací celočíselný identifikátor (*file descriptor*) přidělené krychle.

`int nodeset (int cube, char* node_set)`

Přiděluje *file descriptor* `cube` získaný voláním `nopen()` množině procesorů popsaných v řetězci `node_set`. Například řetězec $\{0, 3-7-2, 10\}$ znamená procesory 0,3,5,7,10, popřípadě {`all`} znamená všechny. Vrací deskriptor využitelný dále v `rexec()`.

`int nclose (int cube)`

Čeká na ukončení všech procesů založených v krychli, poté uvolňuje přidělenou krychli s deskriptorem `cube`.

```
int nwrite (int fildes, char* buff, int nbytes, int dest,
                                                int type)
```

Funkce pro odeslání zprávy (analogická k již uváděné) kde:
`fildes` je deskriptor krychle dříve přidělené pomocí `nopen()`
`buff` je adresa vyr. paměti z které se odesílá zpráva,
`nbytes` je délka zprávy v počtu byte,

`dest` je logické číslo cílového procesoru,
`type` je typ zprávy.

Ostatní komunikační funkce podobně jako `nwrite()` se liší přidaným parametrem `fildes` a nebudeme je zde uvádět.

Dále je v popisované knihovně množina funkcí `rexec()`. Všechny funkce z množiny slouží k zavedení specifikovaného programu do specifikované množiny procesorů. Poslední přidané písmeno (písmena) názvu funkce specifikuje informaci přenášenou spolu se zaváděným programem:

- `l` ... přenáší se seznam parametrů programu (pointery na char),
- `v` ... přenáší se pointer na vektor parametrů,
- `e` ... přenáší se pointer na vektor prostředí,
- `p` ... hledá zaváděný program ve specifikovaném seznamu cest.

Na ukázku uvedeme jednu z funkcí „rodiny“ `rexec()`.

```
npid_t rexecv (int nodeset_ID, char* path, char** argv)
```

kde `nodeset_ID`je deskriptor (získaný z `nodeset()`) množiny procesorů, do které se zavádí program ze souboru se jménem (včetně cesty) uloženým v řetězci `path`. Do procesů vytvořených ve specifikované množině procesorů se přenáší argumenty uložené ve vektoru řetězců `argv`.

Příklad 5.3

Jako příklad využití funkcí z popisované knihovny uvedeme jednoduchý program `host.c`, který běží na hostitelské pracovní stanici (a je tudíž překládán překladačem `cc` a nikoliv `ncc`). Program `host.c` zavede program *myprog* (s činností blíže nespecifikovanou) do množiny procesorů sestávající ze všech sudě očíslovaných prvků pětirozměrné krychle.

```
#include <ncube/nhost.h>

main () {
  int cube, set;
  npid_t ret;
```

```
  /* žádost o přidělení podkrychle s rozměrem 5 */
  if ((cube = nopen (5)) < 0)
    nperror ("Chyba v nopen()");

   /* vytvoření a označení množiny uzlů */
   set = nodeset (cube, "0-31-2");

  /* zavedení programu */
  ret = rexecl (set, "myprog", "myprog", "arg1", 0);
  if (ret.np_pid == -1)
    nperror ("Chyba v~rexecl()");

  /* čekání na ukončení výpočtu a uvolnění podkrychle */
  nclose (cube);
}
```

5.3.6 Parallelization library

Jedná se o knihovnu funkcí využitelnou v paralelních programových modelech SPMD, tj. v modelech, kde podle stejného programu probíhá několik procesů (každý nad svou částí dat). *Z toho vyplývá, že dále uváděné funkce jsou volány (téměř) současně na všech zúčastněných procesorech.* Funkce v knihovně jsou implementovány pomocí základních komunikačních primitivů z Extended Run-time library a odlehčují aplikačního programátora od složitého a pracného programování komunikace mezi procesy. Dále má uživatel k dispozici knihovny matematických funkcí – **blas** (Basic Linear Algebra Subroutines) a **sss64** (Seismic Subroutine Standard) implementovaných pro práci nad distribuovanou pamětí.

Příklad 5.4

Jako ukázku využití (několika málo) funkcí z popisované knihovny uvedeme dále jednoduchý program, který dokáže najít vektor maximálních prvků z vektorů „rozptýlených“ jako lokální data ve všech prvcích alokované podkrychle. Program by měl fungovat po zavedení do krychle s libovolným rozměrem. Lokální vektory jsou inicializovány (každý jinak) v úvodní části výpočtu. Zvolený způsob nicializace zde simuluje jiný (smysluplnější) způsob, kterým by se vektory naplnily.

```
#include <stdio.h>
#include <npara/npara_prt.h>

#define  MULT  3     /* konstanta pro inicializaci vektorů */
#define  LEN   4     /* délka vektorů */

main () {
  int  j, mynode, mypid, myhost, cube_dim, vec[LEM];

  /* identifikace okolí */
  whoami (&mynode, &mypid, &myhost, &cube_dim);

  /* inicializace vektorů */
  for (j=0; j<LEN; j++)
    vec[j] = MULT * (j+1) * (mynode + 2);

  /* zobrazení vektorů na stand. výstup */
  seqstart (mynode - 1, mynode + 1, 1);
    /* Volání způsobí,  že  následující  část  programu
    až  do  nejbližšího  seqend()} bude počítána sekvenčně
    v~pořadí logických čísel procesorů). Pro koordinaci
    se využívají zprávy nulové délky s~kódem mtype=1 */

    if (mynode == 0)  printf ("\nLokální vektory :\n");
    printf ("\nUzel : %d  ", mynode);
    for (j=0; j<LEN; j++)
      printf ("  %d", vec[j]);
  seqend ();  /* konec sekvenční části */

  nimaxn (vec, LEN, 0, 1, -1);
    /* nalezení vektoru maximálních hodnot - LEN je délka,
    výsledkem se přepíše vec[] v procesoru 0, pro koordinaci
    se (skrytě) použijí zprávy s kódem 1,
    maska -1 (samé  jedničky)  znamená provedení operace
    ve všech prvcích aktuální krychle */
```

```
  /* výstup výsledků z procesoru 0 */
  if (mynode == 0) {
    printf ("\n\n Max      :");
    for (j=0; j<LEN; j++)
      printf ("  %d", vec[j]);
   }
}
```

Poznámka na závěr:

Struktura programového vybavení pro systém nCUBE je lehce komplikovaná. Je třeba si uvědomit, že jde o kompozici dvou unixovských programových systémů (v pracovní stanici a v procesorech – prvcích procesorové sítě nCUBE). Vytvářené aplikační programy mohou běžet (a spolupracovat) v obou komponentách systému. Některé funkce potřebné pro realizaci paralelního výpočtu tudíž mohou být využívány v obou komponentách, jmenují se stejně (např. `nwrite()`, `rexec()`), ale jsou v různých knihovnách (jednou přeloženy do cílového kódu pracovní stanice, podruhé do kódu procesorů nCUBE). Podobně i některé utility **Unix**u (typicky samostatné programy vyvolávané z interpretu příkazů – zde např. `ls`, `cp`, `cat`) existují ve dvou verzích – jedné „normální“ určené pro výpočet v pracovní stanici a druhé „speciální“, určené pro zavedení a výpočet v procesorech sítě nCUBE.

Kapitola 6

High Performance Fortran

Hlavní potíží při využívání paralelních počítačů je nutnost rozsáhlých úprav programů při jejich přenosu do paralelního prostředí. Navíc výsledek nebývá vždy uspokojivý, zejména z hlediska počítačové nezávislosti. Tyto obtíže jsou dnes hlavním důvodem omezeného využívání paralelních počítačů a představují omezení i pro blízkou budoucnost.

High Performance Fortran (**HPF**) je prvním pokusem o standardizaci vyššího programovacího jazyka vhodného pro dostatečně širokou třídu paralelních počítačových architektur. Hlavním záměrem tvůrců standardu bylo zpřístupnění prostředků pro přenositelné vyjádření a využití paralelizmu běžným aplikačním programátorům bez nutnosti zacházení s podrobnostmi na úrovni architektury cílového počítače. Dále byla respektována poptávka po možnostech přenosu stávajících, převážně fortranských, programů pro náročné vědecké výpočty do paralelního prostředí. Vedle toho pro konečnou volbu jazyka **Fortran** jako výchozího jazyka ještě přispěly zkušenosti z již uskutečněných rozšíření a svým způsobem spíše jednoduchá struktura jazyka **Fortran** samotného.

6.1 Historie HPF

Za počátek diskuzí o **HPF** je považována schůzka zájemců a výrobců na výstavě Supercomputing '91. Na této schůzce byla založena skupina High Performance Fortran Forum (HPFF), která zahájila práce počátkem roku 1992. Aktivní práce v HPFF se zúčastnili zástupci asi 40 subjektů, většinou výrobců paralelních počítačů nebo univerzitních vývojových laboratoří. Velký důraz byl kladen na rychlý postup prací, výsledkem byl návrh **HPF** Specification v. 1.0 z května 1993, v listopadu 1994 následovala verze 1.1, nejnověji pak v listopadu 1996 verze 2.0 δ.

Zatím je asi předčasné vyvozovat nějaké závěry o úspěchu či neúspěchu práce HPFF. Některá rozšíření **HPF** byla zahrnuta do standardu Fortran 95, trh s kompilátory **HPF** je dostatečný a zahrnuje podporu většiny paralelních platforem. Díky včasnému začátku má **HPF** trochu náskok před obdobnými rozšířeními pro další jazyky (Parallel C apod.), významný je také fakt, že HPFF je sdružení celosvětové. Na straně druhé, kompilátory **HPF** a pomocí nich sestavené programy vyžadují velké zdroje (paměť, výkon v pevné řádové čárce, rychlé komunikace), daleko obtížnější je ladění. Výsledné programy nebývají vždy efektivní a stabilní, využití **HPF** je zatím omezeno spíše na menší zájmové skupiny. Paralelizace některé komerční aplikace např. z oblasti výpočtové mechaniky pomocí **HPF** se zatím neobjevila, zde zatím slaví úspěch programový model SPMD realizovaný pomocí prostředků *message-passing* (PVM, MPI apod.).

6.2 Programový model HPF

Za výchozí programový model byl vybrán model SPMD, který má, zejména z pohledu programátora, tyto vlastnosti:

- výpočetní postup je popisován obvyklým jednoduchým (jednovláknovým) programem. Program může být zpracováván skupinou spolupracujících, volně synchronizovaných, abstraktních procesorů.
- proměnné programu jsou viditelné v rámci jednotného globálního prostoru jmen.
- proměnné, zejména rozměrná pole, lze distribuovat mezi více abstraktních procesorů. Konečná distribuce a implementace přístupu k proměnným pro celou skupinu procesorů je však ponechána výhradně na kompilátoru.
- operace nad částmi distribuovaných polí mohou být podle možnosti vykonávány současně různými procesory.
- synchronizace procesorů je zabezpečena pro programátora transparentním způsobem a plně ponechána na starost kompilátoru. Mimo místa synchronizace mohou procesory provádět různé instrukce.

- mapování abstraktních procesorů na fyzické je počítačově závislé a není prostředky modelu řešeno.

Takto upřesněný programový model SPMD bývá zvykem označovat jako programový model HPF.

Vedle podpory programového modelu HPF, postupovalo HPFF ještě podle dalších záměrů doplňujících:

- orientace návrhu jazyka pro architektury SIMD a MIMD s nerovnoměrným přístupem do paměti
- poskytnutí prostředků pro výhodné přizpůsobení programu podle vlastností architektury cílového počítače
- zajištění prostředků pro vývoj vysoce portabilních programů a pro usnadnění přenosu již existujících programů
- přednostně využít již existující standardy
- umožnění snadné a efektivní implementace kompilátoru, vytvoření podmínek pro postupné ověřování nových možností i mimo HPFF
- návrh prostředků pro styk s jinými programovacími jazyky

Požadavky na vlastnosti datově paralelních prostředků byly při návrhu nového jazyka stanoveny podle zkušeností s reálnými aplikacemi z oblasti náročných výpočtů a s ohledem na podporu efektivní implementace paralelizmu. Podle toho byly při dalším návrhu sledovány především tyto okruhy:

- kontrola rozdělení dat. Paralelní algoritmus nemůže být dostatečně efektivní, pokud není možné zajistit dostatečnou lokalitu dat pro jednotlivé procesory. Přístupem k nelokálním datům může docházet k velkým časovým ztrátám při výměně dat mezi procesory. Algoritmus, který nevykazuje dostatečnou lokalitu dat, není vhodný pro model datového paralelizmu.

- operace nad vícerozměrnými maticemi dat, na jejichž zpracování je jazyk **Fortran** již tradičně zaměřen. Elementární operace nad maticemi, prováděné opakováním jednoduché operace na jednotlivé prvky (součet, negace apod.) bez ohledu na pořadí, dovolují velmi snadné uplatnění paralelizmu. Dále je třeba dát k dispozici prostředky pro tyto operace:

  - práce s pravoúhlými výseky matic. Zde stačí prostředky pro vyjádření výseku, výsledkem je matice.
  - práce s neregulárními (nepravoúhlými, nesouvislými) oblastmi matic. Tento požadavek lze uspokojit podmíněnými operacemi podle maticové masky.
  - vyjádření redukčních operací pro matice, např. součet prvků matice. Obvyklý způsob provedení těchto operací může být pro distribuovaná pole značně neefektivní.
  - operace pro regulární přesun prvků v matici (rotace řádků apod.)
  - prefixové resp. postfixové operace, při kterých jsou matice zpracovávány např. po řádcích a nové hodnoty prvků matice vznikají na základě příspěvků od předchozích resp. následujících prvků v řádku.
  - operace pro nepřímé adresování prvků matice, které jsou nutné pro nestrukturované přesuny prvků.
- další prostředky pro explicitní vyjádření paralelizmu.

Ukázalo se, že většinu těchto požadavků splňuje v roce 1992 nově přijatý standard jazyka Fortran 90. Bylo proto rozhodnuto provést návrh **HPF** jako rozšíření k jazyku Fortran 90, která lze rozdělit zhruba do těchto skupin:

- direktivy pro *mapování* dat rozdělením na části lokálně přístupné abstraktními procesory
- nové paralelní konstrukce
- zavedení dalších *intrinsických* funkcí[1], umožňujících větší úroveň abstrakce
- dodefinování nových atributů pro styk s jinými programovacími modely a jazyky
- direktivy podporující (v případě potřeby) sekvenční charakter globálních dat

[1]Z angl. *intrinsic*, někdy bývají tyto funkce označovány jako *vnitřní* nebo *zabudované*.

