

Supercomputing: Hybrid Parallel Programming Models – OpenMP/MPI

Marcelo Ponce

March 2021

Department of Computer and Mathematical Sciences - UTSC

Today's lecture

On this lecture we will discuss the following topics:

- Brief introduction to Supercomputing
- Supercomputer Architectures
- Quick overview of shared-memory programming
- A brief introduction to distributed memory programming
- Hybrid Parallel Programming
- Summary/Overview

Today's lecture

On this lecture we will discuss the following topics:

- Brief introduction to Supercomputing
- Supercomputer Architectures
- Quick overview of shared-memory programming
- A brief introduction to distributed memory programming
- Hybrid Parallel Programming
- Summary/Overview

"Advanced Parallel Scientific Computing"

Explore the use and examples of parallel computing in scientific research applications.

- HPC tools (optimization and performance tools)
- Advanced MPI, MPI-file IO.
- Hybrid MPI-openMP implementations
- CUDA/OpenACC/OpenCL
- Scientific Applications: Smoothed Particle Hydrodynamics, N-body simulations Molecular Dynamics Computational Fluid Dynamics

Prerequisites

- Knowledge of compiled languages, such as C/C++ and/or Fortran, and scientific programming applications.
- Experience editing and compiling code in a Linux environment.
- Previous knowledge/experience of parallel programming, including shared memory (openMP), distribute MPI, and heterogeneous architectures (openACC, GPU, etc...) is desirable.
- Numerical Analysis, Analysis of Algorithms

Supercomputing

Supercomputing

Supercomputing, a.k.a. High Performance Computing, is leveraging larger and/or multiple computers to solve computations in parallel.

What does it involve?

- hardware – pipelining, instruction sets, multi-processors, inter-connects
- algorithms – concurrency, efficiency, communications
- software – parallel approaches, compilers, optimization, libraries

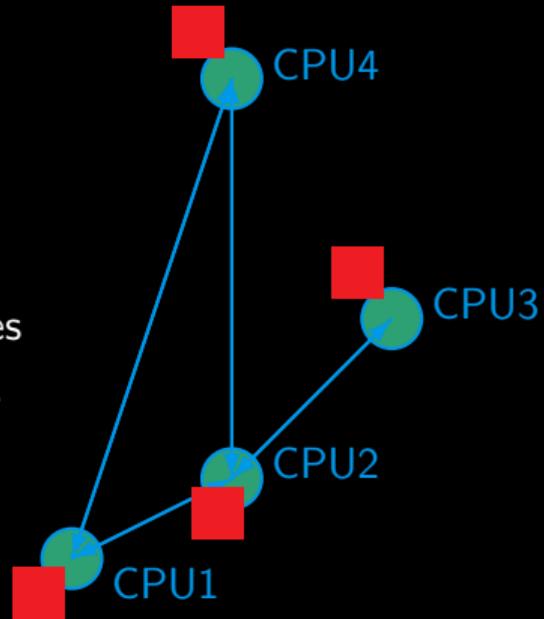
When do I need Supercomputing/HPC?

- My problem takes too long: more/faster computation
- My problem is too big: more memory
- My data is too big: more storage

Supercomputers Architectures

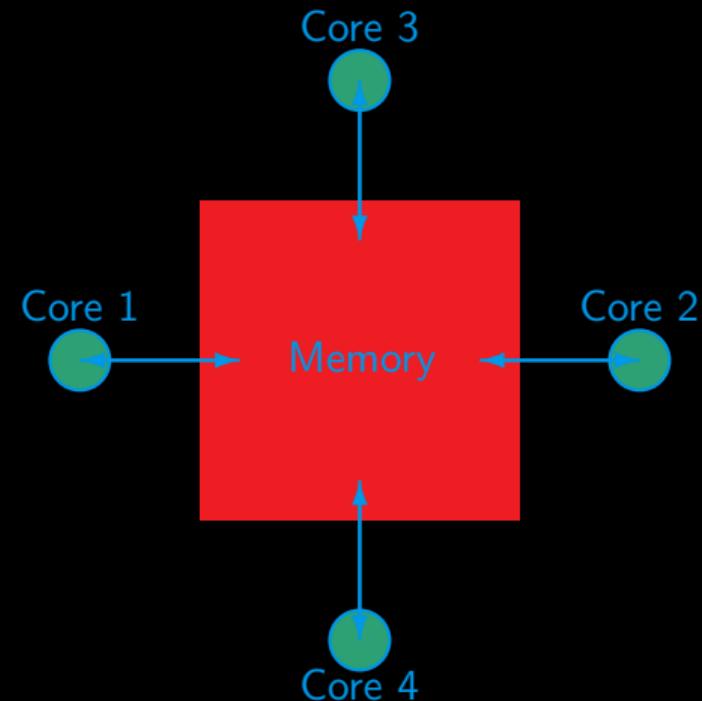
Clusters

- Take existing powerful standalone computers (called a "node"),
- Link them together through a high-speed network (or "interconnect").
- Easy to build and easy to expand.
- Because each node has its own memory that the other nodes cannot see, these are called **distributed memory systems**.
- Nodes communicate and transfer data through messages.
- Programming Model: Message Passing Interface (MPI)



Multi-core Computers

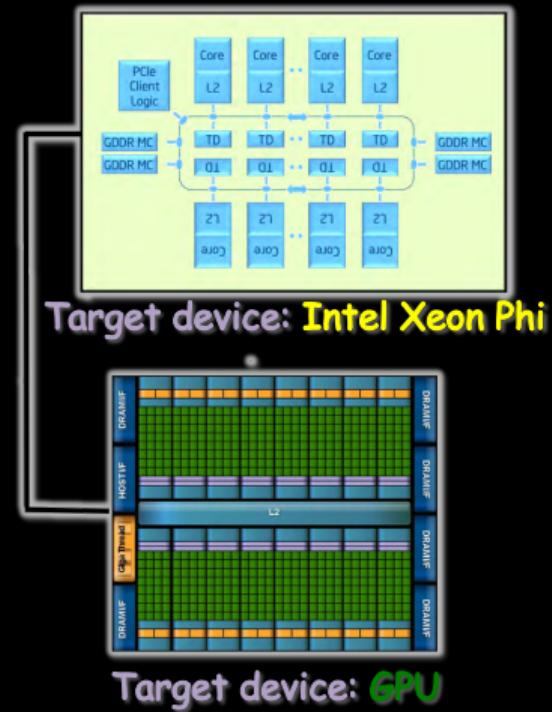
- A collection of processors that can see and use the same memory.
- Limited number of cores, and much more expensive when the number of cores is large.
- Coordination/communication done through memory.
- Also known as **shared-memory systems**.
- Programming model: Threads (e.g. OpenMP)



Your desktop, laptop and cell phone likely use this kind of architecture.

