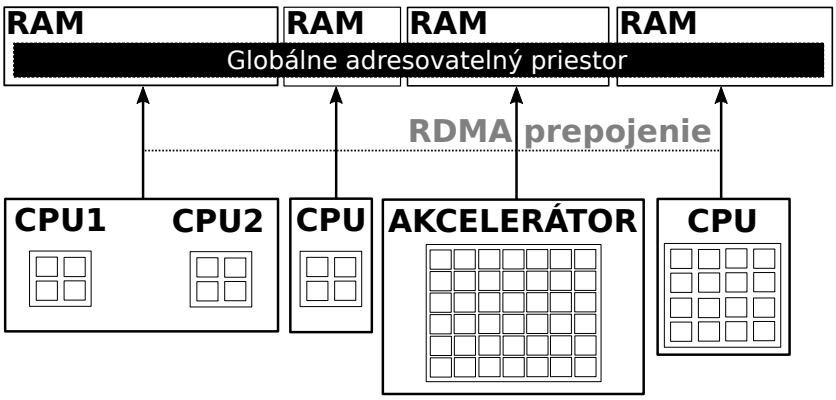
Rozdelený globálny adresný priestor

Alternatínym prístupom k vývoju programov určených pre počítačové systémy s distribuovanou pamäťou je model rozdeleného globálneho adresovateľného priestoru (z anglického Partitioned Global Address Space – PGAS). Táto abstrakcia bola navrhnutá s cieľom uľahčiť implementáciu paralelných algoritmov prostredníctvom poskytnutia prístupu k (emulovanému, virtuálnemu) zdieľanému pamäťovému priestoru naprieč výpočtovými uzlami. Využitím PGAS funkcií s analogickou funkcionalitou, akú poskytuje na jednom výpočtovom uzle funkcia `malloc`, je možné virtuálne alokovať segment pamäte naprieč výpočtovými uzlami, pričom programátor nemusí dbať o to, kde sa fyzicky príslušné údaje nachádzajú, má k nim prístup z ľubovoľného výpočtového uzla. Transfer údajov, nevyhnutný na koordináciu vykonávania paralelného programu, prostredníctvom správ ako v MPI je teda nahradený zdieľaním spoločného pamäťového priestoru. Schématicky je vytvorenie PGAS znázornené na obrázku 1, kde je situácia zovšeobecnená na heterogénne výpočtové uzly, poprípade akcelerátory (napr. GPGPU).



Obr. 1: Vytvorenie rozdeleného globálneho adresovateľného priestoru

Informácia o lokalite údajov (t.j. fyzického umiestnenia na konkrétnom výpočtovom uzle) sa v PGAS prístupe nestráca, je možné (nie však nutné) ju využiť, ak je to prínosom pre zvýšenie efektivity vykonávania príslušného programu. Fyzický presun údajov pri čítaní alebo zápise do zdieľanej pamäte je (podľa okolností) vykonaný nízkoúrovňovými funkciami implementovanými v príslušnej PGAS knižnici, častokrát asynchrónne, pomocou jednostranných (z anglického one-sided alebo single-sided) operácií. PGAS knižnice využívajú na komunikáciu buď (proprietárne) komunikačné primitívy príslušných sietí alebo univerzálne MPI.

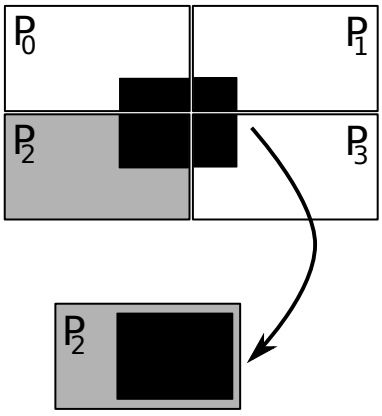
Konkrétnych možností realizácie PGAS, či už ako dedikovaných programovacích jazykov alebo knižníc, je viacero, napríklad Chapel [1], Unified Parallel C [2] (UPC), X10 [3], Charm++ [4], Global Arrays [5], GASPI [6], atď. V tejto kapitole detailnejšie predstavíme posledné dva

zo spomenutých prístupov. Výhodou Global Array a GASPI (nie však exkluzívnou) je mimo iného aj ich plná interoperabilita s MPI, t.j. oba prístupy je možné kombinovať s existujúcimi MPI programami, či už formou hybridného zdrojového kódu alebo externých modulov vyvinutých pomocou príslušného PGAS prístupu.

1 Global Arrays

Prehľad a história

Knižnica Global Arrays bola a stále je vyvíjaná na pôde Pacific Northwest National Laboratory (Washington, USA) a od roku 1994 je verejne dostupná. Cieľom vývoja bola implementácia (čo najjednoduchšieho) API určeného pre distribuované paralelné systémy, ktoré umožní vývoj softvéru analogicky k systémom so zdieľanou pamäťou. Jadrom Global Arrays je efektívna práca s viacdimenzionálnymi poliami (ako už čiastočne vyplýva aj z názvu), nie je preto prekvapením, že pôvodne bola knižnica určená na použitie v kombinácii s jazykom Fortran, neskôr však pribudli rozhrania aj pre C, Python, C++ a Babel (a teda aj Fortran 90 a Java).



