High-**P**erformance **C**omputing 20**25**

Distributed Memory Parallelism with MPI

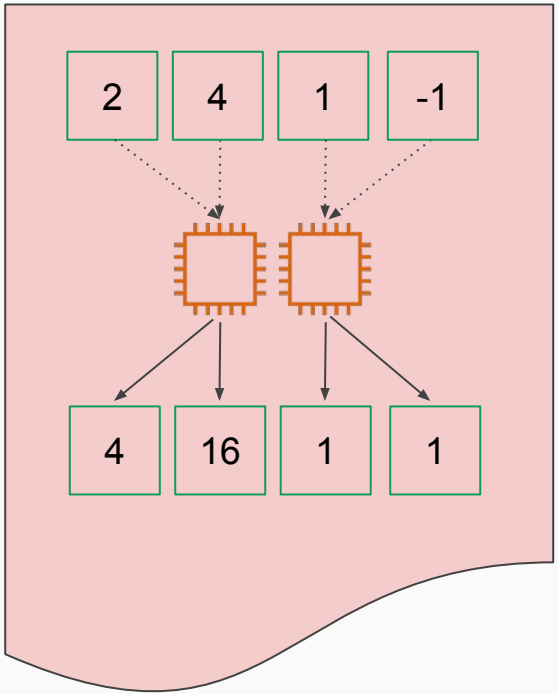
Aryan Eftekhari| **U**niversità della **S**vizzera **i**taliana (USI) | **I**nstitute of **C**omputing (CI)

Review
OpenMP

Shared Memory Parallelization

*Computation is distributed along **threads**.*

Synchronization between threads.



OpenMP is easy to use …

```
#include <omp.h>
#include <vector>

int main(){
    std::vector<double> val(1e8,0);
    #pragma omp parallel for
    for (int i = 0; i < val.size(); i++)
        val[i] = COSTLY_OPERATION(i);
    return 0;
}
```

Parallel Region

```
// In Terminal/Command line
// Compile via command line (or makefile)
g++ -fopenmp -O3 main.cpp -o main.exe

// Run
export OMP_NUM_THREADS=2; ./main.exe
```

Parallel computing with **MPI**

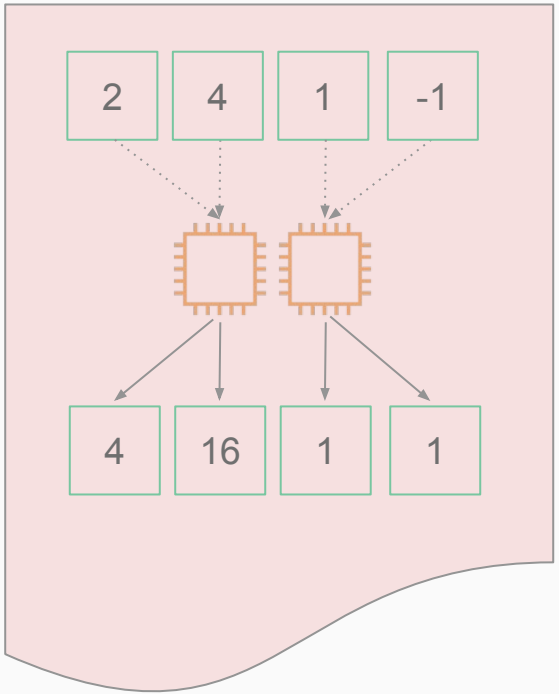
The Message Passing Interface standard was established in mid 90s.
OpenMPI is common implementation of this standard.

Distributed/Shared-Memory Parallelism
Parallelization paradigms

Shared Memory Parallelization

*Computation is distributed along **threads**.*

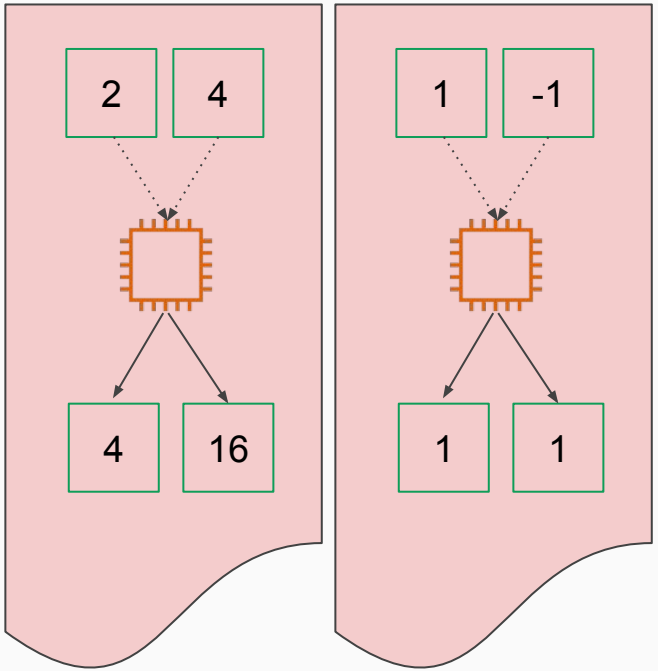
Synchronization between threads.