Pro rychlejší přijetí **HPF** širší programátorskou veřejností a k rychlému vytvoření trhu s dostupnými kompilátory byla současně vytvořena specifikace podmnožiny **Subset HPF**, která nevyžaduje implementaci obtížných konstrukcí a přitom příliš neomezuje použití paralelního programového modelu `HPF`[2].

Při návrhu syntaktických pravidel pro rozšiřující specifikace a příkazy bylo postupováno tak, aby byla co nejméně dotčena možnost kompilace programu obvyklým kompilátorem jazyka Fortran 90.

Rozšiřující nevýkonné direktivy **HPF**, sloužící převážně ke specifikaci mapování dat ve formě atributů, původní syntaktický ani sémantický význam programu neovlivňují vůbec, jejich zápis má tvar zvlášť označené celořádkové poznámky podle pravidel jazyka Fortran 90. I když možností se nabízí více, přidržíme se nejčastěji užívaného označení řetězcem `!HPF$`, kterým musí, až na nevýznamné úvodní mezery a tabelátory, řádek s direktivou **HPF** začínat. Při nedostatku místa lze zápis direktivy jako obvykle[3] dočasně přerušit znakem `&` a pokračovat na dalším řádku, který musí opět začínat, jinak dále bezvýznamným, řetězcem `!HPF$`[4]. Obvyklé příkazy a poznámky mezi pokračovacími řádky direktiv **HPF** nelze používat.

Nové výkonné příkazy a vnitřní funkce vyžadují doplnění syntaktických a sémantických pravidel[5], byla však dodržena sémantická shoda mezi jejich seriovým a paralelním provedením. Zároveň lze očekávat, že nové příkazy budou s velkou pravděpodobností zahrnuty do dokončovaného standardu jazyka Fortran 95.

6.3 Distribuce dat

Hlavní představu programového modelu **HPF** lze jednoduše popsat ve dvou krocích. Před paralelním zpracováním velkého masívu dat jsou tato data nejdříve rozdělena na disjunktní části a následně je pro obvyklé

[2]Další vývoj ovšem ukázal „zdatnost“ výrobců kompilátorů, téměř ve všech dostupných kompilátorech je implementován větší rozsah funkcí než vyžaduje **Subset HPF**.

[3]Tím se rozumí: podle syntaxe jazyka Fortran 90.

[4]Jde svým způsobem o výjimku, na obvyklém pokračovacím řádku jazyka Fortran 90 je v rozpracovaném příkazu pokračováno plynule.

[5]Jinak by bylo asi nutné využít direktivy pro podmíněný překlad alternativních částí programu, tento způsob však mj. značně snižuje přehlednost.

sériové zpracování každé části určen některý z dostupných procesorů. Jednotlivé procesory přistupují ke svěřené části dat lokálně, pokud potřebují jiná data než lokální, musí je vyžádat komunikací. Mechanismus dvojího přístupu k datům, lokálního a vzdáleného, je však pro programátora úplně transparentní a plně ponechán na starost kompilátoru. Při nevhodném umístění dat však může časově náročná meziprocesorová komunikace a synchronizace zmenšit nebo zcela znehodnotit úspory paralelním zpracováním. Pro složitost problému je nasnadě, že je třeba vytvořit prostředky pro vyjádření vhodné lokality dat. Programátor pak může, při znalosti detailů řešeného problému, navrhnout kompilátoru vhodné rozdělení dat s ohledem na co největší omezení komunikace. V **HPF** jsou tyto požadavky zajištěny direktivami pro *distribuci* dat. Jde o velmi „populární" možnost **HPF**, není však správné označovat ji jako nejdůležitější.

Direktiva pro distribuci dat má obecně jeden z těchto tvarů[6] :

`!HPF$ DISTRIBUTE` *jméno*`(`*seznam-distribucí*`)`
nebo
`!HPF$ DISTRIBUTE (`*seznam-distribucí*`)` :: *jméno, ...*

kde *jméno* značí proměnnou typu pole. Položky seznamu *seznam-distribucí* určují po řadě způsob distribuce vyjmenovaných proměnných odděleně v každém jejich rozměru. K distribuci jsou využity všechny dostupné abstraktní procesory.

Jednotlivé způsoby distribuce jsou uvedeny dále. Pro stručnost budeme používat termín část pole náhradou za podmnožinu pole, určené podmnožinou indexového prostoru, souvislá část pole pak odpovídá souvislé podmnožině indexů. Bližší určení abstraktních procesorů zatím ponecháme stranou, rovněž se také nebudeme zabývat „topologickými" problémy, tj. vztahem mezi tvarem (počtem dimenzí) distribuovaného pole a prostorovým uspořádáním procesorů. Všechny tyto okolnosti lze ponechat na starost kompilátoru, explicitní možnosti budou uvedeny později.

6.3.1 Bloková distribuce

Toto rozdělení přiděluje každému procesoru stejně velkou, souvislou část pole – samozřejmě, poslední úsek může být trochu menší. Bloková dis-

[6] Dále použitý symbol : : není definičním symbolem Backus–Naurovy notace, nýbrž konstatním symbolem, který je součástí syntaxe jazyka Fortran 90.

tribuce je v seznamu distribucí specifikována slovem `BLOCK`, jako příklad uveďme distribuci jednorozměrného pole `A` o 16 prvcích:

```
      REAL, DIMENSION(16) :: A
!HPF$ DISTRIBUTE(BLOCK) :: A
```

Pokud budou k dispozici 4 procesory, budou prvky `A(1), ..., A(4)` lokálně přístupné z prvního procesoru, další čtyři z druhého atd.. Obdobně, dvourozměrné pole `B` lze distribuovat pomocí direktivy:

```
      REAL, DIMENSION(16, 24) :: B
!HPF$ DISTRIBUTE(BLOCK, BLOCK) :: B
```

U vícerozměrných polí lze distribuci v některé dimenzi vynechat, takovou distribuci nazýváme *degenerovanou* nebo *kolabovanou*. Nedistribuovaná dimenze se vyznačí znakem `*` a z hlediska distribuce dat není tato dimenze dále uvažována. Tímto způsobem můžeme pole distribuovat po celých řádcích, sloupcích apod.. Distribuci po celých „listech“ ukazuje příklad:

```
      REAL, DIMENSION(1000, 1000, 1000) :: C
!HPF$ DISTRIBUTE(*, *, BLOCK) :: C
```

kdy bude, opět pro 4 procesory, prvních 25 listů umístěno u prvního procesoru, dalších 25 listů u druhého atd..

Je možné velikost distribuovaných bloků přímo stanovit:

```
      REAL, DIMENSION(16) :: D
!HPF$ DISTRIBUTE(BLOCK(5)) :: D
```

Zde jsou distribuovány bloky po 5 sousedních prvcích, na pátý a případně další procesory už nedojde. Velikost bloku nesmí být menší než hodnota, zajišťující pro daný počet procesorů distribuci všech dat. V našem případě by tedy při 4 procesorech nešlo pole `D` distribuovat jako `BLOCK(3)`.

Bloková distribuce je výhodná při řešení problémů pomocí prostorové dekompozice (výpočtová mechanika, výpočtová mechanika tekutin, kvantová chromodynamika), kdy většina výpočetních operací vyžaduje hodnoty z několika málo sousedních výpočtových bodů. Při blokové distribuci jsou takové výpočty většinou lokální, meziprocesorová komunikace je vyžadována pouze na hranicích oblastí.

6.3.2 Cyklická distribuce

Při tomto způsobu, určeném slovem `CYCLIC`, jsou prvky pole přidělovány po jednom na cyklicky uspořádanou množinu procesorů:

```
      REAL, DIMENSION(16) :: A
!HPF$ DISTRIBUTE(CYLIC) :: A
```

Při 4 procesorech, jsou na druhém procesoru lokálně přístupné prvky `A(2), A(6), A(10), A(14)`.

Obdobně jako pro blokovou distribuci, tak i pro distribuci cyklickou lze použít modifikaci degenerovanou a s určením velikosti (počtem prvků) dělení:

```
      REAL, DIMENSION(1000, 1000, 1000) :: C
!HPF$ DISTRIBUTE(*, *, CYCLIC(128)) :: C
```

Velikost cyklického dělení však nepodléhá, na rozdíl od blokové distribuce, žádnému omezení.

Cyklická distribuce je vhodná pro problémy mající periodický charakter, jako typický příklad lze uvést algoritmy pro zpracování signálu. S výhodou lze někdy cyklickou distribuci použít také tehdy, pokud je požadováno co nejdelší rovnoměrné rozdělení zátěže mezi procesory.

6.3.3 Poznámky a shrnutí

Je třeba doplnit, že direktivy pro distribuci dat nejsou pro kompilátor závazné, jsou pouze návodem k jeho práci. Pokud by kompilátor našel nějaké vhodné rozdělení sám, může direktivy úplně ignorovat. Tato volnost je velmi aktuální pro případ, kdy je počet procesorů znám až za běhu programu. Bez ohledu na navrhované rozdělení, může kompilátor mnohdy zvolit nejjednodušší řešení velmi časově náročnou globální replikací všech proměnných na všech procesorech.

Direktivy v rámci jedné programové jednotky pracují nad stejnou množinou procesorů, při použití direktiv:

```
      REAL, DIMENSION(16) :: D
!HPF$ DISTRIBUTE(BLOCK(5)) :: D
      REAL, DIMENSION(16) :: A
!HPF$ DISTRIBUTE(CYLIC) :: A
```

je při 4 procesorech druhý 5–prvkový blok pole `D` lokálně přístupný ze stejného procesoru jako prvky `A(2), A(6), ...`.

Implicitní distribuce jednoduchých proměnných a polí, která nebyla distribuována explicitně, není standardem **HPF** stanovena. Lze předpokládat, že výrobci kompilátoru budou v tomto případě většinou volit mezi replikovanou distribucí kopií proměnných na všech procesorech (např. Digital **HPF**) nebo umístěním na jednom procesoru. Jak zabránit implicitní distribuci bude ukázáno později.

Hlavní nevýhodou výše uvedených způsobů distribuce dat, je jejich omezení na regulární oblasti. Novější verze **HPF 2.0** δ některá z těchto omezení odstraňuje.

6.4 Paralelní příkazy HPF

Dalším hlavním úkolem HPFF byl návrh jazykových konstrukcí pro jednoduché vyjádření operací, které mohou být v případě dostupnosti prostředků prováděny paralelně. Pro dosažení co největší přenositelnosti byly především využity velmi dobré možnosti, které pro tento případ nabízí již Fortran 90.

6.4.1 Maticové operace, příkaz WHERE

Maticovými operacemi zde budeme rozumět provedení nějaké jednodušší operace nad všemi prvky matice příp. dvojicemi prvků matic apod., bez předem stanoveného pořadí výběru prvků. Poprvé (z hlediska vývoje jazyka **Fortran**) byly maticové operace zahrnuty do standardu Fortran 90 jako významný prostředek pro tvorbu vědeckých aplikací a pro možnost uplatnění optimalizace maticových operací na některých typech počítačů (např. vektorových).

Je přirozené, že tyto maticové operace byly plně zahrnuty do návrhu **HPF**, protože při vhodné lokalitě dat představují nejjednodušší možnost pro uplatnění paralelizmu. Navíc, ze syntaktického i sémantického hlediska je jejich použití velmi jednoduché:

```
REAL :: S
REAL, DIMENSION (N) :: A, B
INTEGER, DIMENSION (N) :: IP
```

```
...
A = SQRT(A) + 2.0 * B + S  ! elementální použití SQRT
A = B(IP)                  ! nepřímé indexování
```

Velmi významnou možnost představují prostředky pro práci s obdélníkovými výseky matic. Pomocí trojice *začátek:konec:krok* lze z dané matice vyčlenit požadovanou podmatici, lze rovněž použít vektorové indexy[7]:

```
REAL, DIMENSION (10, 10) :: A, B
INTEGER, DIMENSION(5), PARAMETER ::  &
                       SUDE = (/(I, I = 2,10,2)/)
...
A(2 : I, 1:8:2) = B(3 : I + 1, 2:9:2)
A(SUDE - 1,:) = TRANSPOSE(B(:, SUDE))
```

Vedle tzv. *elementálního* použití intrinsických funkcí na matice, tj. aplikace funkce na každý prvek matice, obsahuje Fortran 90 další užitečné vnitřní funkce pro operace s maticemi, které jsou samozřejmě i součástí **HPF**:

- funkce `TRANSPOSE` pro transpozici matice, `CSHIFT` a `EOSHIFT` pro přesuny podél jednotlivých os;
- redukční funkce `SUM, PRODUCT, MAXVAL, MINVAL, COUNT` pro aritmetické operace (součet, součin atd.) a redukční funkce `ALL, ANY` pro logické operace;
- informační funkce `SHAPE, LBOUND, UBOUND, SIZE, ALLOCATED`;
- funkce pro různé způsoby přiřazení hodnot prvkům matice `MERGE, SPREAD, RESHAPE, PACK, UNPACK`;
- funkce pro operace lineární algebry `DOT_PRODUCT, MATMUL`;

Pro velmi nestrukturovaný přístup je určen příkaz `WHERE`, který umožňuje podmíněný výběr prvků matice podle masky:

```
REAL, DIMENSION (100) :: A, B
LOGICAL, DIMENSION (100) :: L
...
```

[7]Všimněte si umístění příkazu na dvou řádcích. Pokračování příkazu je nutné vyznačit koncovým `&`.

```
WHERE (A  /= 0.0) B = 1.0 / A    ! příkaz  WHERE
...
WHERE (L)                        ! konstrukt WHERE
   A = B
   B = B / 2.0
ELSE WHERE
   A = 0.0
END WHERE
```

6.4.2 Příkaz FORALL

Vedle převzatých maticových operací, nabízí **HPF** příkaz `FORALL`, poskytující možnost paralelizmu na jemnější úrovni. Motivaci pro zavedení tohoto dalšího paralelního příkazu lze trochu objasnit pokusem explicitního vyjádření paralelizmu v této smyčce:

```
REAL, DIMENSION (100, 100) :: A, B
INTEGER I, J
...
DO I = 2, N
  DO J = 1, I - 1
    A(I, J) = A(I, J) / A(I, I)
  END DO
END DO
```

Obtíž spočívá v tom, že běžnými prostředky zatím neumíme vymezit trojúhleníkovou část matice jinak než použitím složitějších prostředků pro neregulární podoblasti matice (pro náš příklad se nabízí např. použití příkazu `WHERE`). Trochu předběhneme a ukážeme řešení pomocí nového příkazu:

```
FORALL (I = 2:N, J = 1:N, J.LT.I) &
                A(I,J) = A(I, J) / A(I, I)
```

Podle uvedeného příkladu by se mohlo zdát, že příkaz `FORALL` nahrazuje smyčku. Toto zdání je však velmi chybné, příkaz `FORALL` nepoužívá vůbec pojem iterace, nýbrž představuje *zobecněný přiřazovací příkaz* pro podmnožiny matic bez předem stanoveného pořadí pro zpracování jednotlivých prvků. Příkaz `FORALL` je dovoleno použít jen tehdy, jsou-li zaručeny identické výsledky jak při sériovém tak paralelním provedení.

