Programming Models

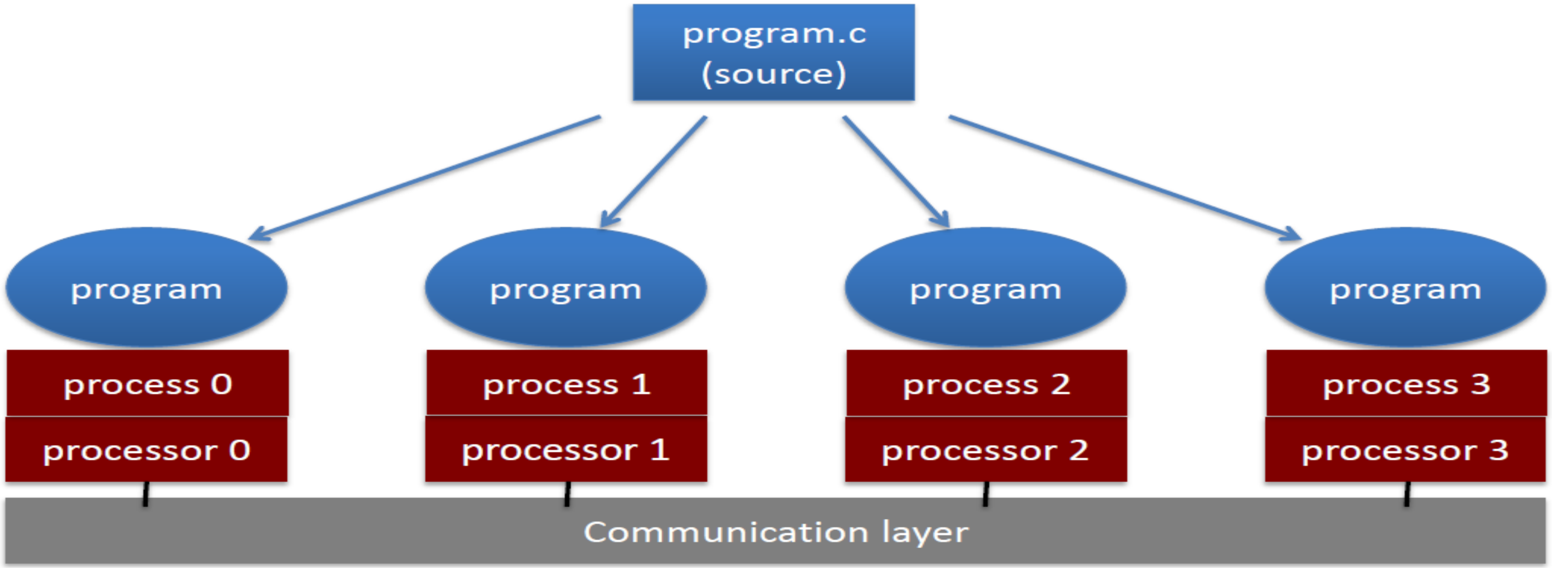
Types of parallelism

- Data Parallelism
  - Each processor performs the same task on different data (remember SIMD, MIMD)
- Task Parallelism
  - Each processor performs a different task on the same data (remember MISD, MIMD)
- Many applications incorporate both

Implementation: Single Program Multiple Data

- Dominant programming model for shared and distributed memory machines
- One source code is written
- Code can have conditional execution based on which processor is executing the copy
- All copies of code start simultaneously and communicate and synchronize with each other periodically

SPMD Model


program.c
(source)
program
program
program
program
process 0
process 1
process 2
process 3
processor 0
processor 1
processor 2
processor 3
Communication layer

Data Parallel Programming

- Example
  - One code will run on 2 CPUs
  - Program has array of data to be operated on by 2 CPUs so array is split into two parts.

```
program:
…
if CPU=a then
   low_limit=1
   upper_limit=50
elseif CPU=b then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU A

```
program:
…
low_limit=1
upper_limit=50
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

Task Parallel Programming

- Example
  - One code will run on 2 CPUs
  - Program has 2 tasks (a and b) to be done by 2 CPUs

```
program.f:
…
initialize
...
if CPU=a then
   do task a
elseif CPU=b then
   do task b
end if
….
end program
```