Distributed Memory Parallelization

*Computation is distributed along **processes**.*

Communication via message passing.



Three considerations: **code**, **compiler**, and **execution**.

```cpp
#include <mpi.h>
#include <vector>

int main(){
    int size, rank;
    MPI_Init(NULL,NULL); // Needs to be called
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<double> val(size,0);
    val[rank]=some_func(rank);
    MPI_Allreduce( MPI_IN_PLACE,&val[0],size,MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD);
    MPI_Finalize(); // Needs to be called
    return 0;
}
```

Initialize, assign rank and size.
… common start to MPI code.

```
// Compile via command line/makefile
mpic++ -O3 main.cpp -o main.exe

// Run
mpirun -np 2 ./main.exe
```

MPI Programming = **Planning** + **Functions**

Provides control over **program flow** and **communication**.

```
for ( int i = 0; i < n; i++){
 val[i] = some_func(i)
}
```

```
val[rank] = some_func(rank)
MPI_Allreduce( MPI_IN_PLACE,
&val[0],size,MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
```

- **Planning:** How will you split the computation?
- **Communication:** How will you aggregate the results?

MPI requires more planning (*as compared to OpenMP*).

Program Flow

Execute n of the same program

```
#include <mpi.h>
int main(){
    int size, rank;
    MPI_Init(NULL,NULL);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Finalize();
    return 0;
}
```

```
// via terminal/command line
mpirun -np 3 ./main.exe
```

```
#include <mpi.h> //rank=0
#include <mpi.h> //rank=1
#include <mpi.h> //rank=2
int main(){
    int size, rank;
    MPI_Init(NULL,NULL);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Finalize();
    return 0;
}
```

1) **Code:** There is one program/class/cpp file etc.

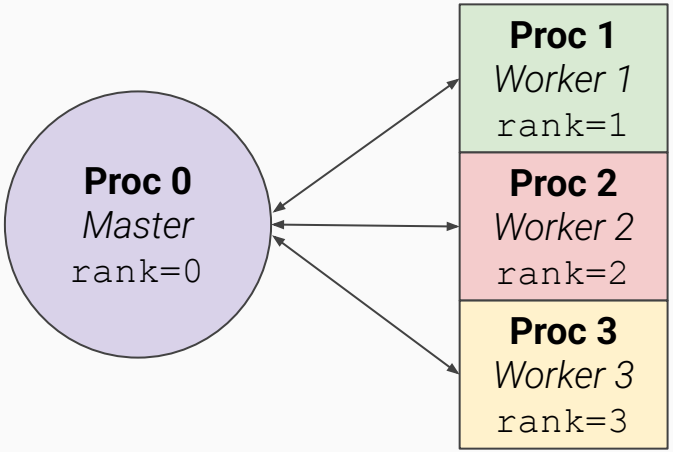
2) **Execution:** `np 3` executions of the program.

3) **Runtime**: each process has an ID, i.e a **`rank`**.

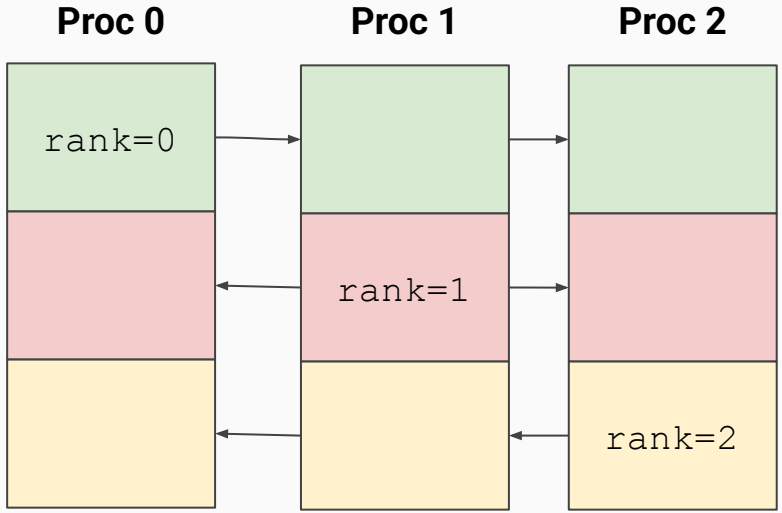
The control of program flow is via **`rank`**.