Obr. 2: Kopírovanie údajov pomocou funkcie GA_Get do lokálneho poľa

Zjednodušenie paralelného programovania pomocou Global Arrays je možné demonštrovať na použití funkcie GA_Get, viď. obrázok 2, ktorej úlohou je skopírovať údaje zo zdieľaného pamäťového poľa (údaje sa fyzicky nachádzajú v pamäti prislúchajúcej každému paralelnému procesu) do lokálneho poľa na jednom z výpočtových uzlov, označenom ako P_2. Nasledujúce symbolické programy realizujúce túto operáciu pomocou MPI a Global Array ukazujú rozdiel vo vynaloženom programátorskom úsilí (syntax funkcie NGA_Get je vysvetlená v ďalšom texte).

Zdrojový kód 1: MPI

```
if ( rankID != 2 ) then
  - priprav údaje do formátu správy
  - pošli (send) správu procesu s rankID = 2
else
  skopíruj údaje do lokálneho poľa
  do rankID = 0,1,3
    - prijmi (receive) správu od procesu s rankID
    - extrahuj údaje zo správy
  end do
end if
```

Zdrojový kód 2: Global Arrays

```
if ( rankID == 2 ) then
  call GA_GET (ga, lo, hi, array, slice)
end if
```

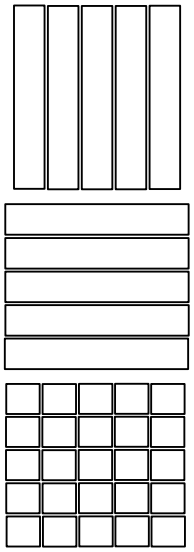
Kým v MPI je nutné časť kódu adresovať pre odosielateľa správy a časť kódu pre prijímateľa, Global Arrays túto prácu spraví za programátora, ktorý sa nemusí starať, kde sa dané údaje fyzicky nachádzajú a ako sa k prijímateľovi dostanú. Ako už bolo spomenuté, informácia o lokalite údajov sa v Global Arrays nestráca a je možné ju využiť, ak si to efektivita programu vyžaduje.

Základné funkcie

Jadrom funkcionality Global Arrays je vytvorenie zdieľaného pamäťového poľa. Toto pole môže byť vytvorené rovnomerne (vzhľadom k fyzickej distribúcii blokov poľa naprieč paralelnými procesmi), alebo nerovnomerne. Syntax najjednoduchšej (kolektívnej) funkcie vytvárajúcej rovnomerne distribuované globálne pole je nasledovná:

```
int NGA_Create(int type, int ndim, int dims[],
          char *array_name, int chunk[])
```

kde `array_name` je jedinečný reťazec identifikujúci pole; `type` je jeden z údajových typov podporovaných knižnicou Global Arrays (napr. `MT_F_DBL` - tzv. double-precision desatinné číslo, `MT_F_INT` - celé číslo, atď.); `ndim` je počet dimenzií vytvoreného poľa; `dims[ndim]` je pole rozmerov v jednotlivých dimenziách a `chunk[ndim]` je pole rozmerov určujúce minimálnu veľkosť blokov, na ktoré bude globálne pole rozdelené v príslušnej dimenzii. Napríklad, pri dvojrozmernom poli (t. j. matici), priradenie `chunk[0]=dims[0]` zabezpečí, že stĺpce matice nebudú fyzicky rozdelené medzi paralelné procesy, naopak `chunk[1]=dims[1]` zachová celistvé riadky matice na každom z paralelných procesov. Výstupom je nenulový, celočíselný manipulátor (tzv. handle), ktorým sa v ďalších volaniach odkazujeme na vytvorené pole. Globálne pole je možné vytvoriť aj pomocou funkcie `GA_Duplicate`, pričom novovytvorené pole zdedí všetky atribúty (veľkosť, distribúcia blokov a pod.) kopírovaného poľa.



Obr. 3: Príklady rovnomernej distribúcie pri vytváraní globálneho poľa

Ďalšími základnými funkciami Global Array sú funkcie na zápis a čítanie údajov do, resp. z globálneho poľa:

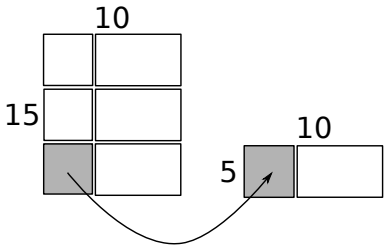
```
void NGA_Get (int g_a, int lo[], int hi[],
              void* buf, int ld[])
void NGA_Put (int g_a, int lo[], int hi[],
              void* buf, int ld[])
```

kde `g_a` je manipulátor (handle) globálneho poľa; `ndim` je počet dimenzií globálneho poľa, `lo[ndim]` (z anglického low, spodný) je pole počiatočných indexov (po dimenziách) v globálnom poli; `hi[ndim]` (tzv. vrchný) je analogické pole koncových indexov; `buf` je ukazovateľ na lokálny zásobník údajov z / do ktorého sa operácia vykoná a `ld[ndim-1]` je `ndim-1`-rozmerné pole "určujúcich rozmerov" (z anglického leading dimensions) poľa `buf`.