CPU A

```
program.f:
…
initialize
…
do task a
…
end program
```

CPU B

```
program.f:
…
initialize
…
do task b
…
end program
```

Shared Memory Programming: pthreads

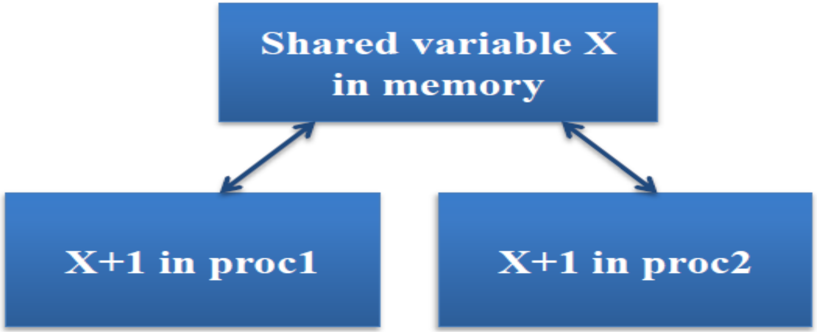
- Shared memory systems (SMPs, ccNUMAs) have a single address space
- Applications can be developed in which loop iterations (with no dependencies) are executed by different processors
- Threads are 'lightweight processes' (same PID)
- Allows 'MIMD' codes to execute in shared address space

Shared Memory Programming: OpenMP

- Built on top of pthreads
- Shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
- OpenMP is a standard for shared memory programming (compiler directives)
- Vendors offer native compiler directives

Accessing Shared Variables

- If multiple processors want to write to a shared variable at the same time, there could be conflicts :
  - Process 1 and 2
  - read X
  - compute X+1
  - write X



- Programmer, language, and/or architecture must provide ways of resolving conflicts (mutexes and semaphores)

OpenMP Example #1: Parallel Loop

```
!$OMP PARALLEL DO
do i=1,128
        b(i) = a(i) + c(i)
end do
!$OMP END PARALLEL DO
```

```
void simple(int n, float *a, float *b)
{
   int i;
#pragma omp parallel for
   for (i=1; i<n; i++) /* i is private by default */
      b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- The first directive specifies that the loop immediately following should be executed in parallel.
- The second directive specifies the end of the parallel section (optional).
- For codes that spend the majority of their time executing the content of simple loops, the PARALLEL DO directive can result in significant parallel performance.

OpenMP Example #2: Private Variables

```
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(I,TEMP)
do I=1,N
      TEMP = A(I)/B(I)
      C(I) = TEMP + SQRT(TEMP)
end do
!$OMP END PARALLEL DO
```

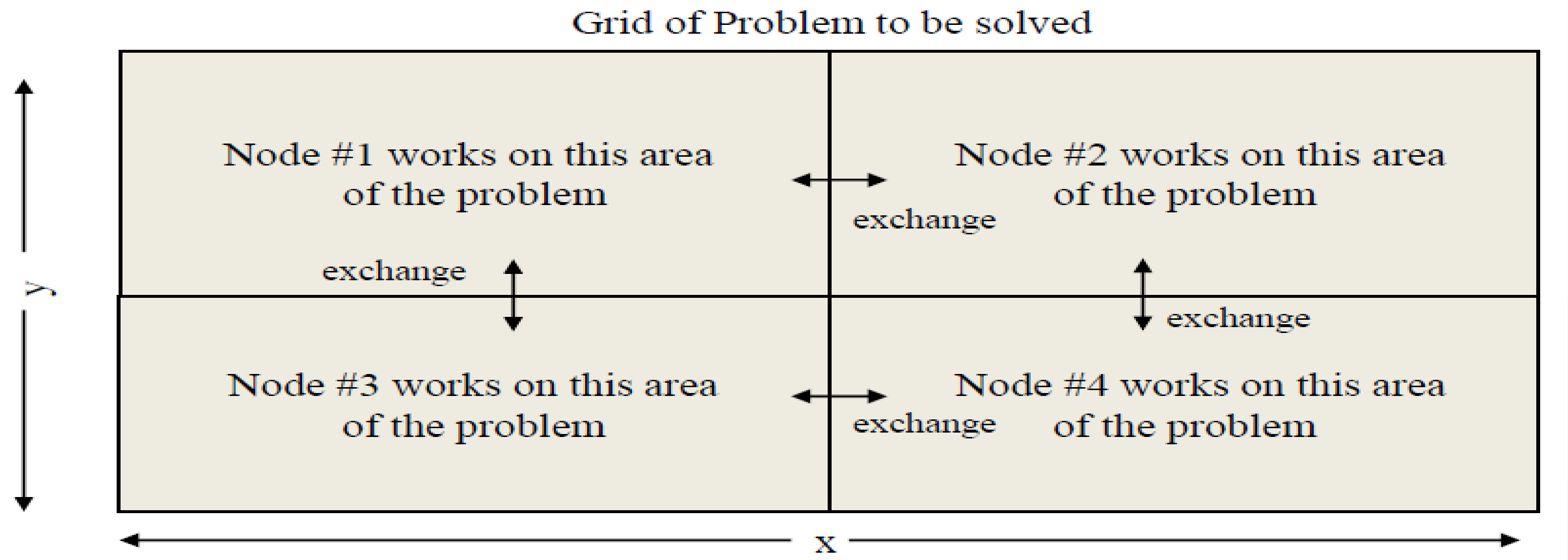
- In this loop, each processor needs its own private copy of the variable TEMP.
- If TEMP were shared, the result would be unpredictable since multiple processors would be writing to the same memory location.

Distributed Memory Programming: MPI

- Distributed memory systems have separate address spaces for each processor
- Local memory access is faster than remote memory
- Data must be manually decomposed
- MPI is the de facto standard for distributed memory programming (library of subprogram calls)
- Vendors typically have native libraries such as SHMEM (T3E) and LAPI (IBM)

Data Decomposition

- For distributed memory systems, the 'whole' grid is decomposed to the individual nodes
  - Each node works on its section of the problem



Typical Data Decomposition

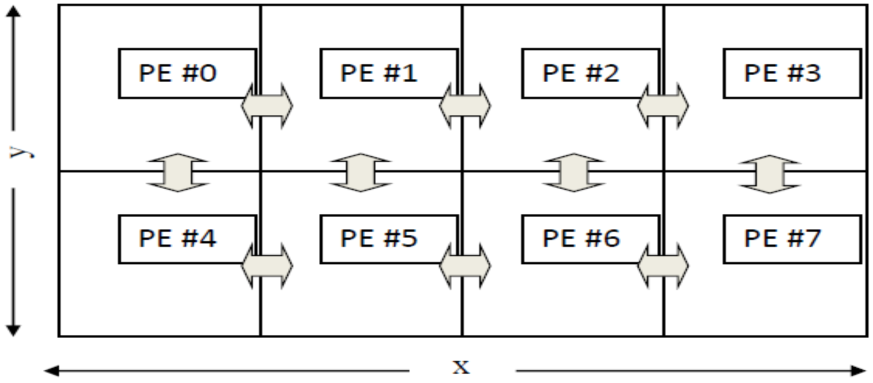
- Example: integrate 2-D propagation problem:

Starting partial differential equation:

$$\frac{\partial \Psi}{\partial t} = D \cdot \frac{\partial^2 \Psi}{\partial x^2} + B \cdot \frac{\partial^2 \Psi}{\partial y^2}$$

Finite Difference Approximation:

$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} = D \cdot \frac{f_{i+1,j}^{n} - 2 f_{i,j}^{n} + f_{i-1,j}^{n}}{\Delta x^2} + B \cdot \frac{f_{i,j+1}^{n} - 2 f_{i,j}^{n} + f_{i,j-1}^{n}}{\Delta y^2}$$



MPI Example #1

- Every MPI program needs these:

```
#include “mpi.h”
int main(int argc, char *argv[])
{
int nPEs, iam;
/* Initialize MPI */
ierr = MPI_Init(&argc, &argv);
/* How many total PEs are there */
ierr = MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What node am I (what is my rank?) */
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
ierr = MPI_Finalize();
}
```

MPI Example #2

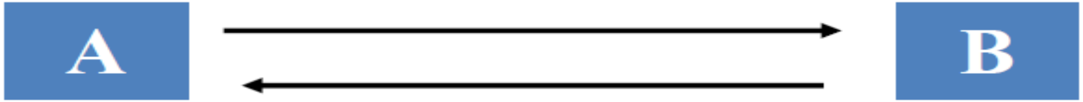
```
#include “mpi.h”
int main(int argc, char *argv[])
{
int numprocs, myid;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
/* print out my rank and this run's PE size */
printf("Hello from %d of %d\n", myid, numprocs);
MPI_Finalize();
}
```

MPI: Sends and Receives

- MPI programs must send and receive data between the processors (communication)
- The most basic calls in MPI (besides the three initialization and one finalization calls) are:
  - MPI_Send
  - MPI_Recv
- These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor issues the matching receive/send.

Message Passing Communication

- Processes in message passing programs communicate by passing messages



- Basic message passing primitives: MPI_CHAR, MPI_SHORT, …
- Send (parameters list)
- Receive (parameter list)
- Parameters depend on the library used
- Barriers

MPI Example #3: Send/Receive

```
#include "mpi.h"

int main(int argc,char *argv[])
{
     int numprocs,myid,tag,source,destination,count,buffer;
     MPI_Status status;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
     tag=1234;
     source=0;
     destination=1;
     count=1;

     if(myid == source){
          buffer=5678;
          MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
          printf("processor %d sent %d\n",myid,buffer);
     }
     if(myid == destination){
          MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
          printf("processor %d got %d\n",myid,buffer);
     }
     MPI_Finalize();
}
```

QUESTIONS?