Podobně jako u příkazu WHERE má i příkaz FORALL dvě podoby: příkaz a konstrukt. Konstrukt FORALL lze ekvivalentně přepsat jako posloupnost příkazů FORALL.

Příkaz FORALL má tento syntaktický zápis:

příkaz-forall	::	FORALL <*hlavička*> *přiřazovací-příkaz*
hlavička	::	(<*seznam-trojic-indexů*>[,*skalární-logický-výraz*])
trojice-indexů	::	*jméno-indexu = dolní-mez : horní-mez*[:*krok*]

Je třeba zdůraznit, že výkonná část příkazu FORALL je tedy omezena pouze na přiřazení.

K těmto syntaktickým pravidlům patří ještě omezení pro zajištění korektního paralelního zpracování:

1. Při vícečlenném seznamu indexových trojic, nesmí výraz pro mezní hodnoty ani pro hodnotu krokovou odkazovat na jména indexů v rozsahu celého seznamu. Již uvedený příkaz

   ```
   FORALL (I = 2 : N, J = 1 : N, J.LT.I)    &
                A(I,J) = A(I, J) / A(I, I)
   ```

   nelze nahradit zdánlivě jednodušším tvarem

   ```
   FORALL (I = 2 : N, J = 1 : I - 1)         &
                A(I,J) = A(I, J) / A(I, I)
   ```

2. pokud je na levé straně přiřazovacího příkazu použito volání funkce, musí to být funkce s atributem PURE, který bude vysvětlen později.

3. je nutné vyloučit vícenásobné přiřazení pro jednu jednoduchou proměnnou. Obě následující použití příkazu FORALL jsou tedy chybná:

   ```
   INTEGER, DIMENSION (N) :: IP
   DOUBLE PRECISION, DIMENSION (N) :: A, B
   REAL :: X
   ...
   IP(1) = 1
   IP(2) = 1
   ...
   FORALL (I = 1:N) A(IP(I)) = B(I)
   ...
   FORALL (I = 1:N) X  = A(I)
   ```

Konstrukt `FORALL` je posloupnost příkazů přiřazení, `FORALL` a `WHERE` uzavřených mezi hlavičku příkazu `FORALL` a terminál `END FORALL`. Při zběžném pohledu nejde vlastně o nic nového, možnost použití vnořených příkazů `FORALL` a `WHERE` však činí celou věc velmi zajímavou:

```
FORALL (I = 1:8)
  A(I, I) = SQRT(A(I,I))
  FORALL (J = I-3:I+3, J /= I .AND. J >= 1 .AND. J <= 8)
    A(I, J) = A(I, I) * A(J, I)
  END FORALL
END FORALL
```

Vyhodnocení příkazu `FORALL` probíhá postupně takto:

1. Podle trojic indexů je vyhodnocena platná množina indexů.
2. Podle logické podmínky je vyhodnocena aktivní množina indexů.
3. Pro množinu aktivních indexů jsou s možností paralelního zpracování vyhodnoceny pravé strany.
4. Je provedeno přiřazení pravých stran levým opět s případným uplatněním paralelizmu.

Mezi těmito fázemi vyhodnocení příkazu `FORALL` a na jeho konci je prováděna implicitní meziprocesorová synchronizace, uvnitř jednotlivých fází není pořadí výpočtů nijak stanoveno. Konstrukt `FORALL` je vyhodnocen jako posloupnost příkazů `FORALL`.

Rozdíl mezi příkazem `FORALL` a iterační smyčkou lze ukázat na jednoduchém příkladu:

```
! varianta s DO
      DO I = 2, N
         A(I, I) = A(I - 1, I - 1)
      END DO
! varianta s FORALL
      FORALL(I = 2:N)   A(I, I) = A(I - 1, I - 1)
```

Po provedení varianty s `DO` bude celá diagonála matice `A` obsazena hodnotou prvku `A(1, 1)`, provedení varianty s `FORALL` diagonálu posune směrem k pravému dolnímu rohu. Při detailním pohledu najdeme ještě jednu odlišnost – po ukončení příkazu `FORALL` není, na rozdíl od smyčky `DO`, hodnota proměnné `I` definována.

Konstrukt `FORALL` není součástí **Subset HPF**, u dostupných kompilátorů bývá bez omezení implementován až ve vyšších verzích a pro výkonnější počítačové platformy. Obě modifikace `FORALL` jsou však součástí připravovaného standardu Fortran 95.

6.4.3 Direktiva INDEPENDENT

Direktiva `!HPF$ INDEPENDENT` může být uvedena na samostatném řádku jedině před smyčkou `DO` nebo příkazem resp. konstruktem `FORALL`, takto modifikované příkazy nazýváme `INDEPENDENT DO` resp. `INDEPENDENT FORALL`. Pomocí této direktivy může programátor předat kompilátoru dodatečné informace o možnosti paralelizace následujícího příkazu i v situaci, kdy paralelizmus není úplně zřejmý anebo je požadována znalost okolností, velmi často aktuálních hodnot proměnných, za běhu programu. Použití této direktivy[8] je omezeno na případy, kdy je programátorem zaručena sémantická shoda při sériovém a paralením provedení. Pokud je tato shoda porušena, není použití direktivy korektní a je na programátorovi, aby vyšetřil, proč nebyly dodrženy předpovězené podmínky.

Z hlediska sémantického jsou příkazy `INDEPENDENT DO` a `INDEPENDENT FORALL`[9] velmi podobné a jsou prováděny ve stejných etapách jako příkaz `FORALL`, avšak s podstatným omezením synchronizace. Po vyhodnocení platné množiny indexů je až na dodržení pořadí příkazů pro jeden index[10] odstraněna implicitní synchronizace jednotlivých etap výpočtu pro různé hodnoty indexu. Například při velmi rozdílné časové náročnosti výpočtu pravých stran (v závislosti na indexu), může být v jinak jednoduché smyčce

```
!HPF$ INDEPENDENT
      DO I = 1, N
         A(I) = PURE_FUNA(I)
```

[8] Direktiva `INDEPENDENT` je vlastně jediná výkonná direktiva.

[9] Každý příkaz `INDEPENDENT FORALL` lze nahradit pomocí `INDEPENDENT DO`, opak není, pro bohatší možnosti příkazů ve smyčce, pravdou.

[10] U obvyklé smyčky tedy pořadí příkazů během jedné iterace. V zájmu co nejvyšší paralelizace by šlo mnohdy uvolnit i toto pořadí, **HPF** to však neumožňuje.

```
      C(I) = PURE_FUNB(I)
    END DO
```

v některém okamžiku prováděn výpočet pro A(2) a zároveň již pro C(3).

Pro posouzení správného použití direktivy `INDEPENDENT`, je vhodné uvažovat ideální podmínky pro paralelizaci, tedy dostupnost procesorů v počtu přesahujícím mohutnost paralelní operace[11]. Z těchto poměrů lze odvodit velmi důležité omezení pro použití direktivy `INDEPENDENT`:

> *atomický datový prvek, do kterého bylo alespoň jednou zapisováno, nesmí být použit pro čtení ani zápis při jiné hodnotě indexu*

Toto omezení je porušeno např. v těchto případech:

```
    DO I = 2, N - 1
      A(I) = A(I+1)
    END DO

    DO I = 1, N
      S = S + A(I) * B(I)
    END DO
```

Výše uvedené omezení by vylučovalo, pokud nebudou poskytnuty dodatečné prostředky, např. použití vnořených smyček v příkazech `INDEPENDENT DO`, neboť iterační proměnná vnořené smyčky je zapisována pro každý index nadřazeného příkazu. Pro překonání této nežádoucí vlastnosti, lze příkaz `INDEPENDENT DO` rozšířit pomocí klauzule `NEW` o seznam proměnných, jejichž životnost je omezena na jednu iteraci smyčky, avšak majících schopnost neustálého „znovuzrození“. Jinými slovy, proměnné ze seznamu `NEW`, jsou lokálními proměnnými v rozsahu jedné iterace smyčky. Oprávněnost použití proměnné v seznamu `NEW` je možné prokázat ověřením nezávislosti výsledků smyčky na hodnotách této proměnné po ukončení jednotlivých iterací. Použití klauzule `NEW` je vidět z těchto jednoduchých příkladů:

```
!HPF$ INDEPENDENT, NEW(J)
      DO I = 1, N
```

[11] Mohutností paralelní operace rozumíme počet základních operací nad již dále nedělitelnými prvky datového objektu (např. prvky matice). V nejjednodušších případech je mohutnost paralelní operace rovná mohutnosti množiny aktivních indexů.

```
         DO J = 1, N
            A(I, J) = A(I, J) + ...
         END DO
      END DO
```

případně

```
!HPF$ INDEPENDENT, NEW(TEMP)
      DO I = 1, N
         TEMP = C(I)
         C(I) = D(I)
         D(I) = TEMP
      END DO
```

Druhý z uvedených příkladů lze bez použití `INDEPENDENT DO` přepsat např. takto:

```
      TA = C ; C = D ; D = TA
```

kde jsou všechny pomocné proměnné realizovány pro každou iteraci odděleně a uloženy do pomocného pole `TA` stejného rozměru jako pole `C`. Použití `INDEPENDENT DO` s klauzulí `NEW` má sémanticky úplně stejný význam, kompilátor má však možnost realizovat lokální proměnné daleko úspornějším způsobem – pro každý zúčastněný procesor stačí vytvořit jednu lokální instanci. Klauzule `NEW` však nemůže odstranit problémy s redukčními proměnnými, např. při postupné akumulaci mezivýsledků (součet prvků pole apod.).

Implementace direktivy `INDEPENDENT` je poměrně náročná a není zahrnuta do specifikace **Subset HPF**.

6.4.4 Atribut PURE

Atribut `PURE` není sám o sobě paralelním příkazem, přispívá však výrazně k obecnosti příkazu `FORALL` tím, že zajišťuje možnost označení funkcí bezpečně použitelných uvnitř tohoto příkazu. Tak je možné odstranit obtíže vyplývající z omezení příkazu `FORALL` na pouhé přiřazení, neboť pro výpočet pravých stran lze použít funkci, zakrývající jinak nedovolené podrobnosti (logické příkazy, iterační výpočty apod.). Obvykle se `PURE` funkce charakterizují jako funkce, které nevykazují žádné tzv. vedlejší efekty.

Tyto funkce nesmějí provádět I/O operace, musí mít pouze vstupní argumenty a při opakovaném použití se stejnými hodnotami argumentů musí dávat vždy stejné výsledky. Uvedeme případ funkce, která takovým pravidlům nevyhovuje:

```
      FUNCTION NON_PURE(X) RESULT(F)
        REAL, INTENT(in) :: X
        REAL :: F
        REAL :: Y
        COMMON /WIZARD/ Y
        ...
! nedovolený přístup ke globálním datům
        Y = Y + 1.0
! výsledek je závislý na celkovém počtu volání
        F = F + Y
      END FUNCTION NON_PURE
```

Všechny vnitřní funkce **HPF** jsou implementovány s atributem `PURE`.

6.4.5 Příklad: trojúhelníkový rozklad matice

Podle vět lineární algebry lze každou regulární matici rozložit na součin dvou trojúhelníkových matic $A = LU$, kde L je spodní trojúhelníková matice s jednotkovou diagonálou, U je horní trojúhelníková matice. Tato dekompozice se nazývá trojúhelníkový rozklad, zkráceně LU-rozklad, a je účinným prostředkem při opakovaném řešení velkých soustav lineárních rovnice pro více pravých stran.

Pokud pro jednoduchost odhlédneme od problémů numerické stability a použijeme postup bez pivotizace, lze sekvenční LU-rozklad matice A na „místě“ provést tímto programem v jazyce Fortran 90:

```
      PROGRAM LU_ROZKLAD
      IMPLICIT NONE          ! zákaz explicitního typování

      INTEGER, PARAMETER :: N = 1000
      REAL, DIMENSION(N, N) :: A
      INTEGER :: I, J, K

!     inicializace matice A
```

```
      ....

      DO  K = 1, N - 1
!       sloupcová normalizace
        DO I = K + 1, N
          A(I, K) = A(I, K) / A(K, K)
        END DO
!       modifikace čtvercových submatic na diagonále,
!       submatice se postupně zmenšují od celé matice
!       až na jednoprvkovou matici v pravém dolním rohu
        DO I = K + 1, N
          DO J = K + 1, N
            A(I, J) = A(I, J) - A(I, K) * A(K, J)
          END DO
        END DO
      END DO

!     další výpočty
      ...

      END PROGRAM LU_ROZKLAD
```

Po provedení těchto příkazů bude matice A transformována tak, že její čistě dolní trojúhelníková část bude překryta maticí L (jednotkovou diagonálu není třeba ukládat), zatímco horní trojúhelník (tentokrát včetně diagonály) bude překryt maticí U.

Pomocí explicitních paralelních prostředků lze výše uvedený postup přepsat bez větších potíží takto:

```
      DO  K = 1, N - 1
        FORALL (I = K + 1 : N)                             &
            A(I, K) = A(I, K) / A(K, K)
        FORALL (I = K + 1 : N, J = K + 1 : N)              &
            A(I, J) = A(I, J) - A(I, K) * A(K, J)
      END DO
```

Jediným menším problémem bylo jen rozhodnutí, jak nahradit smyčku se sloupcovou normalizací. Vedle uvedené možnosti příkazem `FORALL` se nabízí ještě sémanticky shodná náhrada pomocí maticového výrazu

```
A(K + 1 : N, K) = A(K + 1 : N, K) / A(K, K)
```

Pro opakovaný výskytu indexu `K` se zdá, že nakonec zvolený způsob usnadňuje analýzu závislostí během kompilace.