Accelerators

- Systems with accelerators are machines which contain an "off-host" accelerator, such as a GPU or Xeon Phi.
- These accelerator devices are very fast and good at massively parallel processing (having 500-2000+ cores).
- Complicated to program.
- Programming model: CUDA, OpenACC, and OpenCL.
- Needs to be combined with at least some 'host' code: **heterogeneous computing**.

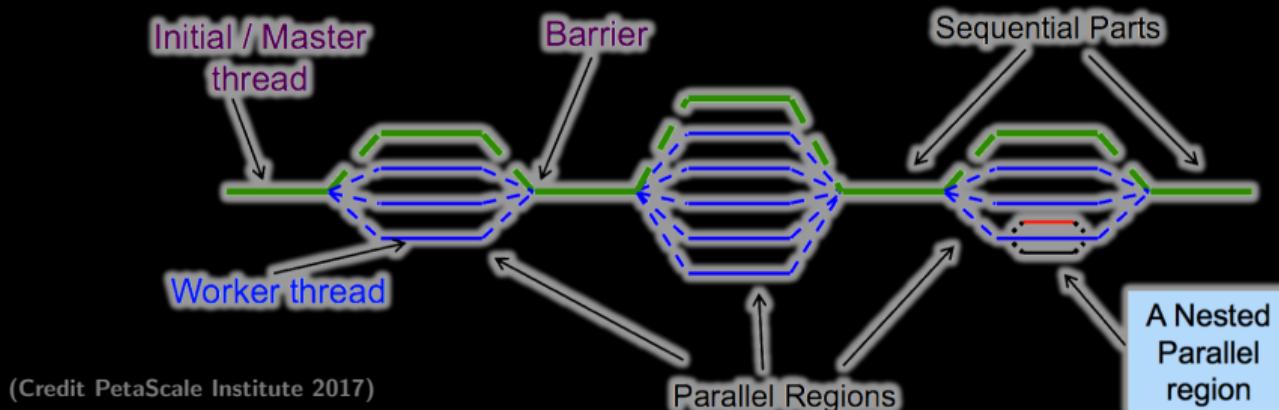


OpenMP for shared-memory Parallel Programming

- For shared memory systems.
- Add parallelism to functioning serial code.
- Compiler, run-time environment does a lot of work for us (divides up work)
- But we have to tell it how to use variables, where to run in parallel, . . .
- Works by adding compiler directives to code.
- Industry standard specifying directives (pragmas) to create parallel Fortran, C and C++ programs
- Directives are instructions to a compiler
- API also has library routines and environment variables
- <http://www.openmp.org>

OpenMP Execution Model

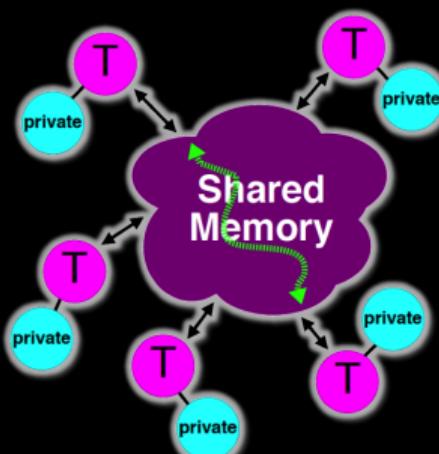
- Execution starts with single thread (the initial / master thread)
- Master thread spawns multiple worker threads as needed, together they form a team
team = master + workers
- Parallel region is a block of code executed by all threads in a team simultaneously



- Number of threads in a team may be dynamically adjusted

OpenMP Memory Model

All threads access the same, globally shared memory



(Credit PetaScale Institute 2017)

Data can be shared or private

- **Shared** – only one instance of data
 - Threads can access data simultaneously
 - Changes are visible to all threads
 - Not necessarily immediately
 - **Private** - Each thread has copy of data
 - No other thread can access it
 - Changes only visible to the thread owning the data
- OpenMP has **relaxed-consistency** shared memory model
- Threads may have a temporary view of shared memory that is not consistent with that of other threads
 - These temporary views are made consistent at certain places in code

MPI for distributed-memory Parallel Programming

Improving scalability

Issues with shared memory programming

- Parallel tasks are run by threads.
- All threads live on the same node and share the memory.
- Limited to the resources of a single node.
- Creation and deletion of threads can cause overhead.
- Can lead to bugs like race conditions.

