Cornell University
Center for Advanced Computing

Introduction to Parallel Computing

Susan Mehringer
Cornell University
Center for Advanced Computing
May 28, 2009

What is Parallel Computing?

- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on its section of the problem or data
  - Processors can exchange information

Why Do Parallel Computing?

- Limits of single CPU computing
  - performance
  - available memory
- Parallel computing allows one to:
  - solve problems that don't fit on a single CPU
  - solve problems that can't be solved in a reasonable time
- We can solve…
  - larger problems
  - faster
  - more cases

Parallel Terminology (1 of 3)

- **serial** code is a single thread of execution working on a single data item at any one time
- **parallel** code has more than one thing happening at a time. This could be
  - A single thread of execution operating on multiple data items simultaneously
  - Multiple threads of execution in a single executable
  - Multiple executables all working on the same problem
  - Any combination of the above
- **task** is the name we use for an instance of an executable. Each task has its own virtual address space and may have multiple threads.

Parallel Terminology

- **node**: a discrete unit of a computer system that typically runs its own instance of the operating system.
- **core**: a processing unit on a computer chip that is able to support a thread of execution; can refer either to a single core or to all of the cores on a particular chip.
- **cluster**: a collection of machines or **nodes** that function in some way as a single resource.
- **scheduler**: assigns **nodes** of a cluster to a user who submits a **job**
- **grid**: the software stack designed to handle the technical and social challenges of sharing resources across networking and institutional boundaries. **grid** also applies to the groups that have reached agreements to share their resources.

Parallel Terminology

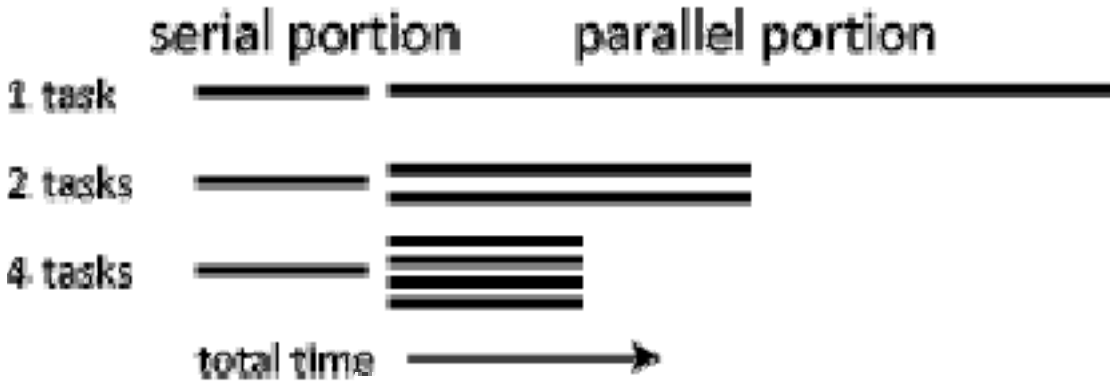
- **synchronization**: the temporal coordination of parallel tasks. It involves waiting until two or more tasks reach a specified point (a sync point) before continuing any of the tasks.
- **parallel overhead**: the amount of time required to coordinate parallel tasks, as opposed to doing useful work, including time to start and terminate tasks, communication, move data.
- **granularity**: a measure of the ratio of the amount of computation done in a parallel task to the amount of communication.
  - fine-grained (very little computation per communication-byte)
  - coarse-grained (extensive computation per communication-byte).

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections
- Other Considerations
  - time to re-write code

Theoretical Upper Limits to Performance

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness



- Amdahl's Law states this formally

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Effect of multiple processors on run time