Dosud jsme provedli přepis posloupnosti příkazů pomocí nových prostředků, pro využití paralelizmu je dále nutné navrhnout vhodnou distribuci dat[12]. Při volbě způsobu distribuce je přednostním hlediskem co největší minimalizace meziprocesorové komunikace nebo řečeno jinak, zabezpečení co nejčastější lokality datových referencí. Jako pomocné kritérium může často posloužit požadavek na co největší „rozumné" vytížení všech procesorů.

Požadavky na meziprocesorovou komunikaci bývají často velmi nepřehledné, k jejich vyjasnění pomůže mnohdy stanovení těchto základních charakteristik analyzovaného výpočtu:

- orientace výpočtu, tj. postup zpracování prvků v maticích. Některé výpočty lze označit za např. řádkově orientované, kdy jsou spíše prováděny operace nad celými řádky nebo jejich částmi. Je nasnadě, že pro omezení komunikace bude v tomto případě lépe vyhovovat distribuce ponechávající řádky pohromadě, tedy degenerovaná v dalších dimenzích, např. typu `(BLOCK, *)` resp. `(CYCLIC, *)` apod..

- omezení výpočtů hodnot prvku pole na nejbližší sousedy. V tomto případě více vyhovuje bloková distribuce. V případě opačném, kdy jsou spíše požadovány hodnoty vzdálených prvků, může z důvodu lepšího a trvalejšího rozdělení zátěže vyhovovat distribuce cyklická.

Vrátíme se k našemu příkladu. LU-rozklad matice nevykazuje výraznou orientaci v žádném směru, díky sloupcové normalizaci bude trochu výhodnější řádkově degenerovaná distribuce, tj. buď `(*, BLOCK)`, nebo `(*, CYCLIC)`. Z hlediska lokality výpočtů nepřináší bloková distribuce žádné výhody. Během transformace submatic jsou výpočty vedeny ve stále menších submaticích v okolí pravého dolního rohu matice A, což by při blokové distribuci vedlo k tomu, že poslední bloky by byly zpracovávány pouze některými procesory za nečinného přihlížení ostatních. Z hlediska rozdělení zátěže je tedy výhodnější zvolit distribuci cyklickou a doplnit část specifikací našeho programu takto:

[12]Pokud ovšem nepoužíváte ideálně dokonalý kompilátor **HPF**, který by vše provedl mnohem lépe sám a vaše „nedokonalé" návrhy nebude akceptovat.

```
      INTEGER, PARAMETER :: N = 1000
      REAL, DIMENSION(N, N) :: A
      INTEGER :: I, J, K
!HPF$ DISTRIBUTION (*, CYCLIC) :: A
```

Je třeba uvést, že LU-rozklad není z hlediska paralelizace vůbec jednoduchý problém a vyvozuje každopádně velký objem komunikace. Je tedy žádoucí, aby kompilátory **HPF** zajišťovaly tuto komunikaci co nejúspornějším způsobem např. snížení režie sdružováním zpráv, vytvářením redundantních replik často čtených proměnných pro jednotlivé procesory apod..

6.5 Mapování dat

Zatím popsané možnosti **HPF** jsou naprosto nedostatečné k vývoji programů se složitými datovými strukturami, chybějí zejména prostředky pro jemnější kontrolu umístění dvou polí navzájem. **HPF** proto obsahuje vedle direktivy `DISTRIBUTE` další prostředky pro tzv. *mapování dat*, jejichž použitím lze pro zajištění dobré efektivity výsledného paralelního výpočtu navrhnout kompilátoru vhodné umístění dat.

Umístění dat do paměti spolupracujících procesorů lze rozdělit na dvě etapy. První etapa zahrnuje formulaci požadavků na vzájemnou polohu dat navzájem, v ideálním případě by měla být navzájem související data umístěna na stejných procesorech. Prostředky této etapy tedy musí umožňovat přidružení datových struktur navzájem. Ve druhé etapě je pak třeba podle povahy řešeného problému provést umístění základních datových struktur, tedy i dat přidružených, na množinu abstraktních procesorů. Počítačově závislou etapu mapování abstraktních procesorů na fyzické již datový model **HPF** nezahrnuje a k jejímu provedení lze použít specifické prostředky cílové počítačové architektury.

Popis nových prostředků (direktiv) budeme v dalším provádět pomocí příkladů, přesné definice jsou mnohdy nepřehledné a z hlediska praktického použití zahrnují zbytečně mnoho rovnocenných variant.

6.5.1 Direktiva PROCESSORS

Tato direktiva slouží k pojmenování vícerozměrných polí abstraktních procesorů a lze jí použít pro přesnější kontrolu distribuce dat v jednotlivých dimenzích. Někdy lze pomocí této direktivy využít specifické vlastnosti koncové počítačové architektury.

Pravidla zápisu direktivy PROCESSORS jsou jednoduchá, jak ukazuje příklad:

```
      !HPF$ PROCESSORS, DIMENSION(5, 2) :: PROCARR
! ekvivalentní zápis: !HPF$ PROCESSORS PROCARR(5, 2)
      !HPF$ PROCESSORS, DIMENSION(3) :: PROCLIN
      !HPF$ PROCESSORS :: PROCSCAL
```

kde je pod jmény PROCARR, PROCARR, PROCSCAL zpřístupněno postupně dvourozměrné pole procesorů o rozměrech 5 * 2, lineární pole délky 3 a skalární procesor (lineární pole délky 1). Nyní již lze takto definované procesory použít pro distribuci dat:

```
      COMPLEX, DIMENSION(100, 100) :: A
      !HPF$ DISTRIBUTE (CYCLIC, BLOCK) ONTO PROCARR :: A
```

Pro použití direktivy PROCESSORS platí tato poměrně samozřejmá pravidla:

- počet dimenzí pole procesorů a distribuovaného pole se musí shodovat, degenerovaná distribuce v některé dimenzi se nezapočítává:

  ```
  !HPF$ DISTRIBUTE (*, BLOCK) ONTO PROCLIN :: A
  ```

- dvě pole procesorů se shodným tvarem[13] odkazují na totožné pole procesorů. Direktiva

  ```
  !HPF$ PROCESSORS PROCARR_ALIAS(5, 2)
  ```

  tedy zavádí pouze „alias“ pro pole procesorů PROCARR.

- pokud jsou dva objekty mapovány na stejný abstraktní procesor, budou lokálně přístupné z téhož fyzického procesoru.

- osamocený procesor lze použít pro data, která není třeba distribuovat:

  ```
  !HPF$ DISTRIBUTE (*, BLOCK) ONTO PROCSCAL :: A
  ```

[13] Musí být shodný jak počet dimenzí, tak rozměry v jednotlivých dimenzích.

- lze použít informace dostupné až za běhu programu:[14]

```
!HPF$ PROCESSORS                                     &
!HPF$ ALL_PROCS(NUMBER_OF_PROCESSORS() / 2, 2)
```

6.5.2 Direktiva ALIGN

Direktiva `ALIGN` slouží k umístění polí na procesory nikoliv přímou distribucí, jak je tomu u direktivy `DISTRIBUTE`, nýbrž tzv. *přidružením*, kdy je procesor pro každý prvek pole, tzv. *obrazu*, určen nepřímo procesorem prvku referenčního pole, tzv. *předlohy*[15]. Již na tomto místě uvedeme nejběžnější omezení pro použití direktivy `ALIGN` – pole na místě obrazu nelze explicitně distribuovat a nesmí být použito v žádné formě, ani jako obraz ani jako předloha, v dalších přidruženích.

Obecný tvar direktivy `ALIGN` lze ukázat několika ekvivalentními způsoby na nejjednodušším případu totožného umístění obrazů `B, C` a předlohy `A`:

```
!           obraz    předloha
   !HPF$ ALIGN B WITH    A
   !HPF$ ALIGN B(:, :) WITH A(:, :)
   !HPF$ ALIGN B(I, J) WITH A(I, J)
!              fiktivní indexy      předloha  obrazy
   !HPF$ ALIGN     (I, J)      WITH  A(I, J) :: B, C
```

Je samozřejmé, že tyto způsoby nepřináší nic nového, šly by nahradit pečlivou, ale méně přehlednou distribucí každého pole zvlášť.

Obecně lze ke specifikaci přidružení použít podmnožinu lineárních zobrazení jedné dimenze indexového prostoru obrazu do libovolné dimenze indexového prostoru předlohy. Tyto bohatší výrazové prostředky jsou už naznačeny v posledních dvou specifikacích z předchozího příkladu. Zde je vzájemná poloha obrazu a předlohy v obou dimenzích explicitně určena identickým zobrazením. K popisu zobrazení byly užity tzv. *fiktivní* indexy

[14] Snad není na škodu, že trochu předbíháme: funkce `NUMBER_OF_PROCESSORS()` vrací počet fyzických procesorů a je jednou z přidaných intrinsických (vnitřních) funkcí ke standardu Fortran 90.

[15] Možná existuje lepší český překlad, uvedené volby dostaly přednost pro zachycení jemné nesymetrie. V angličtině je vše jednodušší: *align* – *alignee* – *align-target*.

`I, J`, které jsou určeny uvedením jmen jednoduchých proměnných na místě indexů při specifikaci obrazu. Na místě specifikace indexů předlohy jsou uvedeny požadované lineární výrazy (ve výše uvedených příkladech identity) v jednom fiktivním indexu. Uveďme nejprve několik typických příkladů:

```
!  posunutí
   !HPF$ ALIGN (I, J) WITH A2(I + 1, J + 1) :: B2
!  opačné pořadí
   !HPF$ ALIGN (I, J) WITH A2(M - I, N - J) :: B2
!  transpozice
   !HPF$ ALIGN (I, J) WITH A2(J, I) :: B2
!  obecný lineární výraz
   !HPF$ ALIGN (I, J) WITH A3(3*J - 2, I) :: B3
!  totéž pomocí indexových  trojic
   !HPF$ ALIGN (I, :) WITH A3(1:3:1000, I) :: B3
```

Lineární výraz ve specifikaci předlohy musí být aritmeticky ekvivalentní výrazu `M * I + N`, kde `I` značí jméno fiktivního indexu a `M, N` celočíselné konstanty, s těmito omezeními:

- lze použít jen aritmetická znaménka `+, -, *`, lomítko nesmí být užito ani např. při dělení konstantou
- indexy v různých dimenzích je třeba označit různými fiktivními indexy
- každý fiktivní index smí být v rozsahu specifikace celé předlohy (tj. všech dimenzí) referován nejvýše jednou

Je asi namístě, uvést příklady několika nedovolených přidružení:

```
!   lomítko
!   !HPF$ ALIGN (I, J) WITH A2(I/3, J) :: B2
!   dvakrát I, musí být explicitně 2 * I
!   !HPF$ ALIGN (I, J) WITH A2(I + I, J) :: B2
!   dvakrát J
!   !HPF$ ALIGN (I, J) WITH A2(J, J) :: B2
!   nedeklarovaný fiktivní index K
!   !HPF$ ALIGN (I, J) WITH A2(K, J) :: B2
```

Zatím uvedené možnosti se týkaly spíše přidružení polí se stejným počtem dimenzí. Podobně jako u direktivy `DISTRIBUTE` lze však podle potřeby využít přidružení degenerované v některé dimenzi. V tomto případě jsou pak k prvkům předlohy přidružovány celé řádky, listy apod., výsledkem je degenerovaná distribuce obrazu. Degenerované přidružení lze vyznačit, podobně jako distribuci, znakem `*` na místě indexu obrazu, nebo označením tohoto indexu dále nepoužitým fiktivním indexem. Uveďme tři ekvalentní způsoby pro přidružení po celých řádcích:

```
!HPF$ ALIGN (:, *) WITH E(:) :: F
!HPF$ ALIGN (:, J) WITH E(:) :: F
!HPF$ ALIGN (I, J) WITH E(I) :: F
```

Někdy nastává situace svým způsobem opačná, kdy pro zpracování každého prvku jemně distribuovaného pole potřebujeme mnohonásobný přístup k nějakým, pro jednoduchost si představme spíše statickým (tj. převážně čteným), datům se všeobecnou platností. Pokud bychom se rozhodli pro jedno pevné přidružení, nevyhneme se objemné komunikaci při zpracování většiny zbývajících prvků distribuovaného pole. Pro tyto případy umožňuje **HPF** přidružování replikací, kdy je naráz provedeno přidružení potřebného počtu kopií obrazu s každým prvkem předlohy. Přidružování replikací pro nějaký index je naznačeno znakem `*` na jeho pozici[16] . Oproti přidružení degenerovanému nelze namísto `*` použít dosud nevyužitý fiktivní index, protože by byla porušena úmluva o „deklaraci“ fiktivního indexu na místě obrazu.

Pokud je tedy nutné „půjčit“ kód, skládající se ze tří skupin čísel, deseti skupinkám po třech zpravodajích v různých místech, nezbývá než zajistit v nejhorším případě deset kopií:

```
INTEGER :: K
INTEGER, DIMENSION (3) :: TABLE
INTEGER, DIMENSION (10, 3) :: ZPRAVODAJ
!HPF$ ALIGN TABLE(K) WITH ZPRAVODAJ(*, K)
!HPF$ DISTRIBUTE ZPRAVODAJ(BLOCK, *)
```

Je samozřejmé, že ve skutečnosti není nutné vytvářet kopii obrazu pro každou předlohu, ale jen pro každý procesor použitý pro distribuci předlohy. Pokud se vrátíme k našemu příkladu, mohou dvě skupinky zpravodajů,

[16]Čtenář už asi ví proč – je tím vyznačena nevýznamnost hodnot tohoto indexu, všichni budou „obsloužení“ stejně, bez rozdílu.

pokud jsou ve skutečnosti ve stejné budově, používat jednu kopii kódovacích čísel.

