Programovanie pre model s distribuovanou pamäťou – MPI

V kapitole **??** sme si uviedli dva typy paralelných systémov, konkrétne systémy so zdieľanou pamäťou a systémy s distribuovanou pamäťou. Z pohľadu programátora je možné systémy s distribuovanou pamäťou vnímať ako dvojicu tvoriacu procesorom a jemu prislúchajúcou pamäťou, do ktorej má prístup a žiadny iný procesor do tejto pamäte nemôže priamo pristupovať. Takéto dvojice môžu navzájom komunikovať medzi sebou pomocou **posielania správ** po sieti. Takýto systém môže byť reprezentovaný integrovanými počítačmi s distribuovanou pamäťou alebo počítačovým klastrom s navzájom prepojenými výpočtovými uzlami pomocou prepojovacej siete.

Pri použití systémov s distribuovanou pamäťou je zvyčajne inštanciou vykonávaného programu **proces**. Každý proces má prístup k svojím údajom uloženým v pamäti a môže komunikovať s ostatnými procesmi pomocou posielania správ. Tento model komunikácie je pomerne zdĺhavý a komplikovaný avšak zároveň veľmi flexibilný. Už v 80-tých rokoch 20. storočia bola potreba vzniku jazyka, ktorý by umožnil vytváranie paralelných programov. Takýmto jazykom bol aj jazyk occam navrhnutý spoločnosťou Inmos [1]. Tento jazyk bol založený na CSP notácií (Communicating Sequential Processes). S postupným vývojom hardvéru sa však časom ukázalo, že tento jazyk nie je z rôznych dôvodov vhodný.

Okrem prístupu založenom na posielaní správ nie je jedinou možnosťou písanie programov pre počítače s distribuovanou pamäťou. Vzniklo niekoľko špecializovaných jazykov, ako napríklad High Performance Fortran (HPF), Co-Array Fortran (CAF) [2], Uified Parallel C (UPC) [3] a mnohé ďalšie. Ani tieto jazyky si však nezískali veľkú priazeň používateľov. To všetko viedlo k vzniku požiadavky na vytvorenie jednotného rozhrania, ktoré by umožňovalo jednoduchú prenositeľnosť a škálovateľnosť paralelných programov.

Súčasné masívne paralelné programy sú väčšinou písané v jazykoch, ako je C, C++ alebo Fortran s využitím ďalších knižníc zabezpečujúcich kooperáciu paralelných procesov. Dvojicu najviac používaných knižníc predstavujú **PVM** (Parallel Virtual Machine), vyvíjanú v Oak Ridge National Laboratory a **MPI** (Message Passing Interface), ktoré predstavuje univerzálnu knižnicu používanú pri písaní paralelných programov.

V priebehu apríla 1992 sa pod záštitou Centra pre výskum paralelného počítania uskutočnil workshop na tému štandardov pre posielanie správ v prostredí s distribuovanou pamäťou, ktorého sa zúčastnilo viac než 80 ľudí s približne 40-tich organizácií zaoberajúcich sa distribuovaným počítaním. Účastníci zastupovali rôzne skupiny, ako napríklad výrobcov paralelných počítačov, vedcov z univerzít, vládu alebo aj priemyselné laboratóriá. Výsledkom workshopu bol vznik

pracovnej skupiny, ktorej úlohou bolo vytvorenie štandardizovanej knižnice. Koncom roku 1992 bol touto skupinou predstavený predbežný návrh, ktorý bol ďalej vylepšovaný a prediskutovaný až do mája roku 1994, kedy bola uverejnená prvá verzia štandardu knižnice **MPI-1.0**. Neskôr boli publikované novšie verzie štandardu: MPI-1.1 (jún 1995), MPI-1.2 (júl 1997), **MPI-2.0** (júl 1997), MPI-2.1 (september 2008), MPI-2.2 (september 2009), MPI-3.0 (september 2012) a **MPI-3.1** (jun 2015). V súčasnosti v roku 2019 sa vedú diskusie a pracuje sa na vývoji nového štandardu MPI-4.0.

Zásadným prínosom MPI-2 v porovnaní s MPI-1 bolo rozšírenie o možnosť dynamického vytvárania úloh, paralelných vstupno-výstupných operácií, možnosť použitia MPI v jazyku C++ a Fortran 90. Štandard MPI-3 so sebou priniesol zásadné zmeny v podobe neblokujúcich funkcií kolektívnej komunikácie, možnosť jednostrannej komunikácie alebo kompatibilitu s jazykom Fortran 2008. Okrem toho bolo zo štandardu odstránených množstvo zastaralých funkcií a objektov.

Rôzne implementácie štandardu MPI je možné nájsť takmer u každého producenta paralelných počítačov vrátane tých so zdieľanou pamäťou. Bezplatnú, prenosnú verziu knižnice predstavujú MPICH a MPICH2 vyvíjané v Argonne National Laboratory a na Štátnej Univerzite Mississippi [4]. Ďalšou známou implementáciou je verzia LAM vyvíjaná v Ohio Supersomputer Center alebo verzia OpenMPI vyvíjaná konzorciom zloženým z akademických, výskumných a priemyselných zástupcov [5]. Súčasný štandard MPI obsahuje viac než 500 funkcií a nie je v možnostiach tejto publikácie ich všetky obsiahnuť. Našou snahou je zamerať sa na základné koncepty vytvárania paralelných programov s použitím MPI a poskytnúť tak základné poznatky k ďalšiemu štúdiu špecializovaných učebníc a dokumentácie.

1 Jednoduchý program v prostredí MPI

Väčšina začiatkov kurzov programovania je sprevádzaná programom 1 – *Hello world*, ktorý vychádza z textov autorov Kernighana a Ritchieho [6]. V nasledujúcej časti tento program upravíme tak, aby sme si na ňom demonštrovali rôzne funkcie MPI programu.

Zdrojový kód 1: Program Hello world

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello world\n");

  return 0;
}
```

Inicializácia a ukončenie paralelného programu

Program MPI je vykonávaný podľa modelu SPMD (Single Program Multiple Data), čiže každý z procesorov vykonáva ten istý program s rôznymi údajmi. Hoci by sa mohlo zdať, že je to dosť obmedzujúce, nepredstavuje to žiadne obmedzenia v porovnaní s modelom MPMD (Multiple Program Multiple Data), pretože každý z procesorov môže vykonávať inú časť toho istého programu. Uvedený sériový program *Hello world* môžme prerobiť na paralelný program 2 s použitím MPI tak, že každý proces vypíše pozdrav.

Zdrojový kód 2: Paralelný program Hello world

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char* argv[])
5  {
6    //inicializácia MPI
7    MPI_Init(&argc, &argv);
8
9    printf("Hello world\n");
10
11   //ukončenie MPI
12   MPI_Finalize();
13
14   return 0;
15 }
```

Vidíme, že pôvodný program bol doplnený o niekoľko nových riadkov. prvom rade je potrebné do MPI programu vložiť ďalší hlavičkový súbor `#include <mpi.h>` (riadok 2). Tento obsahuje funkčné prototypy MPI funkcií, definície makier, údajových typov a podobne, teda všetky definície a deklarácie potrebné pri kompilovaní MPI programu. Všetky identifikátory definované v MPI začínajú predponou `MPI_`, za ktorou pokračuje identifikátor, pričom prvé písmeno za podčiarkovníkom je veľké pre funkcie a MPI údajové typy. Pre konštanty a makrá sú celé identifikátory napísané veľkými písmenami. Vďaka tejto konvencii sa dá ľahko odlíšiť, čo je definované v MPI a čo je definované používateľom [7].

MPI_Init

```
int MPI_Init(int *argc, char ***argv)
```

Volaním funkcie `MPI_Init` na riadku 6 inicializujeme celé prostredie MPI. Pri tomto procese sa napríklad vyhradí v pamäti miesto pre bufer správ alebo sa jednotlivým procesom vykonávajúcim MPI program určí poradie. V prípade použitia viacerých vlákien, túto funkciu môže zavolať len jedno z nich. Ostatné funkcie je možné volať až po vykonaní inicializácie.

Poznámka: Všetky funkcie MPI (okrem MPI_Wtime a MPI_Wtick) ako návratovú hodnotu funkcie vracajú chybový kód. V prípade úspešného vykonania funkcie vracajú hodnotu MPI_SUCCESS.

Funkcia má dva argumenty `argc` a `argv`, ktoré predstavujú ukazovatele na odpovedajúce argumenty funkcie `main`. Pomocou týchto ukazovateľov je možné získať hodnoty z argumentov funkcie `main` vo všetkých MPI procesoch paralelného programu. V prípade, ak nie je potrebné používať tieto argumenty, MPI od verzie 2.0 umožňuje ich nahradenie nulovými ukazovateľmi `NULL`.

MPI_Finalize

```
int MPI_Finalize()
```

Volaním funkcie `MPI_Finalize` na riadku 8 ukončíme prostredie MPI. To znamená, že sa uvoľní alokovaná pamäť pre bufer správ a počká sa na ukončenie všetkých procesov vytvorených v rámci MPI. Po zavolaní tejto funkcie už nie je možné volať žiadne ďalšie funkcie MPI. Funkciu `MPI_Finalize`, ako aj funkciu `MPI_Init` môžme zavolať z ľubovoľného miesta programu, nemusia byť volané z funkcie `main`.

Kompilovanie a spúšťanie MPI programov

Poznámka: Pre písanie, kompilovanie a spúšťanie MPI programov je možné použiť aj integrované vývojové prostredie – IDE, ktoré umožní niektoré činnosti zjednodušiť.

Podrobnosti ohľadne kompilácie a spúšťania MPI programov je najlepšie zistiť priamo od správcu systému alebo iného skúseného používateľa systému, pretože tieto sa môžu v závislosti od systému líšiť. Predpokladajme, že náš program píšeme v ľubovoľnom editore a budeme ho kompilovať a spúšťať pomocou štandardného príkazového riadku v operačnom systéme.

Na mnohých systémoch je možné použiť na skompilovanie MPI programu príkaz `mpicc`, ktorý môžme doplniť o ďalšie parametre bežne používané s príkazom `gcc` podľa nasledujúceho príkladu:

```
mpicc -g -Wall -o hello hello.c
```

Úloha: Napíšte a skompilujte paralelný MPI program *Hello world* s použitím príkazu mpicc a potom s použitím príkazu gcc (prípadne inej alternatívy dostupnej v použitom systéme).

Príkaz `mpicc` je zväčša implementovaný ako skript (wrapper), ktorého úlohou je spustiť kompilátor jazyka C. Okrem toho zjednodušuje prácu tým, že kompilátoru dá informáciu o tom, kde hľadať hlavičkové súbory, a ktoré knižnice použiť pri linkovaní s objektovým kódom MPI programu. V prípade potreby použitia iného kompilátora je možné zistiť parametre pomocou príkazov:

```
mpicc --showme:compile
```

```
mpicc --showme:link
```

Väčšina systémov umožňuje spúšťanie MPI programu pomocou príkazu `mpiexec`, `mpirun` alebo `poe` podľa nasledujúceho príkladu:

```
mpiexec -n počet_procesov program
```

Takže na spustenie nášho programu ako jedného procesu môže použiť príkaz:

```
mpiexec -n 1 ./hello
```

a program vygeneruje nasledovný výstup:

```
Hello world
```

V prípade, že by sme chceli, aby náš MPI program bol vykonávaný viacerými procesmi, bude príkaz a výstup programu vyzerať nasledovne:

```
mpiexec -n 4 ./hello
```

```
    Hello world
    Hello world
    Hello world
    Hello world
```

Po spustení príkaz `mpiexec` zabezpečí, aby systém spustil požadovaný počet inštancií paralelného programu, ktoré môžu navzájom komunikovať. Tiež je možné určiť, ktorý proces bude vykonávaný ktorým procesorom. Pre viac podrobností ohľadne možností spúšťania MPI programov odporúčame preštudovať manuál príkazu.

Práca so vstupmi a výstupmi

V paralelných MPI programoch nie je určené, ktorý z procesov môže vypisovať výsledky na štandardný (`stdout`) alebo chybový (`stderr`) výstup. Spravidla k tomu používame štandardné funkcie `printf(...)` alebo `fprintf(stderr, ...)`, ktoré môžu byť zavolané v ktoromkoľvek procese vykonávajúcom paralelný program. V momente, keď sa pokúsia všetky procesy zapísať výstup v rovnakom čase, pokúsia sa pristupovať k tomu istému zdieľanému výstupnému zariadeniu. MPI neposkytuje samo o sebe nástroj pre plánovanie vykonávania týchto procesov, a preto môžu byť tieto výstupy vypísané v náhodnom poradí pri každom jednom spustení paralelného programu. Tento fakt si môžme overiť na programe 3.

Ak by sme chceli presne určiť poradie, v akom sa majú výstupy jednotlivých procesov vypísať, museli by sme to v programe špeciálne ošetriť. Jednou z možností je odoslanie výstupov zo všetkých procesov jednému z procesov, ktorý následne výsledky vypíše v programátorom definovanom poradí tak, ako je uvedené v programe 4.

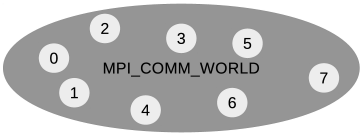
Náročnejšia situácia môže nastať, pokiaľ by paralelný program potreboval načítať údaje zo štandardného vstupu (`stdin`) pomocou štandardnej funkcie `scanf(...)`. V prípade, že by mali všetky procesy prístup k tomuto vstupu, nie je jasné, ktorý z procesov by mal načítať ktorú časť vstupu. Preto je v takomto prípade potrebné v programe ošetriť pomocou vetvenia to, aby vstupné údaje načítal iba jeden z procesov a následne ich predal ostatným procesom pomocou posielania správ. Na tento účel sú ideálne vhodné operácie kolektívnej komunikácie.

Komunikátor

Po spustení MPI programu a vykonaní funkcie `MPI_Init` je v rámci programu vytvorený spoločný priestor, v rámci ktorého sa uskutočňuje všetka komunikácia medzi všetkými procesmi MPI programu. Tento priestor sa označuje ako **komunikátor** `MPI_COMM_WORLD` a združuje všetky procesy, ktoré boli spustené systémom pri spustení paralelného programu. Každému z n procesov je pri inicializácií MPI prostredia priradené poradové číslo z intervalu od 0 do $n - 1$ označované ako **rank**. Celkový počet všetkých procesov združených v komunikátore označujeme ako **size**.