Poznámka: Určujúce rozmery rozhodujú o tom, ako budú fyzicky uložené údaje (ktoré sú v počítačovej pamäti vždy 1-rozmerné) rozdelené do viacerých dimenzií n-rozmerného poľa. V najjednoduchšom prípade 2-rozmerného poľa, je to rozdelenie na stĺpce a riadky matice. Stačí špecifikovať jednu určujúcu dimenziu (napr. počet prvkov v riadku) a druhá dimenzia, v tomto prípade počet riadkov, je automaticky určená celkovou veľkosťou 1-rozmerného poľa. Dôležité je dodať, že radenie prvkov n-rozmerného poľa v jazyku Fortran je opačné ako v jazyku C, t.j. rýchlejšie sa inkrementuje "skorší index", inak povedané, v pamäti sú za sebou uložené jednotlivé stĺpce matice.

V Global Arrays sa na nízkoúrovňovú jednostrannú komunikáciu medzi procesmi používala knižnica ARMCI (Aggregate Remote Memory Copy Interface), ktorá bola nedávno nahradená novšou knižnicou ComEx (Communication runtime for Extreme Scale). Obe spomenuté

Poznámka: Kolektívne operácie v Global Arrays su implementované iba pomocou MPI.



Obr. 4: Príklad operácie NGA_Get, čítanie z globálneho poľa 15x10 do lokálneho, s rozmermi 5x10

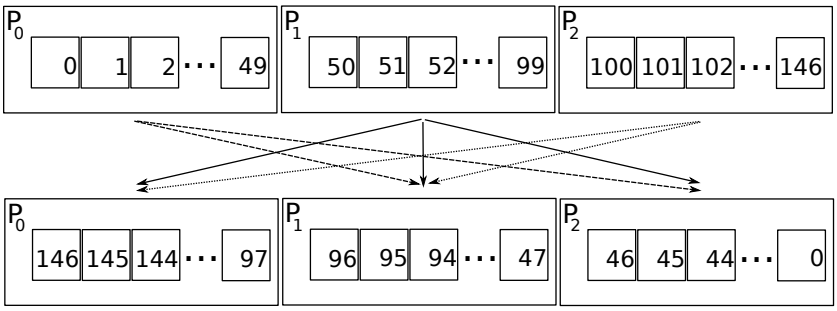
knižnice využívajú buď komunikačné primitívy konkrétnych sietí (Infiniband, Ethernet a pod.) alebo univerzálne dostupné MPI, ako v tomto prípade.

Ako príklad (viď. obrázok 4, môžme uviesť čítanie prvkov [10:14,0:4] z globálneho poľa rozmerov 15x10 do 2-rozmerného lokálneho zásobníka s rozmermi 5x10. Parametre funkcie GA_Get sú: lo=10,0, hi=14,4, ld=10.

Funkcie NGA_Get a NGA_Put (ako aj ďalšie funkcie v knižnici Global Arrays) existujú pre jazyk Fortran aj v zjednodušených, 2-dimenzionálnych variantách – GA_Get a GA_Put, kde sa už neuvádza počet rozmerov globálneho poľa a indexy lo a hi sú celými číslami a nie poliami. Funkcie na čítanie a zápis sú jednostranné, t.j. z pohľadu programátora nevyžadujú koordináciu ďalších paralelných procesov participujúcich na operácii. V knižnici Global Arrays je celkovo viac než 200 rôznych funkcií (blokujúcich, neblokujúcich, kolektívnych, synchronizačných a pod.), ktoré však z dôvodu obmedzeného rozsahu tejto učebnice neuvádzame.

Riešená úloha

Efektívnosť a jednoduchosť paralelného programovania s použitím Global Arrays demonštrujeme aj na reálnom príklade transpozície jednorozmerného poľa. Na obrázku 5 je znázornené (usporiadané) jednorozmerné pole celých čísel, od 0 po 146, ktoré je rozdelené medzi paralelné procesy P_0 až P_2. Úlohou programu je preusporiadať ich v opačnom poradí tak, aby zostali zachované počty jednotlivých prvkov na príslušných paralelných procesoch. Implementovať túto úlohu



Obr. 5: Počiatočná a konečná distribúcia údajov v príklade na transpozíciu jednorozmerného poľa

pomocou MPI by bolo pomerne jednoduché, ak by celkový počet prvkov bol (vždy) deliteľný počtom paralelných procesov. Každý paralelný proces by tak lokálne usporiadal prvky, ktoré má fyzicky uložené v pamäti, a následne by ich poslal príslušnému cieľovému procesu. Ak nie je počet prvkov deliteľný počtom procesov, opäť viď. obrázok 5, úloha je zložitejšia, lebo každý z rankov (okrem posledného, ktorý ma menej prvkov) musí posielať údaje na viac ako jedenému paralelnému procesu.

Nasledujúci výpis uvádza implementáciu transpozície v Global Arrays. Na riadkoch 18 až 22 sa inicializuje MPI prostredie a následne samotné Global Arrays. Po vytvorení globálneho poľa (riadok 34) a jeho následnom naplnení (riadky 2 až 8 v časti II.), funkcia `NGA_Distribution`

vráti každému z paralelných procesov interval indexov (`lo1` a `hi1`) ohraničujúcich údaje, ktoré sú fyzicky uložené v pamäti prislúchajúcej danému paralelnému procesu.