Distributed memory programming approach

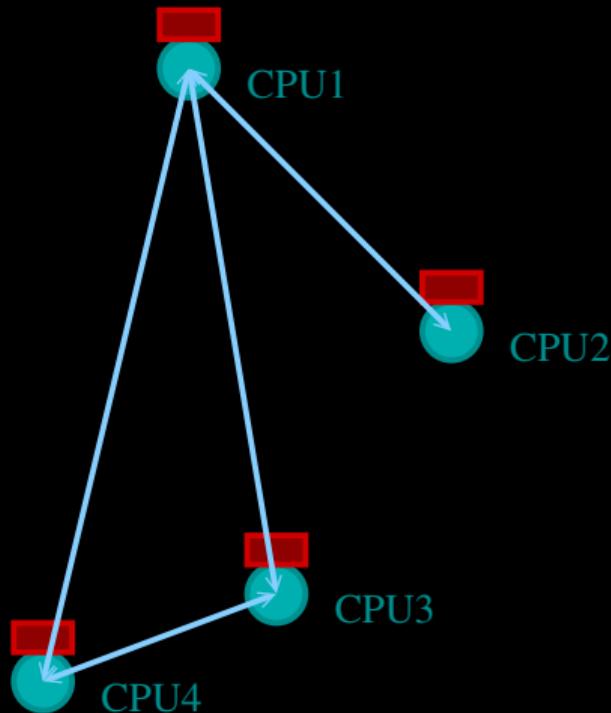
- Parallel tasks are processes.
- Each process has only its own, private memory.
- Processes need not be on the same node.
- You can scale up the size of your system to as many resources as you have.
- Harder to create race condition bugs, but now you get new bugs like dead-lock.
- Must explicitly code in the communication between processes: **Message Passing Interface**

MPI is a Library for Message-Passing

- An open standard library interface for message passing, ratified by the MPI Forum
- Not built in to compiler.
- Function calls that can be made from any compiler, many languages.
- "Just" link to it.
- Multiple implementations: OpenMPI, MPICH, ...
- Wrappers: mpicc, mpif90, mpicxx.

```
1 #include <iostream>
2 #include <string>
3 #include <mpi.h>
4 using namespace std;
5
6 int main(int argc, char **argv) {
7
8     int rank, size;
9
10    MPI_Init(&argc, &argv);
11    MPI_Comm_size(MPI_COMM_WORLD, &size);
12    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14    cout << "Hello_World,_from_task_" +
15        to_string(rank) + "_of_" +
16        to_string(size) + "\n";
17
18    MPI_Finalize();
19 }
```

MPI is a Library for Message Passing



- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.

Messages



- Messages have a **sender** and a **receiver**.
- When you are sending a message, you don't need to specify the sender (it is the current processor).
- A sent message has to be actively received by the receiving process

Size of MPI Library

- Many, many functions (>200).
- Not nearly so many concepts.
- We'll get started with just 10-12, use more as needed.

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Ssend()
MPI_Recv()
MPI_Sendrecv()
MPI_Finalize()

// Special destinations
MPI_PROC_NULL()
MPI_ANY_SOURCE()
```

Example: Hello World

Compile with MPI

MPI provides compiler wrappers

- mpicc
- mpicxx
- mpiif90

that set all the -I, -L, -l, etc. options properly for the base compiler.

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;

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

    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

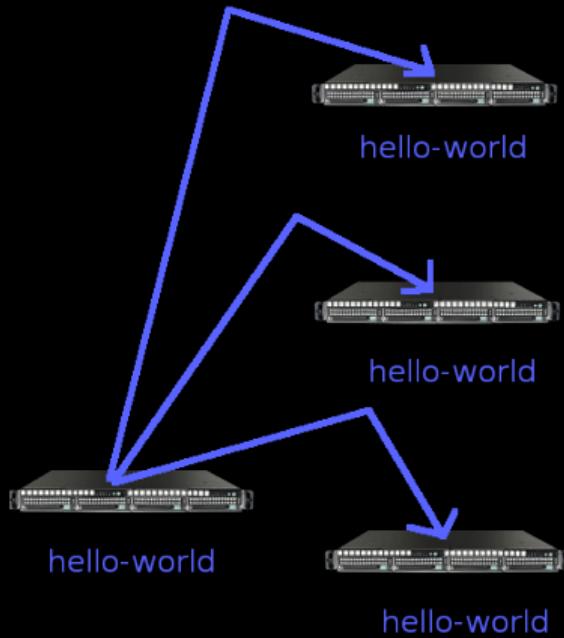
    cout << "HelloWorld, from task " +
        to_string(rank) + " of " +
        to_string(size) + "\n";

    MPI_Finalize();
}
```

```
# Compilation:
mpicxx mpi-hello-world.cc -O2 --std=c++14 -o mpi-hello-world

# Execution:
mpirun -np 16 ./mpi-hello-world
```

What `mpirun` Does



- Launches n processes, assigns each an MPI **rank** and starts the program.
- Usually, the processes run the same executable, therefore **each process runs the exact same code**.
- For multinode runs, has a list of nodes, and logs in (effectively) to each node, where it launches the program.
- `mpirun` can run **any** program, eg.
`mpirun -np 8 hostname`
`mpirun -np 4 ls`

MPI - Hello World (cont)

```
$ mpirun -np 4 ./mpi-hello-world
Hello World, from task 2 of 4
Hello World, from task 1 of 4
Hello World, from task 0 of 4
Hello World, from task 3 of 4

$ mpirun --tag-output -np 4 ./mpi-hello-world
[1,2]<stdout>:Hello World, from task 2 of 4
[1,3]<stdout>:Hello World, from task 3 of 4
[1,0]<stdout>:Hello World, from task 0 of 4
[1,1]<stdout>:Hello World, from task 1 of 4
```

The `--tag-output` flag is specific for the OpenMPI implementation of MPI.