6.5.3 Direktiva TEMPLATE

Pomocí této direktivy lze označit abstraktní indexový prostor, jehož životnost je omezena na dobu kompilace a kterému zejména neodpovídá žádné paměťové místo. Pojmenovaný indexový prostor nemá jiný význam, než jako abstraktní předloha pro přidružování proměnných a následnou distribuci. Využití této direktivy lze však velmi doporučit zejména při řešení vzájemné polohy více polí:

```
!HPF$ TEMPLATE, DISTRIBUTE(CYCLIC),                  &
!HPF$ DIMENSION(N) :: PLAYGROUND
DOUBLE PRECISION A1(ND1, N), A2(ND2, N), A3(ND3, N)
!HPF$ ALIGN (*, :) WITH PLAYGROUND(:) :: A1, A2, A3
```

Jak je ukázáno, lze direktivu `TEMPLATE` použít v tzv. kombinovaném tvaru, kdy lze najednou určit tvar šablony i její distribuci.

6.5.4 Dynamické mapování

V předchozím textu jsme se zabývali *statickým* mapováním dat, které muselo být určeno v době kompilace programu. Tomu odpovídá také to, že příslušné direktivy `DISTRIBUTE, ALIGN` a `PROCESSORS` musely být uvedeny ve specifikační části programu.

Úplná množina jazyka **HPF** obsahuje ještě direktivy `REDISTRIBUTE` a `REALIGN`, které umožňují *dynamické* přemapování proměnných. Tyto direktivy jsou tedy výkonné a musí být umístěny ve výkonné části programu nikoliv ve specifikační. Vykonání direktivy (příkazu) `REDISTRIBUTE` zahrnuje i implicitní přemapování všech přidružených proměnných. Na rozdíl od toho direktiva `REALIGN` ovlivňuje pouze umístění vyjmenovaného objektu. Pokud je dynamické přemapování uplatněno na dynamicky alokované pole za běhu programu, je při uvedení příkazů `ALLOCATE` a `!HPF$ REDISTRIBUTE` bez přerušení společně za sebou, doporučena implementace paralelizovaného provedení obou příkazů.

Pro explicitní dynamické mapování lze použít jen proměnné s atributem `DYNAMIC`, který lze specifikovat pro určenou proměnnou buď v místě statického mapování (jako atribut direktivy `DISTRIBUTE` nebo `ALIGN`) anebo samostatně stejnojmennou direktivou.

Pro podobnost se statickým mapováním stačí vše dokreslit pomocí několika jednoduchých příkladů:

```
!HPF$ PROCESSORS PROC1(3, 3)
REAL, ALLOCATABLE(:)  :: A
!HPF$ DYNAMIC         :: A

REAL, DIMENSION(1000) :: B, C
!HPF$ DISTRIBUTE(BLOCK), DYNAMIC :: B
!HPF$ ALIGN (I) WITH B(I) :: C
...
ALLOCATE(A(M,N))
!HPF$ REDISTRIBUTE A(BLOCK, CYCLIC)
...
!HPF$ REDISTRIBUTE B(CYCLIC) ONTO PROC1
...
!HPF$ REALIGN  C(:) WITH A(:)
```

Je asi vidět, že implementace dynamického mapování, zvláště direktivy `REDISTRIBUTE` není jednoduchá. Rovněž provedení dynamických direktiv bude velmi náročné na počítačové zdroje (zásobník) a může být zdrojem velké komunikační zátěže. Z těchto důvodů není dynamické mapování zahrnuto do **Subset HPF** a možná žádný kompilátor **HPF** není v tomto místě implementován bez omezení. V případě nevyhnutelnosti lze dynamické mapování nahradit posloupností přiřazovacích příkazů, ovšem za cenu velké spotřeby paměti.

6.5.5 Příklad: násobení matic

Obvyklý algoritmus pro násobení matic `C = A * B` (pro jednoduchost pouze čtvercových)

```
C = 0

DO I = 1, N
  DO J = 1, N
    DO K = 1, N
```

```
          C(I, J) = C(I, J) + A(I, K) * B(K, J)
        END DO
      END DO
    END DO
```

neobsahuje žádný explicitní paralelismus a kompilátoru **HPF** příliš možností pro urychlení výpočtu nenabízí. Pokud chceme výpočet urychlit, je třeba provést úpravy ...– čeho vlastně ? Algoritmu či programu ?

Pokusme se podívat na celou věc z poněkud širšího hlediska a nenechme se zmást zdánlivou jednoduchostí našeho problému. Všeobecně použitelný postup lze rozdělit na tyto etapy:

1. Průzkum všech algoritmických možností pro řešení daného problému. Je zapotřebí vyřadit algoritmy bez možnosti paralelizace a podle možnosti také ty, které dovolují pouze omezený paralelizmus (neadaptovatelný na takřka libovolný počet procesorů, příliš zaměřený pouze na jednu architekturu). Pro vhodné algoritmy je zapotřebí vyhodnotit typ paralelismu (datový, úlohový), požadavky na komunikaci, apod.. Pro náš případ sice zůstaneme pro jednoduchost u klasického algoritmu, čtenář by však neměl nabýt dojmu, že tak bylo učiněno podle výpisu části programu – i zde se nabízí možností více. Zvolený algoritmus poskytuje možnost masívního paralelizmu, pro každý prvek výsledné matice lze vyčlenit naprosto samostatný procesor, vždy je však třeba zařídit přístup k jednomu řádku a k jednomu sloupci násobených matic. Poslední podmínka v sobě skrývá velké nebezpečí nadměrných komunikačních požadavků, které v tomto případě mohou zcela přesáhnout časové nároky výpočtu.

2. Ve zvoleném algoritmu najdeme místa pro explicitní použití paralelizmu a soustředíme se na jeho vyjádření programovými prostředky. V našem případě je tímto místem samozřejmě nejhlouběji vnořená smyčka s nepříjemnou redukční operací pro akumulaci součinů. K náhradě zde použijeme s výhodou vnitřní funkci `DOT_PRODUCT` pro výpočet skalárního součinu vektorů. Tato funkce je jednou z bohaté standardní výbavy jazyka Fortran 90 a lze předpokládat, že je implementována co nejefektivněji. Paralelizmus podle naší volby z předchozí etapy lze snadno vyjádřit příkazem `INDEPENDENT FORALL`:

```
!HPF$ INDEPENDENT
```

```
FORALL(I = 1 : N, J = 1 : N)                              &
        C(I, J) = DOT_PRODUCT(A(I, :), B(:, J))
```

či dvojnásobným příkazem `INDEPENDENT DO`:

```
!HPF$ INDEPENDENT, NEW(J)
DO I = 1, N
  !HPF$ INDEPENDENT
  DO J = 1, N
    C(I, J) = DOT_PRODUCT(A(I, :), B(:, J))
  END DO
END DO
```

3. Dále je třeba zajistit co největší lokalitu datových struktur a vyjasnit požadavky na komunikaci. Pokud pomineme velmi diskutabilní otázku počáteční komunikace, bude z hlediska našeho výpočtu výhodné přidružit ke každému prvku výsledné matice `C(I, J)` celý `I`-tý řádek matice `A` a celý `J`-tý sloupec matice `B`. To lze řešit replikami:

```
!HPF$ ALIGN A(:, *) WITH C(:, *)
!HPF$ ALIGN B(*, :) WITH C(*, :)
```

4. V poslední etapě lze již návrh přizpůsobit podle konkrétních podmínek (typ architektury, počet procesorů apod.) a zajistit oddělené vykonání paralelních operací. Tomu v programovém modelu `HPF` odpovídá vhodná distribuce vybraných polí. V našem případě lze zvolit téměř libovolnou distribuci matice `C` v obou dimenzích, záleží jen na počtu procesorů.

Výsledek naší práce můžeme tedy představit programem tohoto tvaru:

```
PROGRAM MATRIX_MULTIPLICATION

IMPLICIT NONE
INTEGER, PARAMETER :: N = 1000
REAL, DIMENSION (N, N) :: A, B, C
INTEGER :: I, J

!HPF$ PROCESSORS SQUARE(2, 2)
```

```
      !HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO SQUARE :: C

!     Replika řádků A(I, *) na procesory s C(I, *)
      !HPF$ ALIGN A(:, *) WITH C(:, *)

!     Replika sloupců B(*, J) na procesory s C(*, J)
      !HPF$ ALIGN B(*, :) WITH C(*, :)

      A = 1
      B = 2
      C = 0

      FORALL(I = 1 : N, J = 1 : N)          &
             C(I, J) = DOT_PRODUCT(A(I, :), B(:, J))

      END PROGRAM MATRIX_MULTIPLICATION
```

6.6 Mapování argumentů procedur

Funkce a podprogramy (pro obě možnosti budeme užívat souhrnný název procedura) jsou nezastupitelnými prostředky při modulárním návrhu programů. Při vyvolání procedury je nutné dodržet zejména stanovená pravidla pro náhradu formálních argumentů fiktivními, jinak nelze zaručit správnost dalšího běhu programu. V případě **HPF** lze očekávat další komplikace, neboť data jsou oproti běžnému jazyku vybavena dalšími mapovacími atributy a při přenosu argumentů musí být pamatováno na více možností. V dalším výkladu proto uvedeme pravidla pro mapování (distribuci i přidružování) fiktivních argumentů procedur. Je vždy dodržena hlavní zásada, která nedovoluje, aby vykonáním procedury došlo k trvalé změně mapování dat ve volajícím (nadřazeném) programu. Je zaručeno, že mapování dat bude po návratu z procedury stejné jako před jejím vyvoláním.

Způsoby mapování argumentů

Při použití procedur se můžeme setkat s těmito dvěma krajními možnostmi:

- procedura vyžaduje (např. pro dobrou efektivitu) pevně stanovené mapování některého argumentu. V tomto případě je nutné, při nesouhlasu mezi mapováním předepsaným[17] a mapováním skutečného argumentu, provést implicitní přemapování např. ihned po vstupu do procedury. Toto mapování fiktivních argumentů nazýváme *preskriptivní.*

- procedura nevyžaduje žádné speciální mapování fiktivního argumentu, umí provést výpočet s použitím přenesené distribuce. Tento způsob mapování fiktivních argumentů označujeme jako *transkriptivní.* Při transkriptivním mapování není tedy prováděno žádné implicitní přemapování skutečných argumentů.

Mezi těmito dvěma krajnostmi, první klade nároky spíše na zdroje za běhu programu, druhá na kompilátor, je ještě jedna možnost:

- mapování *deskriptivní.* Programátor se zavazuje, že data předá s mapováním jak specifikováno (popsáno) mapováním fiktivních argumentů v proceduře. Na začátku procedury bude tedy mapování skutečných argumentů stejné jako pro argumenty fiktivní.

O chvíli později uvidíme, že tento způsob lze k používání doporučit nejvíce.

K tomuto výčtu možností je ještě nutné přidat dvě poznámky o používání direktivy `PROCESSORS` v procedurách:

- jméno skupiny procesorů nemá žádný počítačový obraz, nelze ho tedy přenášet pomocí argumentů. Lze však využít ekvivalenci mezi skupinami procesorů stejného tvaru a pro přehlednost přenášet jména skupin pomocí modulů.

- procesory lze mapovat preskriptivně nebo deskriptivně, mapování procesorů se nemusí shodovat s mapováním argumentů

Je třeba přiznat, že mapování argumentů je velmi komplikované, mnohdy potenciální zájemce odradí a vyžaduje spíše vážný zájem.

Explicitní interface

I bez mapování argumentů je volání procedur a zejména zajištění správné korespondence fiktivních a skutečných argumentů vždy dost ožehavým

[17]Zapamatujte si prosím na chvíli anglický překlad: *prescribed.*

místem. Proto je v tomto případě velmi doporučeno využívání předběžných deklarací procedur, podle terminologie jazyka **Fortran** *explicitních interface* procedur, které **HPF** plně převzalo. Používání explicitních interface procedur má blahodárný vliv i z dalších hledisek a nebude asi na škodu uvést stručný popis těchto prostředků. Uživatel může poskytnout explicitní interface procedury pomocí tzv. bloku `INTERFACE`, který je, až na lokální objekty, shodný se specifikací procedury zapsané mezi omezovače `INTERFACE` a `END INTERFACE`. Jednoduchý příklad vše vysvětlí lépe:

```
INTERFACE
   REAL FUNCTION FUN(I, A, B)
   INTEGER, INTENT(INOUT) : I
   REAL, DIMENSION (:,:) :: A, B
   !HPF$ DISTRIBUTE (BLOCK, *) :: A
   !HPF$ ALIGN WITH A :: B
   END FUNCTION FUN
END INTERFACE
```

Specifikaci bloku `INTERFACE` je nutné umístit ve specifikační části programové jednotky (program, procedura, modul), tím je také vymezena jeho platnost. V jednom bloku `INTERFACE` lze umístit explicitní interface pro více procedur. Vedle toho zajišťuje kompilátor **HPF** explicitní interface pro tyto procedury:

- všechny vnitřní (zabudované) procedury
- procedury exportované z modulu. Takto je možné zajistit explicitní interface pro velkou většinu uživatelských procedur bez nějakého nadbytečného úsilí.
- lokální procedury v rámci nadřazené procedury

6.6.1 Preskriptivní mapování

Preskriptivní mapování je určeno pro ty případy, kdy programátor potřebuje zajistit pevné mapování argumentu uvnitř specifikované procedury, nejčastěji tak bývá pro zajištění efektivity výpočtu. Přítomnost explicitního interface ve volající jednotce není nutná. Pokud mapování skutečného argumentu nesouhlasí s mapováním fiktivního argumentu, je provedeno počáteční i úklidové přemapování na začátku a konci procedury implicitně.

Syntakticky je preskriptivní mapování shodné s obyklým mapováním dat a je užito v příkladu bloku INTERFACE v předchozí části. Můžeme tedy uvést trochu složitější příklad:

```
MODULE GRID_OF_PROC
  !HPF$ PROCESSORS  PROC_GRID(2, 2)
END MODULE GRID_OF_PROC

SUBROUTINE ONE_STEP(A, B)
  USE GRID_OF_PROC
  REAL, DIMENSION(:,:) :: A, B
  !HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO PROC_GRID :: A
  !HPF$ ALIGN WITH A :: B
...
END SUBROUTINE ONE_STEP
```

6.6.2 Deskriptivní mapování

Deskriptivní mapování specifikuje s jakým mapováním bude skutečný parametr do procedury předán, programátor chce vyloučit úvodní implicitní přemapování. Správné předání argumentů lze vedle úplné shody mapování skutečného a fiktivního argumentu dosáhnout ještě uvedením explicitního interface procedury ve volající jednotce. V druhém případě bude při nesouhlasném mapování fiktivního a skutečného argumentu provedeno implicitní přemapování podle explicitního interface těsně před vyvoláním a ihned po skončení procedury. Při rozdílném mapování fiktivních a skutečných argumentů a bez přítomnosti explicitního interface, není volání procedury korektní a nelze nic předvídat o dalším průběhu výpočtu.