Zdrojový kód 3: Transpozícia 1D poľa pomocou Global Arrays - časť I.

```
1  #define   TOTALELEMS   147
2  #define   MAXPROC      32
3
4  #include <stdio.h>
5  #include <math.h>
6  #include "ga.h"
7  #include <mpi.h>
8
9  int main(int argc, char **argv)
10 {
11     int me, nprocs;
12
13     int dims[1], chunk[1], ld[1], lo, hi;
14     int lo1, hi1, lo2, hi2;
15     int g_a, g_b, a[MAXPROC*TOTALELEMS],b[MAXPROC*
       TOTALELEMS];
16     int nelem, i;
17
18     // Inicializácia MPI
19     MPI_Init(&argc, &argv);
20
21     // Inicializácia GA
22     GA_Initialize();
23
24     // Priradenie ID lokálneho ranku a celkového počtu
       rankov
25     me = GA_Nodeid();
26     nprocs = GA_Nnodes();
27
28     // Konfigurácia rozmerov polí
29     dims[0]  = nprocs*TOTALELEMS + nprocs/2;
30     ld[0]    = dims[0];
31     chunk[0] = TOTALELEMS;
32
33     // Vytvor Global Array g_a a duplikuj do g_b
34     g_a = NGA_Create(C_INT, 1, dims, "array A", chunk);
35     g_b = GA_Duplicate(g_a, "array B");
36 ...
```

Následne volanie `NGA_Get` (riadok 17) v druhej časti výpisu teda neprenáša údaje po sieti, ale kopíruje ich v rámci výpočtového uzla. Po lokálnom invertovaní údajov (riadok 21) sa vypočítajú nové indexy `lo2` a `hi2` (riadky 26 a 27), ktoré určujú pozíciu dátového segmentu v invertovanom globálnom poli `g_b`. Volanie funkcie `NGA_Put` (riadok 28) zabezpečí prenos údajov cieľovým paralelným procesom bez toho, aby programátor musel poznať príslušné mapovanie.

Zdrojový kód 4: Transpozícia 1D poľa pomocou Global Arrays - časť II.

```
...
    // Inicializuj údaje v g_a
    if (me==0) {
       for(i=0; i<dims[0]; i++) a[i] = i;
       lo = 0;
       hi = dims[0]-1;
       NGA_Put(g_a, lo, hi, a, ld);
    }

    // Synchronizuj všetky procesy pred ďalším pokračovaním
    GA_Sync();

    // Zisti, ktoré údaje sú na lokálnom ranku
    NGA_Distribution(g_a, me, lo1, hi1);

    // Kopíruj lokálne údaje z g_a do lokálneho zásobníka a
    NGA_Get(g_a, lo1, hi1, a, ld);

    // Invertuj údaje lokálne
    nelem = hi1 - lo1 + 1;
    for (i=0; i<nelem; i++) b[i] = a[nelem-1-i];

    // Invertuj údaje globálnym uložením invertovaných
     blokov na správne miesto v global array g_b
    lo2 = dims[0] - hi1 -1;
    hi2 = dims[0] - lo1 -1;
    NGA_Put(g_b, lo2, hi2, b, ld);

    // Synchronizuj všetky procesy
    GA_Sync();

    // Dealokuj polia
    GA_Destroy(g_a);
    GA_Destroy(g_b);

    GA_Terminate();
    MPI_Finalize();
}
```

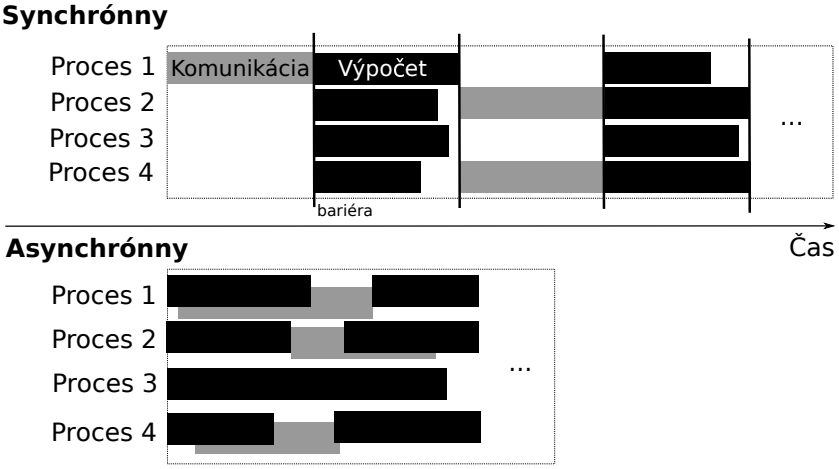
2 GASPI

Prehľad a história

GASPI (z anglického Global Address Space Programming Interface) je API (Aplication Programming Interface), čiže sada funkcií definovaných na prácu s PGAS. GASPI bolo vyvinuté na pôde inštitútu Fraunhofer ITWM (Institut für Techno- und Wirtschaftsmathematik) v Nemeckom Kaiserslauterne, ako odpoveď na podobné iniciatívy, ako napríklad Berkley Unified Parallel C (UPC). Vývoj má počiatky v

roku 2005, kedy vznikol projekt s názvom FVM (Fraunhofer Virtual Machine), neskôr (od roku 2009) premenovaný na GPI (Global address Programming Interface). Prvá špecifikácia API tak, ako je známa dnes a implementovaná do open-source knižnice GPI-2, bola formulovaná v spolupráci s priemyselnými partnermi v roku 2013.