Program Flow
Some examples (not standards)

The master process is the conductor



All processes carry the (full) result



What if the result does not fit on a single process?

Communication
A communication “channel”

`MPI_COMM_WORLD`: The global communication scope between processes.

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

int main(){
   int size, rank;
   MPI_Init(NULL,NULL); // Needs to be called
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   std::vector<double> val(size,0);
   val[rank]=some_func(rank);
   MPI_Allreduce( MPI_IN_PLACE,&val[0],size,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
   MPI_Finalize(); // Needs to be called
   return 0;
}
```

You can split `MPI_COMM_WORLD` into **groups** (see `MPI_Comm_split`)

groups: Local channels of communication having their own `rank` and `size`.
See here for an excellent tutorial.

All processes within `comm` are required **not** to proceed to the next line of code until **all processes** in `comm` are at this line.

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

A parallel operation is, in general, **not synchronized**.

Communication
Point-to-Point

Send **`snd_data`** to rank **`destination`** and
write it to **`rcv_data`** at the rank **`source`**.

```
MPI_Send(void* snd_data,int count, MPI_Datatype datatype,int destination,int tag, MPI_Comm communicator)
MPI_Recv(void* rcv_data,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm communicator,MPI_Status* status)
```

`MPI_Send` and `MPI_Recv` **must be matching**, or else the program hangs!

***Note**: There are asynchronous versions of multiple functions in MPI.*

Communication
Point-to-Point Deadlocks

```
if (rank == 0){
  destination=1;
  source=1;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#1
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#2
}else if(rank == 1){
  destination=0;
  source=0;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#3
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#4
}
```

```
if (rank == 0){
  destination=1;
  source=1;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#1
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#2
}else if(rank == 1){
  destination=0;
  source=0;
  // Note the order of MPI_Send and MPI_Recv
  MPI_Send(&sendbuf,sendbuf_size,MPI_INT,destination,tag,MPI_COMM_WORLD);              //#4
  MPI_Recv(&recvbuf,recvbuf_size,MPI_INT,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); //#3
}
```

*Pair **Sends** with **Receives**!*

```
rank 0 → rank 1 #1
rank 1 ← rank 0 #3

rank 0 ← rank 1 #2
rank 1 → rank 0 #4
```

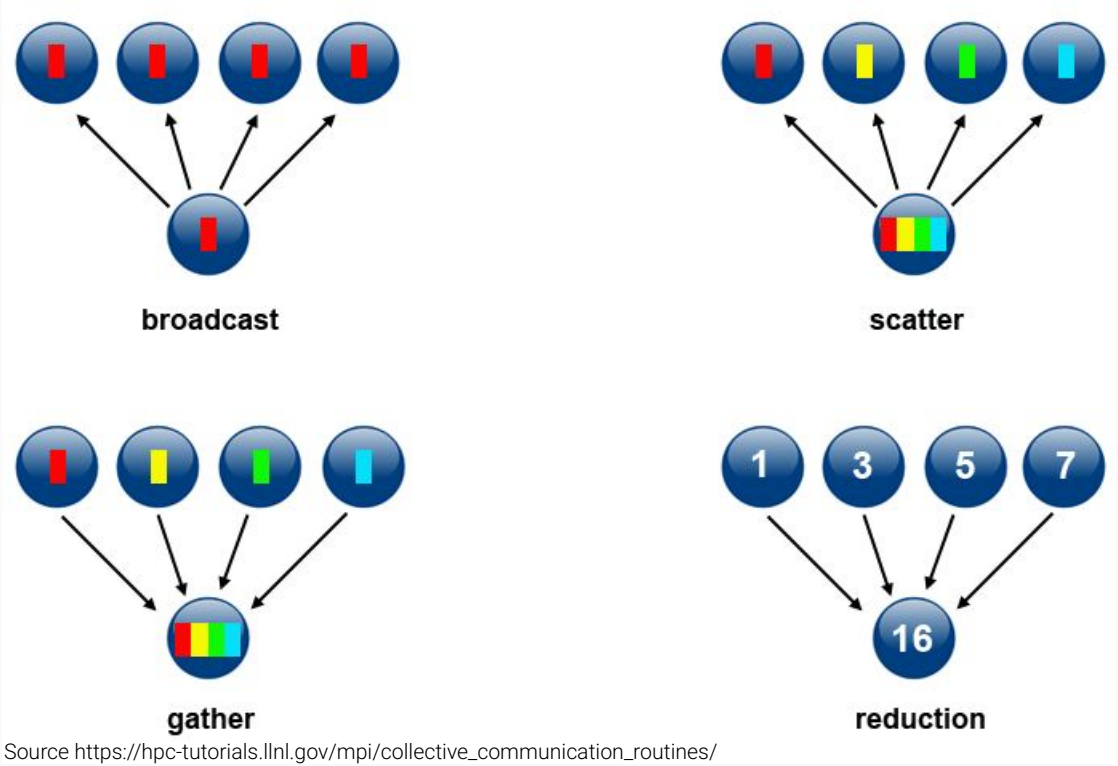
This results in