Deskriptivní mapování je specifikováno uvedením znaku * před seznamem distribucí:

```
!HPF$ DISTRIBUTE *(BLOCK, BLOCK) ONTO *PROC :: A
!HPF$ ALIGN WITH *A :: B
```

Lze používat i kombinované způsoby, např. preskriptivní pro argumenty a deskriptivní pro procesory:

```
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO *PROC :: A
```

6.6.3 Transkriptivní mapování

Pro programátora je transkriptivní mapování zdánlivě nejjednodušší. Má nejjednodušší syntaktický zápis, např:

```
!HPF$ INHERIT A
```

Rovněž volání s libovolně mapovanými argumenty jsou korektní a nikdy není nutné provádět žádné implicitní přemapování.

Zdálo by se, že transkriptivní mapování nabízí všeobecně použitelnou možnost. Jak se však náš názor změní, pokud budeme postaveni před úkol kompilátoru uvnitř procedury! I v případě těch nejjednodušších příkazů, např.:

```
A(1) = A(2)
```

nejsou totiž po ruce žádné informace pro generaci alespoň trochu efektivního kódu a je zapotřebí zabývat se mnoha možnostmi. Asi hlavně z tohoto důvodu není transkriptivní mapování zahrnuto do **Subset HPF** a v dostupných kompilátorech téměř nikdy implementováno.

6.7 Vnitřní funkce

Jeden z hlavních rozdílů mezi jazyky typu **Fortran** a ostatními jazyky spočívá v implementaci běžných matematických funkcí, procedur pro získání informací o systému a konečně i operací I/O.

Jak známo, že např. jazyk C odsouvá řešení těchto otázek na úroveň běžných uživatelských procedur. Tím je sice dosaženo značného zjednodušení jazyka, avšak separátní provedení procedur neumožňuje využít případné specifické podmínky k optimalizovanému volání (vyřešení konstantních výrazů, inline volání).

Na rozdíl od toho poskytuje jazyk **Fortran** tyto funkce a procedury již na úrovni kompilátoru (odtud pochází i jejich pojmenování *intrinsické*, *vnitřní* nebo také *zabudované*) a je umožněna optimalizace při znalosti všech detailů programu ve zdrojovém tvaru. Podle okolností (typ skutečných argumentů, vícenásobné volání se stejným parametrem) lze pak dosáhnout generování efektivnějšího kódu. V řadě neposlední pak přímý přístup k těmto funkcím umožňuje snažší zachycení chyb již během kompilace. Je nepochybné, že popularita jazyka **Fortran** mezi programátory vědeckých aplikací je z větší části dílem koncepce vnitřních funkcí.

Pro **HPF** mají vnitřní funkce ještě větší význam, neboť lze do nich zahrnout i mnohé základní paralelní operace, které lze v programovém modelu `HPF` obvyklým postupem naprogramovat jen velmi neefektivně, (příkladem může sloužit paralelizovaný součet prvků pole). Aplikační programátor je tak osvobozen od problémů souvisejících s využitím detailů na nízké úrovni implementace paralelních výpočtů a výrobci kompilátoru je umožněna optimalizace s využitím všech vlastností cílové architektury. Toto asi chápe každý, kdo si přečetl dost zjednodušené pojednání o mapování argumentů procedur v předchozí podkapitole.

Zhruba řečeno, můžeme vnitřní procedury **HPF** rozdělit na tyto skupiny:

- syntakticky a sémanticky doplněné vnitřní procedury převzaté z jazyka Fortran 90
- systémové dotazovací funkce
- nové vnitřní procedury, které jsou zahrnuty do knihovny `HPF_LIBRARY`

V následující části se pokusíme o stručnou charakteristiku jednotlivých skupin. I když se jedná o jednu z hlavních částí **HPF**, o podrobnějším popisu na tomto místě nemůže být pro značnou rozsáhlost uvažováno.

6.7.1 Vnitřní procedury jazyka Fortran 90

HPF samozřejmě umožňuje využívat všechny tyto běžné vnitřní procedury jazyka Fortran 90:

- obvyklé numerické a konverzní vnitřní funkce lze v **HPF** používat jako elementální funkce, tj. jako funkce provedené způsobem prvek po prvku pro celé pole bez rozdílu pořadí. Příkaz

  ```
  REAL, DIMENSION (100, 100) : A
  A = ABS(A)
  ```

  je tedy ekvivaletní, případně až na pořadí, posloupnosti příkazů

```
DO I = 1, 100
  DO J = 1, 100
     A(I, J) = ABS(A(I, J))
  END DO
END DO
```

- **HPF** zahrnuje paralelní implementaci transformačních funkcí, např. `DOT_PRODUCT` či `MATMUL`, optimalizovaných pro danou počítačovou architekturu. Mezi tyto procedury patří i redukční funkce (`SUM`, `PRODUCT` apod.), jejichž použití je vysoce a výhradně doporučeno.
- dotazovací funkce na vlastnosti datových objektů, např. `PRECISION(X)` navrátí počet platných desetinných míst podle typu proměnné `X`. Velmi užitečné jsou dotazovací funkce na parametry polí, např. rozměr pole `ARRAY` v dimenzi `DIM` lze zjistit pomocí `SIZE(ARRAY, DIM)`, oproti tomu `SIZE(ARRAY)` navrátí vektor s rozměry pole `ARRAY` v jednotlivých dimenzích.
- vnitřní subroutiny jazyka Fortran 90 pro zjištění datumu a času, výpočet náhodných čísel a kopírování s bitovou přesností zdrojové a cílové adresy.

6.7.2 Dotazovací funkce HPF

Tato skupina je tvořena pouze dvěma funkcemi:

`NUMBER_OF_PROCESSORS([DIM])`

Navrací počet procesorů přístupných pro program buď celkově nebo pouze v uvedené dimenzi

`PROCESSORS_SHAPE()`

Navrací tvar multiprocesoru, tj. počet procesorů v každé dimenzi.

6.7.3 Vnitřní procedury HPF z knihovny HPF_LIBRARY

Knihovna **HPF_LIBRARY** není sice součástí **Subset HPF**, většina dostupných kompilátorů **HPF** ji však obsahuje úplně nebo alespoň z převážné části. Procedury v této knihovně lze rozdělit do těchto skupin:

- mapovací dotazovací podprogramy

- funkce pro manipulaci s bity
- redukční funkce pro pole
- funkce pro kombinaci a rozmístění prvků polí
- funkce pro prefixové a sufixové operace s poli
- funkce pro řazení polí

Mapovací dotazovací podprogramy

Aktuální mapování pole za běhu programu lze zjistit pomocí podprogramů `HPF_ALIGNMENT`, `HPF_TEMPLATE` a `HPF_DISTRIBUTION`.

Funkce pro manipulaci s bity

Tři funkce této skupiny doplňují sortiment bitových funkcí jazyka Fortran 90. Všechny mají argument typu `INTEGER`, `LEADZ` navrací počet úvodních nulových bitů, `POPCNT` počet jednotkových bitů a `POPPAR` paritu argumentu.

Redukční funkce pro pole

HPF poskytuje rozšíření redukčních funkcí jazyka Fortran 90. Pro binární operace `AND`, `OR` a `XOR` jsou určeny funkce po řadě `IALL`, `IANY` a `IPARITY`, čtvrtá zbývající, `PARITY`, navrací výsledek redukce pro logické `XOR`.

Funkce pro kombinaci a rozmístění prvků polí

Tyto funkce `ALL_SCATTER`, `ANY_SCATTER`, ..., představují zobecnění libovolné z dvanácti redukčních funkcí `ALL`, `ANY`, `COPY`, `COUNT`, `IALL`, `IANY`, `IPARITY`, `PARITY`, `MAXVAL`, `MINVAL`, `PRODUCT` a `SUM` pro pole. Nejdříve jsou prvky kombinovaného pole rozeslány do těch míst výsledné matice, kde má být uplatněna jejich hodnota (každý prvek lze poslat nejvýše jednou). Hodnoty výsledné matice jsou pak stanoveny provedením příslušné redukční funkce v každém místě se všemi přidělenými hodnotami. Tím způsobem lze tedy provést redukční kombinaci libovolných nepřekrývajících se částí pole.

Funkce pro prefixové a sufixové operace s poli

Tyto funkce umožňují postupné zpracování prvků pole. Pro prefixové resp. sufixové funkce je výsledná hodnota pro daný index dána výsledkem tzv. prohledávací operace s předchozími resp. následujícími prvky. Prohledávací funkcí může být libovolná funkce z již zmíněných dvanácti redukčních funkcí, je tedy definováno dvanáct prefixových (`ALL_PREFIX, ...`) a stejný počet sufixových funkcí. Je možné použít volitelné argumenty `DIM, MASK` a `SEGMENT`, které umožňují omezené prohledávání na některou dimenzi, vybrané prvky, případně provádět operace odděleně pro několik částí pole. Volitelný skalární parametr `EXCLUSIVE` dovoluje hodnotou `.TRUE.` určit, že do výsledku prohlížecí operace není zahrnována hodnota na právě zpracovávané pozici.

Funkce pro řazení polí

HPF zahrnuje funkce `GRADE_DOWN` resp. `GRADE_UP` pro sestupné resp. vzestupné třídění vícerozměrných polí. Tyto funkce navrací permutační matici označující pořadí prvků při požadovaném řazení. Pole lze třídit jako celek nebo podle jednotlivých dimenzí.

6.8 Použití jiných programových modelů

Některé algoritmy nelze v programovém modelu `HPF` realizovat dostatečně efektivně, jindy se nelze vyhnout použití sériových procedur dostupných pouze ve zkompilovaném tvaru a konečně nelze ani vyloučit použití procedur vyvinutých v jiném jazyce než **HPF**.

Uvedené situace jsou v jazyce **HPF** řešeny prostředky *extrinsických* procedur, u kterých je možné v čase deklarace specifikovat programový model pomocí prefixu `EXTRINSIC(`*model*`)` před jménem procedury. Návrh **HPF** zatím podporuje tři programové modely: `HPF_LOCAL`, `HPF_SERIAL` a `HPF`. Použití extrinsických procedur je možné jen při dostupnosti explicitního interface.

EXTRINSIC(HPF_SERIAL)

Vykonání procedury s tímto modelem je rozděleno do těchto postupných kroků:

1. všechny procesory jsou synchronizovány
2. aktuální parametry procedury jsou přemapovány na určený procesor
3. na určeném procesoru jsou vykonány příkazy uvnitř procedury
4. po návratu z procedury jsou všechny procesory opět synchronizovány
5. je obnoveno původní mapování aktuálních argumentů

EXTRINSIC(HPF_LOCAL)

Procedury s prefixem `HPF_LOCAL` umožňují dočasné použití programového modelu `SPMD`. Lokálně prováděné kopie procedur na jednotlivých procesorech nejsou uvnitř, na rozdíl od modelu `HPF`, navzájem nijak implicitně synchronizovány. Z globálního jmenného prostoru jsou pro ně viditelné pouze lokální kopie aktuálních argumentů. Uvnitř procedury lze používat všechny běžné prostředky (např. message-passing) pro meziprocesorovou komunikaci. Po návratu z lokálních procedur jsou všechny procesory synchronizovány a je provedena aktualizace hodnot distribuovaných proměnných.

EXTRINSIC(HPF)

Tato eventualita řeší případ použití HPF-procedury v běžném sériovém programu. Před provedením procedury je provedena distribuce aktuálních argumentů a je zahájen výpočet podle programového modelu `HPF`. Nakonec je proveden přenos hodnot aktualizovaných argumentů na původní místo.

Příklad: součet dlouhého pole prvků

```
PROGRAM SUM_OF_LONG_ARRAY

! Výpočet součtu prvků distribuovaného pole
! v programovém modelu SPMD.
! Lokálně spuštěné podprogramy provedou nezávisle
! mezisoučet prvků velkých bloků pole.
! Hlavní program provede pouze nenáročný konečný součet.
```

```
      IMPLICIT NONE
      USE HPF_CPUTIME
      REAL, DIMENSION(1000) :: A
      REAL TOTAL
      INTEGER I

      !HPF$ DISTRIBUTE (BLOCK) :: A

! Interface blok je pro EXTRINSIC funkce povinný
INTERFACE
      EXTRINSIC(HPF_LOCAL) FUNCTION SREDUCE(A) RESULT(R)
      REAL, DIMENSION(:), INTENT(IN) :: A
      REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
      !HPF$ DISTRIBUTE (BLOCK) :: A, R
      END FUNCTION SREDUCE
END INTERFACE

      FORALL (I = 1 : 1000) A(I) = I

      ! Součet redukovaného pole mezisoučtů
      ! vnitřní procedurou SUM
      TOTAL = SUM(SREDUCE(A))
      WRITE(*, *) TOTAL
      ! Výpis času CPU pro každý procesor
      WRITE(*, *) HPF_ETIME()

END PROGRAM SUM_OF_LONG_ARRAY

EXTRINSIC(HPF_LOCAL) FUNCTION SREDUCE(A) RESULT(R)
! Lokální funkce pro součet prvků pole
! Pole je předáváno jako vstupní argument A

      IMPLICIT NONE
      REAL, DIMENSION(:), INTENT(IN) :: A
      REAL, DIMENSION(1) :: R
```

```
      INTEGER :: COPIES
      INTEGER :: J

      ! Zjištění počtu kopií pole A
      CALL GLOBAL_ALIGNMENT(A, NCOPIES = COPIES)

      ! Funkci lze použít jen pro nereplikovaná pole
      IF (COPIES <= 1) THEN
        ! Akumulace všech prvků
        R(1) = 0.0
        DO J = LBOUND(A, 1), UBOUND(A, 1)
          R(1) = R(1) + A(J)
        END DO
      END IF
END FUNCTION SREDUCE
```

Příklad: modul pro zjištění spotřeby času CPU

```
MODULE HPF_CPUTIME

! Modul s funkcemi pro měření (lokálního) času CPU.
! Je předpokládáno, že vykonávání hlavního HPF-programu
! je provedeno synchronizovaným během jinak samostatných
! procesů. Je lokálně přistupováno ke zdrojům jednotlivých
! procesorů.

  IMPLICIT NONE
  REAL, DIMENSION(2), PRIVATE :: TARRAY(2)

CONTAINS

  EXTRINSIC(HPF_LOCAL) FUNCTION HPF_DTIME() RESULT(R)
    ! Zjištění přírůstku spotřebovaného času CPU
    ! lokálním procesem od okamžiku předchozího dotazu
    IMPLICIT NONE
    REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
    REAL, EXTERNAL :: DTIME
```

```
    !HPF$ DISTRIBUTE (BLOCK) :: R

    ! Dosazení hodnoty spotřebovaného času CPU
    ! od předchozího volání DTIME
    R(1) = DTIME(TARRAY)

  END FUNCTION HPF_DTIME

  EXTRINSIC(HPF_LOCAL) FUNCTION HPF_ETIME() RESULT(R)
    ! Zjištění celkového spotřebovaného času CPU
    ! lokálním procesem od počátku existence procesu
    IMPLICIT NONE
    REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
    REAL, EXTERNAL :: ETIME

    !HPF$ DISTRIBUTE (BLOCK) :: R

    ! Dosazení hodnoty spotřebovaného času CPU
    ! od počátku existence procesu
    R(1) = ETIME(TARRAY)

  END FUNCTION HPF_ETIME

END MODULE HPF_CPUTIME
```

6.9 Další vývoj HPF

Výhledy na všeobecné uplatnění **HPF** nejsou příliš jednoznačné, na jeho nasazení při vývoji nebo přenosu větší aplikace se teprve čeká. Na jedné straně má **HPF** jistý náskok před ostatními programovacími jazyky, na straně druhé se jeho „pronásledovatelé“ mohou vyhnout jím vyjeveným úskalím.

