OpenMPI

Veljko Petrović
Novembar, 2022

Ako imamo OpenMP...

- ...čemu ovo?
- OpenMPI je optimizovan za situacije gde je model deljene memorije jednostavno nije primenjljiv.
- MPI je skraćenica od Message Passing Interface i predstavlja metod kojim se rešava programiranje masivno paralelnih arhitektura.
- U jednom trenutku udarimo u limit na broj procesora koji možemo nagurati u jedno kućište: ako ništa drugo, svi ti procesori se bore oko iste memorije, a transfer stope memorije su prilično oštro ograničene.
- Postoje situacije gde ta ograničenja ne važe (i to će biti vrlo detaljno istraženo kada budemo pričali o OpenACC-

OpenMPI

Model prosleđivanja poruka

u)ali u opštem slučaju podeljena memorija i vrlo pažljiva komunikacija su naša jedina opcija.

Istorija i poreklo

- MPI nije biblioteka nego specifikacija.
- Kolekcija standarda koja specificira način programiranja i protokol mrežne komunikacije.
- Entoni Hoar je principe ‘message passing’ modela postavio 70-tih godina prošlog veka, a 1992 je tim predstavnika akademije i industrije je postavio MPI specifikaciju vođen stručnjacima kao što je Viljem D. Grop.
- MPI ima verzije (MPI-1, MPI-2, MPI-3 itd.) koje mogu biti implementirane različitim skupovima alata i biblioteka.
- MPICH je referentna implementacija.
- OpenMPI je opšta implementacija.
- Mnogo specijalizovanih implementacija postoji.

Struktura

- Kao i OpenMP, OpenMPI nije jezik.
- Odavno je primećeno da posebni jezici retko doživljavaju široku prihvaćenost.
- OpenMPI je:
  - Protokol (OSI nivo 5)
  - Biblioteka koja proširuje postojeće jezike sa konstruktima za paralelizam
  - Alati

Osnovna implementacija

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

int main(int argc, char** argv){
  MPI_Init(&argc, &argv);
  printf("Hello, World\n");
  MPI_Finalize();
  return 0;
}
```

Osnovna implementacija

```
$ mpicc hello.c -o hello
$ mpirun -np 4 ./hello
Hello, World
Hello, World
Hello, World
Hello, World
```

Osnovna implementacija

```
9165 pts/0    00:00:00 mpirun
9170 pts/0    00:00:00 hello
9171 pts/0    00:00:00 hello
9172 pts/0    00:00:00 hello
9173 pts/0    00:00:00 hello
```

Komunikacija

- Kao i većina ovakvih programa, ovo ne radi ništa korisno.
- Da bi OpenMPI bio koristan, programi koji se paralelno izvršavaju moraju da komuniciraju.
- Ovo i nije baš 100% tačno. Moguće je držati programe potpuno nezavisno u tkzv. ‘share nothing’ modelu koji rešava prilično širok dijapazon problema poznat kao ‘throughtput.’
- Rendering je dobar primer.
- Ali za ovo bi koristili SLURM i ništa drugo. OpenMPI je čist višak.
- Ne mogu komunicirati na način na koji to radi OpenMP, preko deljene memorije, zato što iako u našem primeru malopre sve četiri instance su na istom računaru (i istoj ps tabeli) OpenMPI je namenjen da radi na potencijalno udaljenim računarima.

Komunikator

- Komunikator u OpenMPI-u je kolekcija procesa tj. nezavisnih pokrenutih instanci našeg programa i može se porediti sa radio frekvencijom ili kanalom na `IRC/Discord` serveru.
- Proces može učestvovati u proizvoljnom broju komunikatora: oni su tu da bi se komunikacija lakše organizovala.
- Minimalan broj komunikatora je jedan: svaki MPI program apsolutno mora imati barem jedan komunikator koji ne moramo da stvaramo: `MPI_COMM_WORLD`.
- `MPI_COMM_WORLD` je globalni kanal komunikacije: svi procesi su deo njega.

API Komunikatora

- MPI_Comm je tip identifikatora komunikatora
- Postoje komande koje služe za manipulaciju komunikatorom koje sve primaju kao parametar identifikator komunikatora

Veličina komunikatora