MPI Basics

Basic MPI Components

- `#include <mpi.h>`
MPI library definitions
- `MPI_Init(&argc, &argv)`
MPI Initialization, must come first
- `MPI_Finalize()`
Finalizes MPI, must come last

Communicator Components

- A communicator is a handle to a group of processes that can communicate.
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
- `MPI_Comm_size(MPI_COMM_WORLD, &rank)`

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;

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

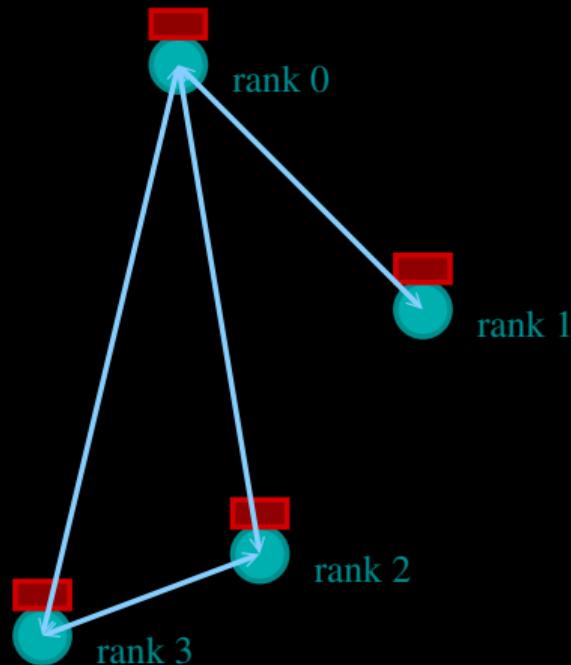
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    cout << "Hello_World, _from_task_" +
        to_string(rank) + "_of_" +
        to_string(size) + "\n";

    MPI_Finalize();
}
```

Communicators



- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank $0..size-1$
- Every task in your program belongs to MPI_COMM_WORLD.

MPI_COMM_WORLD:

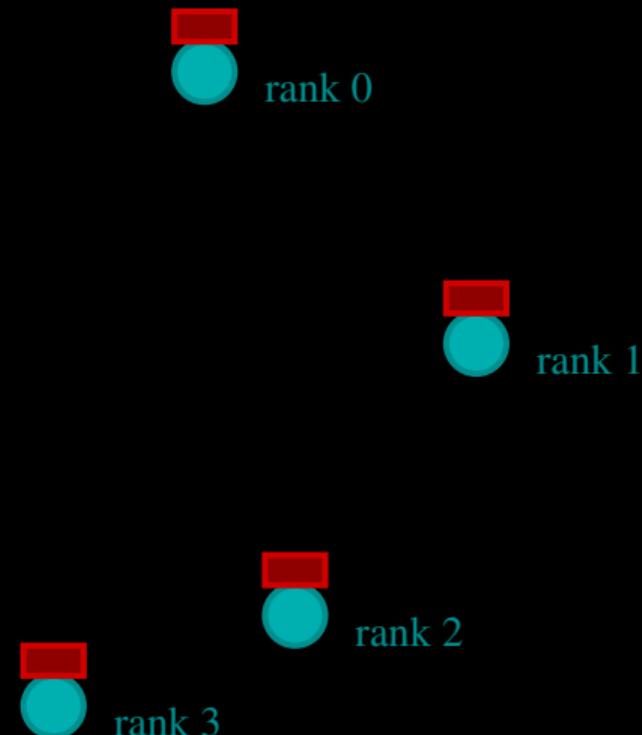
size = 4, ranks = 0..3

- MPI_COMM_WORLD: Global Communicator
- MPI_Comm_rank(MPI_COMM_WORLD,&rank)
Get current tasks rank
- MPI_Comm_size(MPI_COMM_WORLD,&size)
Get communicator size

MPI = Rank and Size

Rank and Size are much more important in MPI than in OpenMP

- In OpenMP, the compiler assigns jobs to each thread; you do not need to know which one is which (usually).
- In MPI, all processes run the same code.
- In MPI, processes determine amongst themselves which piece of puzzle to work on, based on their **rank**, then communicate with appropriate others.



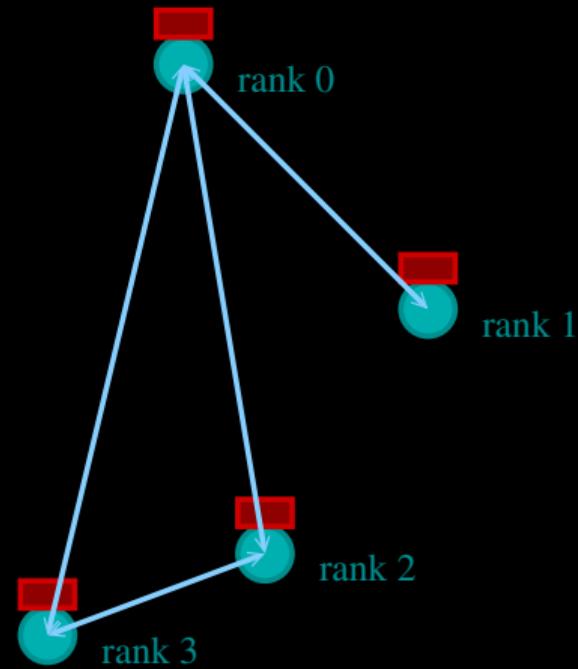
MPI = Communication

Explicit Communication between Tasks

- In OpenMP, threads can communicate using the memory.
- In MPI, a process which needs data of another process needs to communicate with that process by passing messages.

`MPI_Ssend(...)`

`MPI_Recv(...)`



MPI: Send & Receive

```
MPI_Ssend(sendptr, count, MPI_TYPE,  
destination,tag, Communicator);  
MPI_Recv(recvptr, count, MPI_TYPE, source,  
tag, Communicator, MPI_status)
```

MPI: Send & Receive

```
MPI_Ssend(sendptr, count, MPI_TYPE,  
destination,tag, Communicator);  
MPI_Recv(recvptr, count, MPI_TYPE, source,  
tag, Communicator, MPI_status)
```

- sendptr/recvptr: pointer to message
- count: number of elements in message
- MPI_TYPE: one of MPI_DOUBLE,
MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- destination/source: rank of
sender/receiver
- tag: unique id for message pair
- Communicator: MPI_COMM_WORLD or
user created
- status: receiver status (error, source, tag)

MPI: Send & Receive

```
MPI_Ssend(sendptr, count, MPI_TYPE,  
destination,tag, Communicator);  
MPI_Recv(recvptr, count, MPI_TYPE, source,  
tag, Communicator, MPI_status)
```

- sendptr/recvptr: pointer to message
- count: number of elements in message
- MPI_TYPE: one of MPI_DOUBLE,
MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- destination/source: rank of
sender/receiver
- tag: unique id for message pair
- Communicator: MPI_COMM_WORLD or
user created
- status: receiver status (error, source, tag)

