PARALLEL PROGRAMMING...

Parallel Programming: Overview

SESSION 2/3

GOAL

Programming Interface for parallel computing

- MPI (Message Passing Interface)
- OpenMP (Open Multi-Processing)
- CUDA and ROCm

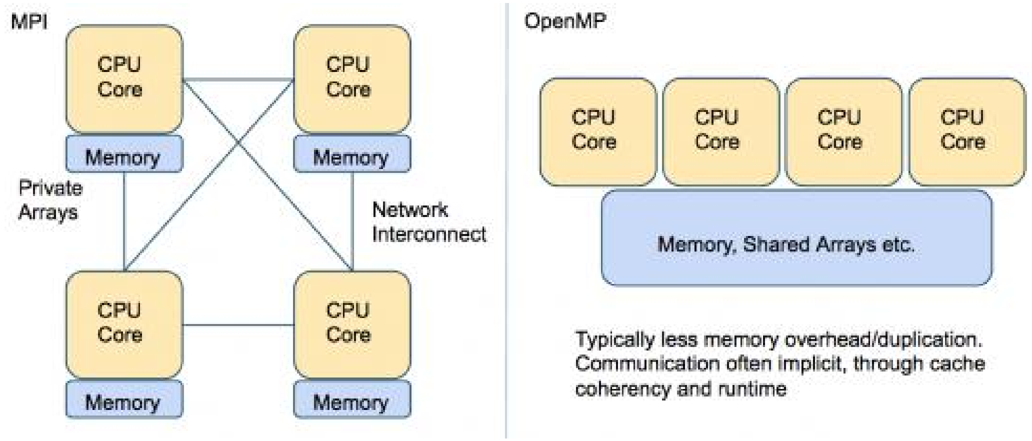
Programming interface for parallel computing

병렬 컴퓨팅을 위한 프로그래밍 인터페이스

Programming interface…

Remember

.



MPI
CPU Core
Memory
Private Arrays
Network Interconnect
OpenMP
Memory, Shared Arrays etc.
Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime

.

MPI (Message Passing Interface) is a multi-process model whose mode of communication between the processes is **explicit.**

==> communication management is the responsibility of the user.

.

OpenMP (Open Multi-Processing) is a multitasking model whose mode of communication between tasks is **implicit**

==> communications is the responsibility of the compiler.

MPI (Message Passing Interface)

MPI (Message Passing Interface)

MPI is a library of subroutines (in Fortran,C)

MPI allows the coordination of a program running as multiple processes in a distributed-memory environment.

MPI is flexible enough to also be used in a shared-memory environment.

MPI programs can be used and compiled on a wide variety of single platforms or (homogeneous or heterogeneous) clusters of computers over a network.

MPI library is standardized, should work (without further changes!) on any machine on which the MPI library is installed.

MPI: Basic Environment

```
MPI_Init(&argc, &argv)
```

Initializes MPI environment
Must be called in every MPI program
Must be first MPI call
Can be used to pass command line arguments to all

```
MPI_Finalize()
```

Terminates MPI environment
Last MPI function call

MPI: Basic Environment

```
MPI_Comm_rank(comm, &rank)
```

Returns the rank of the calling MPI process
Within the communicator, comm
MPI_COMM_WORLD is set during Init(...)
Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

Returns the total number of processes
Within the communicator, comm

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI: Point-to-Point Communication

```
MPI_Send(&buf, count, datatype, dest, tag, comm)
```

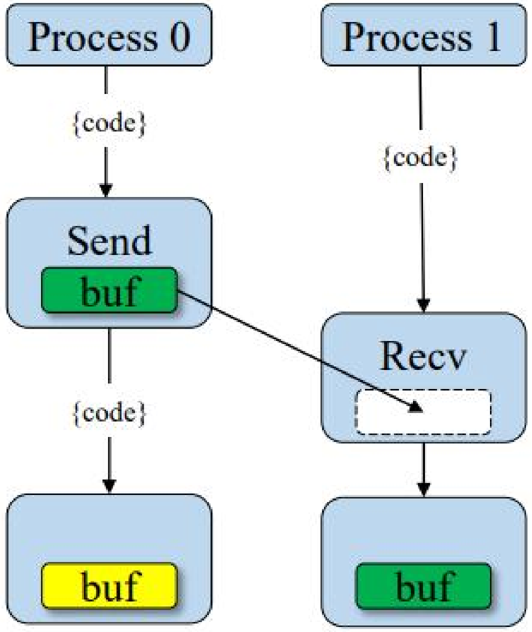
Send a message
Returns only after buffer is free for reuse (Blocking)

```
MPI_Recv(&buf, count, datatype, source, tag, comm, &status)
```

Received a message
Returns only when the data is avaible
Blocking

```
MPI_SendRecv(...)
```

Two way communication
Blocking



MPI: Point-to-Point Communication

Blocking

- Only returns after completed
  - Received: data has arrived and ready to use
  - Send: safe to reuse sent buffer
- Be aware of deadlocks
- Tip: use when possible

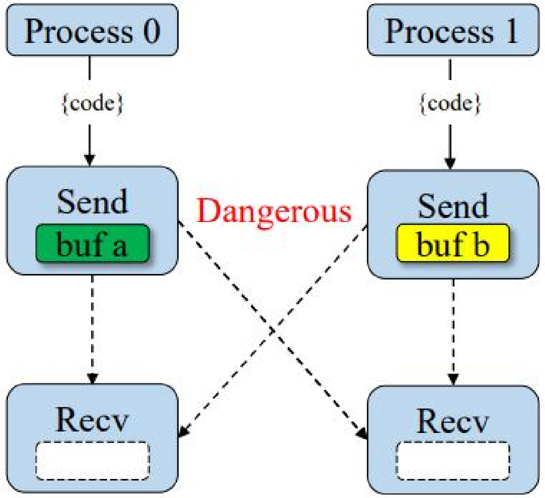
Non-Blocking

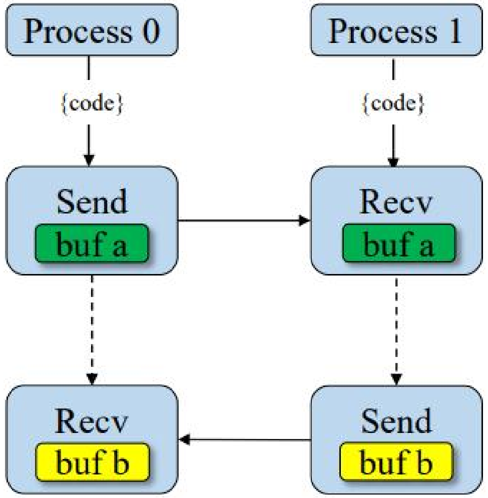
- returns immediately
  - Unsafe to modify buffers until operation is known to be complete
- Allows computation and communication to overlap
- Tip: Use only when needed

MPI: Deadlock

Blocking calls can resluts in deadlock

- One process is waiting for message that will never arrive
- Only option is to abort the interrupt/kill the code (CTRL-c)
- Might not always deadlock - depends on size of system buffer

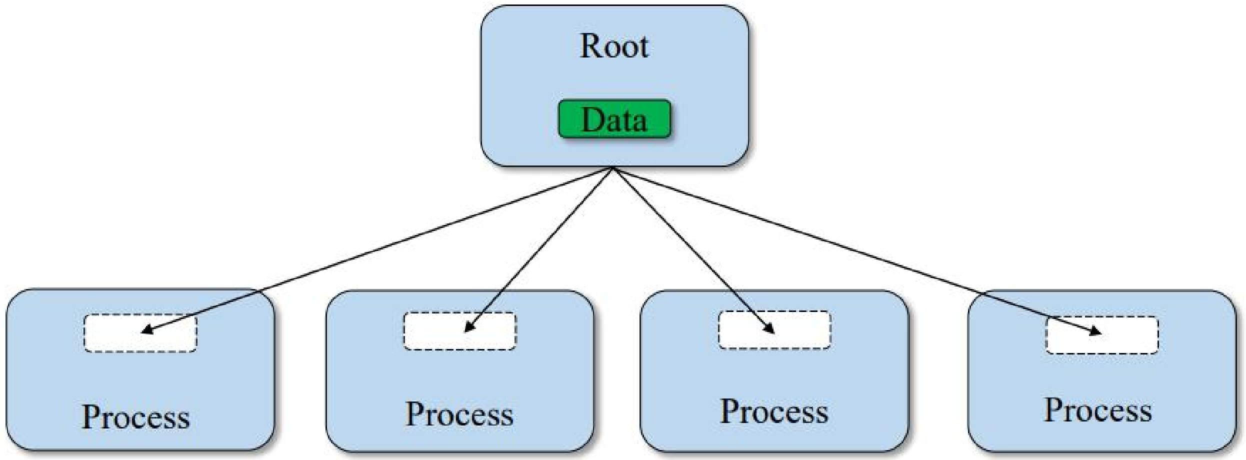




MPI: Coolective Communication (BCast)