```c
MPI_Init(&argc, &argv);
int size;
MPI_Comm_Size(MPI_COMM_WORLD, &size);
printf("%d\n", size);
MPI_Finalize();
return 0;
```

Red procesa

- Red (rank) procesa je njegov identifikator unutar komunikatora.
- U pitanju je ceo broj nasumično dodeljen u okviru komunikatora u opsegu 0..size-1
- Dobavlja se uz pomoć `MPI_Comm_Rank` funkcije koja prima identifikator komunikatora i pokazivač na int gde valja smestiti vrednost.

Nedeterminizam izvršavanja

Slanje poruka na neku adresu

- `MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `message`—pokazivač na poruku generičkog tipa
- `count`—koliko elemenata ima u poruci
- `datatype`—Kog je tipa poruka
- `dest`—red procesa kome se šalje
- `tag`—celobrojna vrednost rezervisana za proizvoljnu upotrebu
- `comm`—komunikator koji se koristi

Primanje poruka

- `MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `message`—pokazivač na poruku generičkog tipa gde smeštamo vrednost koju dobijemo
- `count`—koliko elemenata ima u poruci
- `datatype`—Kog je tipa poruka
- `source`—red procesa od koga se prima
- `tag`—celobrojna vrednost rezervisana za proizvoljnu upotrebu
- `comm`—komunikator koji se koristi

- `status`—podaci o poruci uključujući ko je stvarno šalje i šta je poslat tag

Korisnički tipovi podataka

- Moguće je dodati i naš tip podataka baziran na struct-u
- Nažalost sintaksa je malo kompleksna
- Tip se prvo definiše a zatim upiše

Definisanje tipa

```
MPI_Type_create_struct(
    int number_items,
    const int *blocklength,
    const MPI_Aint *array_of_offsets,
    const MPI_Datatype *array_of_types,
    MPI_Datatype *new_datatype)
```

Upisivanje tipa

```
MPI_Type_commit(MPI_Datatype *new_datatype)
```

Primer

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

int main(int argc, char** argv){
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_Rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_Size(MPI_COMM_WORLD, &size);

  int message[2];
  int dest,src,tag = 0;
  MPI_Status status;

  if(rank != 0){
    message[0] = rank;
```

Primer

```
veljko@HPC:~/c/mpi2$ mpirun -np 4 ./msg
Poruka sa procesa 1 od 4
Poruka sa procesa 2 od 4
Poruka sa procesa 3 od 4
```

Primer

- Ovo je klasična implementacija radnik-menadžer obrasca u MPI.
- Nema ničeg posebnog oko procesa sa redom 0, ali ga je lako naći.
- Primetite da je izlaz ovog procesa sada deterministički.

Barijerna sinhronizacija

- Neke komande nisu namenjene posebnom procesu: neke služe za adresiranje više procesa
- To su kolektivne komande (`collectives`)
- Primer toga je `MPI_Barrier(MPI_Comm communicator)`
- Ovo je zaustavljanje izvršavanja dok svi procesi u komunikatoru nisu stigli na isto mesto.
- Ovo je 1:1 ekvivalentno sa `#pragma omp barrier`

Kolektivna komunikacija

- Kolektivna komunikacija opisuje situaciju gde imamo više procesa koji komuniciraju sa više procesa.
- Može se isprogramirati štagod da je to što želim, ali ovaj tip komunikacije pada uglavnom u četiri obrasca:
  - emitovanje (broadcast)
  - rasipanje (scatter)
  - skupljanje (gather)
  - sinhronizovano skupljanje (allgather)

Broadcast

- Broadcast služi da podatak sa jednog procesa premesti na sve ostale ciljane procese.
- Može da koristi da osveži nekakvu tabelu međurezultata, ali je najkorisnije prilikom početka rada procesa.
- Divan primer broadcast-a se može videti ako razmišljamo o onoj fibonači implementaciji i njenom prvom koraku

Broadcast

`int MPI_Bcast(void* shared_data, int number, MPI_Datatype datatype, int source_process, MPI_Comm communicator)` Svi procesi pozivaju istu funkciju. Razlika je u tome što onaj proces čiji je red jednak `source_process` parametru čita iz `shared_data`, a ostali samo smeštaju ono što dobiju tu.

Scatter

- Scatter, kao i broadcast, jeste slučaj slanja sa jednog na više procesa
- Razlika je u tome što proces koji šalje takođe podeli ono što šalje u onoliko (tipično ne-preklapajućih) podskupa koliko ima procesa kojima se šalje.
- Ako se razmisli, u onom fibonači primeru posle broadcast-a treba scatter koji podeli skup rednih brojeva fibonačijevog broja koji se računa između procesa

Scatter

`MPI_Scatter(void *send_data, int send_number, MPI_Datatype datatype, void *put_data, int put_number, int source_rank, MPI_Comm communicator)` `send_data/number` je bafer za slanje (prazan i nebitan za sve osim za proces koji ima `source_rank` red) `put_data/number` je bafer za primanje koji svi koriste.

Gather

- Gather je obrnut proces od scatter (očigledno).
- Više procesa šalje jednom procesu delove nekog većeg skupa podataka koji se zatim spaja.
- Posle broadcast i scatter faze, naš fibonači algoritam bi imao gather da iz procesa izvadi podatke o sledećih m brojeva i skupi ih u niti koja upravlja podacima.