```
int main(int argc, char **argv) {  
    int rank, size;  
    int tag = 1;  
    double msgsent, msgrcvd;  
    MPI_Status rstatus;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
  
    msgsent = 111.;  
    msgrcvd = -999.;  
  
    if (rank == 0) {  
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, 1,  
                  tag, MPI_COMM_WORLD);  
        cout << "Sent_" + to_string(msgsent) +  
            "_from_" + to_string(rank) + "\n";  
    }  
  
    if (rank == 1) {  
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, 0,  
                 tag, MPI_COMM_WORLD, &rstatus);  
        cout << "Received_" + to_string(  
            msgrcvd) + "_on_" + to_string(rank)  
            + "\n";  
    }  
    MPI_Finalize();  
}
```

MPI: Send right, Receive left

```
int main(int argc, char **argv) {
    int rank, size, left, right, tag = 1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;

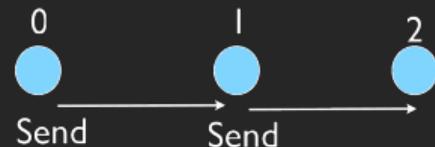
    right = rank + 1;
    if (right >= size) right = MPI_PROC_NULL;

    msgsent = rank*rank;
    msgrcvd = -999.;

    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);

    cout << to_string(rank) + ": Sent " + to_string(msgsent)
        + " and got " + to_string(msgrcvd) + "\n";

    MPI_Finalize();
}
```



MPI: Send right, Receive left

```
$ mpirun -np 3 ./sendLeftreceiveRight
2: Sent 4.000000 and got 1.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000

$ mpirun -np 6 ./sendLeftreceiveRight
4: Sent 16.000000 and got 9.000000
5: Sent 25.000000 and got 16.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
```

MPI: Send Right-Recv Left, with Periodic BCs

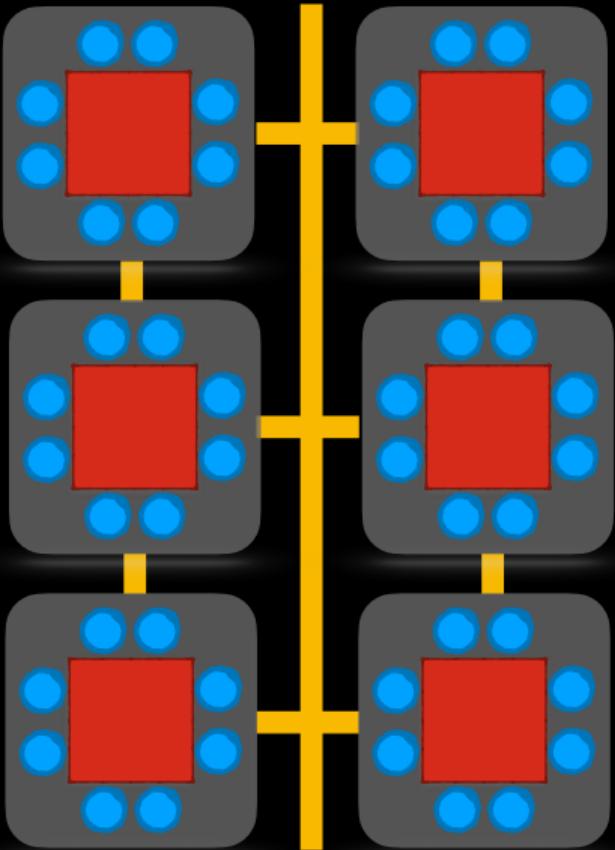
```
// detect identities
left = rank - 1;
if (left < 0) left = size-1; // Periodic BC
right = rank + 1;
if (right >= size) right = 0; // Periodic BC
// messages
msgsent = rank*rank;
msgrcvd = -999.;
// ...
if ((rank % 2) == 0) {
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    // MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
    //             &msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
} else {
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
}
// ...

$ mpirun -np 5 ./sendreceivePBC
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

Hybrid Parallel Programming

Shared and Distributed Memory Systems

- Modern clusters have a hybrid architecture.
- Multicore machines linked together with an (specialized) interconnect.
- Machines with GPU or other coprocessors: GPU is multi-core, but the amount of shared memory is limited.



Hybrid Parallel Programming

On Modern HPC systems, we need a solution to enable parallelism across nodes

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for shared memory or intra-node, and now supports accelerators as well (intra means within)
- Several solutions $\text{MPI} + X$
 - MPI + OpenMP
 - MPI + MPI
 - MPI + other languages
 - Non-MPI solutions such as HPX, ParallelX, ...
- *Hybrid Programming* is when we use a solution with different programming models for inter vs intra-node parallelism

MPI vs OpenMP

We have OpenMP for shared memory programming.

We have MPI to program distributed memory machines.

| model | memory | latency | mem.overhead | scalable | incremental |
|--------|-------------|----------|--------------|----------|-------------|
| OpenMP | shared mem | low | low | limited | yes |
| MPI | distributed | high(er) | higher | yes | no |

- Could we have the best of both worlds?

MPI vs OpenMP

We have OpenMP for shared memory programming.

We have MPI to program distributed memory machines.

| model | memory | latency | mem.overhead | scalable | incremental |
|--------|-------------|----------|--------------|----------|-------------|
| OpenMP | shared mem | low | low | limited | yes |
| MPI | distributed | high(er) | higher | yes | no |

- Could we have the best of both worlds?
 - NUMA/memory hierarchy/cache-lines/...
 - processor affinity
 - memory affinity (NUMA...)