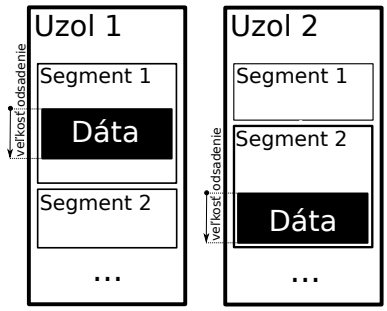
GASPI bolo od počiatku formulované a implementované s cieľom poskytnúť vysokú škálovateľnosť paralelných programov, flexibilitu a odolnosť voči chybám vzniknutým pri vykonávaní paralelného programu. Táto vlastnosť (z anglického fault tolerance alebo aj runtime resilience) je obzvlášť dôležitá pre masívne paralelné výpočty (s tisícmi až státisícmi paralelných procesov), kde je zlyhanie niektorého z hardvérových alebo komunikačných komponentov v časovom horizonte vykonávania programu vysoko pravdepodobné a zväčša vyústi k chybnému ukončeniu aplikácie. Kľúčom k dosiahnutiu škálovateľnosti v GASPI je nahradenie "tradičných" blokujúcich paralelných operácií jednostrannými, asynchrónnymi operáciami spolu a poskytnutím príslušných nízkoúrovňových komunikačných rutín. Asynchrónne, jednostranné operácie umožňujú efektívny prekryv výpočtu (častí programu intenzívne zaťažujúcich CPU, napríklad aritmetické operácie, alebo vykonávajúce I/O) a komunikácie, nakoľko nevyžadujú aktívnu participáciu druhej strany, viď obrázok 6. Z hľadiska efektívneho využitia sú preferované sieťové komponenty, ktoré umožňujú tzv. RDMA (z anglického Remote Direct Memory Access, t.j. priamy prístup do vzdialenej pamäte), čo tradične poskytuje napr. Infiniband, v súčasnosti však aj Ethernet.



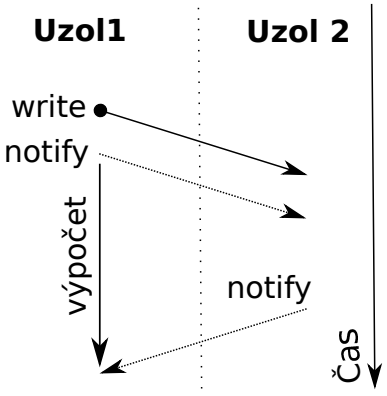
Obr. 6: Prechod od synchrónneho k asynchrónnemu paralelnému modelu – prekryv výpočtu a komunikácie

Segmenty, komunikácia a synchronizácia

Základným elementom GASPI umožňujúcim jednostrannú komunikáciu sú tzv. segmenty – spojité pamäťové bloky alokované v rámci výpočtového uzla. Segmentov je možné alokovať na danom uzle viacero, nakoľko je však samotná alokácia segmentov blokujúca (a relatívne časovo náročná) operácia, vykonáva sa väčšinou hneď po inicializácii GASPI prostredia, t.j. na začiatku programu. K segmentom je možné pristupovať lokálne (analogicky k blokom alokovaným použitím napr. *malloc*) alebo globálne, pomocou GASPI funkcií. Výpočtové uzly môžu



Obr. 7: Adresovanie údajov v GASPI segmentoch



Obr. 8: GASPI zápis (write) a notifikácie (notify)

Poznámka: Alternatívny spôsob synchronizácie je využívanie bariér, ktoré nútia program čakať na dokončenie prenosu údajov, t.j. funkcia gaspi_barrier.

priamo a bez asistencie partnerského uzla buď čítať údaje zo vzdialeného segmentu, alebo doň zapisovať a to bez nutnosti použitia vyrovnávacej pamäte (tzv. zero-copy prenos).

Syntax jednej z dostupných funkcií v GASPI na vytvorenie segmentu je nasledovná:

```
gaspi_return_t gaspi_segment_create (
          gaspi_segment_id_t segment_id ,
          gaspi_size_t size ,
          gaspi_group_t group ,
          gaspi_timeout_t timeout ,
          gaspi_alloc_t alloc_policy
    )
```

kde `segment_id` je (zvolený) identifikátor segmentu; `size` je veľkosť segmentu (v bajtoch); `group` je skupina rankov (paralelných procesov), na ktorých sa segment vytvorí; `timeout` je časový limit (v milisekundách), po ktorom je možné získať návratovú hodnotu funkcie (môže mať aj špeciálne hodnoty ako `GASPI_TEST` – najkratší možný čas, teoretickú nula milisekúnd, alebo `GASPI_BLOCK` – nekonečne dlhý časový interval), ak sa (úspešne alebo neúspešne) už nevykonala skôr; `alloc_policy` je parameter určujúci, či má byť pamäťový segment pri alokácii aj inicializovaný, t.j. zapísaný nulami. GASPI samozrejme obsahuje aj ďalšie funkcie na prácu so segmentami, ako napríklad `gaspi_segment_ptr`, ktorá pre segment (podľa identifikátora) vráti ukazovateľ na jeho polohu v lokálnej pamäti.