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

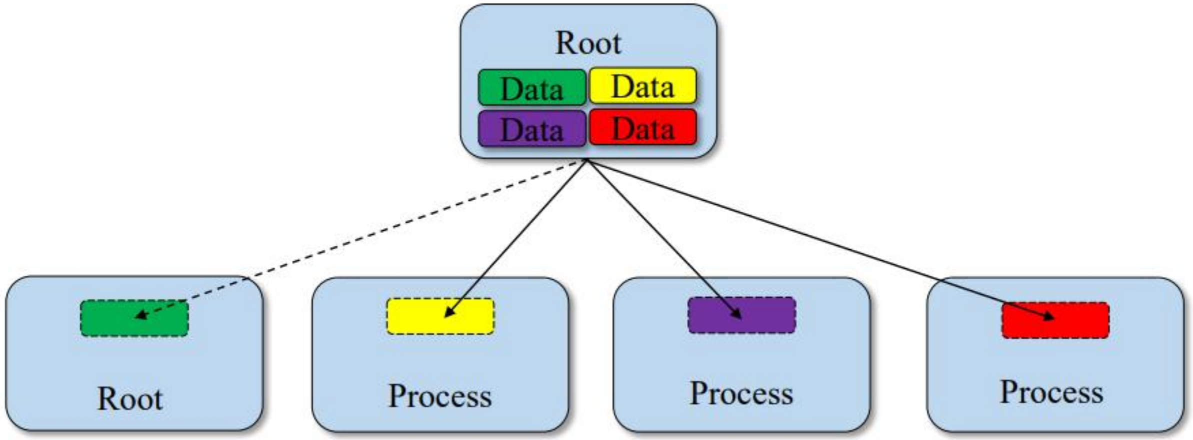
Broadcast a message from the root process to all other processes.
Useful when reading in input parameters from file.



MPI: Collective Communication (Scatter)

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

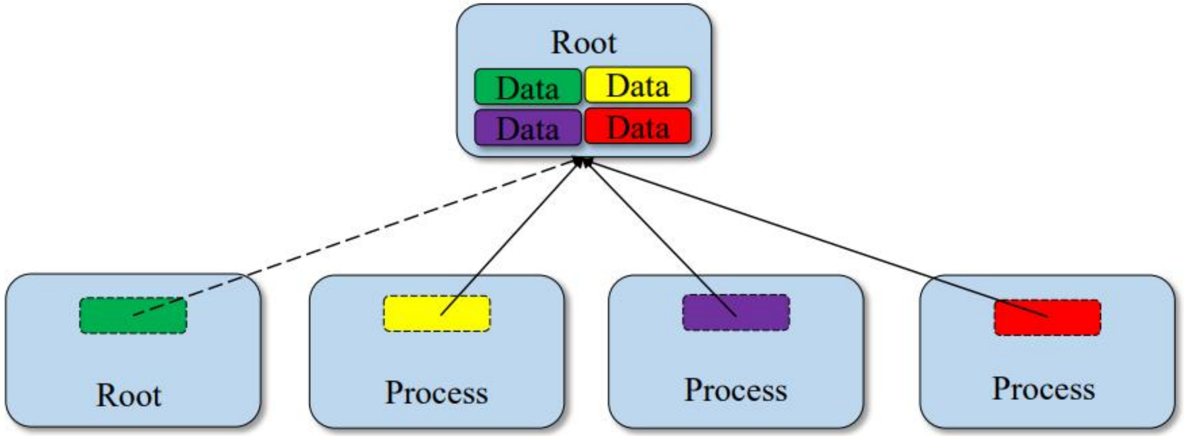
Sends individual messages from the root process to all other processes.



MPI: Collective Communication (Gather)

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

Opposite of Scatter.

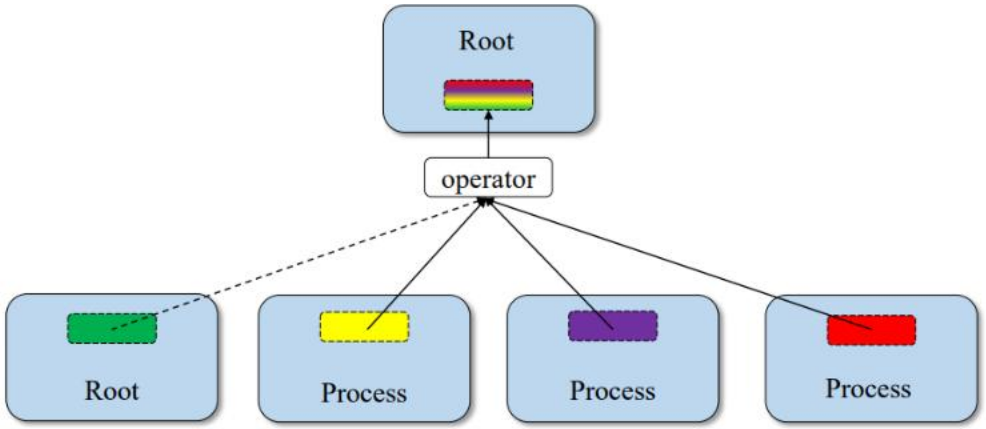


MPI: Collective Communication (Reduce)

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

Applies reduction operation on data from all processes.
Puts results on root process.

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

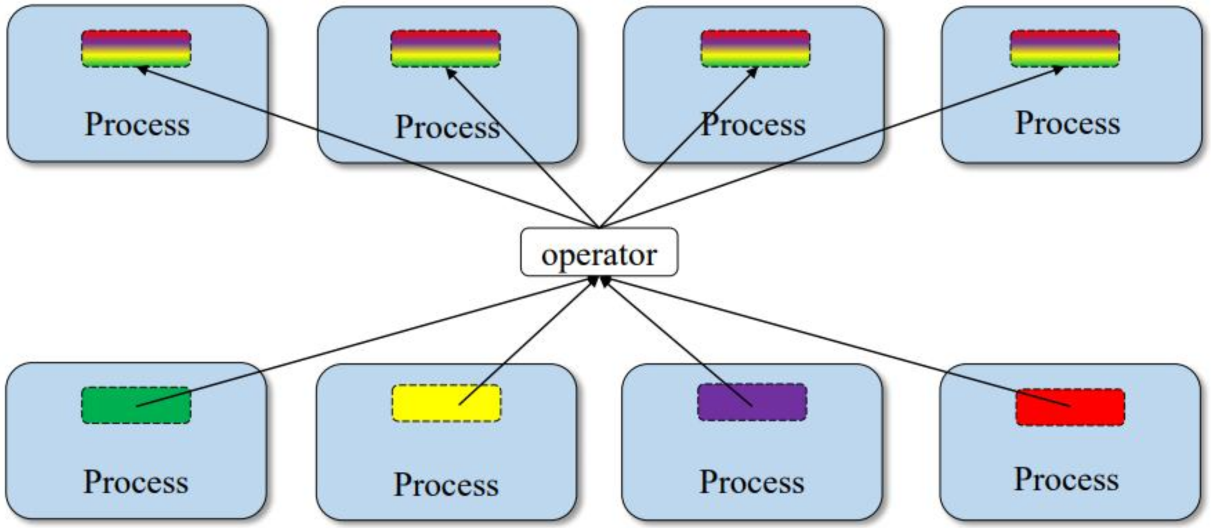


MPI: Collective Communication (Allreduce)

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

Applies reduction operation on data from all processes.
Store results on all processes.