Hybrid Programming

Example

```
1 #include <mpi.h>
2 #include <omp.h>
3 #include <iostream>
4
5 int main(int argc, char ** argv) {
6
7     int size,rank;
8
9     MPI_Init(&argc,&argv);
10    MPI_Comm_get_rank(MPI_COMM_WORLD ,&rank);
11    MPI_Comm_get_size(MPI_COMM_WORLD ,&size);
12
13 #pragma omp parallel for
14 for (int i=0;i<4;i++)
15     std::cout << "Hello_world_from_thread_"
16     << omp_get_thread_num() << std::endl;
17
18 MPI_Finalize();
19 }
```

PseudoCode (Fortran-ish...)

Program hybrid

```
call MPI_INIT (ierr)
call MPI_COMM_RANK (...)

call MPI_COMM_SIZE (...)

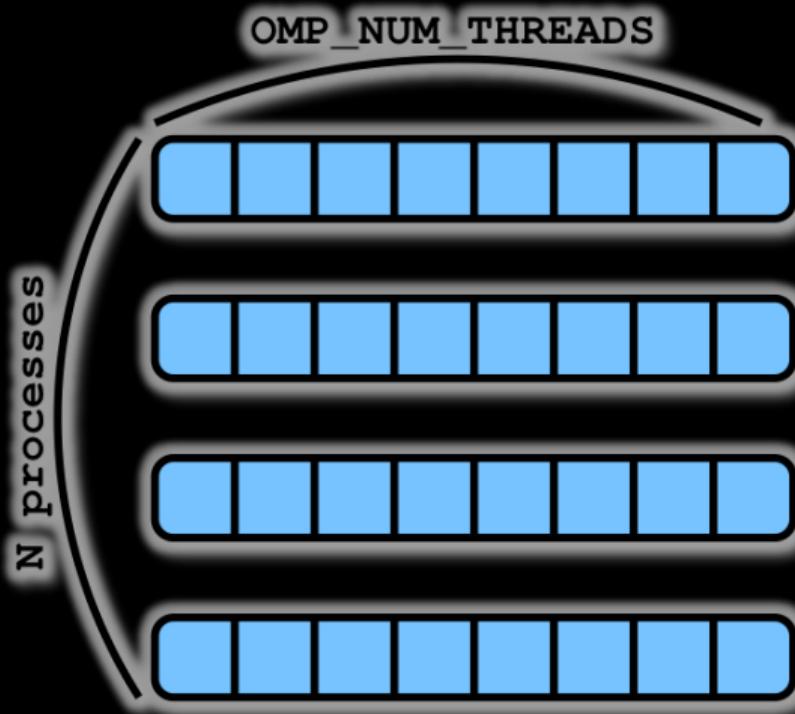
... some computation and MPI communication

call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO PRIVATE(i)
!$OMP&           SHARED(n)
do i=1,n
    ... computation
enddo
!$OMP END PARALLEL DO
... some computation and MPI communication
call MPI_FINALIZE (ierr)
end
```

endp

```
call MPI_FINALIZE (ierr)
... some computation and MPI communication
!$OMP END PARALLEL DO
enddo
    ... computation
do i=1,n
```

Hybrid Programming



- Memory shared among threads of same process
- Memory not shared among threads of different processes

Hybrid Programming – Technicalities

- Note: OpenMP inside MPI
- Often, one starts with an MPI code and adds in OpenMP.
- Compilation:

```
mpicxx -fopenmp [filename] -o [executable]
```

- Execution:

```
export OMP_NUM_THREADS=M
```

```
mpirun -np N --bynode [executable]
```

- This starts N processes
- Between `MPI_Init` and `MPI_Finalize`, each process spawns `OMP_NUM_THREADS` threads in `#pragma omp parallel` blocks.

Summary

Summary

- Supercomputing/HPC
- Shared memory programming: openMP (pragmas, directives to the compiler)
- Distributed memory programming: MPI (library)
- Hybrid parallel programming: MPI+ X , X =openMP (openACC, CUDA, ...)

MPI Application

MPI Example

Let's consider a diffusion equation with an explicit **finite-difference**, **time-marching** method.

We'll imagine that the problem is too large to fit in the memory of one node, so we need to do **domain decomposition**, and use **MPI**.

Discretizing Derivatives

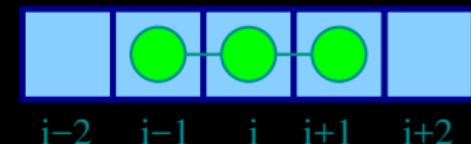
- Partial Differential Equations like the diffusion equation

$$\frac{\partial T}{\partial t} = D \frac{\partial^2 T}{\partial x^2}$$

are usually numerically solved by finite differencing the discretized values.

- Implicitly or explicitly involves interpolating data and taking the derivative of the interpolant.
- Larger ‘stencils’ → More accuracy.

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$



Diffusion equation in higher dimensions

Spatial grid separation: Δx . Time step Δt .

Grid indices: i, j . Time step index: (n)

1D

$$\frac{\partial T}{\partial t} \Big|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\frac{\partial^2 T}{\partial x^2} \Big|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



Diffusion equation in higher dimensions

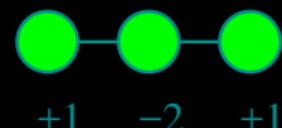
Spatial grid separation: Δx . Time step Δt .

Grid indices: i, j . Time step index: (n)

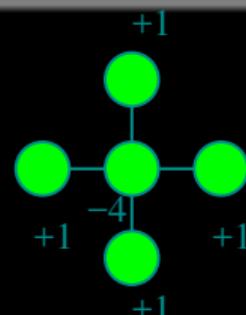
1D

$$\frac{\partial T}{\partial t} \Big|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\frac{\partial^2 T}{\partial x^2} \Big|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



Diffusion equation in higher dimensions

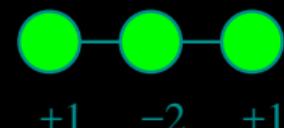
Spatial grid separation: Δx . Time step Δt .