```
rank 0 → rank 1  #1
rank 1 → rank 0  #4
```

This *could* result in a **deadlock** (not recommended). Both rank 1 and rank 2 send without receiving, and thus wait forever!

Implementation-dependent, see https://web.stanford.edu/class/cme194/cgi-bin/lecture5.pdf

Communication
Collective



Source https://hpc-tutorials.llnl.gov/mpi/collective_communication_routines/

For `MPI_Allreduce` and `MPI_Allgather` the results of the base operations are broadcasted i.e., all ranks have the **full** result.

References

Many online references are available

See mpitutorial.com for practical examples and references, and rookiehpc.org for more details on both MPI and openMP.

Hybrid MPI and OpenMP Parallel Programming

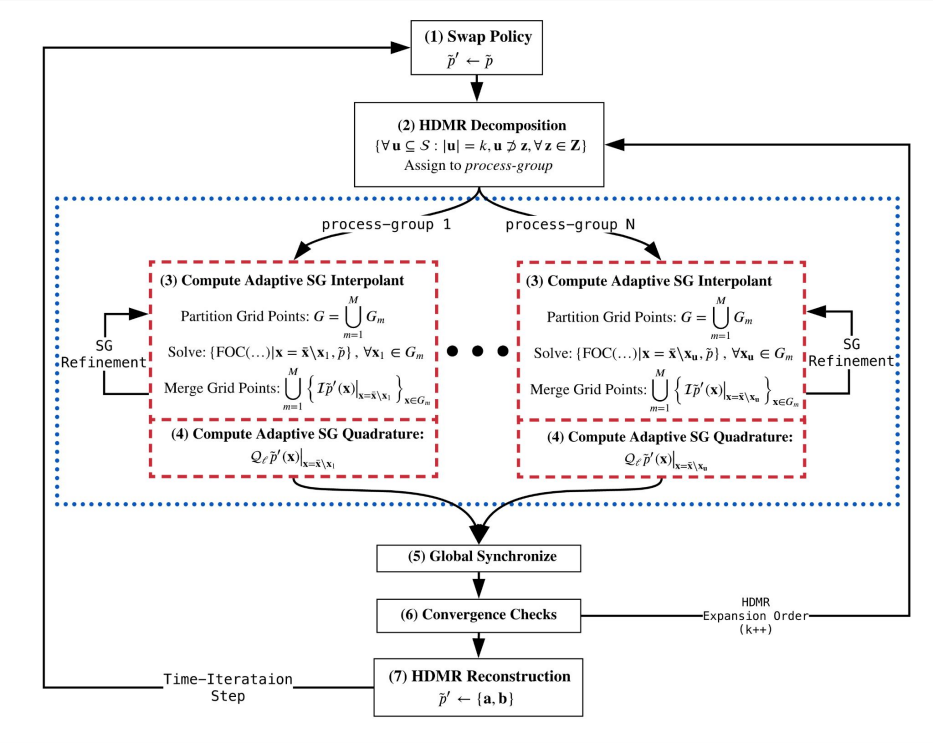
An example

MPI + OpenMP: Two layers of parallelism

(1) The primary layer uses MPI (dotted blue lines), which ideally splits independent workloads across processes.

(2) Within each workload, the secondary layer uses OpenMP (dashed red lines) to perform parallel numerical operations within the workload.

Note the need for “global synchronization” across processes.



E. A., Scheidegger S. Siam SISC, 2022.

Motivation

MPI is designed to scale and

is fundamental for HPC.