MPI: Collective Communication (Barrier)

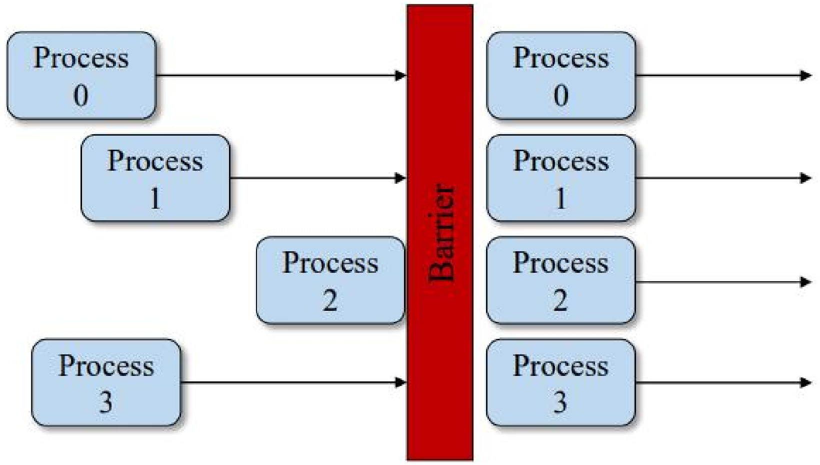
```
MPI_Barrier(comm)
```

Process synchronization (blocking).

All processes forced to wait for each other.

Use only where necessary.

Will reduce parallelism.



MPI: keywords

1 environment
- MPI Init: Initialization of the MPI environment
- MPI Comm rank: Rank of the process
- MPI Comm size: Number of processes
- MPI Finalize: Deactivation of the MPI environment
- MPI Abort: Stopping of an MPI program
- MPI Wtime: Time taking

2 Point-to-point communications
- MPI Send: Send message
- MPI Isend: Non-blocking message sending
- MPI Recv: Message received
- MPI Irecv: Non-blocking message reception
- MPI Sendrecv and MPI Sendrecv replace: Sending and receiving messages
- MPI Wait: Waiting for the end of a non-blocking communication
- MPI Wait all: Wait for the end of all non-blocking communications

3 Collective communications
- MPI Bcast: General broadcast
- MPI Scatter: Selective spread
- MPI Gather and MPI Allgather: Collecting
- MPI Alltoall: Collection and distribution
- MPI Reduce and MPI Allreduce: Reduction
- MPI Barrier: Global synchronization

4 Derived Types
- MPI Contiguous type: Contiguous types
- MPI Type vector and MPI Type create hvector: Types with a con-standing
- MPI Type indexed: Variable pitch types
- MPI Type create subarray: Sub-array types
- MPI Type create struct: H and erogenous types
- MPI Type commit: Type commit
- MPI Type get extent: Recover the extent
- MPI Type create resized: Change of scope
- MPI Type size: Size of a type
- MPI Type free: Release of a type

MPI: Keywords

5 Communicator
• MPI Comm split: Partitioning of a communicator
• MPI Dims create: Distribution of processes
• MPI Cart create: Creation of a Cart´esian topology
• MPI Cart rank: Rank of a process in the Cart´esian topology
• MPI Cart coordinates: Coordinates of a process in the Cart´esian topology
• MPI Cart shift: Rank of the neighbors in the Cart´esian topology
• MPI Comm free: Release of a communicator

6 MPI-IO
• MPI File open: Opening a file
• MPI File set view: Changing the view
• MPI File close: Closing a file

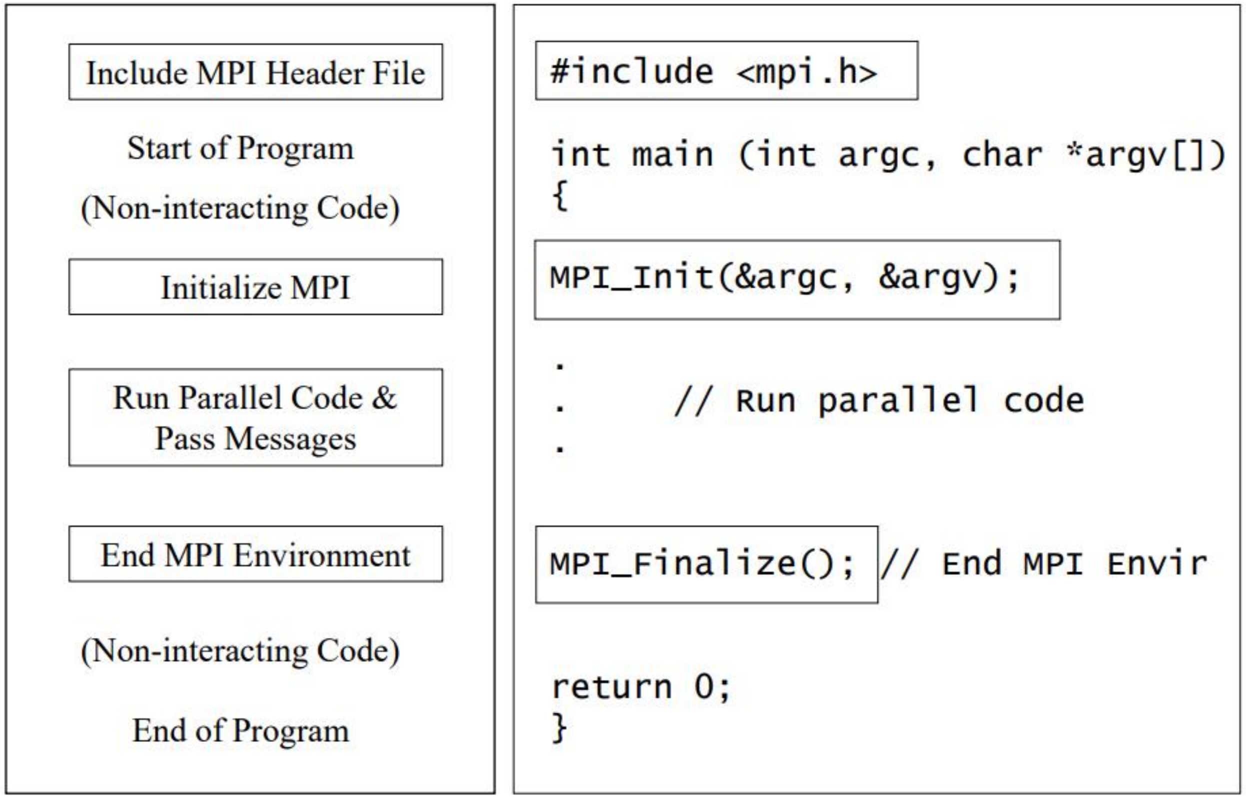
6.1 Explicit addresses
• MPI File read at: Reading
• MPI File read at all: Collective reading
• MPI File write at: Writing

6.2 Individual pointers
• MPI File read: Reading
• MPI File read all: collective reading
• MPI File write: Writing
• MPI File write all: collective writing
• MPI File seek: Pointer positioning

6.3 Shared pointers
• MPI File read shared: Read
• MPI File read ordered: Collective reading
• MPI File seek shared: Pointer positioning

7.0 Symbolic constants
• MPI COMM WORLD, MPI SUCCESS
• MPI STATUS IGNORE, MPI PROC NULL
• MPI INTEGER, MPI REAL, MPI DOUBLE PRECISION
• MPI ORDER FORTRAN, MPI ORDER C
• MPI MODE CREATE,MPI MODE RONLY,MPI MODE WRONLY

MPI: Program Basics



Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code &
Pass Messages

End MPI Environment

(Non-interacting Code)

End of Program

```
#include <mpi.h>

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

MPI_Init(&argc, &argv);

.
.      // Run parallel code
.


MPI_Finalize(); // End MPI Envir


return 0;
}
```

MPI: Example

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


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

  int rank, size;

  MPI_Init (&argc, &argv);  //initialize MPI library

  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

  //do something
  printf ("Hello World from rank %d\n", rank);
  if (rank == 0) printf("MPI World size = %d processes\n", size);