Grid indices: i, j . Time step index: (n)

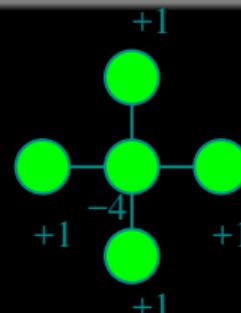
1D

$$\frac{\partial T}{\partial t} \Big|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\frac{\partial^2 T}{\partial x^2} \Big|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



$$\frac{\partial T}{\partial t} \Big|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

$$\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \Big|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution:

Guard cells:

- Pad domain with these guard cells so that stencil works even for the first point in domain.
- Fill guard cells with values such that the required boundary conditions are met.



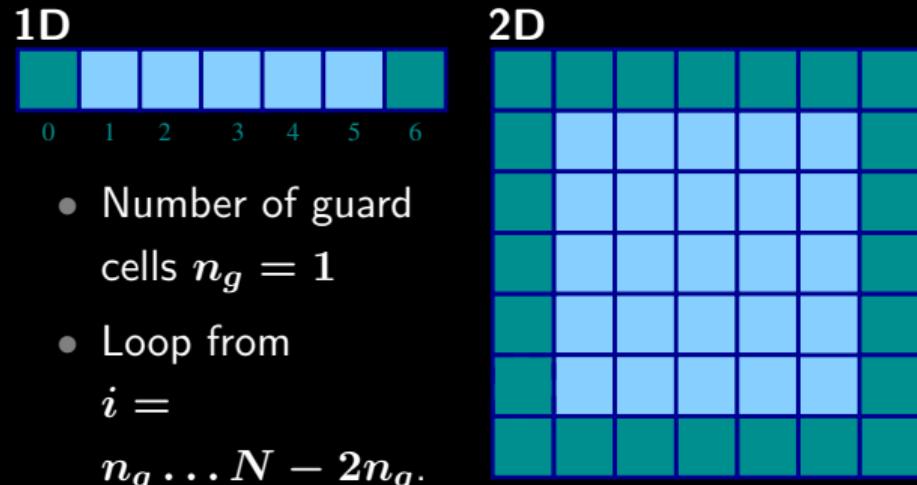
- Number of guard cells $n_g = 1$
- Loop from
 $i = n_g \dots N - 2n_g$.

Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution:

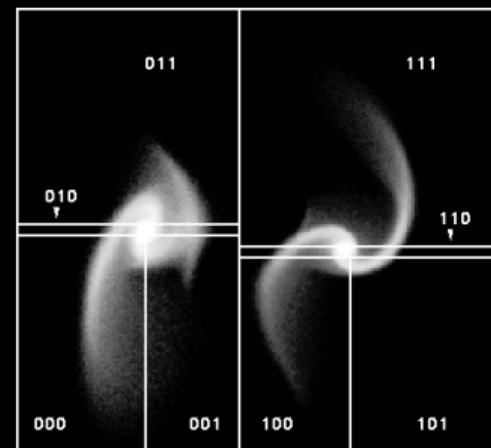
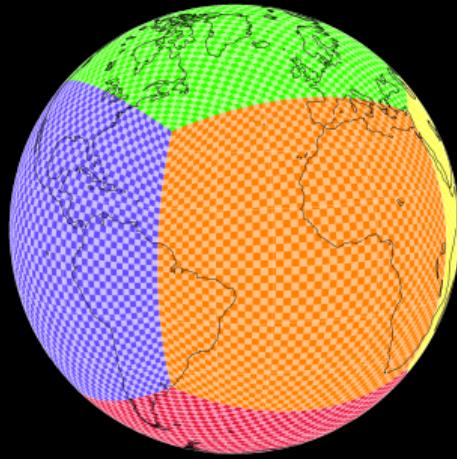
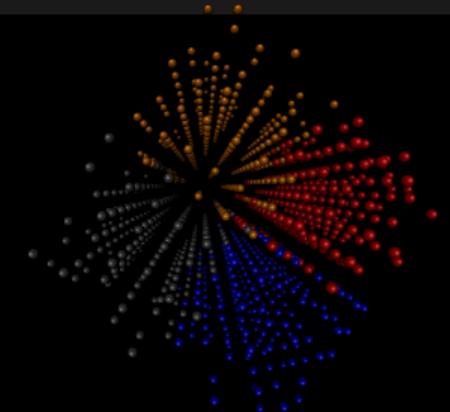
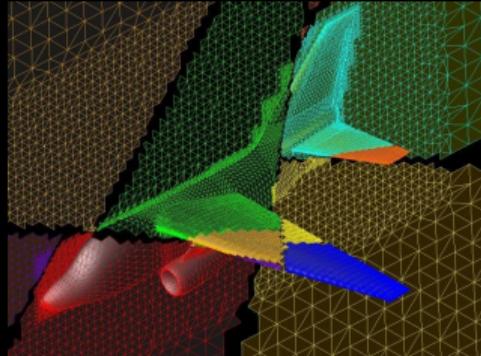
Guard cells:

- Pad domain with these guard cells so that stencil works even for the first point in domain.
- Fill guard cells with values such that the required boundary conditions are met.



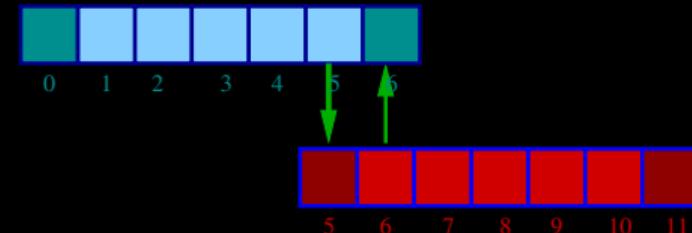
Domain decomposition

- A very common approach to parallelizing on distributed memory computers.
- Subdivide the domain into contiguous subdomains.
- Give each subdomain to a different MPI process.
- No process contains the full data!
- Maintains locality.
- Need mostly local data, ie., only data at the boundary of each subdomain will need to be sent between processes.



Guard cell exchange

- In the domain decomposition, the stencils will jut out into a neighbouring subdomain.
- Much like the boundary condition.
- One uses guard cells for domain decomposition too.
- If we managed to fill the guard cell with values from neighbouring domains, we can treat each coupled subdomain as an isolated domain with changing boundary conditions.