Priebeh zápisu (write) a analogicky čítania (read) do GASPI segmentu je znázornený na obrázku 8. Uzol 1 iniciuje zápis údajov do segmentu Uzla 2 volaním príslušnej GASPI funkcie (pri čítaní je to, napríklad `gaspi_write`). Uzol 1 nečaká na ukončenie zápisu (volanie nie je blokujúce), okamžite pokračuje vo vykonávaní nasledujúcich inštrukcií programu (spomenutý prekryv výpočtu a komunikácie). Uzol 2 sa doposiaľ na komunikácii nepodieľal, nie je si teda ani vedomý prenosu údajov. Synchronizácia medzi uzlami je zabezpečená posielaním notifikácií. Keď Uzol 2 obdrží notifikáciu (presnejšie, notifikáciu musí očakávať), môže si byť istý, že zápis údajov z Uzla 1 je už ukončený. V prípade, že Uzol 1 potrebuje vedieť, že prenos údajov na segment Uzla 2 bol ukončený, môže taktiež očakávať a následne aj obdržať notifikáciu z Uzla 2.

Syntax funkcie na zápis a údajov a súčasné odoslanie notifikácie je nasledovná:

```
gaspi_return_t gaspi_write_notify (
          gaspi_segment_id_t segment_id_local,
          gaspi_offset_t offset_local,
          gaspi_rank_t rank,
          gaspi_segment_id_t segment_id_remote,
          gaspi_offset_t offset_remote,
          gaspi_size_t size,
          gaspi_notification_id_t notification_id,
          gaspi_notification_t notification_value,
          gaspi_queue_id_t queue_id,
```

```
        gaspi_timeout_t timeout
    )
```

kde `segment_id_local` a `offset_local` sú identifikátor a odsadenie segmentu odosielajúceho procesu; `rank` identifikuje paralelný proces adresáta údajov; `segment_id_remote` a `offset_remote` sú identifikátor a odsadenie segmentu procesu prijímateľa údajov; `size` je veľkosť prenášaných údajov (v bajtoch); argumenty `notification_id` a `notification_value` sú identifikátor a hodnota posielanej notifikácie; `queue_id` je identifikátor komunikačného frontu a `timeout` má rovnaký význam, ako pri funkcii `gaspi_segment_create`.

Synchronizácia pomocou notifikácií je vlastnosťou GASPI, ktorá ju výrazne odlišuje od iných PGAS knižníc. Vďaka notifikáciam je možné uskutočniť a synchronizovať komunikáciu výlučne v lokálnom procese, t.j. bez aktívnej participácie vzdialeného paralelného procesu. Všetky žiadosti o zápis, čítanie údajov, ako aj notifikácie sú zaznamenané v lokálnom komunikačnom fronte paralelného procesu. Týchto frontov spravuje každý proces niekoľko, čo umožňuje separovať komunikáciu rôznych častí programu (napr. externých knižníc). Žiadosti nemusia nutne byť vykonané v rovnakom poradí, v akom boli vložené do frontu, GASPI implementácia frontov však zaručuje poradie notifikácií a žiadostí vložených skôr ako príslušná notifikácia.

Kolektívne operácie

GASPI ponúka okrem jednostranných aj kolektívne operácie, čiže také, ktoré vykonáva súčasne skupina alebo všetky paralelné procesy. Konkrétny paralelný proces môže byť súčasťou viacerých GASPI skupín rankov, každá skupina však môže vykonávať iba jeden typ kolektívnej operácie v daný čas. Jedinečnosť GASPI kolektívnych operácií spočíva v tom, že sú na rozdiel od tradičných MPI ekvivalentov, asynchrónne - neblokujúce. Príklad kolektívnej operácie je tzv. Allreduce, známej z MPI (vykoná zvolenú operáciu použitím prvkov poľa naprieč paralelnými procesmi a identický výsledok rozdistribuuje späť na každý proces), ktorej syntax je nasledovná:

```
gaspi_return_t gaspi_allreduce (
          gaspi_pointer_t buffer_send,
          gaspi_pointer_t buffer_receive,
          gaspi_number_t num,
          gaspi_operation_t operation,
          gaspi_datatype_t datatyp,
          gaspi_group_t group,
          gaspi_timeout_t timeout
)
```

kde `buffer_send` a `buffer_receive` sú ukazovatele na zdrojové a cieľové zásobníky údajov; `num` je počet elementov v zásobníku; `operation` je jedna z možných operácií podporovaných GASPI – `GASPI_OP_MIN`, `GASPI_OP_MAX` a `GASPI_OP_SUM`, t.j. minumum, maximum a suma prvkov v zásobníku; `datatyp` je jeden z možných údajových typov podporovaných GASPI (napríklad `GASPI_TYPE_INT`, `GASPI_TYPE_LONG`, atď.);

group je skupina rankov zúčastňujúcich sa operácie a timeout má rovnaký význam, ako v predchádzajúcich prípadoch. Návratom z funkcie je buď GASPI_SUCCESS, v prípade úspešného ukončenia, GASPI_ERROR, v prípade chyby alebo GASPI_TIMEOUT, v prípade vypršania časového limitu. Návratová hodnota GASPI_TIMEOUT neznamená nutne chybu, iba indikuje, že sa funkcia ešte stále vykonáva, čo je opäť možné využiť na prekryv prenosu údajov a výpočtu, viď. výpis 5.