  MPI_Finalize(); //MPI cleanup

  return 0;
}
```

- 4 processes

```
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 2
Hello World from rank 1
```

- Code ran on each process independently
- MPI Processes have *private* variables
- Processes can be on completely different machines

COMPILING an MPI Program

- **Compiling a program** for MPI is almost just like compiling a regular C or C++ program
  - The C compiler is **mpicc** and the **C++** compiler is **mpic++**.
    - For example, to compile **MyProg.c** you would use a command like
      - **mpicc** - O2 -o **MyProg MyProg . c**

OpenMP (Open Multi-Processing)

OpenMP (Open Multi-Processing)

OpenMP is a programming interface for parallel computing on shared memory architecture.

- **It allows you to manage:**

the creation of light processes
the sharing of work between these lightweight processes
synchronizations (explicit or implicit) between all light processes
the status of the variables (private or shared).

OpenMP (Open Multi-Processing)

- **Shared memory model**
  - ➢ Threads communicate by accessing shared variables

- **The sharing is defined syntactically**
  - ➢ Any variable that is seen by two or more threads is shared
  - ➢ Any variable that is seen by one thread only is private

- **Race conditions possible**
  - ➢ Use synchronization to protect from conflicts
  - ➢ Change how data is stored to minimize the synchronization

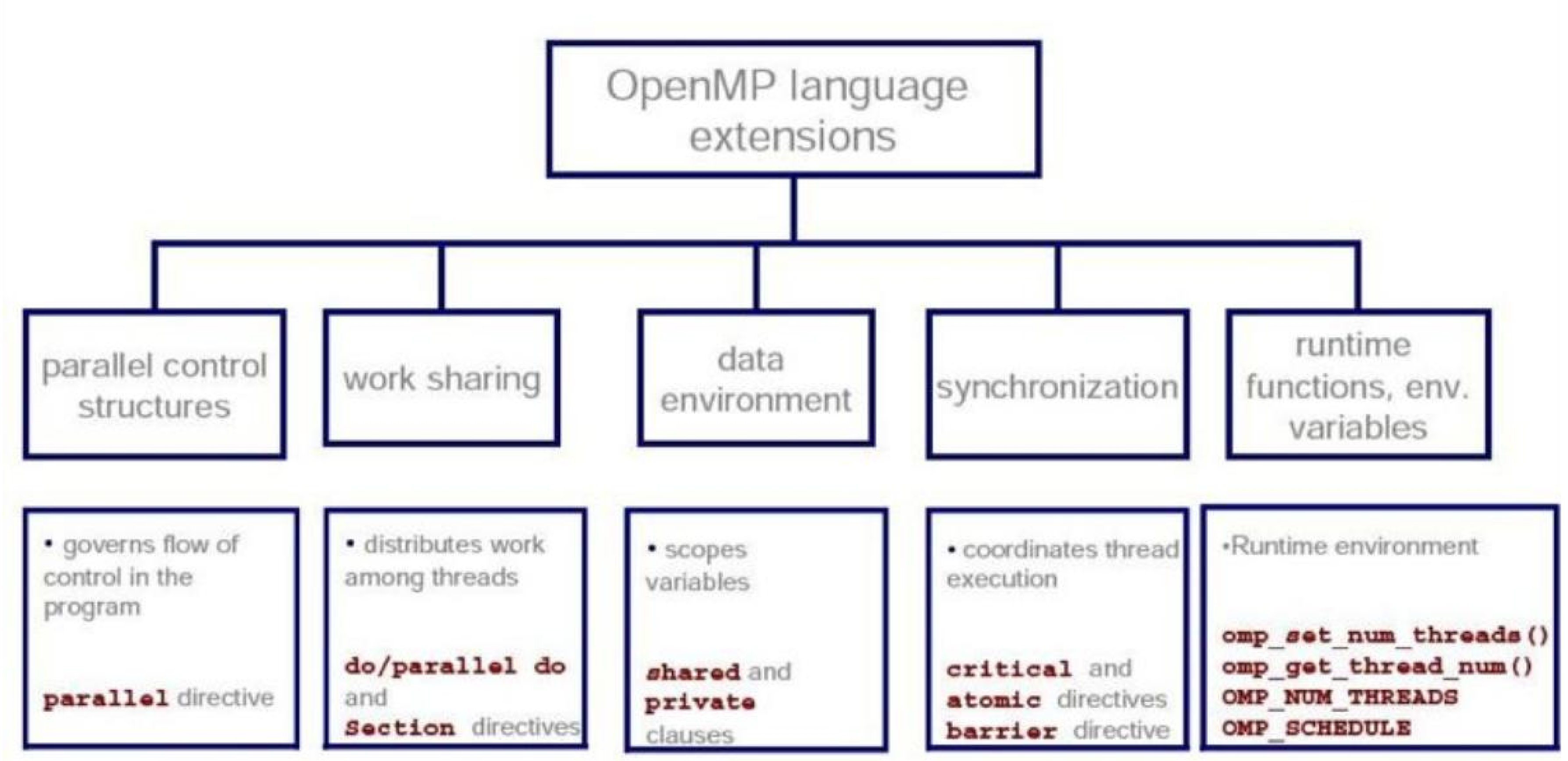
OpenMP (Open Multi-Processing)

- **Multicore CPUs are everywhere:**
  - Servers with over 100 cores today and more
  - Even smartphone CPUs have 8 cores

- **Multithreading, natural programming model**
  - All processors share the same memory
  - Threads in a process see same address space
  - Many shared-memory algorithms developed

- **Multithreading is hard**
  - Lots of expertise necessary
  - Deadlocks and race conditions
  - Non-deterministic behavior makes it hard to debug

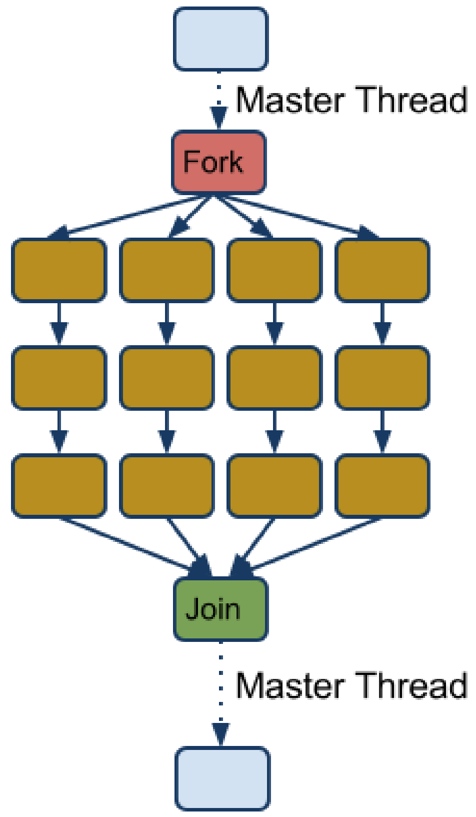
OpenMP: Architecture


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
• governs flow of control in the program
parallel directive
• distributes work among threads
do/parallel do and Section directives
• scopes variables
shared and private clauses
• coordinates thread execution
critical and atomic directives barrier directive
•Runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

OpenMP: Terminology and behavior

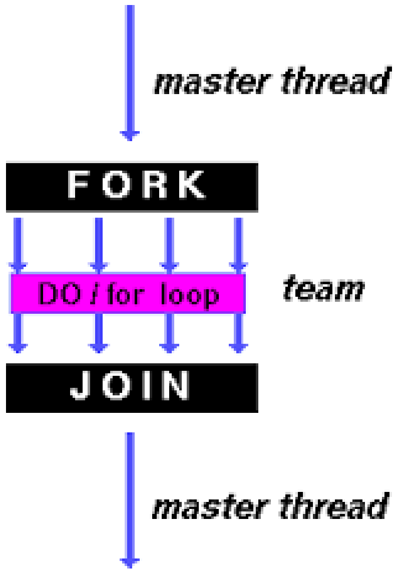
- **OpenMP Team** = Master + Worker

- **Parallel Region** is a block of code executed by all threads simultaneously (has implicit barrier)
  - The master thread always has thread id 0
  - Parallel regions can be nested
  - If clause can be used to guard the parallel region



OpenMP: Terminology and behavior

- A **Work-Sharing construct** divides the execution of the enclosed code region among the members of the team. (Loop, Section etc.)



OpenMP: Preprocessor Directives

Preprocessor directives tell the compiler what to do.
Always start with #
You have already seen one:

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

OpenMP directives tell the compiler to add machine code for parallel execution of the following block.