MPI_Comm_rank, MPI_Comm_size

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
```



Obr. 1: Komunikátor MPI_COMM_WORLD

Na zistenie poradového čísla procesu a celkového počtu spustených procesov paralelného MPI programu môžme použiť funkcie `MPI_Comm_rank` a `MPI_Comm_size`. Prvý argument oboch funkcií predstavuje MPI komunikátor a pomocou druhého argumentu je formou ukazovateľa vrátená požadovaná hodnota `rank` a `size`. Pre lepšiu názornosť je na obrázku 1 znázornený komunikátor s 8 procesmi s poradovými číslami 0 až 7.

Úloha: Napíšte, skompilujte a spustite paralelný MPI program číslo 3 tak, že bude vykonávaný postupne 1, 2 a 4 procesmi.

Úloha: Spustite program z predchádzajúcej úlohy niekoľkokrát po sebe tak, aby bol vykonávaný viacerými procesmi. Sledujte poradie vypísaných správ od jednotlivých procesov. Sú správy vypísané vždy v rovnakom poradí alebo sa poradie náhodne mení? Pokúste sa to zdôvodniť.

Pomocou týchto funkcií môžme výpis pozdravu nášho programu doplniť o informáciu s hodnotou `rank` a `size` podľa nasledujúceho programu 3.

V tejto časti je vhodné opäť pripomenúť, že sme kompilovali a spúšťali iba jeden program a všetky procesy paralelného MPI programu vykonávajú ten istý program podľa modelu SPMD. Ako sme už uviedli, nepredstavuje to významnú nevýhodu v porovnaní s modelom MPMD, pretože jednotlivým procesom môžme určiť vykonávanie odlišných častí toho istého programu na základe ich poradového čísla `rank` a s použitím vetvenia programu (if-else). Každá vetva tohto programu môže obsahovať iné podúlohy, ktoré budú riešené procesmi paralelne.

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

V predchádzajúcom prípade patrili všetky vytvorené procesy do jedného komunikátora `MPI_COMM_WORLD`. Môže však nastať prípad, kedy nie je potrebné, aby spolu komunikovali všetky procesy a je výhodnejšie rozdeliť procesy z komunikátora `MPI_COMM_WORLD` do viacerých komunikátorov. To umožní efektívnejšiu kolektívnu komunikáciu medzi procesmi, pretože správy budú predávané len medzi procesmi v rámci novovytvorených komunikátorov. Tým môžme zabezpečiť, aby neboli zbytočne všetky procesy zaťažované komunikáciou, ktorá sa ich netýka.

Funkcia `MPI_Comm_split` rozdelí procesy z pôvodného komunikátora `comm` do samostatných podskupín, pričom pre každú zadanú farbu určenú argumentom `color` (reprezentovanú nezáporným celým číslom) vytvorí samostatný nový komunikátor. Tento bude procesu vrátený pomocou ukazovateľa `newcomm`. Každý novovytvorený komunikátor bude obsahovať všetky procesy s rovnakou farbou. Poradové číslo procesov `rank` v novom komunikátore bude procesom pridelené na základe hodnoty argumentu `key`. Previazanosť na `rank` v rámci pôvodného komunikátora `comm` nie je nutné zachovať.

Funkcia `MPI_Comm_split` je kolektívna operácia, to znamená, že ju musia zavolať všetky procesy z pôvodného komunikátora, avšak každý proces môže uviesť svoje vlastné hodnoty argumentov `color` a `key`. V prípade, že niektorý proces nemá byť súčasťou žiadneho z novovytvorených komunikátorov, je možné zadať hodnotu farby `MPI_UNDEFINED`. V takomto prípade bude vrátená hodnota pre nový komunikátor `MPI_COMM_NULL`. Po vytvorení nových komunikátorov ostáva naďalej funkčný aj pôvodný komunikátor `MPI_COMM_WORLD` obsahujúci všetky procesy. Novovytvorený komunikátor je možné zrušiť zavolaním funkcie `MPI_Comm_free`.

Zdrojový kód 3: Paralelný program Hello world s informáciou o komunikátore

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int rank, size;

  //inicializácia MPI
  MPI_Init(&argc, &argv);

  //zistenie poradového čísla procesu - rank
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //zistenie celkového počtu procesov - size
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Hello world, proces %d z %d\n", rank, size);

  //ukončenie MPI
  MPI_Finalize();

  return 0;
}
```

2 Komunikácia

Komunikácia medzi MPI procesmi je zabezpečená pomocou posielania správ. Podľa toho, koľko procesov navzájom komunikuje a akým

spôsobom je komunikácia synchronizovaná, môžme rozdeliť komunikáciu medzi procesmi do určitých kategórií [8]. Podľa počtu procesov, ktoré spolu komunikujú, môže byť komunikácia:

- **adresná medzi dvoma procesmi** – navzájom komunikujú práve dva. Jeden proces správu odosiela druhému procesu, ktorý ju prijíma. Pri odosielaní je potrebné uviesť presnú hodnotu `rank` procesu, ktorému má byť správa odoslaná. Prijatie správy môže byť buď od konkrétneho procesu alebo aj od ľubovoľného procesu v rámci toho istého komunikátora.
- **kolektívna (skupinová)** – navzájom komunikujú všetky procesy v rámci jedného komunikátora alebo skupiny procesov. Pri skupinovej komunikácií sa komunikácie zúčastňujú všetky procesy. Použitie špecializovaného hardvéru môže viesť k vyššej efektívnosti kolektívnej komunikácie v porovnaní s použitím komunikácie medzi dvoma procesmi.

Ďalší rozdiel v komunikácií je v závislosti od spôsobov synchronizácie. Komunikáciu delíme na:

- **blokujúcu** – vykonávanie procesu je pozastavené až do momentu, kým nie je komunikácia ukončená, čiže kým nie je správa doručená a prijatá adresátom.
- **neblokujúcu** – vykonávanie procesu pokračuje ďalej okamžite po zavolaní funkcie a komunikácia je vykonaná na pozadí. Pomocou `MPI_Request` je možné overiť stav komunikácie alebo počkať na jej dokončenie podobne ako v blokujúcej komunikácií.

Pri komunikácii medzi procesmi ide v podstate o prenos určitých údajov, ktoré sú uložené v pamäti (bufer) jedného procesu – odosielateľa a sú doručené a uložené do pamäte druhého procesu – prijímateľa. Okrem údajovej časti každá správa obsahuje aj obálku, ktorá obsahuje informáciu o poradovom čísle procesu odosielateľa a prijímateľa správy, ako aj o údajovom type správy. Pre udávanie typu prenášaných údajov sa používajú špeciálne typy definované v MPI a nie štandardné typy jazyka C. Prehľad typov MPI a ich vzťah k štandardným typom jazyka C je uvedený v tabuľke 1.

Pri posielaní viacerých správ medzi dvoma procesmi je podstatné, že správy sú doručené v takom istom poradí, ako boli odoslané, a teda odovzdávanie správ je deterministické. Toto platí iba v prípade, že používame jednovláknové procesy. V prípade viacvláknových procesov, môže nastať situácia, že niekoľko paralelných vlákien odosiela správy adresované tomu istému prijímateľovi súčasne. V prípade, že prijímateľ prijíma správy sekvenčne, nie je zaručené poradie doručenia týchto správ, a teda odovzdávanie správ je nedeterministické.

Upravme program 3 tak, aby všetky správy na obrazovku vypisoval iba jeden proces. Ostatné procesy svoje správy odošlú tomuto procesu podľa programu 4.

Úloha: Čo by sa stalo, ak by sme nahradili hodnotu argumentu strlen(sprava)+1 na riadku 23, určujúceho dĺžku správy, hodnotou strlen(sprava)? Čo by sa stalo, ak by sme použili hodnotu MAXSTR? Vysvetlite, či by program pracoval správne, a či by bol efektívny.

Úloha: Upravte program tak, aby všetky správy vypísal proces s hodnotou rank rovnou 1.

Tabuľka 1: Prehľad údajových typov MPI

Typ MPI	Typ v jazyku C
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG	signed long long
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	byte
MPI_PACKED	(nemá ekvivalent)

Zdrojový kód 4: Paralelný program Hello world s posielaním správ

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <mpi.h>
4
5  #define MAXSTR 100
6
7  int main(int argc, char* argv[])
8  {
9    char sprava[MAXSTR];
10   int rank, size;
11
12   //inicializácia MPI
13   MPI_Init(&argc, &argv);
14
15   //zistenie poradového čísla procesu - rank
16   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17   //zistenie celkového počtu procesov - size
18   MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20   if(rank != 0) {
21     sprintf(sprava, "Hello world, proces %d z %d",
              rank, size);
22     //odoslanie správy
23     MPI_Send(sprava, strlen(sprava)+1, MPI_CHAR,
                0, 0, MPI_COMM_WORLD);
24   } else {
25     printf("Hello world, proces %d z %d\n", rank, size);
26     for(int i=1; i<size; i++) {
27       //prijatie správy
28       MPI_Recv(sprava, MAXSTR, MPI_CHAR,
                  i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29       printf("%s\n", sprava);
30     }
31   }
32
33   //ukončenie MPI
34   MPI_Finalize();
35   return 0;
36 }
```

Blokujúca komunikácia dvoch procesov

Blokujúca komunikácia dvoch procesov slúži na predanie správy od jedného procesu druhému, pričom je známe, ktorý proces správu odosiela, a ktorému procesu má byť správa doručená. Na to, aby sme mohli uskutočniť predanie správy medzi dvoma procesmi, musia byť splnené určité predpoklady:

- odosielateľ musí poznať poradové číslo (rank) prijímateľa správy,
- prijímateľ musí uviesť správneho odosielateľa,
- oba procesy musia mať rovnaký komunikátor,
- tag správy sa musí zhodovať u oboch procesov,
- typ odosielaných údajov uložených v pamäti sa musí zhodovať s údajovým typom správy,
- údajový typ musí byť rovnaký u oboch procesov,
- bufer prijímateľa musí byť dostatočne veľký na uloženie celej prijatej správy.

Predanie správy pri adresnej komunikácií sa môže uskutočniť v niektorom zo štyroch režimov:

- štandardný `MPI_Send`,
- bufrovaný (asynchrónny) `MPI_Bsend`,
- synchrónny `MPI_Ssend`,
- a režim pripravenosti `MPI_Rsend`.

MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm)
```

Funkcia `int MPI_Send` slúži na odoslanie správy štandardným spôsobom. Predanie správy procesu sa môže uskutočniť buď v synchrónnom alebo bufrovanom režime. Výber režimu je závislý na zdrojoch systému, dĺžke správy a pod., a nie je možné ho ovplyvňovať z úrovne programátora. V závislosti na voľbe režimu môže toto odosielanie v prípade použitia bufrovaného režimu skončiť skôr, ako bude zavolané odpovedajúce prijatie správy pomocou príkazu `MPI_Recv` druhým procesom. V prípade, že bude zvolené nebufrované odoslanie správy, napríklad z dôvodu výkonu alebo nedostatočnej veľkosti bufra, je toto dokončené až po zavolaní odpovedajúceho príkazu `MPI_Recv` druhým procesom.

Prvé tri argumenty určujú obsah posielanej správy, pričom `buf` predstavuje ukazovateľ na miesto v pamäti, kde sú údaje uložené. Nasledujúce parametre `count` a `type` určujú množstvo údajov, ktoré sa budú odosielať. Argument `type` udáva údajový typ položiek správy, teda napríklad `MPI_CHAR` a zároveň určuje veľkosť jednej položky správy. Argument `count` udáva počet položiek, z ktorých správa pozostáva. V programe 3 počet položiek odpovedá dĺžke reťazca správa plus ukončovací znak reťazca `\0` v jazyku C. Dĺžka správy sa následne vypočíta ako súčin veľkosti položky daného typu v B a počtu položiek.

Štvrtý argument `dest` udáva poradové číslo (rank) procesu, ktorému je správa odosielaná. Piaty argument `tag` môže nadobúdať hodnotu nezáporného celého čísla. Toto číslo slúži na rozlíšenie správ, ktoré by inak boli rovnaké, napríklad správy obsahujúce výsledok, ktorý sa má vypísať na obrazovke budú mať `tag = 0` a správy obsahujúce výsledok, ktorý sa má použiť v ďalšom výpočte budú mať `tag = 1`. Posledný argument `comm` je komunikátor. Pripomeňme, že ten obsahuje množinu procesov, ktoré môžu navzájom komunikovať. Správu odoslanú v rámci jedného komunikátora nie je možné prijať procesom v inom komunikátore.

Pre lepšie pochopenie si predstavme príklad, že máme dva paralelné programy, pomocou ktorých chceme študovať klimatické zmeny na Zemi. Prvý program modeluje zemskú atmosféru a druhý program slúži na modelovanie oceánov. Našou úlohou je tieto dva programy navzájom prepojiť, pretože oba zložky zohrávajú významnú úlohu pri klimatických zmenách, avšak chceme ich prepojiť len do takej miery, aby sme nenarušili ich správnu funkčnosť neúmyselným posielaním správ z druhého programu. Prvým riešením by bolo rozlišovať správy na základe hodnoty tag, pričom časť procesov modelujúcich atmosféru bude sledovať len správy s tag-om 0 a naopak. Elegantnejšie riešenie však predstavuje použiť dva rôzne komunikátory, čím zabezpečíme, aby žiadna zo správ nebola neúmyselne doručená inej skupine procesov.

MPI_Bsend

```
int MPI_Bsend(void *buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm)
```

Úloha: Upravte a odskúšajte program 4 s použitím funkcie MPI_Bsend.

Argumenty funkcie `MPI_Bsend` sú rovnaké ako pri použití funkcie `MPI_Send`, rozdiel je len v spôsobe, akým sa uskutoční predanie správy medzi komunikujúcimi procesmi. Pri použití bufrovaného režimu posielania správ sa po odoslaní správy táto uloží do alokovaného opakovane použiteľného bufra knižnice MPI. Počas kopírovania správy do bufra je vykonávanie procesu blokované a hneď po jeho dokončení je vrátené riadenie procesu, ktorý môže pokračovať ďalej bez nutnosti toho, aby bola zavolaná odpovedajúca funkcia na prijatie správy `MPI_Recv` druhým procesom. Preto toto odosielanie označujeme ako lokálne. Bufer používaný na uloženie správy je alokovaný používateľom pomocou volania funkcie `MPI_Buffer_attach` a uvoľnený pomocou volania funkcie `MPI_Buffer_detach`. Ak nie je alokovaný dostatočne veľký bufer pre uloženie správy, operácia bude ukončená s chybou.

MPI_Ssend

```
int MPI_Ssend(void *buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm)
```

Úloha: Upravte a odskúšajte program 4 s použitím funkcie MPI_Ssend.

Argumenty funkcie `MPI_Ssend` sú rovnaké ako pri predchádzajúcich funkciach. Funkciu `MPI_Ssend` je možné zavolať bez ohľadu na to, či bola predtým zavolaná funkcia pre prijatie správy odpovedajúcim

druhým procesom. Riadenie oboch procesov je blokované až do momentu, kedy je odoslaná správa prijatá druhým procesom pomocou funkcie `MPI_Recv`.

MPI_Rsend

```
int MPI_Rsend(void *buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm)
```

Úloha: Upravte a odskúšajte program 4 s použitím funkcie MPI_Rsend.

Na rozdiel od predchádzajúcich funkcií, funkciu `MPI_Rsend` môžme zavolať až v momente, keď sme si istý, že už bola zavolaná odpovedajúca funkcia `MPI_Recv`, v opačom prípade operácia skončí s chybou. Tento režim funguje podobne ako synchrónny režim avšak poskytuje lepší výkon.

MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype type,
          int src, int tag, MPI_Comm comm, MPI_Status *stat)
```

Funkcia pre prijatie správy `MPI_Recv` je vo všetkých štyroch uvedených blokujúcich režimoch rovnaká. Táto funkcia blokuje vykonávanie procesu do momentu, kedy je správa tomuto procesu doručená. Prvý argument `buf` funkcie predstavuje ukazovateľ na blok pamäte, kde má byť prijatá správa uložená. Argument `count` udáva maximálny počet položiek správy, ktoré je možné prijať a `type` je rovnaký ako v odpovedajúcej funkcii odosielania správy.

Úloha: Upravte a odskúšajte program 4 s použitím makier MPI_ANY_SOURCE a MPI_ANY_TAG pri prijímaní správ.

Argument `src` určuje, od ktorého odosielateľa má byť správa prijatá. V prípade, že nezáleží na tom, od ktorého z procesov s rovnakým komunikátorom bude správa prijatá, môžme túto hodnotu nahradiť makrom `MPI_ANY_SOURCE`. Podobne sa musí pre úspešné predanie správy zhodovať aj hodnota `tag` medzi odosielanou a prijímanou správou. Pre prijatie správy s ľubovoľnou hodnotou `tag`, ju môžme nahradiť makrom `MPI_ANY_TAG`. Hodnoty `MPI_ANY_SOURCE` a `MPI_ANY_TAG` je možné použiť len s funkciou prijímania správy `MPI_Recv` a nie odosielania správy.

Úloha: Napíšte program, pomocou ktorého otestujete rýchlosť komunikácie medzi dvoma procesmi. Proces 0 odošle správu procesu 1 a ten po doručení správy odošle správu späť procesu 0.

V prípade, že bola správa prijatá od ľubovolného procesu s ľubovoľnou hodnotu tag, túto informáciu je možné získať pomocou posledného argumentu `stat`. Typ `MPI_Status` je štruktúra, ktorá obsahuje tri položky `MPI_SOURCE`, `MPI_TAG` a `MPI_ERROR`. Na základe tohto vieme o doručenej správe zistiť informácie o:

- množstve údajov v správe,
- odosielateľovi správy,
- hodnote tag správy.

Uvedené hodnoty získame ako položky premennej `stat.MPI_SOURCE` a `stat.MPI_TAG`. Množstvo údajov prijatých v správe nie je priamo uložený v premennej `stat`, ale pomocou funkcie `MPI_Get_count` a údajového typu ho vieme jednoducho zistiť nasledovne:

```
MPI_Get_count(&stat, type, &count)
```

pričom do premennej `count` bude uložený počet položiek prijatej správy. Táto hodnota sa nenachádza priamo v premennej `stat` z jednoduchého dôvodu, pretože je závislá od údajového typu správy a je potrebné ju vypočítať ako pomer množstva prijatých údajov a veľkosti jednej položky správy.

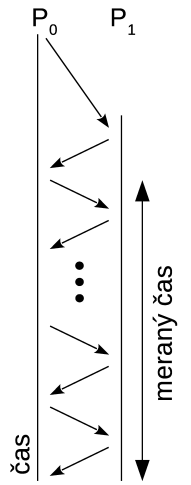
Premennú `stat` je možné pri volaní funkcie `MPI_Recv` nahradiť makrom `MPI_STATUS_IGNORE`, pokiaľ nebudeme potrebovať informácie z tejto premennej.

Pri použití blokujúcej komunikácie medzi dvoma procesmi môže nastať prípad uviaznutia a to v prípade, keď si oba procesy budú súčasne navzájom predávať správy tak, že najprv zavolajú funkciu `MPI_Send` a následne `MPI_Recv`. V prípade, že bude použitý synchrónny režim komunikácie, dôjde k uviaznutiu procesov, pretože oba procesy budú blokované až do momentu prijatia správy, čiže do volania odpovedajúcej funkcie `MPI_Recv`, ktoré z dôvodu blokovania procesov nenastane.

Riešená úloha

V MPI programoch komunikujú dva procesy pomocou posielania správ cez prepojovaciu komunikačnú sieť. Každá z použitých sietí ma svoje vlastnosti a charakteristiky v závislosti na type použitej siete. Medzi dve základné vlastnosti patrí aj jej latencia a šírka komunikačného kanálu.

Vytvorte paralelný program 5 podľa obrázku 2, ktorý spustíte tak, aby bol vykonávaný dvoma procesmi. Proces P_0 odošle správu procesu P_1. Proces P_1 správu príjme a následne ju vráti späť procesu P_0. Zopakujte tento proces v cykle 50-krát. Odmerajte čas tesne pred a po volaní tohto cyklu. V procese P_0 vypíšte čas potrebný na predanie jednej správy medzi procesmi. Na meranie času môžte použiť rozdiel časov získaných zavolaním funkcie, ktorá vracia ako návratovú hodnotu čas wall-clock v sekundách. `double MPI_Wtime(void)`



Obr. 2: Schéma posielania správ medzi dvoma procesmi

Ako môžeme vidieť na obrázku 2, čas predania prvej správy môže byť skreslený z dôvodu oneskoreného štartu procesu P_1. Preto čas prenosu prvej správy z nášho merania vylúčime. Latenciu siete vieme overiť pri predávaní krátkych správ. Šírku komunikačného kanálu vieme vypočítať ako podiel veľkosti správy a času potrebného na jej prenos. Upravte program tak, aby proces P_0 vypísal čas prenosu a prenosovú šírku.

Merania zopakujte niekoľkokrát pre rôzne veľkosti správ 8 B (premenná typu `double`), 512 B ($= 8 * 64$), 32kB ($= 8 * 64^2$), 2 MB ($= 8 * 64^3$). Na predávanie správ odskúšajte rôzne režimy blokujúcej komunikácie.

Pri spúšťaní je potrebné zabezpečiť, aby bol na každom uzle vykonávaný práve jeden proces, inak by sme reálne nemerali parametre komunikačnej siete. Za týmto účelom je možné použiť príkaz:

```
mpiexec -n 2 --map-by ppr:1:node --hostfile hosts
```

pričom súbor hosts obsahuje zoznam sieťových adries použitých výpočtových uzlov.

Zdrojový kód 5: Program ping-pong

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int rank, size;
  MPI_Status status;

  //inicializácia MPI
  MPI_Init(&argc, &argv);
  //zistenie poradového čísla procesu - rank
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //zistenie celkového počtu procesov - size
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  double start, stop, sprava=123.456;

  start = MPI_Wtime();  //meranie času
  for(int i=0; i<50; i++) {
    if(!rank) {
      //odoslanie správy procesom 0
      MPI_Send(&sprava, 1, MPI_DOUBLE, 1, 0,
               MPI_COMM_WORLD);
      //prijatie správy procesom 0
      MPI_Recv(&sprava, 1, MPI_DOUBLE, 1, 0,
               MPI_COMM_WORLD, &status);
    } else {
      //prijatie správy procesom 1
      MPI_Recv(&sprava, 1, MPI_DOUBLE, 0, 0,
               MPI_COMM_WORLD, &status);
      //odoslanie správy procesom 1
      MPI_Send(&sprava, 1, MPI_DOUBLE, 0, 0,
               MPI_COMM_WORLD);
    }
  }
  stop = MPI_Wtime();  //meranie času
  if(!rank) {
    printf("Cas prenosu: %f mikrosekund.\n",
           (stop-start)/(2*50)*1e6);
    printf("Bandwidth: %f B/s\n",
           sizeof(sprava)*2*50/(stop-start));
  }

  //ukončenie MPI
  MPI_Finalize();
  return(0);
}
```

Neblokujúca komunikácia dvoch procesov

Hlavným zmyslom použitia neblokujúcej komunikácie medzi dvoma procesmi je zvýšenie výkonnosti paralelného programu spôsobené možnosťou súčasného výpočtu a komunikácie procesov. Toto zvýšenie výkonu však nie je vždy automaticky garantované. Pri použití neblokujúcej komunikácie je riadenie vrátené okamžite po volaní funkcie odosielania správy, bez ohľadu na použitý režim. To znamená, že správa z bufra ešte nemusí byť prenesená celá. Ak by sme chceli bufer znova použiť, je potrebné najprv overiť, či bolo odosielanie správy už ukončené.

Podobne je tomu aj v prípade volania neblokujúcej funkcie na prijímanie správy, kde je riadenie vrátené okamžite po jej zavolaní, teda skôr, ako môže byť správa prenesená do bufra prijímajúceho procesu. Ukončenie prenosu je opäť možné otestovať. MPI umožňuje kombinovanie neblokujúceho odosielania správ s blokujúcim prijímaním správ a opačne.

Zvýšenie efektívnosti paralelného programu je možné dosiahnuť práve využitím možnosti súčasného prenosu údajov na pozadí, zatiaľ čo proces môže vykonávať určité výpočty. Časté volanie funkcií na overovanie ukončenia predávania správ vedie k nežiadúcemu zníženiu výkonu paralelného programu, ktoré môže viesť dokonca k pomalšiemu programu než s použitím blokujúcej komunikácie.

MPI_Isend

```
int MPI_Isend(void *buf, int count, MPI_Datatype type,
         int dest, int tag, MPI_Comm comm, MPI_Request *req)
```

Funkcia `int MPI_Isend` slúži na neblokujúce odoslanie správy štandardným spôsobom. Rozdielom oproti blokujúcim funkciám odosielania správ je posledný argument `req`, ktorý slúži ako komunikačná požiadavka (handle) použitá na overenie stavu ukončenia predávania správy.

Podobne ako pri blokujúcej komunikácii aj tu je možnosť použitia ďalších režimov odosielania správ. Pre bufrovaný režim je potrebné volať funkciu `MPI_Ibsend`, pre synchrónny režim funkciu `MPI_Issend` a pre odoslanie správy v režime pripravenia je to funkcia `MPI_Irsend`.

MPI_Irecv

```
int MPI_Irecv(void *buf, int count, MPI_Datatype type,
          int src, int tag, MPI_Comm comm, MPI_Request *req)
```

Funkcia pre neblokujúce prijatie správy je `MPI_Irecv`. Význam posledného argumentu funkcie `req` je obdobný ako pri funkciách odosielania správ, a to pre kontrolu stavu dokončenia neblokujúcej komunikácie.

Keďže prenos správ medzi procesmi pomocou neblokujúcich operácií sa vykonáva na pozadí a riadenie je odovzdané naspäť procesu okamžite po ich zavolaní, pred samotným použitím údajov zo správy

je potrebné preveriť, či bol prenos správy už úspešne dokončený a s predávanými údajmi je možné bezpečne pracovať.

MPI_Wait

```
int MPI_Wait(MPI_Request *req, MPI_Status *stat)
```

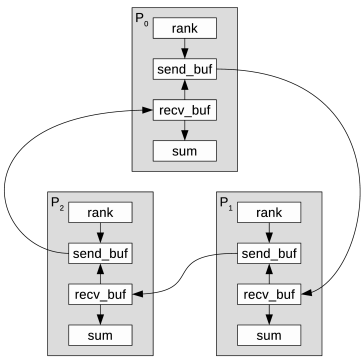
Funkcia `MPI_Wait` slúži na blokovanie vykonávania procesu do momentu, kým nie je dokončená konkrétna požiadavka `req`, ako napríklad neblokujúca verzia funkcií pre odosielanie alebo prijímanie správ. Po úspešnom dokončení aktivity, na ktorú sa čakalo, pokračuje proces vo vykonávaní programu. V prípade, že nie je požadovaná informácia z premennej `stat`, je pri volaní funkcie premennú možné nahradiť makrom `MPI_STATUS_IGNORE`. V prípade, že použijeme túto funkciu na kontrolu požiadavky, ktorá už bola dokončená a hodnota `req` je neaktívna alebo nastavená na `MPI_REQUEST_NULL`, toto volanie nebude mať žiaden efekt.

MPI_Test

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *stat)
```

Funkcia `MPI_Test` slúži podobne ako `MPI_Wait` na overenie stavu dokončenia konkrétnej požiadavky `req`. Rozdiel je v tom, že táto operácia nie je blokujúca, a teda vykonávanie programu po jej zavolaní pokračuje aj v prípade, že vykonávanie testovanej požiadavky ešte stále nebolo dokončené. Výsledkom volania funkcie je logická hodnota argumentu `flag`, ktorá udáva informáciu, či bolo vykonávanie testovanej požiadavky dokončené.

Knižnica MPI obsahuje viacero funkcií, pomocou ktorých môžme kontrolovať dokončenie aj viacerých požiadaviek súčasne. Spomenieme aspoň niektoré z nich. Funkcia `MPI_Waitany` umožňuje zadať ako argument pole obsahujúce viacero požiadaviek a počkať na dokončenie ľubovoľnej z nich, pričom vráti ako argument index dokončenej požiadavky. Naopak funkcia `MPI_Waitall` umožňuje blokovať vykonávanie procesu do momentu, kým nie sú dokončené všetky požiadavky zadané v poli, ako argument funkcie.



Obr. 3: Schéma posielania správ susedným procesom

Riešená úloha

V paralelných MPI programoch môže nastať situácia, keď všetky procesy vykonávajúce program musia medzi sebou súčasne komunikovať. Napíšte program 6 podľa obrázku 3, pomocou ktorého overíte rôzne spôsoby komunikácie susedných procesov a ich rýchlosť. Predstavme si, že všetky procesy vykonávajúce program zoradíme podľa ich hodnoty `rank` do kruhu. Na začiatku každý z procesov vloží svoju hodnotu `rank` do bufra na odosielanie správ. Každý z procesov odošle túto správu svojmu susednému procesu (`rank+1`). Každý proces počíta sumu hodnôt zo všetkých prijatých správ. Prijatú správu vloží do bufra na odosielanie a opäť odošle svojmu susednému procesu.

Tento proces zopakuje každý z procesov toľkokrát, koľko procesov je zapojených do výpočtu.

Program vyskúšajte pre rôzne kombinácie modelov komunikácie:

- `MPI_Irecv + MPI_Send`
- `MPI_Irecv + MPI_Isend`
- `MPI_Isend + MPI_Recv`
- `MPI_Isend + MPI_Irecv`
- `MPI_Sendrcv`
- `MPI_Neighbor_alltoall`

Poznámka: Funkcia MPI_Sendrcv kombinuje spoločne funkciu troch funkcii MPI_Irecv + MPI_Send + MPI_Wait. Funkcia MPI_Neighbor_alltoall patrí do skupiny operácií kolektívnej komunikácie.

Následne odskúšajte program tak, že nahradíte sled funkcií `MPI_Issend` + `MPI_Recv` + `MPI_Wait` funkciami `MPI_Irecv` + `MPI_Ssend` + `MPI_Wait` alebo `MPI_Irecv` + `MPI_Issend` + `MPI_Waitall`.

Poznámka: V pôvodnom programe 6 nie je možné nahradiť funkciu MPI_Issend blokujúcou funkciou MPI_Ssend, pretože by došlo k uviaznutiu procesov.

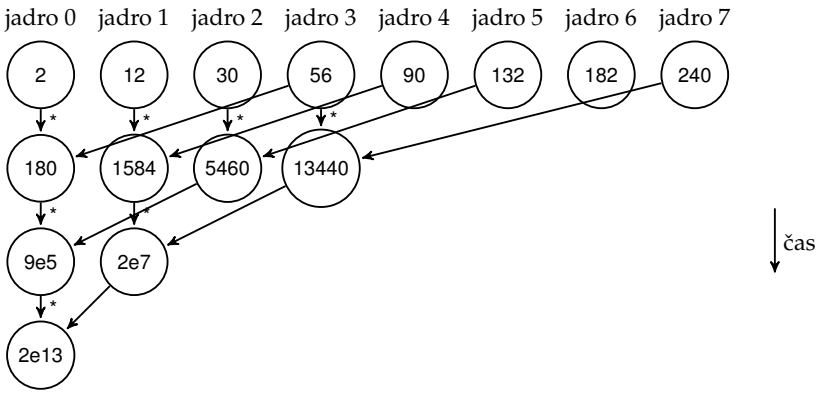
Zdrojový kód 6: Program pre posielanie správ susednému procesu

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int rank, size;
  MPI_Status status;
  MPI_Request req;

  //inicializácia MPI
  MPI_Init(&argc, &argv);

  //zistenie poradového čísla procesu - rank
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //zistenie celkového počtu procesov - size
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int send_buf, recv_buf, sum=0, odosielatel, adresat;

  send_buf = rank;
  //výpočet rankov odosielateľa a adresáta
  odosielatel = (rank - 1 + size) % size;
  adresat = (rank + 1) % size;

  for(int i=0; i<size; i++) {
    //odoslanie správy
    MPI_Issend(&send_buf, 1, MPI_INT, adresat,
               0, MPI_COMM_WORLD, &req);
    //prijatie správy
    MPI_Recv(&recv_buf, 1, MPI_INT, odosielatel,
             0, MPI_COMM_WORLD, &status);
    //čakanie na dokončenie neblokujúcej operácie
    MPI_Wait(&req, &status);
    sum += recv_buf;
    send_buf = recv_buf;
  }

  printf("Proces %d vypocital sumu %d\n", rank, sum);

  //ukončenie MPI
  MPI_Finalize();

  return(0);
}
```