Zdrojový kód 5: Kolektívne operácie - prekryv komunikácie a výpočtu

```
1  while (ret = (gaspi_allreduce (
2                  buffer_send,
3                  buffer_receive,
4                  num,
5                  operation,
6                  datatyp,
7                  group,
8                  GASPI_TEST
9              )) != GASPI_SUCCESS
10       )
11 {
12   if ( ret == GASPI_ERROR )
13   {
14      reakcia_na_chybu (&ret);
15   }
16   vypocet();
17 }
```

Volaním funkcie gaspi_allreduce sa spustí asynchrónny transfer údajov, program však okamžite pokračuje ďalej, nakoľko bola funkcia volaná s (teoreticky) nulovým časový limitom – GASPI_TEST. V prípade, že návratová hodnota funkcie nebola chybová, t.j. GASPI_ERROR (riadok 12), program pokračuje volaním funkcie vypocet(). Po vykonaní výpočtu sa opäť kontroluje návratová hodnota funkcie gaspi_allreduce (riadok 9), ktorá medzičasom (ak bol výpočet dostatočne dlhý) úspešne alebo neúspešne skončila.

Odolnosť voči chybám pri vykonávaní paralelného programu

GASPI ponúka dva základné nástroje na programovanie paralelných aplikácii odolných voči chybám, ktoré sa môžu vyskytnúť pri ich vykonávaní. Základom je už spomenutý "timeout" mechanizmus, ktorý je zabudovaný v každej GASPI funkcii. V prípade, že volaná funkcia nebola dokončená v očakávanom časovom limite (alebo aj z akéhokoľvek iného dôvodu), stav všetkých paralelných procesov je možné zistiť volaním funkcie gaspi_state_vec_get. Návratovou hodnotou je vektor obsahujúci stavy GASPI_STATE_HEALTHY (proces je v poriadku) alebo GASPI_STATE_CORRUPT (proces nekomunikuje) pre každý rank. V prípade, že bol detegovaný chybový stav niektorého z paralelných

procesov, aplikácia môže na vzniknutý stav reagovať, napríklad presunom úlohy zlyhaného procesu na "záložný" rank, preusporiadaním práce pre zostávajúce ranky, alebo návratom k poslednej zálohe (z anglického checkpoint). Reakcia aplikácie na chybový stav je plne v réžii programátora, GASPI neposkytuje žiadne východzie riešenie, iba nástroje potrebné na jeho implementáciu.

Jedným z takýchto nástrojov je modul s názvom GPI CP [7], ktorý zjednodušuje proces vytvárania a manažmentu záloh. Vytváranie záloh a ich klonovanie na viaceré paralelné procesy (z dôvodu redundancie) sa deje asynchrónne, na pozadí vykonávania hlavného programu a nemá prakticky žiadny penalizačný dopad na jeho efektivitu.

Riešená úloha

Pre úplnosť uvádzame aj GASPI implementáciu programu na inverziu jednorozmerného poľa s rovnakým zadaním, ako pre Global Arrays, viď. výpisy 6 a 7. Zložitosť programu je už na prvý pohľad väčšia v porovnaní s Global Arrays, stále však menšia ako analogická implementácia v MPI (kedže nemusíme explicitne implementovať obojstrannú komunikáciu medzi rankami). GASPI automaticky neponúka tak vysokú abstrakciu "emulovaného" programovania systémov so zdieľanou pamäťou, aký poskytuje Global Arrays, t.j. nie je možné jedným volaním GASPI funkcie vytvoriť globálne pole naprieč paralelnými procesmi tak, ako pomocou NGA_Create. Ak by však bola takáto funkcionalita žiadaná, nepochybne môže byť, s primeraným programátorským úsilím, implementovaná pomocou dostupných GASPI funkcií.

Program využíva jednoduchý zápis do segmentov paralelných procesov, bez koordinácie pomocou notifikácií. Koordinácia je zabezpečená až na konci procesu volaním kolektívnej operácie `gaspi_barier`, t.j. čakaním na koniec komunikácie medzi všetkými procesmi. Interoperabilitu s MPI (nie je súčasťou príkladu) je možné zabezpečiť analogicky ako v príklade na Global Arrays. Pred inicializáciou GASPI prostredia (riadok 10) sa najskôr inicializuje MPI (analogicky s príkladom na Global Arrays), vďaka čomu bude priradenie MPI identifikátorov rankov identické s GASPI (riadok 11, funkcia `gaspi_proc_rank`).