```
#pragma omp parallel
```

OpenMP: Some OpenMP Subroutines

```
int omp_get_max_threads()
```

Return max possible

```
int omp_get_num_threads()
```

Returns number of treads in current team \\

```
int omp_get_thread_num()
```

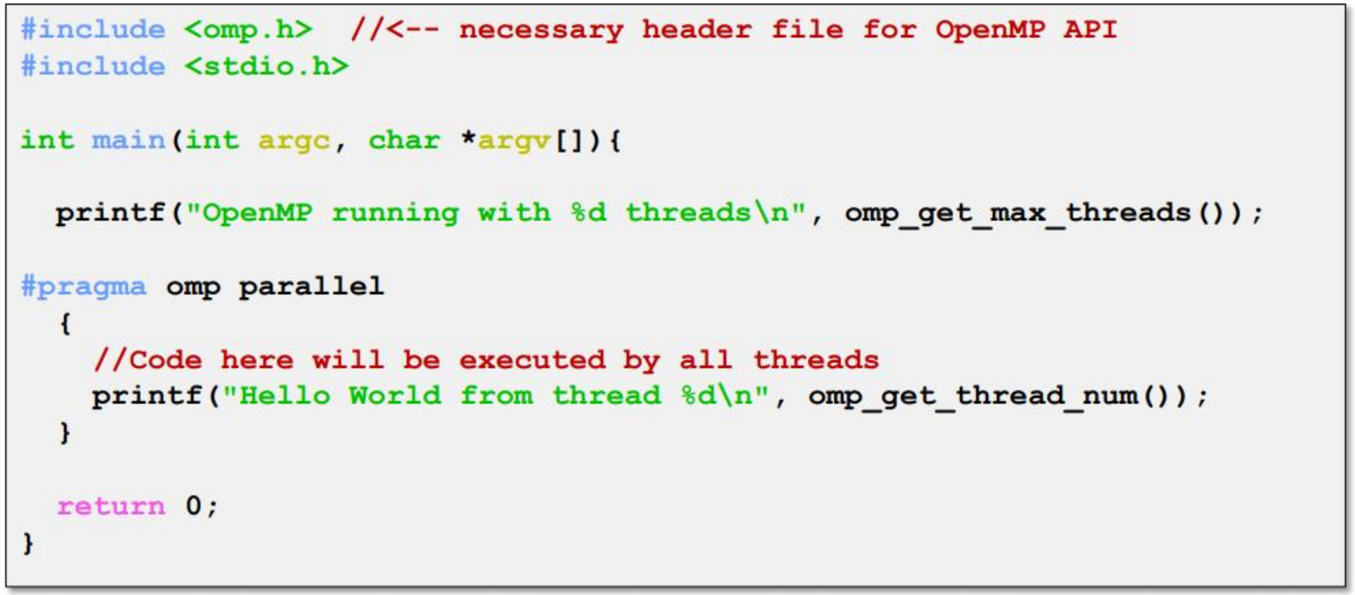
Returns tread id of calling thread
Between () and omp_get_num_threads-1

OpenMP: Process vs. Thread

- MPI = Process, OpenMP = Thread
- Program start with a single process
- Process have their own (private) memory space
- A process can create one or more threads

- Threads created by a process share its memory space
  - Read and write to same memory adresses
  - Share same process ids and file descriptors
- Each thread has a unique counter and stack pointer
  - A tread can have private starage on the stack

OpenMP: Example



```c
#include <omp.h>  //<-- necessary header file for OpenMP API
#include <stdio.h>

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

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP: Constructs

Parallel region

- Thread creates team, and becvomes master (id 0)
- All threads run code after
- Barrier at end of parallel section

```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     num_threads (integer)


   structured_block
```

(not a complete list)

OpenMP Parallel Clauses

```
#pragma omp parallel if (scalar_expression)
```

Only execute in parallel.
Otherwise serial.

```
#pragma omp parallel private (list)
```

Data local to thread.
Value are not guaranted to be defined on exit (even if defined before)
No storage associated with original object
Use firstprivate and/or lastprivate clause to override

```
#pragma omp parallel firstprivate (list)
```

Variables in list are private.
Initialized with the value the variable had before entering the construct.

```
#pragma omp parallel for lastprivate (list)
```

Only in for loops
Variables in list are private.
The thread that executes the sequentially last iteration updates the value of the variables in the list.

OpenMP Parallel Clause 3

```
#pragma omp shared (list)
```

Data is accessible by all threads in team.
All threads access samme address space.

Improperly scoped variables are big source of OMP bugs
- Shared when should be private
- Race condition

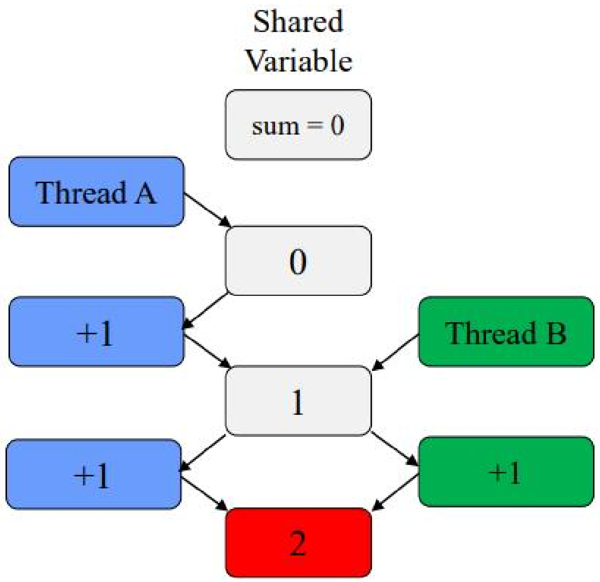
```
#pragma omp default (shared | none)
```

Tip: Safest is to use default (none) and declare by hand.

OpenMP: Caution Race Condition

When multible threads simultaneously read/write
Multiple OMP solutions
- Reduction
- Atomic
- Critical

```
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```



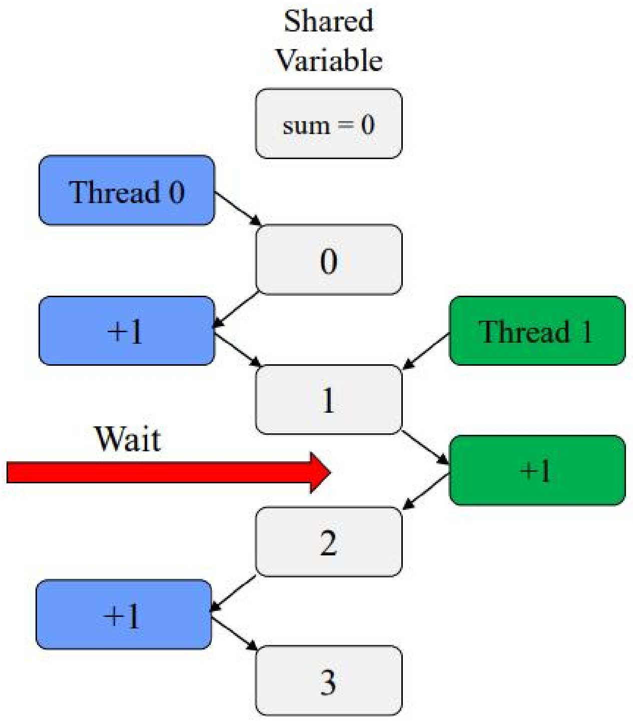
Should be 3!

OpenMP: Critical Section

One solution: use critical
Only one tread at a time can execute
 a critical section