Kolektívna komunikácia procesov

Kolektívna komunikácia procesov umožňuje všetkým procesom v rámci komunikátora, aby si mohli navzájom vymieňať údaje. Typickým príkladom použitia kolektívnej komunikácie môže byť prípad, keď každý proces rieši určitú podúlohu a výsledkom je čiastočný výsledok. Vhodnou kombináciou čiastočných výsledkov potom vieme získať celkový výsledok riešeného problému.

Takúto úlohu je bezpochyby možné riešiť aj pomocou adresnej komunikácie medzi dvoma procesmi. Pripomeňme si príklad znázornený na obrázku **??**, kde sme hľadali efektívny spôsob kombinovania čiastočných výsledkov pri paralelnom výpočte hodnoty faktoriálu. Uvedený spôsob kombinácie výsledkov je efektívnejší z pohľadu počtu súčasne vykonaných prenosov správ a sčítaní pri kombinovaní čiastočných výsledkov v porovnaní s postupom, kde by všetky čiastočné výsledky boli odoslané iba jednému procesu, ktorý ich potom postupne skombinuje. Nie je ťažké si uvedomiť, že takéto riešenie by si vyžiadalo podstatne zložitejší výsledný program. Jednoduchším riešením sa môže javiť kombinácia čiastočných výsledkov podľa schémy 4.



Obr. 4: Alternatívny spôsob kombinovania výsledkov od všetkých procesov

Rozhodnúť o tom, ktorý z uvedených spôsobov kombinácie čiastočných výsledkov je efektívnejší, nie je jednoduché ani v prípade, že by sme sa ich všetky pokúsili naprogramovať a otestovať. Dospeli by sme k záverom, že efektívnosť jedného, či druhého spôsobu je závislá nielen od konkrétneho problému, jeho veľkosti a počtu procesov, ale aj od samotného systému, na ktorom je program vykonávaný. Takže použitie adresnej komunikácie medzi dvoma procesmi sa v tomto prípade javí ako nepraktické.

Nie všetky funkcie kolektívnej komunikácie musia zabezpečovať predávanie správ medzi procesmi. Avšak, podstatný fakt je, že na kolektívnej komunikácií spravidla participujú všetky procesy v rovnakom komunikátore a tieto je preto potrebné synchronizovať. Operácie kolektívnej komunikácie je potrebné zavolať v každom procese a sú ukončené až v momente, keď je komunikácia dokončená každým jedným participujúcim procesom. Preto ich je vhodné používať pre riešenie približne rovnako výpočtovo náročných podúloh. Bežnou začiatočníckou chybou je časté používanie zbytočnej synchronizácie procesov, ktoré ma za následok zvýšenie času réžie paralelného programu, a teda aj dlhší čas potrebný na vyriešenie úlohy.

Poznámka: Od verzie MPI-3.0 je možné použiť aj neblokujúce verzie operácií pre kolektívnu komunikáciu. Názvy týchto funkcií sú tvorené MPI_I a názov operácie. Umožňuje súbežne vykonávať výpočty a prenos údajov alebo vykonávať niekoľko prenosov údajov v rámci prekrývajúcich sa komunikátorov bez problému uviaznutia alebo serializácie programu.

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

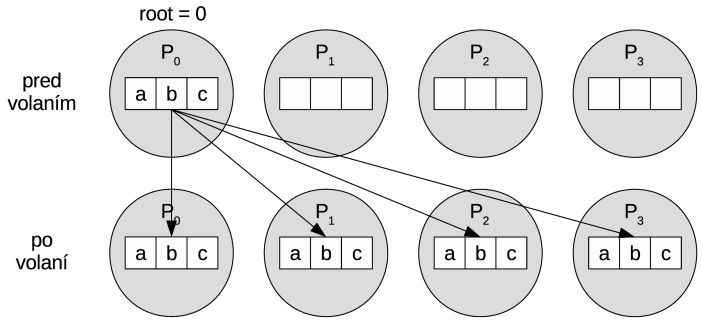
Základnou kolektívnou operáciou na synchronizáciu procesov je funkcia `MPI_Barrier`. Po zavolaní tejto funkcie je blokované vykonávanie volajúceho procesu až do momentu, kým nie je funkcia zavolaná každým procesom v rámci komunikátora. To znamená, že všetky procesy musia v tomto mieste programu na seba počkať a až potom môžu pokračovať v ďalšom vykonávaní programu. Jediným argumentom funkcie `comm` je komunikátor.

Použitie tohto príkazu v praxi nie je bežné, pretože všetky operácie kolektívnej komunikácie, ktoré umožňujú výmenu údajov, majú v sebe implicitne implementovanú synchronizáciu. Praktické využitie nachádza skôr pri ladení alebo profilovaní programu.

MPI_Bcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
              int root, MPI_Comm comm)
```

V paralelných MPI programoch nastáva pomerne často situácia, keď jeden proces načíta vstupné údaje potrebné pre výpočet a tieto následne rozpošle všetkým ostatným procesom. K tomuto účelu je vhodné použiť funkciu kolektívnej komunikácie `MPI_Bcast`, ktorá slúži na vysielanie (broadcast) rovnakej správy všetkým procesom v rámci komunikátora podľa obrázku 5. Je potrebné, aby túto funkciu volali všetky procesy v komunikátore, pričom argument `buf` slúži ako ukazovateľ na bufer pre odoslanie alebo prijatie správy. Argumenty `count` udávajú počet položiek a `type` údajový MPI typ položiek.



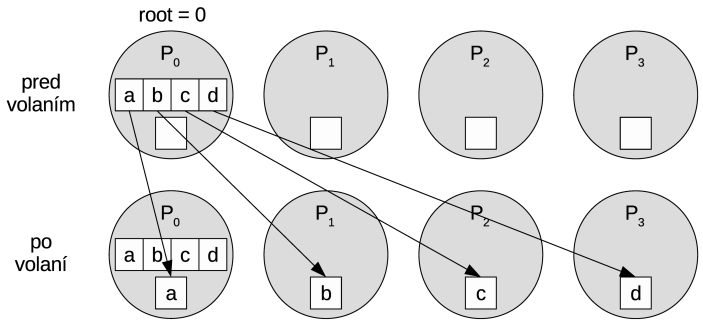
Obr. 5: Schéma činnosti funkcie MPI_Bcast

Správa je vysielaná jedným procesom určeným hodnotou argumentu `root`. Proces, ktorý zhodnú hodnotu `rank` s hodnotou argumentu `root` odošle obsah svojho bufra všetkým ostatným procesom, ktoré správu príjmu do svojho bufra. Hodnota argumentu `root` musí mať vo všetkých funkciu volajúcich procesoch rovnakú hodnotu. Táto operácia automaticky vyžaduje synchronizáciu všetkých procesov v rámci spoločného komunikátora určeného argumentom `comm`.

MPI_Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Operácia kolektívnej komunikácie `MPI_Scatter` slúži na rozdistribuovanie určitých údajov medzi všetkých *n* procesov v komunikátore tak, že jeden proces rozdelí odosielanú správu na *n* častí a každému procesu odošle odpovedajúcu časť na základe jeho hodnoty `rank`. Tento proces je schématicky znázornený na obrázku 6.



Obr. 6: Schéma činnosti funkcie MPI_Scatter

Prvé tri argumenty funkcie udávajú ukazovateľ `sendbuf`, počet položiek odoslaných každému procesu `sendcount` a MPI údajový typ `sendtype` bufra pre údaje, ktoré majú byť rozdistribuované procesom. Nasledujúca trojica argumentov udáva ukazovateľ `recvbuf`, počet prijatých položiek `recvcount` a MPI údajový typ `recvtype` bufra pre prijaté údaje. Argument `root` musí byť rovnaký vo všetkých volajúcich procesoch a určuje proces s hodnotou `rank`, ktorý bude údaje distribuovať. Táto operácia automaticky vyžaduje synchronizáciu všetkých procesov v rámci spoločného komunikátora určeného argumentom `comm`.

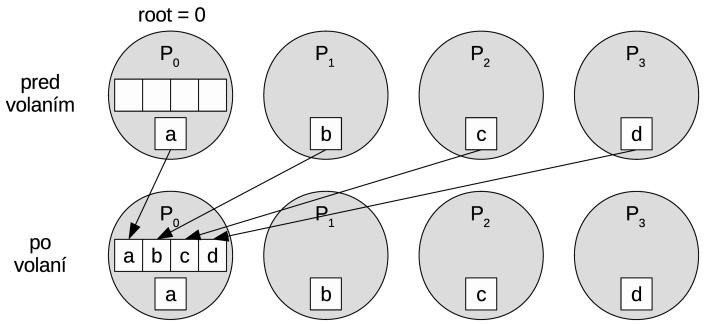
Pre nerovnomerné rozdelenie množstva údajov pre jednotlivé procesy je možné použiť funkciu `MPI_Scatterv`.

MPI_Gather

```
int MPI_Gather(const void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Operácia kolektívnej komunikácie `MPI_Gather` predstavuje opačnú operáciu k operácii `MPI_Scatter`. To znamená, že proces s hodnotou `rank` rovnou hodnote argumentu `root` zozbiera časti údajov od jednotlivých procesov. Ostatné argumenty funkcie sú identické s funkciou `Scatter`. Tento proces je schématicky znázornený na obrázku 7.

Pre nerovnomerné rozdelenie množstva údajov pre jednotlivé procesy je možné použiť funkciu `MPI_Gatherv`.

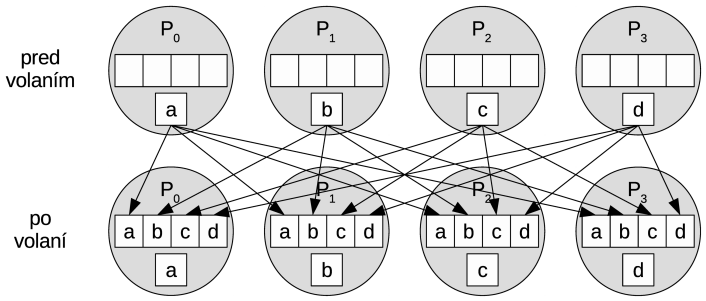


Obr. 7: Schéma činnosti funkcie MPI_Gather

MPI_Allgather

```
int MPI_Allgather(const void *sendbuf, int sendcount,
              MPI_Datatype sendtype, void *recvbuf,
              int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```

Operácia kolektívnej komunikácie `MPI_Allgather` vykoná rovnakú funkciu ako funkcia `MPI_Gather` s tým rozdielom, že jednotlivé časti údajov sú zozbierané v každom z procesov, takže na konci má každý z procesov kompletné údaje, podľa schémy na obrázku 8. Argumenty funkcie majú rovnaký význam ako pri funkcii `MPI_Gather` s tým rozdielom, že sa neuvádza argument `root`, pretože výsledné údaje budú k dispozícii vo všetkých procesoch v rámci komunikátora.



Obr. 8: Schéma činnosti funkcie MPI_Allgather

Pre nerovnomerné rozdelenie množstva údajov pre jednotlivé procesy je možné použiť funkciu `MPI_Allgatherv`.

MPI_Reduce

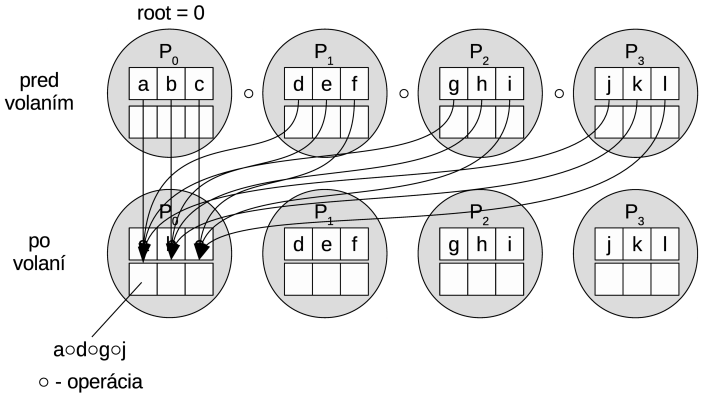
```
int MPI_Reduce(const void *sendbuf, void *recvbuf,
              int count, MPI_Datatype type,
              MPI_Op op, int root, MPI_Comm comm)
```

Operácia kolektívnej komunikácie `MPI_Reduce` umožňuje navzájom kombinovať čiastočné výsledky uložené v bufry s ukazovateľom `sendbuf` z jednotlivých procesov pomocou niektorej z binárnych asociatívnych operácií uvedených v tabuľke 2, určenej pomocou argumentu `op`. Každý proces môže do redukcie prispieť počtom prvkov `count`, ktoré sú údajového typu `type`. Výsledok redukcie je vložený

do bufra s ukazovateľom `recvbuf` v procese s hodnotou `rank` rovnou hodnote argumentu `root`, podľa schémy na obrázku 9.

Tabuľka 2: Prehľad operácií redukcie

Názov operácie	Význam operácie
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	suma
MPI_PROD	produkt
MPI_LAND	logický súčin
MPI_BAND	bitový súčin
MPI_LOR	logický súčet
MPI_BOR	bitový súčet
MPI_LXOR	logický exkluzívny súčet
MPI_BXOR	bitový exkluzívny súčet
MPI_MAXLOC	maximum a pozícia maxima
MPI_MINLOC	minimum a pozícia minima



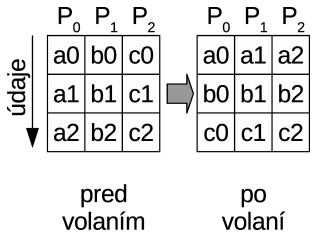
Obr. 9: Schéma činnosti funkcie MPI_Reduce

Okrem uvedených základných operácií redukcie je možné definovať aj svoje vlastné operácie pomocou funkcie `MPI_Op_create`. Vďaka tomu je možné, že údajový typ odosielanej správy nemusí byť len obyčajné číslo, ale aj zložitejší používateľom definový typ.

MPI_Allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf,
              int count, MPI_Datatype type,
              MPI_Op op, MPI_Comm comm)
```

Operácia kolektívnej komunikácie `MPI_Allreduce` umožňuje navzájom kombinovať čiastočné výsledky od všetkých procesov pomocou určenej operácie. Základným rozdielom voči funkcii `MPI_Reduce` je to, že po skončení operácie budú výsledky uložené do bufra vo všetkých procesoch a tie ich následne môžu ďalej použiť. Argumenty funkcie majú rovnaký význam ako pri funkcii `MPI_Reduce` bez potreby zadávania parametra `root`.