- Could use even/odd trick, or sendrecv.

1D diffusion with MPI

Before MPI

```
a = 0.25*dt/pow(dx, 2);
guardleft = 0;
guardright = n+1;

for (int t=0; t<maxt; t++) {
    T[guardleft] = 0.0;
    T[guardright] = 0.0;

    for (int i=1; i<=n; i++)
        newT[i] = T[i] + a*(T[i+1]+T[i-1]-2*T[i]);

    for (int i=1; i<=n; i++)
        T[i] = newT[i];
}
```

Note:

- the for-loop over i could also easily be parallelized with OpenMP (\Rightarrow hybrid MPI-OpenMP code).

1D diffusion with MPI

Before MPI

```
a = 0.25*dt/pow(dx,2);
guardleft = 0;
guardright = n+1;

for (int t=0;t<maxt;t++) {
    T[guardleft] = 0.0;
    T[guardright] = 0.0;

    for (int i=1; i<=n; i++)
        newT[i] = T[i] + a*(T[i+1]+T[i-1]-2*T[i]);

    for (int i=1; i<=n; i++)
        T[i] = newT[i];
}
```

Note:

- the for-loop over i could also easily be parallelized with OpenMP
(\Rightarrow hybrid MPI-OpenMP code).

After MPI

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

left = rank-1; if(left<0)left=MPI_PROC_NULL;
right = rank+1; if(right>=size)right=
            MPI_PROC_NULL;
localn = n/size;
a = 0.25*dt/pow(dx,2);
guardleft = 0; guardright = localn+1;
for (int t=0;t<maxt;t++) {
    MPI_Sendrecv (&T[1], 1, MPI_DOUBLE , left , 11,
                  &T[guardright], 1, MPI_DOUBLE , right , 11,
                  MPI_COMM_WORLD ,MPI_STATUS_IGNORE);
    MPI_Sendrecv (&T[nlocal], 1, MPI_DOUBLE ,
                  right , 11, &T[guardleft], 1, MPI_DOUBLE ,
                  left , 11, MPI_COMM_WORLD ,
                  MPI_STATUS_IGNORE);
    if (rank==0) T[guardleft] = 0.0;
    if (rank==size-1) T[guardright] = 0.0;
    for (int i=1; i<=localn; i++)
        newT[i] = T[i] + a*(T[i+1]+T[i-1]-2*T[i]);
    for (int i=1; i<=n; i++)
        T[i] = newT[i];
}
MPI_Finalize();
```

More on Hybrid Parallel Programming

MPI and OpenMP

Hybrid programming model of using MPI and OpenMP

- MPI across nodes
- OpenMP within nodes
- Minimizes communication
- Scalable
- Not much more complicated than pure MPI
- MPI+ X / X =openMP

MPI and OpenMP

Hybrid programming model of using MPI and OpenMP

- MPI across nodes
- OpenMP within nodes
- Minimizes communication
- Scalable
- Not much more complicated than pure MPI
- MPI+ X / X =openMP

Many Computations are amenable to a Hybrid (often MPI + OpenMP) approach

- Domains (often resulting from the decomposition of PDE's) are spread across the large system and only need to communicate "ghost zone" information as time advances
- MPI is used to communicate these ghost zone values in the form of messages passed among the nodes
- In various hybrid approaches, different programming models can be used for the shared memory region. OpenMP is one choice.

Hybrid Programming

Pros

- No decomposition on node
- Lower latency, less communication
- Less duplication of data (and perhaps computation)
- OpenMP has load balance capabilities

Cons

- One more layer to maintain
- OpenMP has more hidden side effects
- May have to worry about NUMA

MPI + OpenMP has been targeted for optimized use on many systems

- For example, systems with many threads to keep busy
- MPI + OpenMP is one obvious programming model
 - may not fit into node using pure MPI across all HW cores and threads because of the memory overhead for each MPI task.
 - Conceptually nice: OpenMP within node, MPI between
 - Provides a way to increase fine-scale parallel granularity
 - Some problems have natural two-level parallelism;

Programming Tips for Adding OpenMP

- Choose between fine grain or coarse grain parallelism implementation.
- Use profiling tools to find hotspots.

Add OpenMP and check correctness incrementally.

- Parallelize outer loop and collapse loops if possible.
- Minimize shared variables, minimize barriers.
- Decide whether to overlap MPI communication with thread computation.
 - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks.
 - Could use MPI inside parallel region with thread-safe MPI.
- Consider OpenMP TASKing.

Hybrid MPI+OpenMP Scalability Performance

- MPI/OpenMP Code can sometimes be slower
- All threads are idle except one while MPI communication
 - Need overlap computation and communication for better performance.
 - Critical Section for shared variables.
- Thread creation overhead
- Cache coherence, false sharing.
- Data placement, NUMA effects.
- Natural one level parallelism problems.
- Check performance of pure OpenMP in a node to pure MPI in a node

If a Routine Does Not Scale Well

- Examine code for serial/critical sections, eliminate if possible.
- Reduce number of OpenMP parallel regions to reduce overhead costs.
- Perhaps loop collapse, loop fusion or loop permutation is required to give all threads enough work, and to optimize thread cache locality.
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme.
- Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth.
- Test different process and thread affinity options.
- Leave some cores idle on purpose, for memory capacity or bandwidth capacity.

If a Routine Does Not Scale Well

- Examine code for serial/critical sections, eliminate if possible.
- Reduce number of OpenMP parallel regions to reduce overhead costs.
- Perhaps loop collapse, loop fusion or loop permutation is required to give all threads enough work, and to optimize thread cache locality.
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme.
- Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth.
- Test different process and thread affinity options.
- Leave some cores idle on purpose, for memory capacity or bandwidth capacity.

* **Vary MPI ranks and OpenMP threads to find optimal configuration**

~~~ **find the sweet spot for hybrid MPI/OpenMP**