```
#pragma omp critical
   {
     sum += i;
   }
```

Downside ?
SLOOOOWWW
Overhead and serialization

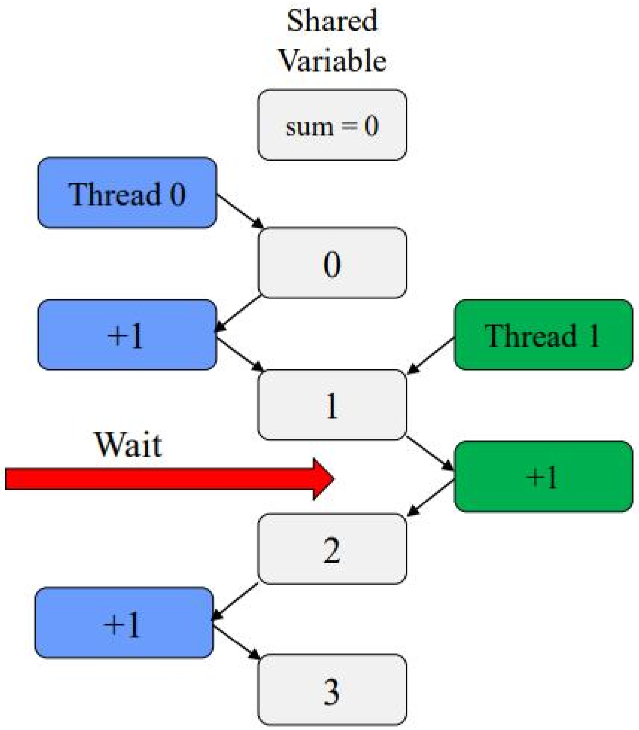


OpenMP: Atomic

Atomics like "mini" critical
Only one line
Certain limitations

```
#pragma omp atomic
    sum += i;
```

Hardware controlled
Less overhead tha critical



OpenMP Reduction

```
#pragma omp reduction (operator:variable)
```

Avoids race condition
Reduce variable must be shared
Makes variable private, then performs operator at end of loop
operator cannot be overloaded (c++)

- One of: +,*,-,/ (and &,^,|,&&,||)
- OpenMP 3.1: added min and max for c/c++

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

int main() {

  int i;
  const int N = 1000;
  int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
  for (i=0; i<N; i++) {
    sum += i;
  }

  printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

OpenMP: Scheduling omp for

How does a loop get split up ?

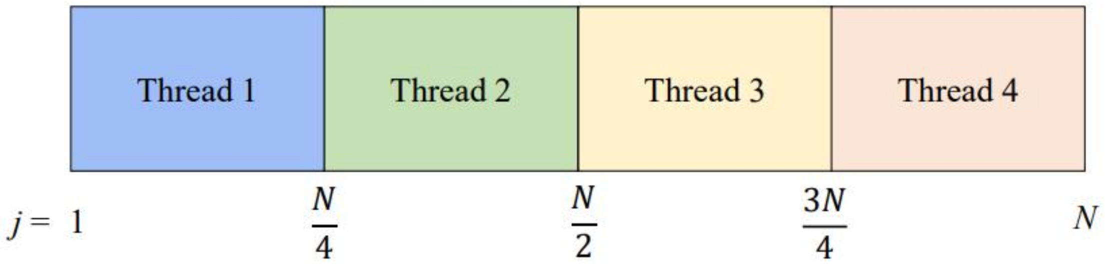
In MPI, we have to do it manually

If you do not tell what to do, the compiler decides

Usually compiler chooses "static" - chunks of N/p

```
#pragma omp parallel for default(shared) private(j)
  for (j=0; j<N; j++) {
        ... // some work here
}
```

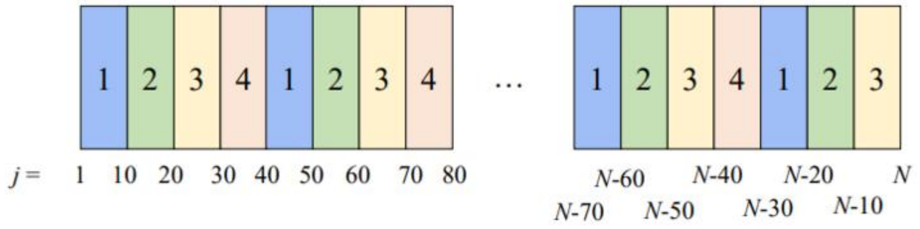
Unspecified schedule



OpenMP: Static Scheduling

You can tell the compiler what size chunks to take.

```
#pragma omp parallel for default(shared) private(j) schedule(static,10)
  for (j=0; j<N; j++) {
           ... // some work here
}
```

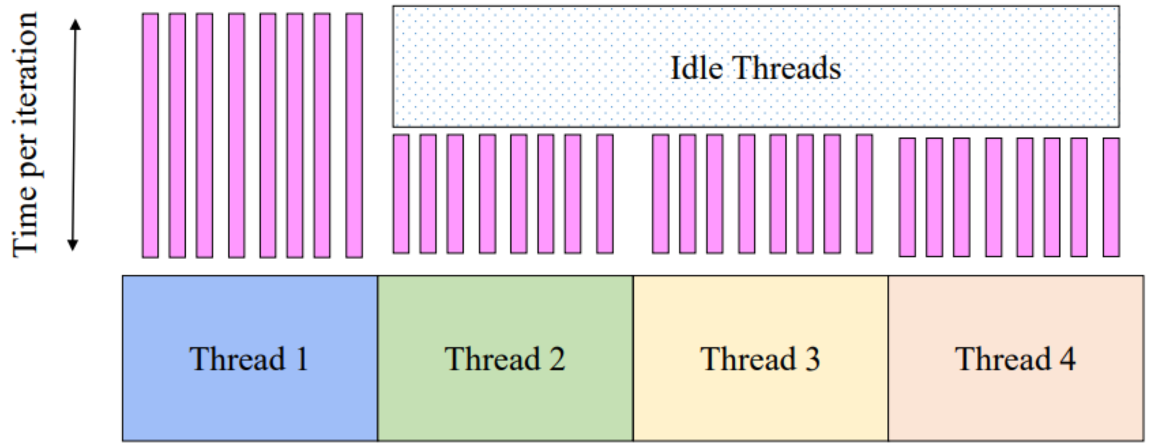


Keeps assigning chunks until done.
Chunk size that is not a multiple of the loop will results in thread with uneven numbers.

OpenMP: Problem with Static Scheduling

What happens if loop iterations do not take the same amount of time ?

Load imbalance



OpenMP: Dynamic Scheduling

Chunks are assigned on the fly, as threads become available
When a thread finishes on chunk, it is assigned another

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic,10)
  for (j=0; j<N; j++) {
          ... // some work here
}
```

Caveat Emptor: higer overhead than static!

OpenMP omp for Scheduling Recap

```
#pragma omp parallel for schedule(type [,size])
```

Scheduling types

Static

-Chucks of specified size assigned round-robin

Dynamic

-Chunks of specified size are assigned when thread finishes previous chunk

Guided

-Like dynamic, but chunks are exponentially decreasing

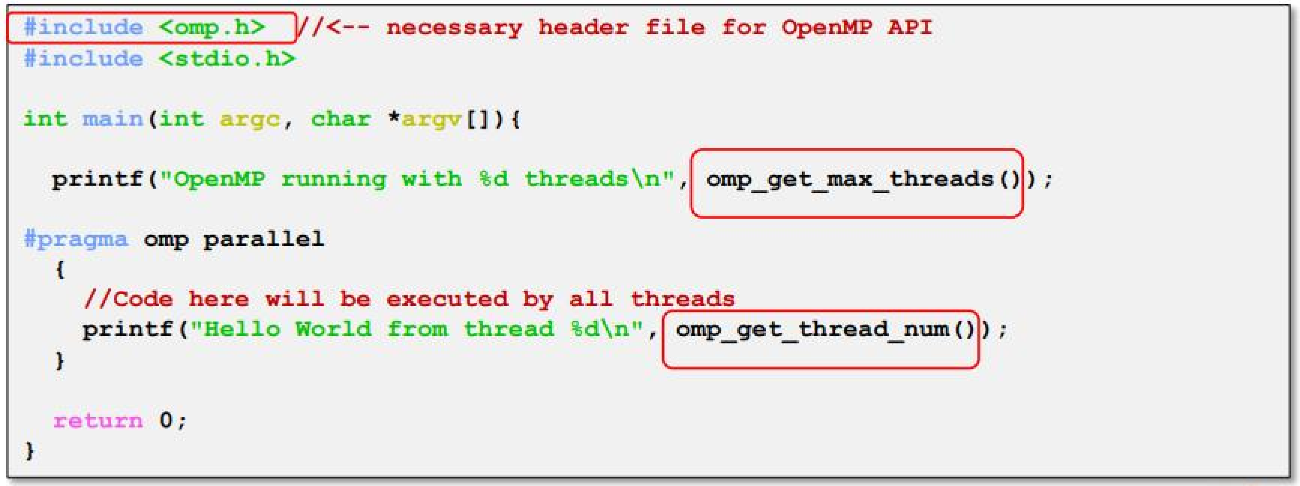
-Chunk will not be smaller than specified size