Obr. 10: Schéma činnosti funkcie MPI_Alltoall

Úloha: Čo vykonajú jednotlivé operácie kolektívnej komunikácie v prípade, ak komunikátor obsahuje iba jeden proces?

MPI_Alltoall

```
int MPI_Alltoall(const void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

Pri operácii kolektívnej komunikácie `MPI_Alltoall` všetky procesy posielajú správy všetkým ostatným procesom. Význam jednotlivých argumentov funkcie je obdobný ako pri predchádzajúcich funkciách. Vykonanie operácie spočíva v tom, že proces i odošle j-tú časť údajov procesu j. Proces j príjme časť údajov od procesu i a uloží ich na pozíciu i, podľa schémy na obrázku 10.

Pre nerovnomerné rozdelenie množstva údajov pre jednotlivé procesy je možné použiť funkciu `MPI_Alltoallv`. Je vhodné poznamenať, že pri komunikácii je vhodné používať v čo najväčšej miere údajové štruktúry s rovnakou komunikačnou náročnosťou.

Riešená úloha

Ako vhodný príklad na demonštráciu použitia kolektívnej komunikácie nám môže poslúžiť výpočet hodnoty skalárneho súčinu dvoch vektorov podľa vzťahu 1.

$$a \cdot b = \sum_{i=1}^{n} a_i b_i \tag{1}$$

Úloha: Upravte program 7 tak, aby výsledky vypísali všetky procesy. Následne nahraďte operáciu MPI_Reduce operáciou MPI_Allreduce a porovnajte získané výsledky.

Ako môžeme vidieť v programe 7, vektory `A` a `B` sú inicializované len procesom číslo 0. Ostatné procesy tieto hodnoty nemajú k dispozícii. Preto je najprv potrebné rozdeliť výpočty jednotlivým procesom a každému procesu odoslať odpovedajúce potrebné údaje (časť vektorov) operáciou `MPI_Scatter` na riadkoch 34 a 35. Následne každý z procesov môže vypočítať skalárny súčin pridelenej časti vektora a čiastočný výsledok uložiť do premennej `workResult`. Následne je potrebné výsledky v tejto premennej zozbierať a vypočítať globálny výsledok. K tomu je vhodné použiť operáciu redukcie `MPI_Reduce` na riadku 43.

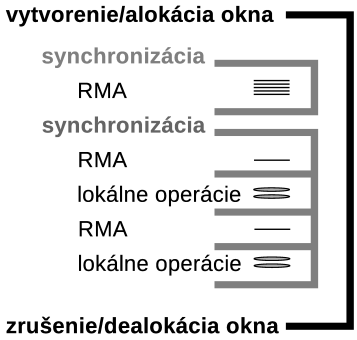
Zdrojový kód 7: Program na výpočet skalárneho súčinu vektorov

```
#include <stdio.h>
#include <mpi.h>

#define MAX 100

int main(int argc, char* argv[])
{
  int rank, size;
  MPI_Status status;
  MPI_Request req;

  //inicializácia MPI
  MPI_Init(&argc, &argv);
  //zistenie poradového čísla procesu - rank
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //zistenie celkového počtu procesov - size
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  double A[MAX], B[MAX], result;

  //údaje uložené vo vektoroch A, B bude mať len proces 0
  if(!rank) {       //túto časť vykoná iba proces 0
    for(int i=0; i<MAX; i++) {
      A[i] = i;     //inicializácia vektora A
      B[i] = i*i;   //inicializácia vektora B
    }
  }

  //počet položiek pridelených každému procesu
  const int count = MAX / size;
  double workA[count], workB[count], workResult=0;

  //rozdelenie vektorov na časti jednotlivým procesom
  MPI_Scatter(A, count, MPI_DOUBLE, workA, count,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(B, count, MPI_DOUBLE, workB, count,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);

  //výpočet čiastkového výsledku
  for(int i=0; i<count; i++) {
    workResult += workA[i] * workB[i];
  }

  //kombinovanie poslaných čiastkových výsledkov procesom 0
  MPI_Reduce(&workResult, &result, 1, MPI_DOUBLE, MPI_SUM,
    0, MPI_COMM_WORLD);

  if(!rank) {printf("Vysledok je: %f\n", result);}

  //ukončenie MPI
  MPI_Finalize();
  return(0);
}
```

Jednostranná komunikácia procesov

Úloha: Upravte programy 6 a 7 tak, aby ste použitú komunikáciu nahradili jednostrannou komunikáciou.

Vytvoreniu štandardu MPI-3.0 predchádzala rozsiahla revízia rozhrania pre **vzdialený prístup do pamäte** (RMA – Remote Memory Access). Výsledkom bolo vytvorenie rozhrania pre jednostrannú komunikáciu procesov, čo prispelo k vyššej efektívnosti a použiteľnosti MPI RMA [9]. Cieľom implementácie rozhrania pre MPI RMA bolo zvýšenie súbežnosti v porovnaní s tradičným prístupom pomocou posielania správ [8]. Tento prístup umožňuje lepšie vyvažovanie efektívnosti a prenositeľnosť na širokú škálu architektúr ako sú SMP, SMP klaster, NUMA, MPP alebo aj bežne pracovné stanice.



Obr. 11: Priebeh jednostrannej komunikácie v MPI

Operácie jednostrannej komunikácie umožňujú procesom vytvoriť tzv. okno v svojej pamäti, z ktorého môžu ostatné procesy čítať alebo doň zapisovať údaje pomocou operácií typu get a put. Tieto operácie sú neblokujúce, a preto je ich potrebné použiť v súčinnosti s funkciami pre synchronizáciu. Proces synchronizácie pri týchto operáciách je teda oddelený od operácií komunikácie. Z pohľadu synchronizácie a zapojenosti zdrojového a cieľového procesu do komunikácie poskytuje MPI dva módy:

- komunikácia s pasívnym cieľom – na komunikácii sa zúčastňuje iba zdrojový proces odosielajúci údaje,
- komunikácia s aktívnym cieľom – na komunikácii sa zúčastňujú oba procesy, odosielajúci aj prijímajúci údaje.

Operácie jednostrannej komunikácie, ako môžeme vidieť na obrázku 11, pozostávajú z troch krokov:

- vytvorenie a alokovanie okna,
- vzdialeného prístupu k pamäti,
- synchronizácie.

MPI_Win_create

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                   MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

Funkcia `MPI_Win_create` umožňuje poskytnúť prístup pre ostatné procesy k určitej časti pamäte, ktorá už bola predtým v procese alokovaná. Jedná sa o kolektívnu operáciu, to znamená, že ju musia zavolať všetky procesy v rámci komunikátora `comm`. Argument funkcie `base` určuje ukazovateľ na začiatočnú adresu okna v pamäti. Argument `size` udáva veľkosť okna v bajtoch a `disp_unit` udáva posunutie (veľkosť jednej položky) v bajtoch. Argument `info` poskytuje informácie o použití okna pre lepšiu optimalizáciu. Po vykonaní funkcie obsahuje argument `win` ukazovateľ na vytvorené okno, ktoré je ďalej možné používať.

MPI_Win_allocate

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit,
                     MPI_Info info, MPI_Comm comm,
                     void *baseptr, MPI_Win *win)
```

Funkcia `MPI_Win_allocate` má podobnú funkciu ako funkcia `MPI_Win_create` s tým rozdielom, že najprv v pamäti pre každý proces alokuje miesto a následne vráti ukazovateľ na okno s alokovanou pamäťou. Význam argumentov je totožný s funkciou, pričom nový argument `baseptr` obsahuje po vykonaní funkcie lokálny ukazovateľ na alokovanú pamäť v procese.

MPI_Win_allocate_shared

```
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit,
                            MPI_Info info, MPI_Comm comm,
                            void *baseptr, MPI_Win *win)
```

Podobný význam má aj funkcia `MPI_Win_allocate_shared`. Rozdiel je ten, že každý proces alokuje miesto v pamäti o veľkosti `size`, pričom toto miesto s ukazovateľom `baseptr` je zdieľané a prístupné v rámci celého uzla so zdieľanou pamäťou.

MPI_Win_shared_query

```
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
                         int *disp_unit, void *baseptr)
```

Pomocou funkcie `MPI_Win_allocate_shared` je možné alokovať pamäť naprieč všetkými procesmi v rámci komunikátora. S použitím funkcie `MPI_Win_shared_query` je možné neskôr zistiť informácie o existujúcom okne. Argument `size` bude po vykonaní funkcie obsahovať veľkosť regiónu alokovanej pamäte v príslušnom procese s hodnotou `rank`. Zároveň je možné získať aj informáciu o veľkosti položiek `disp_unit` a lokálnej adrese `baseptr`.

MPI_Win_create_dynamic

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
                           MPI_Win *win)
```

Funkcia `MPI_Win_create_dynamic` umožňuje vytvorenie okna bez toho, aby doň bola pripojená pamäť. Túto je možné neskôr dodatočne pripojiť pomocou funkcie `MPI_Win_attach` a odpojiť funkciou `MPI_Win_detach`.

MPI_Get, MPI_Put

```
int MPI_Get(const void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)

int MPI_Put(const void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)
```

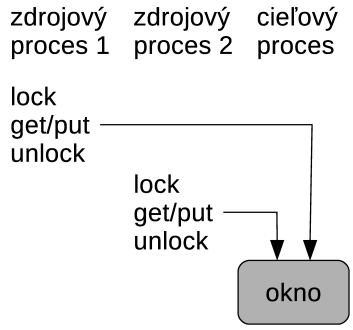
Tieto funkcie slúžia na vykonanie čítania a zápisu údajov z/do vytvoreného okna `win` medzi procesmi pomocou vzdialeného prístupu do pamäte procesov. Argument `origin_addr` predstavuje ukazovateľ na adresu bufra zdrojového procesu, `origin_count` a `origin_datatype` určuje počet a MPI údajový typ prenášaných prvkov na strane zdrojového procesu, `target_rank` určuje `rank` cieľového procesu. Ďalšie tri argumenty `target_disp`, `target_count` a `target_datatype` určuje veľkosť položky, počet a typ prvkov na strane cieľového procesu.

Je potrebné podotknúť, že súbežné volanie funkcie `MPI_Put` s rovnakým cieľovým miestom môže viesť k nepredvídateľnému správaniu.

MPI_Accumulate

```
int MPI_Accumulate(const void *origin_addr, int origin_count,
        MPI_Datatype origin_datatype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

Operácia `MPI_Accumulate` slúži podobne ako redukcie na kombinovanie výsledkov do pamäte v okne cieľového procesu. Argumenty funkcie majú obdobný význam, ako pri predchádzajúcich funkciách. Argument `op` predstavuje jednu z preddefinovaných MPI operácií. Operácia akumulácie je atomická, čo znamená, že sa vykoná v celku, a preto je ju možné volať súčasne z viacerých zdrojových procesov pre jeden cieľový proces.



Obr. 12: Pasívna synchronizácia cieľa

MPI_Win_lock, MPI_Win_unlock

```
int MPI_Win_lock(int lock_type, int rank,
                 int assert, MPI_Win win)

int MPI_Win_unlock(int rank, MPI_Win win)
```

Funkcie je možné použiť na synchronizáciu s pasívnym cieľom podľa obrázku 12. Tento model má istú podobnosť s modelom so zdieľanou pamäťou. Pred vykonaním RMA operácie prenosu údajov je potrebné začať epochu pomocou funkcie `MPI_Win_lock`. Prístup k údajom v okne je možné nastaviť pomocou argumentu `lock_type`

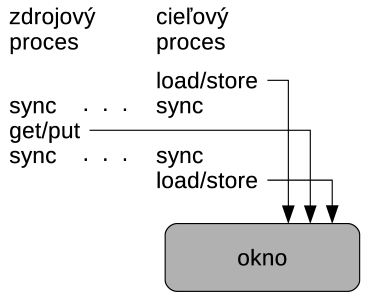
na hodnotu `MPI_LOCK_SHARED` v prípade, ak chcem, aby mohli k údajom v tom istom čase pristupovať aj iné procesy alebo na hodnotu `MPI_LOCK_EXCLUSIVE`, ak chceme zamedziť ostatným procesom prístup. Argument `rank` určuje poradové číslo procesu so zamknutým oknom `win`. Argument `assert` slúži pre optimalizáciu volania, pričom je možné použiť predvolenú hodnotu `0`.

Po vykonaní RMA operácie prenosu údajov je potrebné ukončiť epochu a odomknúť okno pomocou funkcie `MPI_Win_unlock`. Pre začatie a ukončenie epochy vo všetkých procesoch s daným oknom môžme použiť funkcie `MPI_Win_lock_all` a `MPI_Win_unlock_all`. Pre dokončenie všetkých začatých RMA operácií v danom okne môžeme použiť funkcie `MPI_Win_flush` alebo `MPI_Win_flush_all`.

MPI_Win_fence

```
int MPI_Win_fence(int assert, MPI_Win win)
```

Kolektívnu funkciu `MPI_Win_fence` je možné použiť na synchronizáciu s aktívnym cieľom podľa obrázku 13. Tento model má istú podobnosť s modelom posielania správ, kde na komunikácii aktívne participujú oba zúčastnené procesy. Funkcia vykoná synchronizáciu MPI procesov, podobne ako bariéra, na MPI RMA okne `win`. Argument funckie `assert` určuje špeciálne podmienky pre optimalizáciu synchronizácie, pričom je možné použiť predvolenú hodnotu `0`. Táto funkcia by mala byť zavolaná pred aj po volaní RMA operácie.