Gather

`MPI_Gather(void *send_data, int send_number, MPI_Datatype datatype, void *put_data, int put_number, int destination_rank, MPI_Comm communicator)` `send_data/number` je bafer za slanje (koji svi koriste) `put_data/number` je bafer za primanje (koji je prazan i nebitan za sve osim procesa sa redom koji je ravan `destination_rank`)

Allgather

- Allgather je isto što i gather samo što ga odmah prati broadcast onoga što se gather-uje.
- Fibonači program bi, u stvari, imao `allgather` operaciju umesto gather operacije za sve blokove osim poslednjeg.

Allgather

`MPI_Allgather(void *send_data, int send_number, MPI_Datatype datatype, void *put_data, int put_number, MPI_Comm communicator)` `send_data/number` je bafer za slanje (koji svi koriste) `put_data/number` je bafer za primanje (koji svi koriste)
Naravno, mogli bi i da radimo `gather` praćen sa `broadcast` -om, ali nema razloga.

Redukcije

- Redukcije su olakšica koja omogućava kompleksnu komunikaciju neophodnu da se paralelizuje, npr. sumiranje niza i slične operacije.
- Apsolutno je ista namena kao ekvivalentne `OpenMP` konstrukcije.
- Veoma je slična sintaksi `gather` komande:
- `int MPI_reduce(const void *send_data, void *put_data, int send_number, MPI_Datatype, MPI_OP operation, int destination_rank, MPI_Comm comm)`

- Jedina razlika jeste operacija koja služi za kombinovanje koja može biti korisnički definisana ili jedna od osnovnih operacija

Redukcioni operatori

Operacija	Operator
Maksimum	MPI_MAX
Minimum	MPI_MIN
Suma	MPI_SUM
Proizvod	MPI_PROD
Logičko I	MPI_LAND
Bit I	MPI_BAND
Logičko ILI	MPI_LOR
Bit ILI	MPI_BOR
Logičko XOR	MPI_LXOR
Bit XOR	MPI_BXOR
Maksimalna vrednost i lokacija	MPI_MAXLOC
Minimalna vrednost i lokacija	MPI_MINLOC

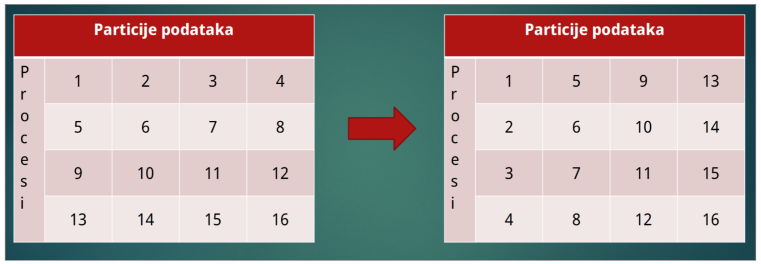
Sinhronizovana redukcija

- Kao što je `Reduce` bazirano na `Gather`, `Allgather` proizvodi `Allreduce`
- `int MPI_Allreduce(const void *send_data, void *put_data, int send_number, MPI_Datatype datatype, MPI_OP operation, MPI_Comm comm)`

Svi/svi komunikaciona šema

- To je šema komunikacije koja:
  - Aranžira sve procese tako da formiraju matricu gde su
    - Redovi procesi
    - Kolone particije podataka kao što bi ih scatter komanda napravila
  - Formira transfer podataka iz izlaznog u ulazni bafer tako da efektno transponira podatke

Obrazac svi/svi komunikacione šeme



Svi/svi komunikaciona šema

```
int MPI_Alltoall(void *send_data, int
send_number, MPI_Datatype send_datatype,
void *put_data, int put_number, MPI_Datatype
put_datatype, MPI_Comm communicator)
```

Neblokirajuća usmerena komunikacija

- Do sada, svo slanje podataka je blokirajuće
- Ako nema Recv za svako Send program stane.
- Takođe, imamo nužno sinhrono ponašanje, to može da uspori program: ako ne moramo da sinhronizujemo, ne treba.
- Radi isto kao ranije, ali vraća `MPI_Request` objekat
- `int MPI_Isend(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *send_request)`
- `int MPI_Irecv(void *message, int count, MPI_Datatype datatype, int source, int tag,`

```
MPI_Comm comm, MPI_Request
*receive_request)
```

`MPI_Request`

- Ovo je promenljiva koja pokazuje na potencijalno neispunjenu operaciju slanja/primanja
- Kada je imamo, možemo je koristiti da sačekamo da se operacija završi, sinhronizujući naš poziv kada zaželimo:
- `int MPI_Wait(MPI_Request *req, MPI_Status *status)`
- Ovo će vratiti status kada se operacija bude završila.
- Moguće je i asinhrono proveriti da li se operacija završila kroz:
- `int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)`
- Ovo se odmah završi i postavi flag na true ako je operacija gotova i status na vrednost statusa ako je flag true.