Runtime

-Type and chunk determined at runtime via environment variables

OpenMP: API

- API for library calls that perform useful functions
- Must include "omp:h"
- Will not compile without OpenMP compiler support



```
#include <omp.h>  //<-- necessary header file for OpenMP API
#include <stdio.h>

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

  printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
  {
    //Code here will be executed by all threads
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }

  return 0;
}
```

OpenMP: API

```
void omp_set_num_threads(int num_threads)
```

Sets number of treads used in next parallel section
Overrides OMP_NUM_THREADS environment variable
Positive integer

```
int omp_get_max_threads()
```

Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_num_threads()
```

Returns number of threads currently in team

```
int omp_get_thread_num()
```

Returns thread id of calling thread
Between 0 and omp_get_num_threads-1

```
double omp_get_wtime()
```

Returns number of secons since some poin
Use inpais time=(t2-t1)

OpenMP: Performance Tips

- Avoid serialization!
- Avoid using `#pragma omp parallel for` before each loop
  - Can have significant overhead
    - Thread creation and scheduling is NOT free!!
  - Try for broader parallelism
    - One `#pragma omp parallel`, multiple `#pragma omp for`
  - Always try to parallelize the outer most loop
- Use reduction whenever possible
- Minimize I/O
- Minimize `critical`
  - Use `atomic` instead of `critical` where possible

Hybrid OpenMP-MPI

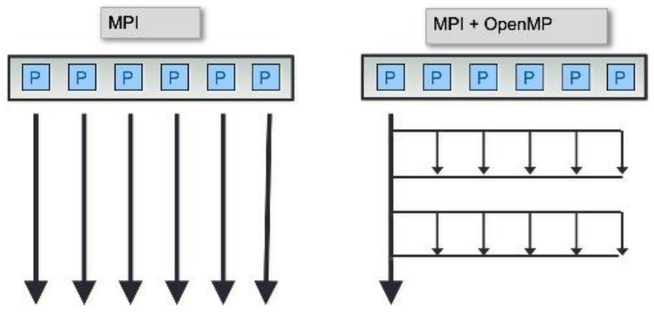
Hybrid MPI and OpenMP

Hybrid application programs using **MPI + OpenMP** are now common place on large HPC systems.

There are basically two main motivations:

1. Reduced memory footprint, both in the application and in the MPI library (eg communication buffers).
2. Improved performance, especially at high core counts where pure MPI scalability runs out.

A common hybrid approach



- From dequential code, alongside MPI first, then try adding OpenMP
- From MPI code, add OpenMP
- From OpenMP code, treat as serial code
- The simplest and least error-prone method is to use MPI outside the parallel region and allow only the master thread to communicate between MPI tasks.
- Could use MPI in parallel region with thread-safe MPI.

Hybrid MPI and OpenMP

- Two-level Parallelization
  - Mimics hardware layout of cluster
  - Only place this really make sense
  - MPI between nodes
  - OpenMP within shared-memory nodes

- Why ?
  - Saves memory by not duplicating data
  - Minimize interconnect communication by only having 1 MPI process per node

- Careful of MPI calls within OpenMP block
- Safest to do MPI calls outside (but not required)

Obviously requires some thought!

Hybrid Programming

In hybrid programming each process can have multiple threads executing simultaneously
All threads within a process share all MPI objects Communicators, requests, etc.

MPI defines 4 levels of thread safety:

- **MPI_THREAD_SINGLE** : One thread exists in program
- **MPI_THREAD_FUNNELED** : Multithreaded but only the master thread can make MPI calls Master is one that calls MPI_Init_thread()
- **MPI_THREAD_SERIALIZED:** Multithreaded, but only one thread can make MPI calls at a time
- **MPI_THREAD_MULTIPLE:** Multithreaded and any thread can make MPI calls at any time. Use MPI_Init_thread instead of MPI_Init if more than single thread

Hybrid Programming

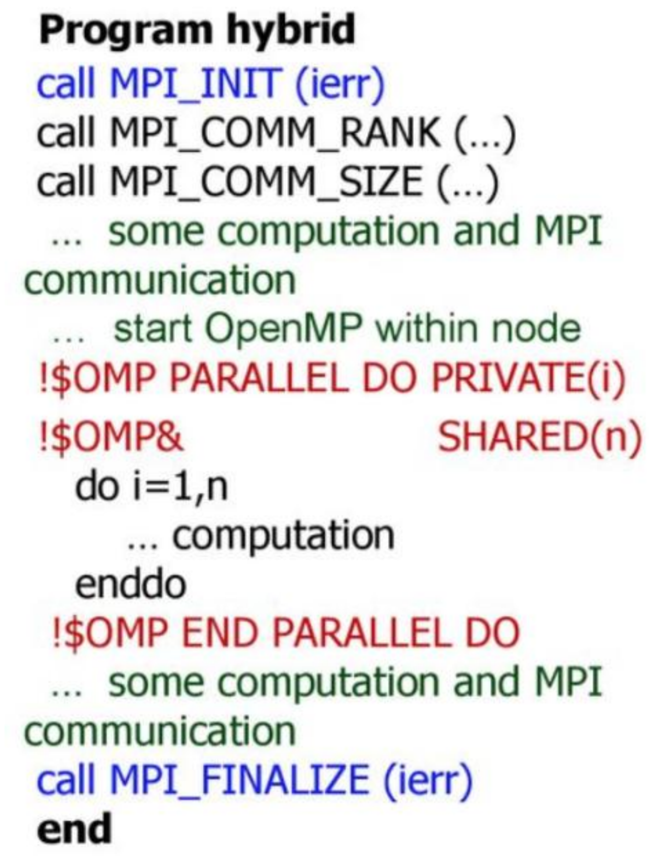
Safest (easiest) to use MPI_THREAD_FUNNLED

- Fits nicely with most OpenMP models
  - ➢ - Expensive loops parallelized with OpenMP
  - ➢ -Communication and MPI calls between loops
- Eliminates need for true “thread-safe” MPI

- Parallel scaling efficiency may be limited (Amdahl’s law) by MPI_THREAD_FUNNLED approach

- Moving to MPI_THREAD_MULTIPLE does come at a performance price (and programming challenge)

Hybrid Programming Example