6.9.1 Celkové hodnocení HPF

Provedeme krátké vyhodnocení výhod a nevýhod **HPF**.

Přednosti HPF

Mezi výhody **HPF** patří zejména:

- **Produktivita.** **HPF** umožňuje využívat běžné metody programování. Při vývoji programů pro oblast jazyka **Fortran** typických, tj. programů pro řešení problémů z oblasti náročných vědeckých výpočtů a s poměrně častou potřebou modifikace, je vyžadována jen minimální asistence počítačového experta.

- **Výkonnost.** Výsledné programy vykazují většinou nejméně stejné výkonnostní parametry (urychlení, adaptibilita na velké problémy) jako programy vyvinuté jinými prostředky (dnes zejména pomocí message-passing).

- **Přenositelnost a snadná údržba.** Zde nemá **HPF** soupeře, zejména díky existenci *de facto* standardu a poměrně bohatému trhu s kompilátory **HPF** pro téměř všechny běžně užívané paralelní platformy.

Slabiny HPF

Mezi nevýhody **HPF** můžeme zařadit především:

- **Nadměrný rozsah specifikace.** Specifikace jazyka Fortran 90 a tím i **HPF** je velmi rozsáhlá, komplikovaná a někdy i nesrozumitelná. Zejména pro nepočítačového specialistu zahrnuje příliš mnoho podrobností. Zátěž dlouhodobé zpětné kompatibility je až příliš citelná.

- **Replikace dat.** Pro značnou obtížnost bude automatická distribuce dat asi ještě delší dobu spíše zbožným přáním. Současné kompilátory řeší problém explicitně nedistribuovaných proměnných jejich replikací pro všechny procesory, což však může mít při méně pečlivém návrhu programu nedozírné následky na jeho paměťové a časové požadavky.

- **Omezený rozsah uplatnění.** Zde se nejedná o nevýhodu, spíše jde o obecnou vlastnost s trvalou platností. **HPF** nemůže zatím zajistit dostatečnou podporu pro zejména „nestrukturovaný“ paralelizmus, pro značnou část aplikací je tedy nutné použít metodu message-passing. V **HPF** nelze spatřovat konečné řešení všech problémů, je však možné počítat s využitím v kombinaci s jinými způsoby.

6.9.2 Oblast použití HPF

Podle uvedených předností a nedostatků lze za současného stavu doporučit nasazení **HPF** v těchto případech:

- **Řešení rozměrných problémů.** I pro **HPF** platí tato obecná zásada, že náklady a časové nároky na vývoj „malých" paralelních aplikací jsou mnohdy neúměrné oproti získaným úsporám.
- **Uplatnění datového paralelizmu.** Při zajištění této podmínky může být pomocí **HPF** dosaženo velmi uspokojivých výsledků, její nedodržení naopak může vést k velkému zklamání.
- **Tvorba prototypů.** Návrh prototypu aplikace pomocí **HPF** lze použít k rychlému rozhodnutí jak pokračovat dále, zda vůbec paralelizmus použít a případně jaký jeho typ.

6.9.3 HPF 2.0

V našem výkladu jsme se omezili na první verzi jazyka **HPF 1.0**, jejíž praktické ověřování po dobu více než dvou let bylo podkladem pro návrh zatím poslední verze **HPF 2.0** δ. Uvedeme jen stručně nejdůležitější rysy této nové verze:

- Není rozlišována specifikace úplná a její podmnožina. Nová verze zahrnuje vlastnosti, jejichž implementaci lze s již získanými zkušenostmi očekávat u většiny kompilátorů přibližně během jednoho roku. Náročnější konstrukce jsou zahrnuty do tzv. schválených doporučení[18].
- Do **HPF 2.0** byly zahrnuty všechny běžné konstrukce z předchozí verze s výjimkou prostředků pro dynamické mapování. Ty byly pro velkou implementační a výpočetní náročnost přesunuty do schválených rozšíření. Příkaz `INDEPENDENT DO` byl rozšířen o klauzuli `REDUCTION` mající obdobnou syntaxi jako klauzule `NEW` a umožňující použití redukčních operací uvnitř smyčky.
- Schválená rozšíření vedle již zmíněného dynamického mapování obsahují základní podporu těchto nových možností:
  - neregulární mapování

[18] z angl. *approved extensions*

– paralelizmus podle úloh
– asynchronní I/O operace

6.10 Odkazy na informace o HPF

Všechny potřebné informace o **HPF** a jazyce **Fortran** vůbec, lze získat na těchto odkazech:

- veškerá dokumentace o standardech **HPF** v elektronické podobě je veřejně přístupná na URL:
  `ftp://titan.cs.rice.edu/public/HPFF/draft`
- celosvětové URL skupiny HPFF je:
  `http://www.crpc.rice.edu/HPFF/home.html`
  resp. v Evropě:
  `http://www.vcpc.univie.ac.at/HPFF/home.html`.
- nakonec URL, které nemůže žádný zájemce o jazyk **Fortran** vynechat:
  `http://www.fortran.com/fortran/`

Dodatek A

A.1 Základní návod pro práci s jazykem Digital ADA

Vytvoření programu v jazyce **ADA** probíhá v následujících krocích [1]:

1. Vytvoření pracovního adresáře (využívaného pro zdrojové soubory programu). Pro každou aplikaci (program) je vhodné vytvořit separátní pracovní adresář.

2. Zdrojové soubory by měly mít příponu `.ada`. Dále je vhodné v názvu souborů obsahujících samostatně překládané moduly rozlišit, zda se jedná o specifikaci nebo o tělo modulu, např. použijeme `modul-s.ada` a `modul-b.ada` pro soubory obsahující specifikaci a tělo modulu (package).

3. Vytvoření knihovny pro objektové moduly (vytvořené překladem zdrojových modulů). Provede se z pracovního adresáře příkazem

   `amklib` *adalib*

   kde *adalib* je zvolené jméno podadresáře, ve kterém bude knihovna vytvořena. Do knihovny se překopírují přeložené standardní moduly (package ap.) dostupné v příslušné instalaci. Dále se pak do ní přidávají objektové moduly vzniklé překladem zdrojových modulů uživatelského programu. Uživatelské moduly se přidávají do knihovny automaticky - každým překladem se nahradí stará verze (pokud tam je) novou. Knihovní adresář *adalib* je vytvořen pod pracovním adresářem.

4. Definování aktuální knihovny (tzv. *kontext*). Provede se z pracovniho adresáře příkazem

[1] Uživatelsky volitelná jména v příkazech jsou psána kurzívou.

`setenv ADALIB @`*adalib*

kde *adalib* je jméno knihovniho adresáře.

!!! Po každém přihlášení do systému je zapotřebí přepnout se na pracovní adresář a definovat aktuální knihovnu.

5. Samostatný překlad zdrojového souboru *file*`.ada` se nejlépe provede příkazem

   `ada -V` *file*`.ada`

   Volba `-V` zajistí vytvoření protokolu o překladu, který se uloží do souboru *file*`.l`. Při bezchybném překladu se automaticky uloží do knihovny všechny moduly (**ADA** units), jejichž text se nachází v souboru *file*`.ada`. Pokud je specifikace modulu v samostatném souboru, musí se přeložit dřív než tělo.

6. Sestavení programu se provede příkazem

   `ald -o` *main_exe main_proc*

   kde *main_exe* je jméno souboru se sestaveným programem a *main_proc* je jméno procedury [2]. (obvykle samostatně kompilované), která slouží jako hlavní program. Všechny moduly (package, samostatně kompilované procedury), které využívá hlavní program už musí být přeložené v knihovně [3]. Sestavený program se nachází v pracovním adresáři.

   Můžeme také použít rovnou příkaz

   `ald` *main_proc*

   a sestavený program nalezneme v pracovním adresáři pod jménem `a.out`.

7. Pokud změníme zdrojový text některého modulu a nechceme opakovat jednotlivě překlad všech závislých modulů, využijeme příkaz

   `amake` em main_proc

[2]!!! Pozor - v příkazu `ald` se uvádí jméno procedury *main_proc* a nikoliv jméno zdrojoveho souboru, ve kterém je procedura obsažena. Nejlépe je dát proceduře hlavního programu i jejímu zdrojovému souboru stejné jméno.

[3]!!! Obecně platí, že v knihovnách jazyka **ADA** se vedou jména modulů (**ADA** units - package, procedury) a nikoliv jména zdrojových sou- borů, ve kterých jsou moduly obsaženy.

pro překlad procedury hlavního programu. Automaticky se vyhledají a ve správném pořadí provedou všechny potřebné překlady. Pokud chceme navíc ještě provést sestavení programu, použijeme příkaz

`amake -l -o` *main_exe main_proc*

8. Spuštění výpočtu se provede příkazem

   main_exe

   kde *main_exe* je jméno použité v příkazu `ald`, tj. jméno souboru se spustitelným programem.

9. Ladění programu je u konkurenčních výpočtů mnohem obtížnější, proto je součástí prostředí ADY ladicí program (*debugger*) `dbx`. Neodladěné moduly je třeba překládat s volbou `-g`, například

   `ada -g` *file*`.ada`

   Program se normálním způsobem sestaví a spustí příkazem:

   `dbx` *main_exe*

   Po ohlášení ladicího programu lze využitím příkazu `help` získat seznam použitelných příkazů (definice *breakpointů* ap.). Návrat z ladicího programu se provede kdykoliv příkazem `quit`.

10. Ještě uvedeme užitečné příkazy potřebné pro práci s knihovnou přeložených modulů. Obsah knihovny se zobrazí příkazem

    `als`

    Knihovní moduly, jejichž jména končí `-s` představují přeložené specifikace, zakončení `-b` představuje přeložené tělo modulu.

    Zajímá–li nás zdrojový text modulu (důležité zejména u specifikací standardních knihovních modulů, např. `TEXT_IO`), využijeme příkaz

    `acat TEXT_IO`

    pro zobrazení zdrojového textu specifikace i těla na standardní výstup, nebo

    `acat TEXT_IO-s`,

    jestliže nás zajímá jen specifikace.

Modul můžeme vypustit (specifikaci i tělo) z knihovny například příkazem

`arm -v` em muj_modul

Volba `-v` zajistí zprávu, které moduly byly vypuštěny. Například jenom specifikace se vypustí příkazem

`arm -v` *muj_modul*`-s`

Celou knihovnu zlikvidujeme příkazem

`armlib -i adalib`

kde `adalib` je jméno adresáře a volba `-i` zajistí dotaz, zda se má akce skutečně provést.

11. Stručný anglicky psaný manuál pro překladač programovacího jazyka **ADA** je možné prohlížet (ovládání klávesou `d`) po použití příkazu

    `man ada`

A.2 Základní návod pro práci s jazykem C na počítači SEQUENT

Vytvoření programu v jazyce **C** probíhá v následujících krocích [4]:

1. Pro každý program je vhodné zřídit separátní adresář V tomto adresáři se vytvoří zdrojové soubory programu, popřípadě se prostřednictvím `ftp` přenesou zdrojové soubory vytvořené pod MS-DOS.

2. Překlad a sestavení programu *file*`.c` určeného pro paralelní výpočet se provede příkazem:

   `cc -Xs -Wc,-seq` *file*`.c -lpps -lseq [-l*]`

   Význam nastavených voleb :

   `-Xs`

   překlad v DYNIX/ptx **C** kompatibilním modu (pre-ANSI **C**, spíše Kerningham a Ritchie),

[4] Uživatelsky volitelná jména v příkazech jsou psána kurzívou.

-Wc,-seq

oznámení překladači, že program je konstruován jako paralelní (umožňuje využití klíčových slov `private` a `shared`),

file.`c`

zdrojový text programu,

`-lpps`

využívá se knihovna paralelních funkcí,

-lseq

využívá se speciální knihovna pro SEQUENT,

-l*

případné další knihovny (viz manuál cc).

Pokud chceme protokol o překladu uložit do souboru, provedeme přesměrování výstupů (přidáme `2>` *file*.`l`). Sestavený program je v pracovním adresáři pod názvem `a.out`.

Vzhledem ke složitosti příkazu se vyplatí vyrobit příkazový soubor (script).

3. Sestavený program se spustí příkazem

   `a.out`

   Pokud chceme monitorovat využití procesorů počítače SEQUENT, spustíme výpočet na pozadí příkazem

   `a.out <`*file_i*`.dat >`*file_o*`.dat &`

   kde *file_i*`.dat` je soubor vstupních dat a *file_o*`.dat` je soubor pro uložení výsledků výpočtu. Potom spustíme monitorovací program:

   `monitor`

   Návrat z monitoru provedeme klávesou `q`.