Zdrojový kód 6: Transpozícia 1D poľa pomocou GASPI - časť I.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <GASPI.h>
4
5  #define TOTALELEMS 147
6
7  int main(int argc,char *argv[])
8  {
9      gaspi_rank_t rank, size;
10     gaspi_proc_init(GASPI_BLOCK);
11     gaspi_proc_rank(&rank);
12     gaspi_proc_num(&size);
13     gaspi_segment_id_t seg_id_a = 0, seg_id_b = 1;
14     gaspi_pointer_t seg_ptr_array, seg_ptr_value;
15     gaspi_size_t seg_length;
16     gaspi_offset_t l_p = 0, r_p = 0;
17
18     int i, n, n_part, s = 0, shift = 0;
19     int *a, *b;
20
21     n = TOTALELEMS;
22
23     // Výpočet dĺžky segmentov
24     if( n%size == 0 )
25         n_part = n/size;
26     else
27     {
28         shift = size-n%size;
29         n_part = (n+shift)/size;
30         s = 1;
31
32         if ( rank == size-1 )
33         {
34             n_part = n-n_part*(size-1);
35             shift = 0;
36         }
37     }
38
39     // Alokácia GASPI segmentov
40     seg_length = sizeof (int);
41
42     gaspi_segment_create ( seg_id_a, n_part*seg_length,
       GASPI_GROUP_ALL,GASPI_BLOCK, GASPI_MEM_INITIALIZED );
43     gaspi_segment_create ( seg_id_b, n_part*seg_length,
       GASPI_GROUP_ALL,GASPI_BLOCK, GASPI_MEM_INITIALIZED );
44 ...
```

Na rozdiel od Global Arrays môžu paralelné procesy priamo zapisovať do segmentov, ktoré sa (fyzicky) nachádzajú na príslušnom ranku, teda aj výpočtovom uzle. Po alokácii segmentov (riadky 42 a 43) sa získajú ukazovatele (volanie `gaspi_segment_ptr` na riadkoch 3 a 6 v druhej časti výpisu), ktoré sa použijú lokálne (polia `a` a `b` na riadkoch

4 a 7).

Zdrojový kód 7: Transpozícia 1D poľa pomocou GASPI - časť II.

```
1  ...
2      // Získanie lokálnych ukazovateľov
3      gaspi_segment_ptr ( seg_id_a, & seg_ptr_array );
4      a = (int *) seg_ptr_array ;
5
6      gaspi_segment_ptr ( seg_id_b, & seg_ptr_array );
7      b = (int *) seg_ptr_array ;
8
9      // Inicializácia údajov
10     for (i = 0; i < n_part; i++)
11     {
12         a[i] = rank*(n+s*size-n%size)/size + i;
13         b[i] = a[i];
14     }
15
16     // Invertovanie lokálnych údajov
17     for (i = 0; i < n_part; i++)
18     {
19         a[i] = b[n_part-i-1];
20     }
21
22     // Invertovanie globálnych údajov
23     if( shift > 0 )
24     {
25         l_p = 0;
26         r_p = (n_part-shift)*sizeof(int);
27         gaspi_write( seg_id_a, l_p, size-2-rank, seg_id_b,
       r_p, shift*sizeof(int), 0, GASPI_BLOCK);
28     }
29
30     l_p = shift*sizeof(int);
31     r_p = 0;
32     gaspi_write( seg_id_a, l_p, size-1-rank, seg_id_b, r_p,
        (n_part-shift)*sizeof(int), 0, GASPI_BLOCK);
33
34     // Synchronizuj všetky procesy //
35     gaspi_barrier ( GASPI_GROUP_ALL, GASPI_BLOCK );
36
37     // Ukončenie GASPI
38     gaspi_proc_term ( GASPI_BLOCK );
39     return 0;
40 }
```

Jadrom programu je časť `Invertovanie globálnych údajov` (riadky 16 až 32 v druhom výpise). Každý z procesov zapisuje zo segmentu `seg_id_a` do `seg_id_b` (paralelného procesu s ID rovným `size-1-rank`, kde `rank` je ID lokálneho procesu, viď. riadok 32) časť invertovaných údajov, ktoré nezávisia od toho, či je ich celkový počet deliteľný počtom procesov. Ak tento počet nie je deliteľný, je nutné vykonať aj zápis (riadok 27) do segmentu procesu s ID rovným `size-2-rank`.

Literatúra

[1] Barbara Chapman et al. “Introducing OpenSHMEM: SHMEM for the PGAS Community”. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. PGAS ’10. New York, New York, USA: ACM, 2010, 2:1–2:3. DOI: 10.1145/2020373.2020375 (cited on page 1).

[2] Tarek El-Ghazawi and Lauren Smith. “UPC: Unified Parallel C”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. DOI: 10.1145/1188455.1188483 (cited on page 1).

[3] Philippe Charles et al. “X10: An Object-oriented Approach to Non-uniform Cluster Computing”. In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 519–538. DOI: 10.1145/1103845.1094852 (cited on page 1).

[4] Laxmikant V. Kale and Sanjeev Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++”. In: *SIGPLAN Not.* 28.10 (Oct. 1993), pp. 91–108. DOI: 10.1145/167962.165874 (cited on page 1).

[5] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. “Global Arrays: A Portable "Shared-memory" Programming Model for Distributed Memory Computers”. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing ’94. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 340–349 (cited on page 1).

[6] Thomas Alrutz et al. “GASPI – A Partitioned Global Address Space Programming Interface”. In: *Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–136. DOI: 10.1007/978-3-642-35893-7_18 (cited on page 1).

[7] Valeria Bartsch et al. “GASPI/GPI In-memory Checkpointing Library”. In: Aug. 2017, pp. 497–508. DOI: 10.1007/978-3-319-64203-1_36 (cited on page 11).