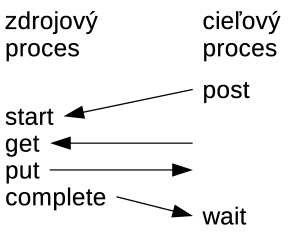


Obr. 13: Aktívna synchronizácia cieľa – fence

MPI_Win_start, MPI_Win_complete
MPI_Win_post, MPI_Win_wait

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_complete(MPI_Win win)

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_wait(MPI_Win win)
```

Funkcie je možné použiť na synchronizáciu s aktívnym cieľom podľa obrázku 14. Cieľový proces volaním funkcie `MPI_Win_post` začína a `MPI_Win_wait` ukončuje epochu, počas ktorej je umožnené pracovať s oknom `win` v skupine zdrojových procesov definovanou argumentom `group`. Ostatné argumenty majú obdobný význam ako pri funkcii `MPI_Win_fence`. Na strane zdrojových procesov je potrebné epochu začať volaním funkcie `MPI_Win_start` a ukončiť volaním funkcie `MPI_Win_complete`. Počas aktívnych epoch na oboch stranách je možné vykonávať RMA operácie na určenom okne `win`.



Obr. 14: Aktívna synchronizácia cieľa – post, wait

3 Odvodené údajové typy

Pri písaní paralelných programov môžeme dospieť k problému, keď sa musíme rozhodnúť, či je z hľadiska efektívnosti výhodnejšie niektoré údaje medzi procesmi predávať pomocou správ alebo ich v každom procese znovu vypočítať z údajov dostupných v lokálnej pamäti. Nie je ťažké si predstaviť, že v prípade jednoduchého výpočtu (napríklad sčítania) nejakého čísla by jeho poslanie v podobe správy trvalo podstatne dlhšie. Ak pôjdeme v našej úvahe ešte ďalej, môžme sa zamyslieť nad tým, či je výhodnejšie poslať určité množstvo údajov vo forme viacerých správ alebo vo forme jednej rovnako veľkej správy. Skúsme porovnať dva spôsoby uvedené v zdrojovom kóde 8.

Zdrojový kód 8: Rôzne spôsoby predania údajov

```
1  double A[1000];
2  ...
3  //1. spôsob - posielanie viacerých správ
4  if(!rank)
5    for(i=0; i<1000; i++)
6      MPI_Send(&A[i], 1, MPI_DOUBLE, 1, 0, comm);
7  else
8    for(i=0; i<1000; i++)
9      MPI_Send(&A[i], 1, MPI_DOUBLE, 0, 0, comm,
       MPI_STATUS_IGNORE);
10
11 //2. spôsob - posielanie jednej správy
12 if(!rank)
13     MPI_Send(A, 1000, MPI_DOUBLE, 1, 0, comm);
14 else
15     MPI_Send(A, 1000, MPI_DOUBLE, 0, 0, comm,
       MPI_STATUS_IGNORE);
```

Po praktickom odskúšaní oboch spôsobov by sme s istotou prišli k záveru, že druhý spôsob predávania všetkých údajov jednou správou je mnohonásobne rýchlejší. To by nás mohlo doviesť k záveru, že čím viac sa nám podarí znížiť počet potrebných predávaných správ medzi procesmi, tým vyššia môže byť efektívnosť nášho paralelného programu.

V MPI existujú tri základné spôsoby, ktoré nám umožňujú konsolidáciu posielaných údajov za účelom minimalizácie počtu predávaných správ [10]:

- argument komunikačných funkcií `count`,
- operácie `MPI_Pack` a `MPI_Unpack`,
- odvodené údajové typy.

Zbaľovanie a rozbaľovanie

Prvý spôsob sme si ozrejmili na príklade zdrojového kódu 8. Druhý spôsob spočíva v zbalení správy pred jej odoslaním a rozbalení po jej prijatí druhým procesom. Výhodou tohto prístupu je, že programátor

nemusí vytvárať nový údajový typ a vystačí si s troma funkciami `MPI_Pack_size`, `MPI_Pack` a `MPI_Unpack`.

MPI_Pack_size

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
                  MPI_Comm comm, int *size)
```

Prv než môžme vytvoriť bufer, do ktorého by sme mohli zbaliť posielané údaje, potrebujeme zistiť, aká veľkosť bufra je potrebná na zbalenie všetkých požadovaných údajov. Funkcii `MPI_Pack_size` je potrebné zadať počet položiek `incount`, údajový typ položiek `datatype` a komunikátor `comm`. Po vykonaní funkcia vráti maximálnu potrebnú veľkosť bufra `size` v bajtoch.

MPI_Pack

```
int MPI_Pack(const void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

Úlohou funkcie `MPI_Pack` je zbalenie správy pred jej odoslaním. Správa uložená na adrese ukazovateľa `inbuf` s počtom prvkov `incount` a údajového typu `datatype` je zbalená a uložená do bufra na adrese ukazovateľa `outbuf` s veľkosťou `outsize` bajtov. Argument `position` ako vstupný argument určuje prvú pozíciu výstupného bufra, ktorá sa má použiť na uloženie zbalenej správy. Hodnota tohto argumentu sa postupne zväčšuje s narastajúcou veľkosťou zbalenej správy. Tento argument zároveň funguje aj ako výstupný a po ukončení funkcie sa v ňom nachádza prvá pozícia za pozíciou obsadenou zbalenou správou (prvá voľná pozícia). Po zbalení správy ju je možné odoslať ako správu MPI údajového typu `MPI_PACKED` v rámci komunikátora `comm`.

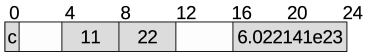
MPI_Unpack

```
int MPI_Unpack(const void *inbuf, int insize,
               int *position, void *outbuf,
               int outcount, MPI_Datatype datatype,
               MPI_Comm comm)
```

Na rozbalenie prijatej správy slúži funkcia `MPI_Unpack`. Prijatá správa je uložená na adrese ukazovateľa `inbuf` s veľkosťou `insize` bajtov. Do bufra `outbuf` je rozbalená správa o dĺžke `outcount` položiek údajového typu `datatype` nachádzajúca sa na pozícií `position` udávanej v bajtoch. Argument `comm` predstavuje komunikátor.

Nové údajové typy

Vytvorenie nového údajového typu spočíva v spojení rôzneho počtu položiek rôznych typov, o ktorých budeme mať uloženú informáciu ako o ich type, tak aj o ich relatívnom umiestnení v pamäti. Princíp spočíva v tom, že pokiaľ funkcia odosielajúca údaje pozná údajový typ a relatívne umiestnenie údajov v pamäti, môže ich pred odoslaním zozbierať a poslať ako jeden celok. Podobne aj funkcia, ktorá takéto údaje príjme, ich následne môže rozdistribuovať na správne miesta v pamäti.



Údajový typ	Posun
char	0
int	4
int	8
double	16

Obr. 15: Štrukúra odvodeného údajového typu

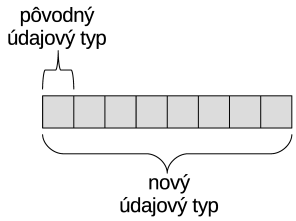
Odvodený údajový typ je v podstate ukazovateľ na zoznam položiek pozostávajúci z údajového typu relatívneho posunu v pamäti, tak ako je znázornené na obrázku 15. Môžeme si všimnúť, že niektoré bloky pamäte (znázornené bielou farbou) ostávajú nepoužité z dôvodu zarovnávania pamäte. Na obrázku vidíme údaje pozostávajúce z jednej položky typu `char` (veľkosť 1 B), dvoch položiek typu `int` (veľkosť 4 B) a jednej položky typu `double` (veľkosť 8 B). Takýto odvodený údajový typ je vhodné použiť, pokiaľ potrebujeme poslať údaje komplexnejšieho charakteru ako len jednoduchú štruktúru blokov prvkov umiestnených v pamäti za sebou. Napríklad stĺpec matice síce pozostáva z prvkov rovnakého údajového typu, ale jeho odoslanie bude vyžadovať preskakovanie niektorých prvkov matice, nakoľko matica je uložená v pamäti po riadkoch, ako jeden dlhý vektor.

Poznámka: Výskyt nepoužitých miest môže viesť k významnému zníženiu prenosovej šírky pásma. Viac informácií je možné nájsť v štandarde MPI-3.0 [11], príklade 4.1.6 v rade pre používateľov.

V MPI máme k dispozícii niekoľko spôsobov pre vytvorenie nového odvodeného údajového typu v závislosti od počtu polí, počtu relatívnych posunov a počtu rôznych typov [12]:

- **spojitý** (`MPI_Type_contiguous`) – je definový jednou hodnotou dĺžky poľa, žiadnym posunom a jedným údajovým typom,
- **vektor s obkrokom** (`MPI_Type_vector`) – je definovaný jednou hodnotou dĺžky poľa, jednou hodnotou posunu a jedným údajovým typom,
- **indexový** (`MPI_Type_indexed`) – je definovaný viacerými dĺžkami polí, viacerými posunmi a jedným údajovým typom,
- **štruktúra** (`MPI_Type_struct`) – je definovaný viacerými dĺžkami polí, viacerými posunmi a viacerými údajovými typmi.

V prvom rade je potrebné vytvoriť premennú typu `MPI_Datatype`, ktorá bude reprezentovať nový údajový typ. Následne vypočítame hodnoty argumentov pre funkcie vytvárajúce nový údajový typ a potom môžme použiť niektorú z nasledujúcich funkcií. Skôr než nový údajový typ môžme začať používať, musíme ho zaznamenať zavolaním funkcie `MPI_Type_commit`. Po skončení práce, keď už nový typ nebudeme potrebovať, ho môžme zrušiť zavolaním funkcie `MPI_Type_free`.



Obr. 16: Štrukúra odvodeného spojitého údajového typu

MPI_Type_contiguous

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

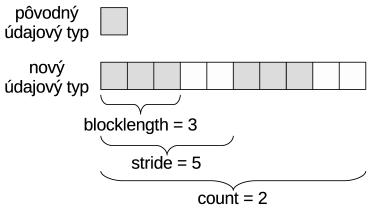
Pomocou funkcie `MPI_Type_contiguous` môžme vytvoriť najjednoduchší odvodený údajový typ, ktorý pozostáva z určitého množstva

položiek určeného hodnotou argumentu `count`, pričom všetky položky musia byť rovnakého údajového typu `oldtype` a nachádzajú sa v pamäti uložené za sebou. Schématické znázornenie položky pôvodného typu a nového typu môžme vidieť na obrázku 16. Argument `newtype` predstavuje ukazovateľ na nový údajový typ.

MPI_Type_vector

```
int MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Pomocou funkcie `MPI_Type_vector` môžme vytvoriť nový údajový typ, ktorý pozostáva z určitého počtu blokov `count` obsahujúcich položky rovnakého údajového typu `oldtype`. Počet prvkov v bloku, ktoré majú byť súčasťou nového typu určuje argument `blocklength` a celkový počet položiek v bloku udáva argument `stride`. Schématické znázornenie položky pôvodného typu a nového typu môžme vidieť na obrázku 17. Bloky znázornené sivou farbou budú súčasťou nového údajového typu a biele bloky budú vynechané pri prenose údajov. Argument `newtype` predstavuje ukazovateľ na nový údajový typ.



Obr. 17: Štrukúra odvodeného údajového typu vektor s obkrokom

MPI_Type_indexed

```
int MPI_Type_indexed(int count,
              const int *array_of_blocklengths,
              const int *array_of_displacements,
              MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Pomocou funkcie `MPI_Type_index` môžme vytvoriť nový údajový typ, ktorý pozostáva z určitého počtu blokov `count` obsahujúcich položky rovnakého údajového typu `oldtype`. Počet prvkov v blokoch, ktoré majú byť súčasťou nového typu, určujú pre jednotlivé bloky hodnoty v poli uvedenom v argumente `array_of_blocklengths` a celkový počet položiek v jednotlivých blokoch udáva pole v argumente `stride`. Argument `newtype` predstavuje ukazovateľ na nový údajový typ.

MPI_Type_struct

```
int MPI_Type_struct(int count,
                    const int *array_of_blocklengths,
                    const MPI_Aint *array_of_displacements,
                    const MPI_Datatype *array_of_types
                    MPI_Datatype *newtype)
```

Pomocou funkcie `MPI_Type_struct` môžme vytvoriť nový údajový typ, ktorý pozostáva z určitého počtu blokov `count` obsahujúcich položky rôznych údajových uvedených v argumente `array_of_types` v poli. Počet prvkov v blokoch, ktoré majú byť súčasťou nového typu určujú pre jednotlivé bloky hodnoty v poli uvedenom v argumente

`array_of_blocklengths` a relatívny posun blokov od začiatku je uvedený v pre každý blok v poli ako argument `array_of_displacements`. Argument `newtype` predstavuje ukazovateľ na nový údajový typ.

Schématické znázornenie položky pôvodného typu a nového typu môžme vidieť na obrázku **??**. V uvedenom príklade budeme pracovať s dvoma blokmi, takže hodnota `count = 2`. Počet položiek blokov zadefinujeme ako pole nasledovne:

```
int array_of_blocklengths[2] = {3, 4}
```

Zistenie relatívneho posunu v bajtoch od začiatku správy nie je úplne jednoduché, pretože musíme brať do úvahy aj nepoužité bloky pamäte (znázornené bielou farbou) vynechané kvôli zarovnávaniu pamäte. Hodnota posunutia pre blok 0 je rovná `0`. Správne hodnoty posunutia pre ďalšie bloky vieme vypočítať ako rozdiel adries $adr1 - adr0$. Hodnoty týchto adries zistíme použitím funkcie:

```
int MPI_Get_address(const void *location, MPI_Aint *address)
```

pričom do ukazovateľa `location` musíme zadať adresu umiestnenia bloku v pamäti a po vykonaní funkcie získame výsledok uložený na adrese ukazovateľa `address`, ktorý predstavuje celé číslo zodpovedajúce adrese umiestnenia. Za určitých okolností môže byť toto číslo rovnaké ako adresa získaná operátorom `&`, čo však záleží od použitého systému. Hodnoty v poli `array_of_displacements` potom budú nasledovné:

```
MPI_Aint array_of_displacements[2] = {0, adr1-adr0}
```

Ďalší argument `array_of_types` určuje údajové typy blokov nasledovne:

```
MPI_Datatype array_of_types[2] = {MPI_INT, MPI_DOUBLE}
```

MPI_Type_commit

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Po zadefinovaní nového údajového typu je potrebné, aby sme ho pred samotným použitím zaregistrovali. Následne môžme správu tohto nového odvodeného údajového typu predať medzi procesmi pomocou štandardných nástrojov MPI na posielanie správ. Ako argument funkcie `datatype` zadávame nový údajový typ, ktorý chceme zaregistrovať a následne používať.

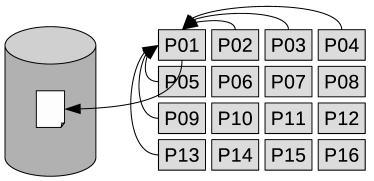
MPI_Type_free

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Po ukončení práce s novým odvodeným údajovým typom ho môžme odstrániť zavolaním funkcie `MPI_Type_free`. Ako argument funkcie `datatype` zadávame nový údajový typ, ktorý chceme odstrániť.

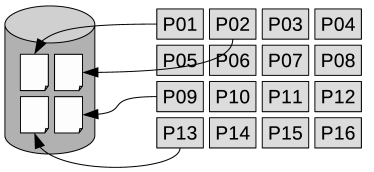
4 Paralelné vstupno-výstupné operácie

Pri písaní paralelných programov je často potrebné zaoberať sa otázkou čítania alebo zápisu údajov do súboru uloženého na disku. Pokiaľ vychádzame zo sériového programu je prirodzené použiť taký model práce so súborom, kde jeden vybraný proces bude vykonávať vstupno-výstupné (I/O) operácie so súborom a ostatné procesy budú posielať svoje požiadavky tomuto procesu podľa obrázky 19. Tento model je pomerne jednoduchý na implementáciu. Avšak v prípade väčšieho počtu procesov, a teda aj požiadaviek na I/O operácie, narazíme na limity prenosovej šírky pre hlavný proces. Ďalšou komplikáciou sú zvýšené náklady na komunikáciu medzi procesmi. V dôsledku uvedených skutočností sa dostávame do stavu, že mnohé procesy musia čakať, kým budú obslúžené ich požiadavky na I/O a nemôžu využívať výpočtové prostriedky.



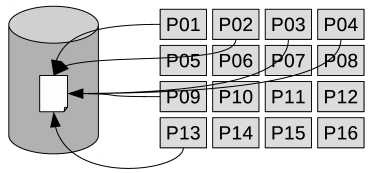
Obr. 19: Prístup jedného procesu k súboru

Ako určité riešenie tohto problému sa ponúka použitie viacerých súborov súčasne, kde každý proces bude pracovať so svojim vlastným súborom podľa obrázku 20. Tento model je tiež pomerne ľahko implementovateľný. Okrem toho nie je potrebná koordinácia medzi procesmi, pretože každý proces pracuje so svojim súborom nezávisle od ostatných procesov. Taktiež nehrozí problém konfliktného zdieľania blokov súborového systému. Nevýhodou tohto modeluje je, že pri väčšom počte procesov môže nastať situácia, že systém nebude schopný manažovať tak veľký počet súborov. Ďalšia komplikácia nastane vtedy, keď budeme potrebovať zjednotiť výstupy do jedného súboru, čo si vyžiada dodatočný výpočtový čas. Tretím problémom tohto prístupu je zápis metadát zapisovaných súborovým systémom, ktorý môže byť vykonávaný sériovo.



Obr. 20: Prístup každého procesu k svojmu súboru

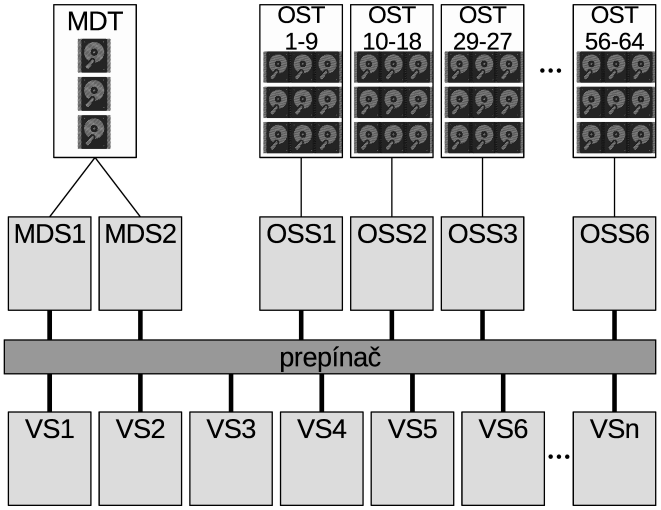
Ako tretie riešenie možno použiť prístup všetkých súborov k jednému zdieľanému súboru podľa obrázku 21. Výhoda tohto modelu spočíva v tom, že počet súborov nie je závislý od počtu procesov a pre vytvorenie výstupu nie je potrebné ďalšie dodatočné spracovanie. Problém však môže nastať pri nekoordinovanom zadávaní požiadaviek na I/O operácie so zdieľaným súborom. Okrem toho hrozí nebezpečenstvo konfliktného zdieľania blokov súborového systému. Veľkosť bloku údajov uložených v súborovom systéme nie vždy korešponduje s veľkosťou bloku používaného súborovým systémom v čoho dôsledku môže byť blok dát uložený čiastočne na niekoľkých blokoch súborového systému. Takto môže nastať situácia, že jeden blok súborového systému obsahuje údaje z dvoch blokov údajov, ku ktorým môžu chcieť pristupovať rôzne procesy. V takomto prípade bude prístup k bloku vybavený sériovo.



Obr. 21: Prístup procesov k zdieľanému súboru

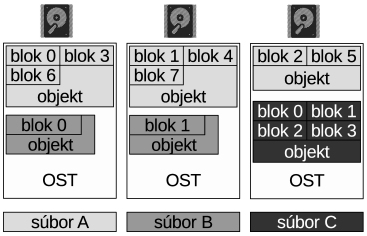
Väčšina v súčasnosti používaných vysoko-výkonných výpočtových systémov disponuje úložiskom údajov s paralelným súborovým systémom, ako je napríklad paralelný súborový systém Lustre [13], GPFS

[14] alebo Globálny RAM disk BeeGFS [15]. Pre lepšie pochopenie je na obrázku 22 znázornená štruktúra systému s paralelným súborovým systémom Lustre.



Obr. 22: Paralelný súborový systém Lustre

Uvedený systém pozostáva z niekoľkých serverov manažujúcich metadáta (MDS – Meta Data Server) a fyzických cieľov – diskov pre ukladanie metadát (MDT – Meta Data Target). Okrem toho systém disponuje viacerými servermi pre ukladanie objektov (OSS – Object Storage Server) a cieľovými zariadeniami (OST – Object Storage Target). Paralelný súborový systém je prepojený prepínačom s výpočtovými servermi (VS).



Obr. 23: Rozdelenie súboru na bloky v paralelnom súborovom systéme Lustre

Takýto systém umožňuje rozdelenie jedného súboru do viacerých blokov, pričom tieto sú ukladané fyzicky na rôzne cieľové disky, podľa obrázku 23. Takáto architektúra paralelného súborového systému umožňuje paralelný prístup k obsahu súboru. Požiadavky na I/O operácie jednotlivých procesov k tomu istému zdieľanému súboru môžu byť takto obslúžené viacerými nezávislými servermi. Úskalím tohto systému je obslúženie veľkého počtu požiadaviek na metadáta.

Špecifikácia MPI umožňuje využívať procesom paralelného programu kolektívne operácie na prácu so súbormi. Vďaka tomu je možné efektívne rozdelenie súboru medzi procesy, prenos globálnych údajov medzi pamäťou a súborom alebo asynchrónny prenos údajov. Pomocou paralelných I/O operácií môžme v jednotlivých procesoch vytvoriť logické pohľady na zdieľaný súbor na základe určitej schémy a určiť, s ktorou časťou súboru bude pracovať ktorý proces.

MPI_File_open

```
int MPI_File_open(MPI_Comm comm, const char *filename,
                  int amode, MPI_Info info, MPI_File *fh)
```

Kolektívna operácia `MPI_File_open` umožňuje otvorenie súboru vo všetkých procesoch v rámci komunikátora `comm`. V prípade potreby otvorenia lokálneho súboru je možné použiť ako hodnotu komunikátora

MPI_COMM_SELF. Keďže sa jedná o kolektívnu operáciu, je nevyhnutné, aby ju zavolali všetky procesy tak, aby argument filename odpovedal tomu istému súboru. Tretí argument amode určuje mód prístupu k súboru, ktorý musí byť zhodný vo všetkých procesoch a môže nadobúdať niektorú z hodnôt uvedených v tabuľke 3. Pomocou argumentu info je možné zadať doplňujúce informácie alebo použiť hodnotu MPI_INFO_NULL. Po úspešnom vykonaní funkcia vráti ukazovateľ na otvorený súbor v argumente fh.

Tabuľka 3: Prehľad MPI módov otvorenia súboru

Mód MPI	Význam
MPI_MODE_RDONLY	prístup iba na čítanie
MPI_MODE_RDWR	prístup na čítanie a zapisovanie
MPI_MODE_WRONLY	prístup iba na zapisovanie
nasledujúce módy je možné kombinovať pomocou operácie bitového súčtu	
MPI_MODE_CREATE	vytvoriť súbor, ak neexistuje
MPI_MODE_EXCL	chyba v prípade vytvárania už existujúceho súboru
MPI_MODE_DELETE_ON_CLOSE	zmazanie súboru pri zatvorení
MPI_MODE_UNIQUE_OPEN	súbor nie je inde otvorený
MPI_MODE_SEQUENTIAL	sekvenčný prístup k súboru
MPI_MODE_APPEND	ukazovateľ je nastavený na koniec súboru

MPI_File_close

```
int MPI_File_close(MPI_File *fh)
```

Kolektívna operácia MPI_File_close slúži na zatvorenie súboru určeného ukazovateľom fh. Používateľ sa musí uistiť, že všetky neblokujúce a údaje rozdeľujúce kolektívne operácie so súborom boli dokončené.

MPI_File_delete

```
int MPI_File_delete(const char *filename, MPI_Info info)
```

Operácia MPI_File_delete slúži na zmazanie súboru určeného argumentom filename. Funkciu je potrebné zavolať len jedným procesom. V prípade, že sa pokúsime zmazať súbor, ktorý je otvorený niektorým iným procesom, záleží od implementácie MPI, či budú operácie so súborom dokončené alebo bude súbor zmazaný. Samotná funkcia však túto skutočnosť neoveruje.

MPI_File_set_view

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                MPI_Datatype etype, MPI_Datatype filetype,
                const char *datarep, MPI_Info info)
```

Definovaním pohľadu kolektívnou operáciou MPI_File_set_view je možné určiť, ktorá časť obsahu súboru fh je viditeľná pre ktorý proces. Definíciu pohľadu určujeme na základe posunu disp v bajtoch počítaných od začiatku súboru, na základe základného údajového typu

a typu súboru. Definovaný pohľad je možné zmeniť aj počas behu programu bez nutnosti súbor zatvoriť a opätovne otvoriť.

Po otvorení súboru je naň nastavený predvolený pohľad pre všetky procesy paralelného MPI programu s nulovým odsadením, pričom hociktorý z procesov môže so súborom pracovať ako so sekvenciou bajtov. Pomocou argumentu `etype` je možné určiť MPI údajový typ obsahu súboru, ktorý bude použitý pri definovaní posunu v násobkoch údajového typu. Tento údajový typ môže byť základný alebo odvodený. Pomocou argumentu `filetype` je možné určiť vzor prístupu k súboru. Vzor môže pozostávať z inštancií určeného MPI údajového typu alebo dier.

Argument `datarep` určuje konverziu údajov uložených v pamäti na údaje uložené na disku. Táto operácia ma vplyv na interoperabilitu I/O operácií medzi rôznorodými časťami výpočtového systému alebo rôznymi systémami. Môže nadobúdať hodnoty:

- `native` – údaje sú uložené rovnakým spôsobom ako v pamäti (výhody: bez straty presnosti, bez potreby konverzie údajov, nevýhody: problematická prenositeľnosť),
- `internal` – údaje sú uložené vo formáte špecifikovanom implementáciou (výhody: vhodné pre homogénne aj heterogénne výpočtové prostredie, implemetácia zabezpečí prípadnú potrebnú konverziu údajov, nevýhody: náklady na konverziu a možná nekompatibilita medzi implementáciami),
- `external32` – údaje sú uložené v štandardnom formáte údajov (IEEE - big endian) (výhody: použiteľná aj s programami mimo MPI, nevýhody: zníženie výkonu kvôli konverzii a nedostupnosť na všetkých implementáciách).

Pre čítanie a zápis údajov do súboru sa štandardne používajú POSIX funkcie `read` a `write`. MPI poskytuje rôzne verzie týchto funkcií. Z pohľadu synchronizácie sú to I/O funkcie:

- blokujúce – vrátia riadenie procesu až po dokončení operácie,
- neblokujúce – nečakajú na dokončenie operácie, pričom dokončenie je potrebné explicitne overiť, bufer nie je možné použiť počas vykonávania operácie,
- kolektívne rozdeľujúce – obmedzená verzia neblokujúcej kolektívnej verzie, bufer nie je možné použiť počas vykonávania operácie, nedovoľuje iný kolektívny prístup k súboru počas vykonávania operácie, začatie a ukončenie musí byť volané z toho istého vlákna.

Z pohľadu vzájomnej koordinácie procesov ide o I/O funkcie:

- nekolektívne – dokončenie je závislé len na aktivitách volajúceho procesu,
- kolektívne – dokončenie je závislé na aktivite všetkých procesov, poskytuje veľký priestor pre optimalizáciu.

Z pohľadu určovania pozície môže ísť o I/O funkcie s:

- explicitným offsetom – bez potreby ukazovateľov, pozícia sa udáva priamo ako argument funkcie,

- individuálnym ukazovateľom – každý proces má vlastný ukazovateľ na súbor, po vykonaní prístupu sa posunie na nasledujúcu položku údajového typu,
- zdieľaným ukazovateľom – všetky procesy zdieľajú spoločný ukazovateľ a rovnaký pohľad, individuálny prístup je riadený sériovo v náhodnom poradí, kolektívny prístup je riadený v poradí `rank` procesov.

Nasledujúca tabuľka 4 sumarizuje prehľad použiteľných funkcií pre paralelné čítanie a zápis na základe predchádzajúcich kritérií.

Tabuľka 4: Prehľad MPI funkcií pre čítanie a zápis do súboru

určovanie pozície	synchronizácia	nekolektívna koordinácia	kolektívna koordinácia
explicitný offset	blokujúca	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	neblokujúca	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`

	kolektívna rozdeľujúca	–	`MPI_File_read_at_all_begin`
			`MPI_File_read_at_all_end`
			`MPI_File_write_at_all_begin`
			`MPI_File_write_at_all_end`

individuálny ukazovateľ	blokujúca	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	neblokujúca	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`

	kolektívna rozdeľujúca	–	`MPI_File_read_all_begin`
			`MPI_File_read_all_end`
			`MPI_File_write_all_begin`
			`MPI_File_write_all_end`

zdieľaný ukazovateľ	blokujúca	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	neblokujúca	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	–

	kolektívna rozdeľujúca	–	`MPI_File_read_ordered_begin`
			`MPI_File_read_ordered_end`
			`MPI_File_write_ordered_begin`
			`MPI_File_write_ordered_end`

Úloha: Napíšte paralelný MPI program, v ktorom každý proces zapíše svoju hodnotu rank do spoločného súboru rank.dat tak, aby boli v súbore zoradené v podarí $0...n-1$. Následne proces 0 tento súbor prečíta a vypíše na štandardný výstup.

Úloha: Napíšte paralelný MPI program, v ktorom procesy prečítajú súbor rank.dat (z predchádzajúcej úlohy) v opačnom poradí, to znamená, že proces 0 prečíta poslednú položku atď. Každý proces vypíše svoju hodnotu rank a načítané číslo. (Pozor: Počet procesov pri spustení nemusí byť rovnaký ako v predchádzajúcej úlohe.)