```
Program hybrid
call MPI_INIT (ierr)
call MPI_COMM_RANK (...)
call MPI_COMM_SIZE (...)
 ... some computation and MPI
communication
 ... start OpenMP within node
!$OMP PARALLEL DO PRIVATE(i)
!$OMP&                 SHARED(n)
   do i=1,n
      ... computation
   enddo
 !$OMP END PARALLEL DO
 ... some computation and MPI
communication
call MPI_FINALIZE (ierr)
end
```

Start with MPI Initialization

Create OMP parallel regions with MPI task (Process):

- Sereal Regionds are the master head or MPI task .

- MPI rank is know to all thread.

Call MPI library in serial and parallel regionds.

Finalize MPI

Hybrid Programming Example

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}dx = \pi$$

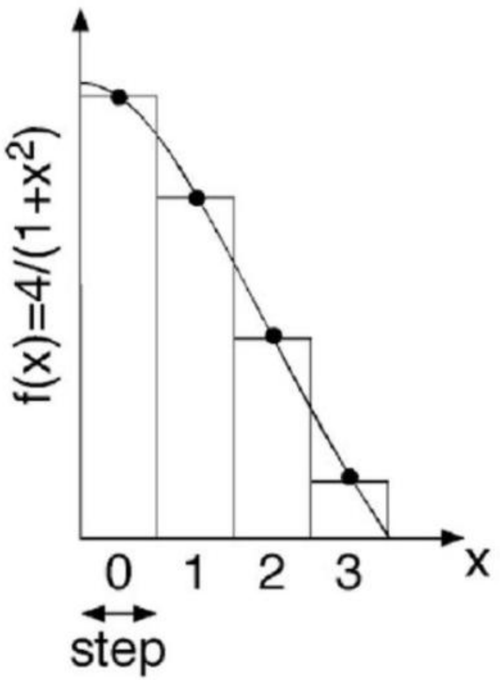
- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta \ (i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



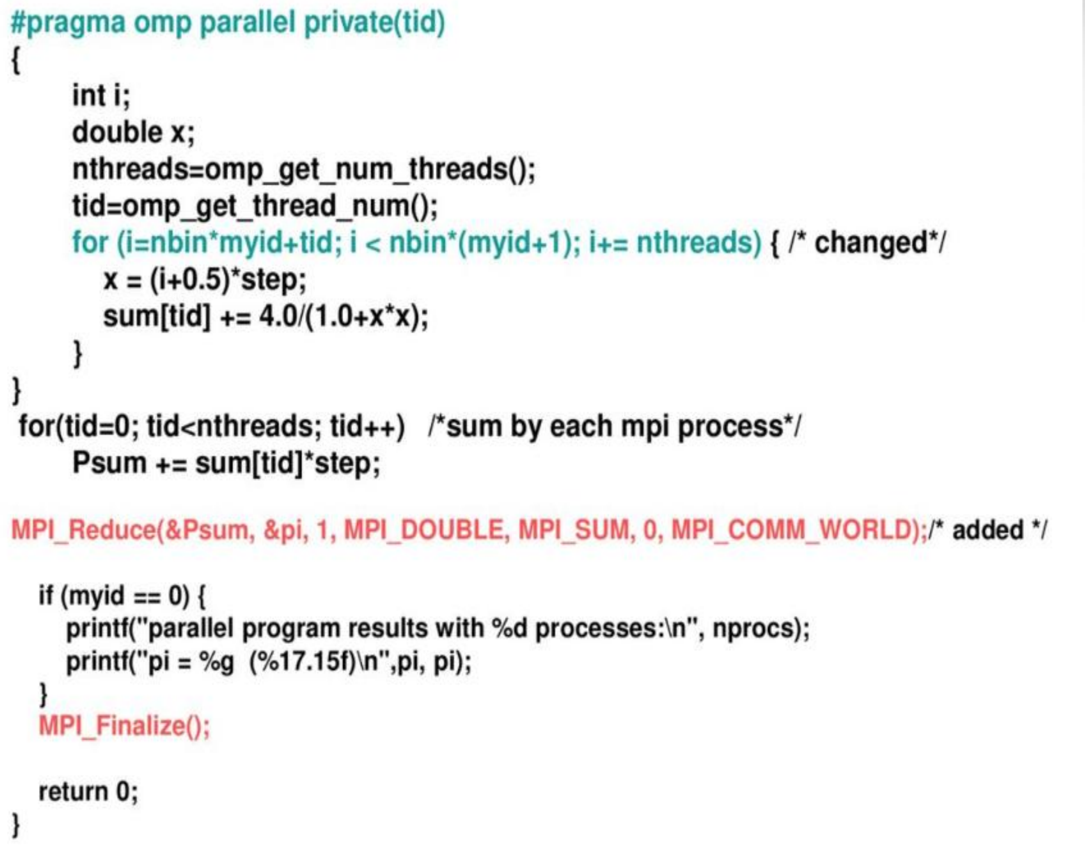
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>              /* MPI header file */
#include <omp.h>              /* OpenMP header file */
#define NUM_STEPS 100000000
#define MAX_THREADS 4

int main(int argc, char *argv[]) {
   int nprocs, myid;
   int tid, nthreads, nbin;
   double start_time, end_time;
   double pi, Psum=0.0, sum[MAX_THREADS]={0.0};
   double step = 1.0/(double) NUM_STEPS;

   /* initialize for MPI */
  MPI_Init(&argc, &argv);     /* starts MPI */
   /* get number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   /* get this process's number (ranges from 0 to nprocs - 1) */
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  nbin= NUM_STEPS/nprocs;
```

Hybrid Programming Example

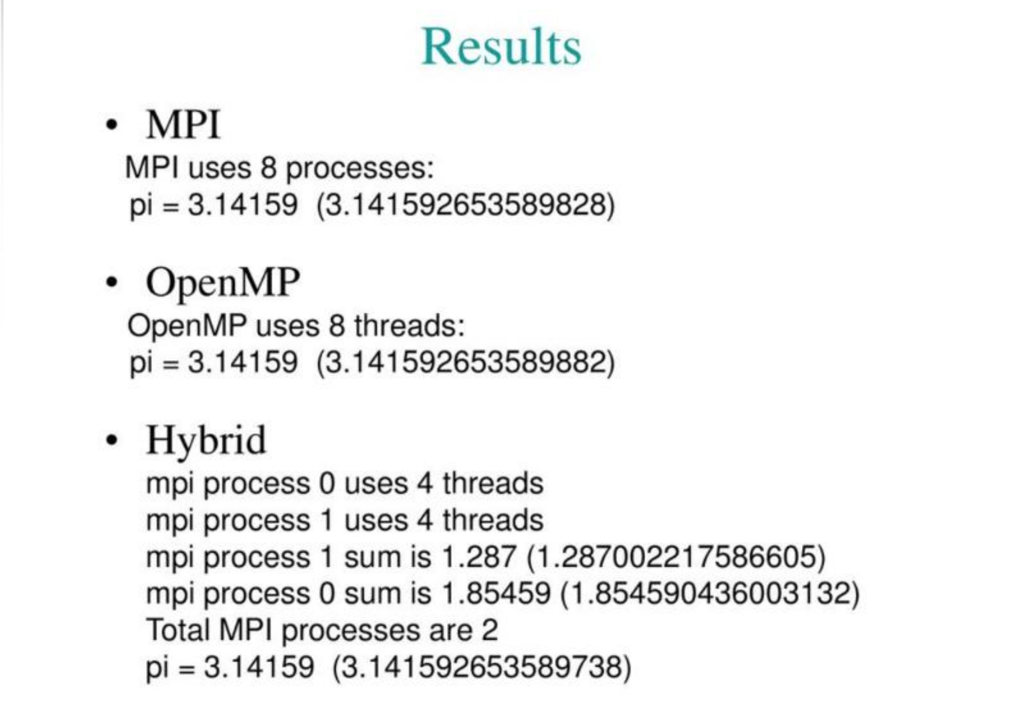


```
#pragma omp parallel private(tid)
{
        int i;
        double x;
        nthreads=omp_get_num_threads();
        tid=omp_get_thread_num();
        for (i=nbin*myid+tid; i < nbin*(myid+1); i+= nthreads) { /* changed*/
           x = (i+0.5)*step;
           sum[tid] += 4.0/(1.0+x*x);
        }
}
 for(tid=0; tid<nthreads; tid++)   /*sum by each mpi process*/
        Psum += sum[tid]*step;

MPI_Reduce(&Psum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

   if (myid == 0) {
      printf("parallel program results with %d processes:\n", nprocs);
      printf("pi = %g  (%17.15f)\n",pi, pi);
   }
   MPI_Finalize();

   return 0;
}
```



Results

- MPI

```
MPI uses 8 processes:
pi = 3.14159  (3.141592653589828)
```

- OpenMP

```
OpenMP uses 8 threads:
pi = 3.14159  (3.141592653589882)
```

- Hybrid

```
mpi process 0 uses 4 threads
mpi process 1 uses 4 threads
mpi process 1 sum is 1.287 (1.287002217586605)
mpi process 0 sum is 1.85459 (1.854590436003132)
Total MPI processes are 2
pi = 3.14159  (3.141592653589738)
```

Thank you for your attention !