4. Stručný anglicky psaný manuál pro překladač cc je možné prohlížet (ovládání klávesou `d`, návrat `q`) po použití příkazu

   `man cc`

A.3 Základní návod pro práci s jazykem ADA na počítači SEQUENT

Vytvoření programu v jazyce **ADA** (kompilátor firmy Meridian) probíhá v následujících krocích [5].

1. Pro každý program je vhodné zřídit separátní adresář. V tomto adresáři se vytvoří zdrojové soubory programu, popřípadě se prostřednictvím `ftp` přenesou zdrojové soubory vytvořené pod MS-DOS. Zdrojové soubory by měly mít příponu `.ada` . Dále je vhodné v názvu souborů obsahujících samostatně překládané moduly rozlišit, zda se jedná o specifikaci nebo o tělo (implementaci) modulu, např. *modul*`-s.ada` a *modul*`-b.ada` pro soubory obsahující specifikaci a tělo modulu (*package*) *modul*. Další možnost je používat přípony ve jménech souborů `.ads` pro specifikace, `.adb` pro těla (angl. *body*) a `.sub` pro *sub–unity* (separátně překládaná těla procedur).

2. Nastavení prostředí pro použití překladače se provede příkazem:

   `usrcfg ada`

3. Vytvoření knihovny pro objektové moduly (vytvořené překladem zdrojových modulů). Provede se z pracovního adresáře pomocí příkazu

   `newlib`

   V pracovním adresáři vznikne podadresář `ada.aux/` a soubor `ada.lib`. Do knihovny se překopírují přeložené standardní moduly dostupné v příslušné instalaci. Dále se pak do ní přidávají objektové moduly vzniklé překladem zdrojových modulů uživatelského programu. Uživatelské moduly se přidávají do knihovny automaticky – každým překladem se nahradí stará verze (pokud tam je) novou.

4. Samostatný překlad zdrojového souboru *file*`.ada` se nejlépe provede příkazem

   `ada -l` *file*`.ada`

   Volba `-l` zajistí vytvoření protokolu o překladu, který se uloží do souboru *file*`.lst`. Při bezchybném překladu se automaticky uloží do

[5] Uživatelsky volitelná jména v příkazech jsou psána kurzívou.

knihovny všechny moduly (**ADA** *units*), jejichž text se nachází v souboru *file*.`ada`. Pokud je specifikace modulu v samostatném souboru, musí se přeložit dřív než tělo.

5. Sestavení programu se provede příkazem

   `bamp` *main_proc*

   kde *main_proc* je jméno procedury (obvykle samostatně kompilované), která slouží jako hlavní program [6]. Všechny moduly (package, samostatně kompilované procedury), které využívá hlavní program už musí být přeložené v knihovně. Sestavený program se nachází v pracovním adresáři pod názvem *main_proc** .

6. Pokud změníme zdrojový text některého modulu a nechceme opakovat jednotlivě překlad všech závislých modulů, využijeme příkaz

   `amake` *main_proc*

   pro překlad procedury hlavního programu. Automaticky se vyhledají a ve správném pořadí provedou všechny potřebné překlady.

7. Spuštění výpočtu se provede příkazem

   main_proc

   kde *main_proc* je jméno použité v příkazu `bamp`.

8. Příkazy potřebné pro práci s knihovnou přeložených modulů jsou `lslib` (informace o knihovně), `lnlib` (přidávání ke knihovně), `rmlib` (vypouštění z knihovny). Nejčastěji potřebujeme zobrazit obsah celé knihovny příkazem

   `lslib -a |more`

   Další informace o uvedených příkazech jsou dostupné z jejich manuálových stránek.

9. Stručný anglicky psaný manuál pro jednotlivé zde popsané (a další příkazy) je možné prohlížet po použití

   `man` *jméno_příkazu*

[6]!!! Pozor - v příkazu `bamp` se uvádí jméno procedury `main_proc` a nikoliv jméno zdrojového souboru, ve kterém je procedura obsažena. Nejlépe je dát proceduře hlavního programu i jejímu zdrojovému souboru stejné jméno.

A.4 Základní návod pro práci s PVM

1. Pro úspěšnou práci s PVM je nutné nejdříve nastavit několik proměnných prostředí. Je to především proměnná `PVM_ROOT`, která udává kořenový adresář lokální instalace PVM a proměnná `PATH` určující adresáře prohledávané interpreterem příkazového jazyka (*shell*). Nastavení těchto proměnných závisí na typu používaného *shellu.*

   Pro *Bourne Shell, Korn Shell a Bourne Again Shell (bash)* stačí do souboru `.profile`, popř. `.bashrc` přidat tyto příkazy :

   ```
   PVM_ROOT=/usr/local/pub/pvm3; export PVM_ROOT
   MANPATH=$MANPATH:$PVM_ROOT/man; export MANPATH
   PATH=$PATH:$PVM_ROOT/lib; export PATH
   ```

   V případě *C Shellu* přidáme do `.cshrc` příkazy :

   ```
   set PVM_ROOT=/usr/local/pub/pvm3
   set MANPATH=($MANPATH $PVM_ROOT/man)
   set path=($path $PVM_ROOT/lib)
   ```

   Přidáním těchto příkazů do uvedených souborů zajistíme, že při každém přihlášení do systému budeme mít PVM ihned k dispozici.

2. Dalším krokem je zajištění možnosti se přihlásit na vzdálený počítač bez nutnosti zadání hesla (pro `pvmd`). To provedeme tak, že do souboru `.rhosts` v domovském adresáři napíšeme seznam počítačů, které budou "zapojeny" do PVM s uvedením uživatelského jména. V případě, že na každém z počítačů vlastníme stejné uživatelské jméno, bude obsah souboru `.rhosts` následující :

   ```
   pallas1.zcu.cz novak01

   pallas2.zcu.cz novak01

   pallas3.zcu.cz novak01
   ```

3. Dále musíme určit počítače začleněné do PVM. Ty uvedeme v souboru *hostfile*, který bude uložen opět v domovském adresáři. Jeho obsah může vypadat následovně :

```
pallas1 ep=/export/home/novak01/pvm/bin
pallas2 ep=/export/home/novak01/pvm/bin
```

Položka `ep` udává adresář, ve kterém se nácházejí spustitelné soubory aplikace pro PVM.

4. Posledním krokem je spuštění démona `pvmd` příkazem :

   ```
   pvmd hostfile &
   ```

5. K usnadnění kompilace aplikací pro PVM lze použít program `make`. Vzorový soubor `Makefile` pro tento program lze najít v adresáři `/usr/local/pub/pvm3/local` na stanicích *Sun*.

A.5 Použití KAPF pod OS Digital UNIX

Vedle samotného preprocesoru **KAPF** je k dispozici řídící program, který umožňuje postupné spuštění preprocesoru a kompilátoru bez přímého zásahu uživatele. Uvedeme jen jednoduché příklady příkazových řádků pro oba způsoby celého procesu transformace, kompilace a sestavení paralelního programu, podrobnější informace lze nalézt v manuálových stránkách.

Oddělený běh preprocesoru a kompilátoru

Pro paralelizaci, kompilaci a sestavení programu řekněme `pprg` ze zdrojového souboru `pprg.f` slouží při odděleném běhu preprocesoru a kompilátoru tato dvojice příkazů:

```
kapf -conc -cmp=pprg_mp.f -list=pprg.lst -lo=o pprg.f
f77 pprg_mp.f -lkmp_osf -threads -o pprg
```

Ve vyvolání preprocesoru `kapf` jsou zahrnuty tyto volby:

`-conc`

je požadována (ve zkrácené formě) transformace programu pro paralelní zpracování

`-cmp=` *jméno-souboru*

následuje název souboru pro uložení transformovaného programu

`-list=` *jméno-souboru*

uvedení názvu souboru pro diagnostický výpis

`-lo=` *volby*

přesnější určení informací, které budou ukládány do souboru pro diagnostický výpis.

`-supress=` *volby*

omezení diagnostických informací

Volby při vyvolání kompilátoru `f77` zajišťují pro fázi sestavení programu:

`-lkmp_osf`

začlenění podpůrné knihovny pro **KAPF**

`-threads`

přednostní použití vícevláknových modifikací knihoven a zařazení podpůrné knihovny pro DECthreads (libpthreads).

Společný běh preprocesoru a kompilátoru

Paralelizaci, kompilaci a sestavení programu lze pro již uvedený příklad provést bez přerušení příkazem[7]:

```
kf77 -fkapargs='-conc'parprog.f -o parprog
```

Pro předání parametrů a voleb pro **KAPF** je vyhrazen přepínač `-fkapargs`, kde lze zřetězeně uvést všechny požadavky.

Běh paralelních programů v prostředí SMP

Před během programu paralelizovaného pomocí **KAPF** je třeba, jako pro každý jiný program využívající v nějaké formě knihovnu DECthreads, nastavit některé proměnné prostředí udávající počet vláken, velikost zásobníku pro vlákna a způsob synchronizace. Uvedeme příklad pro příkazový interpreter C-Shell:

nastavení počtu vláken

```
setenv PARALLEL 4
```

[7] ovšem bez možnosti přímých úprav transformovaného programu

Proces vytvoří 4 vlákna. Počet vláken by neměl převyšovat počet dostupných fyzických procesorů, jinak dochází k degradaci výkonnosti.

velikost zásobníku vláken v bytech

```
setenv KMP_STACKSIZE 100000
```

Nastaví velikost zásobníku na 100 kB.

způsob synchronizace

```
setenv KMP_SPINLOCKS no
```

Pro synchronizaci vláken bude užito pasívní čekání, sice časově méně pohotové, ale vhodné pro víceuživatelský režim chodu systému. Druhou možností je aktivní čekání ve smyčce (hodnota `yes`), vhodnější pro jednouživatelský režim.

Literatura

[ADA91] DEC Ada Language Reference Manual, DEC 1991, 589 s.

[Bach86] BACH. M. J.: The Design of the Unix Operating System, Prentice Hall, 1986, 471 s.

[Bar97] BARNES, J.: Introduction to ADA95, http://www.ada9x.uk, 1997

[Blume94] BLUME, W. et al.: Automatic Detection of Paralelism: A Grand Challenge for HPC, CSRD Report No. 1348, University of Illinois at Urbana-Champaign, 1994.

[Bow87] BOWLER K.C. et al.: An Introduction to OCCAM 2 Programming. Chartwell-Bratt 1987, 109 s.

[Brod89] BRODSKÝ, J. - SKOČOVSKÝ L.: Operační systém Unix a jazyk C. SNTL 1989, 367s.

[Bust88] BUSTARD, D. - ELDER J.- WELSH J.: Concurrent Program Structures, Prentice Hall 1988, 321 s.

[Digital] Firemní materiály Digital.

[DECTh] Guide to DECThreads, Firemní materiály Digital, Feb.94

[Fost95] FOSTER, I.J.: Designing and Building Parallel Programs. Addison–Wesley 1995, 379 s.

[Han96] HANULIAK, I.: Parallel algorithms and their network implementations, Journal of Electrical Engineering, 47, No. 9-10, pp. 269-273, 1996, Bratislava

[Han97] HANULIAK, I.: Paralelné architektury - multiprocesory, počítačové sietě, Knižné centrum Žilina, 1997, 187 s. ISBN: 80-88-723-64-7

[Han98] HANULIAK, I.: Parallel network algorithms, 13. Internationale Fachtagung, 11-14. November 1998, pp. 21-27, Institut for Technic and Economy, Mitweida, Germany

[Hlav94] HLAVIČKA, J.: Architektury počítačů, Vydavatelství ČVUT, Praha 1994

[Hlav96] HLAVIČKA, J. - RACEK, S: Multiprocesorové systémy. Softwarové noviny, 5/1996.

[Isip93] Sborník konference ISIPCALA 93, Praha, červenec 1993.

[JáJá92] JÁJÁ, J.: An Introduction to Parallel Algorithms. Addison–Wesley, 1992, 566 s.

[Jež86] JEŽEK K.: Speciální programovací jazyky. Ediční středisko VŠSE Plzeň 1986, 138 s.

[Jon88] JONES, G. - GOLDSMITH, M.: Programming in OCCAM 2. Prentice Hall 1988, 317 s.

[nCUBE] Firemní materiály nCUBE.

[Pláš91] Plášil, F.: Operační systémy. Skriptum ČVUT, Praha 1991, 377 s.

[PN90] ROZENBERG. G. (ed.): Lecture notes in Comp. Science 424 – Advances in Petri Nets 1989, Springer - Verlag 1990.

[PPL89] Guide To Parallel Programming On Sequent Computer Systems, Prentice Hall 1989.

[PVM94] GEIST, A.: PVM 3 User's guide and reference manual. Report ORNTL/TM-12187, Oak Ridge Nat. Lab., Sept. 1994, 159 s.

[Qui87] QUINN M.J.: Designing Efficient Algorithms for Parallel Computers, McGraw Hill 1987, 280 s.

[SEQUENT] Firemní materiály SEQUENT.

[Ra94] RACEK,S. - JEŽEK,K.: Paralelní programování. ZČU Plzeň, 1994, 176 s.

[Ra95] RACEK,S. - HEROUT,P.: Využití programu PVM pro urychlení diskrétní simulace. Sborník konf. Advanced simulation of systems, Zábřeh na Mor. 1995, ss.16 - 21.

[Skan89] SKANSHOLM, J.: Ada from the beginning. Adison-Wesley 1989, 617s.

[Šmrha94] ŠMRHA, P.: Operační systém Unix. Skriptum ZČU, ediční středisko ZČU, Plzeň, 1994, 129 s.

[Tak95] TAKAYASU,I. - AKINORI, Y. (Eds.): Theory and Practice of Parallel Programming, Lecture Notes in Computer Science 907, Springer 1995.

[Tvrd94] TVRDÍK,P.: Parallel systems and Algorithms. Vydavatelství ČVUT, Praha 1994, 79 s.

[Wirth82] WIRTH N.: Programming in MODULA-2. 3rd ed. Springer–Verlag, Berlin, New York, 1985, 202 s.

[Young88] YOUNG S.J.: Programovací jazyky pro RT aplikace. SNTL, Praha 1988, 388 s.