Ďalšie užitočné funkcie pre paralelné I/O operácie so súbormi sú `MPI_File_get_position` na zistenie súčasnej pozície ukazovateľa v súbore, `MPI_File_seek` na nastavenie pozície ukazovateľa v súbore, `MPI_File_get_byte_offset` na prevod relatívneho offsetu v pohľade na posun v bajtoch, `MPI_File_set_atomicity` na nastavenie módu atomicity a `MPI_File_sync` na zapísanie všetkých operácií zápisu a zmien na disk.

Pre efektívnu prácu s I/O operáciami je dôležité vhodne zvoliť súborový systém tak, aby mal najlepší výkon. Túto skutočnosť je zväčša vhodné konzultovať so správcom systému alebo skúsenejším používateľom systému. Používanie a frekvenciu používania I/O operácií je potrebné čo najviac minimalizovať, niekedy aj za cenu ďalších výpočtov. Ak je to možné, tak treba znížiť presnosť ukladaných údajov. Čítanie a zápis údajov je lepšie vykonávať vo veľkých postupných blokoch. Explicitnú synchronizáciu zápisu pomocou `MPI_File_sync` používať v čo najmenšej možnej miere. Pre otváranie súborov používať správny mód prístupu a doplnkové informácie, čo umožní systému efektívnejšie vykonávanie I/O operácií.

5 Meranie času v MPI programe

Po napísaní funkčného paralelného programu nás zrejme bude zaujímať, do akej miery bola naša snaha o zrýchlenie výpočtov úspešná. Na túto otázku budeme schopný odpovedať, až po vykonaní meraní času vykonávania celého programu alebo niektorých jeho častí.

Najjednoduchším spôsobom merania času vykonávania celého programu je s použitím štandardného príkazu `time` v operačnom systéme UNIX alebo Linux. Ten môžme použiť napríklad nasledovne:

```
time mpiexec -np 8 ./nas_program
```

V rámci MPI je na meranie času určená funkcia

```
double MPI_Wtime()
```

ktorá vracia ako návratovú hodnotu reálne číslo predstavujúce čas v sekundách, ktorý uplynul od určitého časového bodu v minulosti. Počas vykonávania procesov je garantované, že sa tento časový bod nezmení.

Čas vykonávania programu alebo jeho časti potom môžme vypočítať ako rozdiel dvoch hodnôt získaných volaním funkcie hneď tesne pred a po skončení sledovanej časti programu. Uvedený čas v sekundách predstavuje tzv. *wall clock* čas, čo je skutočný čas vykonávania programu. Alternatívou pre zistenie času vykonávania programu je použitie POSIX funkcie `gettimeofday`, ktorá vracia počet mikrosekúnd uplynutých od určitého časového bodu.

Pripomeňme, že okrem *wall clock* času je možné merať aj skutočný CPU čas pomocou funkcie `clock`. Táto udáva čas strávený pri vykonávaní kódu používateľa, volaní funkcií z knižníc a vykonávaní kódu operačného systému. Tento v sebe nezahŕňa čas nečinnosti, napríklad počas čakania procesu na prijatie správy.

Pre presné meranie času je vhodné použiť explicitnú synchronizáciu procesov pomocou kolektívnej operácie `MPI_Barrier`. Tým zabezpečíme, že žiaden proces sa nebude nachádzať v sledovanej časti programu skôr, ako by sme očakávali.

Môžeme si všimnúť, že po niekoľkých opakovaniach merania času toho istého programu na tom istom výpočtovom systéme dostaneme namerané časy s určitou nepresnosťou. Pre opakovanie merania sa treba uistiť, že bude vykonané za rovnakých podmienok, to znamená, že budeme spúšťať ten istý program s tým istým vstupom na tom istom výpočtovom systéme. Napriek tomu sa namerané hodnoty časov od seba budú mierne líšiť z dôvodu interakcie nášho programu so samotným operačným systémom a jeho službami. Okrem toho je dobré uistiť sa, že v systéme nie sú spustené žiadne ďalšie programy, ktoré by mohli skresliť naše merania.

6 Efektívnosť paralelného MPI programu

V mnohých paralelných programoch sa skrýva ešte ďalší potenciál pre lepšiu optimalizáciu programu. V prípade, že vychádzame zo sériového programu, ktorý môžme považovať za takmer optimálne riešenie pre jednojadrový procesor, je nevyhnutné podrobiť vytvorený paralelný program ďalším testom výkonnosti a škálovateľnosti, aby bolo možné zhodnotiť správne fungovanie programu a predísť rôznym problémom spojeným s paralelizáciou. Mnohé z nich nie sú priamo spojené s MPI, ale pramenia napríklad zo sériového vykonávania určitých častí programu, neoptimálneho vyvažovania záťaže, zbytočnej synchronizácie a mnohých ďalších. Naopak mnohé sú zas špecifické pre MPI. Treba si uvedomiť, že hoci MPI je navrhnuté tak, aby umožňovalo vytvoriť efektívne a ľahko prenositeľné paralelné programy, neznamená to automaticky, že aj výkonnosť paralelného programu bude na iných architektúrach rovnaká [16].

V porovnaní so sériovým programom v paralelnom programe nie je možné hneď priamo určiť príčinu problémov týkajúcich sa efektívnosti paralelného MPI programu. Napomôcť nám môže použitie rôznych komerčných alebo voľne dostupných nástrojov na profilovanie MPI paralelného programu, ako sú napríklad Vampir [17–19] alebo Intel Trace Analyzer and Collector [20, 21]. Tieto nám môžu poskytnúť informácie o tom, koľko času program strávi pri vykonávaní MPI funkcií a koľko pri samotnom výpočte úlohy, alebo o množstve vykonanej komunikácie, prípadne či niektoré procesy musia čakať pri komunikácii. Súčasťou štandardu MPI je rozhranie pre profilovanie. Každá MPI funkcia má wrapper, ktorý začína názvom `PMPI_`.

Komunikácia

V programe 5 sme vykonali jednoduchý test latencie a prenosovej šírky siete pri použití MPI komunikácie. Tieto údaje však neodpovedajú skutočným úplne presne, pretože predávanie správ v MPI je komplexnejší proces, než bola naša zjednodušená predstava. Väčšina implementácií MPI používa rôzne možnosti prenosu správ podľa veľkosti samotnej správy, ako aj iných faktorov.

Pri posielaní krátkych správ môžu byť tieto spoločne doplnkovými informáciami (obálkou) odoslané a uložené na strane prijímajúceho procesu v prealokovanom bufri. Následne neskôr musí byť správa prekopírovaná s dočasného bufra do bufra pre prijatú správu. V takomto prípade hovoríme o tzv. **eager protokole**. Jeho výhodou je zníženie času čakania pri synchronizácii. Na druhej strane vyžaduje veľké množstvo prealokovanej pamäte pre bufer. V prípade veľkého množstva prichádzajúcej komunikácie do jedného procesu môže nastať pretečenie bufra, čo môže viesť k chybe programu.

Pri posielaní veľkých správ nemá použitie bufra veľké opodstatnenie. V takomto prípade je správa aj s obálkou okamžite uložená do bufra pre prijaté správy, pričom ďalšie vykonávanie procesov je blokované, kým nie je správa prijatá. Tým sa eliminuje potreba dodatočného

kopírovania správy medzi buframi, čo vedie k vyššej prenosovej rýchlosti, avšak je nevyhnutná synchronizácia komunikujúcich procesov. V takomto prípade hovoríme o tzv. **rendezvous protokole.**

Na základe týchto skutočností je možné prispôsobiť veľkosť správy alebo dočasného bufra tak, aby bola komunikácia medzi procesmi čo najefektívnejšia.

Synchronizácia a serializácia

V paralelných programoch sa bežne vyskytuje neúmyselná synchronizácia procesov alebo dokonca serializácia programu. Názornou ukážkou takéhoto prípadu je predávanie správy v kruhu. V takomto prípade dokonca hrozí riziko uviaznutia pri použití blokujúcej komunikácie.

Je žiadúce vyhýbať sa tomuto javu v maximálnej možnej miere nie len preto, že vedie k vyššiemu času réžie komunikácie, ale hlavne preto, že vedie k zlé vyváženému rozdeľovaniu úloh. Nasledujúce kroky môžu viesť k čiastočnému odstráneniu tohto problému:

- Zmena poradia odosielania a prijímania správ, napríklad iba procesy s nepárnym číslom `rank` budú správu odosielať, pričom párne procesy budú s nimi tvoriť komunikujúce dvojice.
- Použitie neblokujúcej komunikácie. Jej výhodou je, že MPI dokáže manažovať predávanie viacerých takýchto správ v optimálnom poradí.
- Použitie blokujúcej komunikácie, pri ktorej je garantované, že neuviazne bez ohľadu na veľkosť správy. Takúto komunikáciu je možné vykonať použitím funkcií `MPI_Sendrecv` alebo `MPI_Sendrecv_replace`.

Ďalší problém môže nastať v prípade, ak sa viacero procesov bude snažiť o prístup k rovnakému zdroju. Uveďme si niekoľko prípadov:

- Viacero procesov alebo vlákien v rámci jedného výpočtového uzla sa snaží komunikovať s inými uzlami. V prípade, že prenosová šírka neškáluje s počtom spojení, z pohľadu procesu sa bude prenosová šírka znižovať s rastúcim počtom požiadaviek na komunikáciu. S týmto javom sa môžme stretnúť obzvlášť často najmä pri použití komoditného hardvéru, ktorý disponuje jedným sieťovým rozhraním umožňujúcim MPI komunikáciu, pričom toto rozhranie môže zároveň slúžiť aj ako komunikačné rozhranie pre I/O k sieťovému súborovému systému. Riešením je použitie špeciálneho hardvéru pre paralelné počítače, ktoré disponujú viacerými prepojeniami.
- Topológia siete nemusí byť neblokujúca, napríklad v prípade, ak je šírka bisekcie menšia ako súčin počtu uzlov a prenosovej šírky jedného prepojenia. Toto je typické pre siete typu kubickej mriežky alebo tučných stromov, ktoré nie sú plne neblokujúce.
- Hoci by aj šírka bisekcie bola optimálna, v dôsledku statického smerovania môže nastať problém pri použití určitých vzorov komunikácie. Riešením môže byť zmena záznamov v smerovacej tabuľke smerovača.

Vo všeobecnosti platí, že akékoľvek zníženie času réžie komunikácie, zníženie množstva prenášaných údajov zníži riziko týchto problémov. Taktiež môže pomôcť aj preusporiadanie poradia predávania správ tak, aby nastávalo čo najmenej konfliktov.

Optimálna doménová dekompozícia

Za účelom zníženia času réžie komunikácie je potrebné zvoliť optimálnu doménovú dekompozíciu tak, aby bola styčná plocha dvoch vytvorených domén čo najmenšia. Ďalej je potrebné brať na zreteľ, že komunikácia medzi dvoma procesmi v rámci toho istého uzla je väčšinou rýchlejšia ako komunikácia medzi uzlami. Z tohto predpokladu je potrebné vychádzať pri mapovaní jednotlivých podúloh na jednotlivé procesory.

Agregácia správ

Pri potrebe predávania veľkého množstva malých správ medzi procesmi je táto komunikácia v značnej miere závislá na latencii siete, pretože predanie každej správy vyžaduje určitý čas. V takomto prípade je výhodnejšie agregovať väčšie množstvo malých správ do jednej väčšej správy, pretože zdržanie v podobe latencie siete nastane len raz a vzhľadom na veľkosť správy bude možné efektívnejšie využiť prenosovú šírku. K tomuto účelu je vhodné použiť odvodené údajové typy pre vytvorenie správy.

Neblokujúca vs. asynchrónna komunikácia

Spoločne so znížením času réžie komunikácie je možné ďalej zvýšiť efektívnosť paralelného programu použitím súbežného komunikovania a vykonávania výpočtov. Pre tento účel je vhodná práve neblokujúca komunikácia. Je však potrebné rozlišovať medzi neblokujúcou a asynchrónnou komunikáciou. Podľa štandardu, pri neblokujúcej komunikácii nie je možné používať bufer pre odosielanú správu po vykonaní operácie odoslania správy, avšak toto sa môže líšiť v závislosti od implementácie MPI.

Kolektívna komunikácia

Používanie kolektívnych operácií pre komunikáciu je zväčša efektívnejšie ako použitie adresnej komunikácie dvoch procesov, pokiaľ je potrebné, aby spolu komunikovali všetky procesy. Vzhľadom k tomu, že pri komunikácii dvoch procesov nie je možné optimalizovať predávanie správ z globálneho pohľadu, je jeho optimalizácia skôr v rukách programátora. Naopak pri použití kolektívnej komunikácie má MPI lepšiu možnosť optimalizácie prenosu správ obzvlášť v prípade, keď MPI má informácie o použitej topológii siete.

Literatúra

[1] David May and Richard Taylor. "Occam-an overview". In: *Microprocessors and Microsystems* 8.2 (1984), pp. 73–79 (cited on page 1).

[2] Robert W Numrich and John Reid. "Co-Array Fortran for parallel programming". In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM. 1998, pp. 1–31 (cited on page 1).

[3] William W Carlson et al. *Introduction to UPC and language specification*. Tech. rep. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999 (cited on page 1).

[4] MPICH. *High-Performance Portable MPI*. https://www.mpich.org. 2019 (cited on page 2).

[5] Open MPI. *Open Source High Performance Computing*. https://www.open-mpi.org/. 2019 (cited on page 2).

[6] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006 (cited on page 2).

[7] Open MPI Project. *Open MPI v4.0.2 documentation*. https://www.open-mpi.org/doc/current/. 2019 (cited on page 3).

[8] Pawel Czarnul. *Parallel Programming for Modern High Performance Computing Systems*. Chapman and Hall/CRC, 2018 (cited on pages 8, 26).

[9] James Dinan et al. "An implementation and evaluation of the MPI 3.0 one-sided communication interface". In: *Concurrency and Computation: Practice and Experience* 28.17 (2016), pp. 4385–4404 (cited on page 26).

[10] Peter S. Pacheco. *An Introduction to Parallel Programming*. MA USA: Morgan Kaufmann, 2011 (cited on page 30).

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. 2012 (cited on page 32).

[12] Ján Kollár. *Metódy a prostriedky pre výkonné paralelné výpočty*. Košice: elfa, 2003 (cited on page 32).

[13] *About the Lustre File System*. http://lustre.org/about/. 2019 (cited on page 35).

[14] Frank B Schmuck and Roger L Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *FAST*. Vol. 2. 19. 2002 (cited on page 36).

[15] Jan Heichler. *An introduction to BeeGFS*. http://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf. 2014 (cited on page 36).

[16] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cited on page 41).

[17] *Vampir*. https://vampir.eu/. 2019 (cited on page 41).

[18] Andreas Knüpfer et al. "The vampir performance analysis tool-set". In: *Tools for High Performance Computing*. Springer, 2008, pp. 139–155 (cited on page 41).

[19] Holger Brunst and Bernd Mohr. "Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG". In: *International Workshop on OpenMP*. Springer. 2005, pp. 5–14 (cited on page 41).

[20] *Intel Trace Analyzer and Collector*. https://software.intel.com/en-us/trace-analyzer. 2019 (cited on page 41).

[21] Moshe Bach et al. "Analyzing parallel programs with pin". In: *Computer* 43.3 (2010), pp. 34–41 (cited on page 41).