$$t_n = \left(f_p / N + f_s \right) t_1$$

  - Where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors

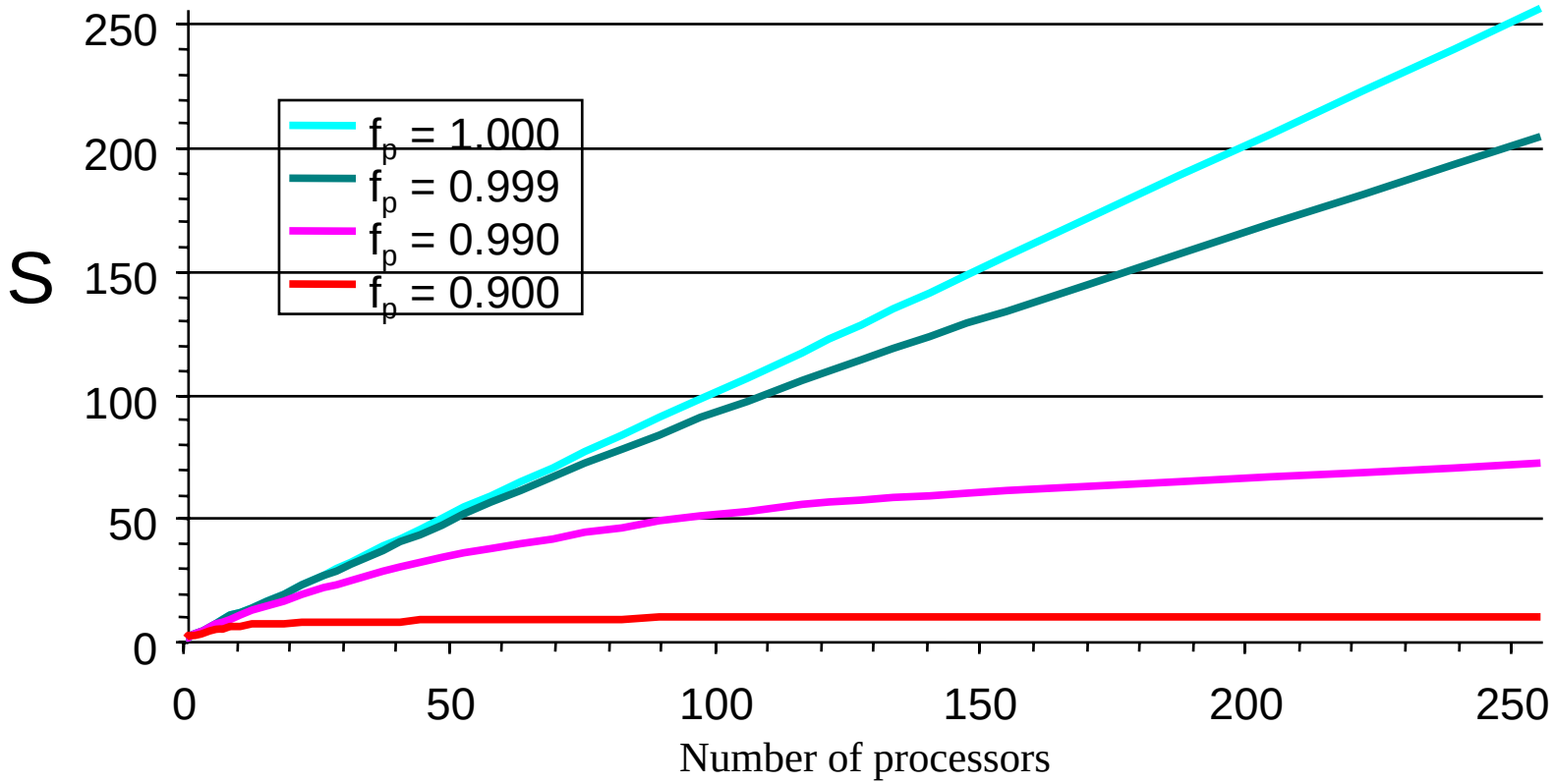
Limit Cases of Amdahl's Law

- Speed up formula:

$$S = \frac{1}{f_s + f_p / N}$$

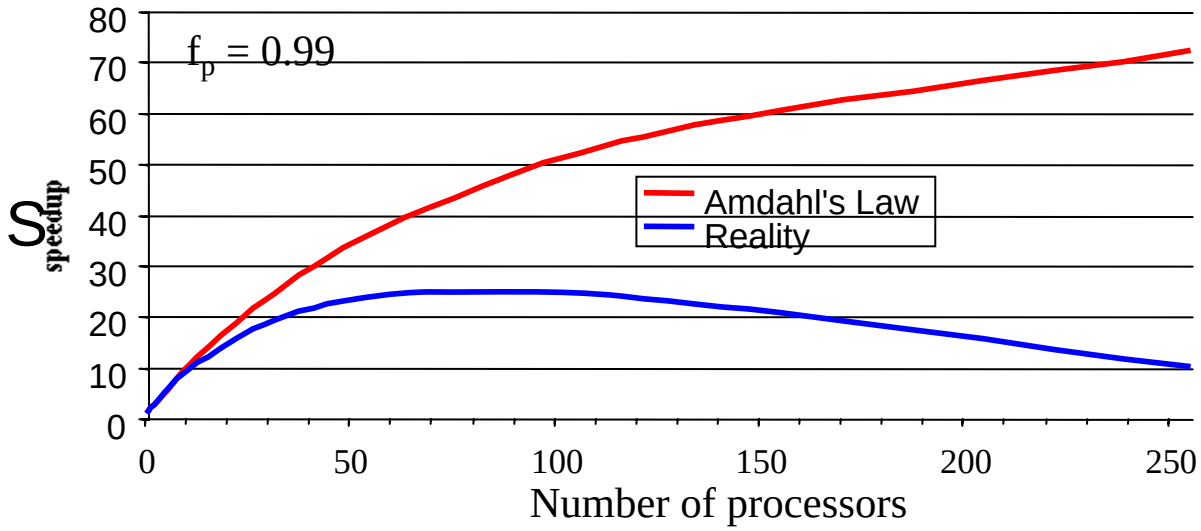
  - Where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors
  - Case:
    - $f_s = 0$, $f_p = 1$, then S = N
    - N goes to infinity: $S = 1/f_s$, so if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors.

Illustration of Amdahl's Law



Practical Limits: Amdahl's Law vs. Reality

- Amdahl's Law shows a theoretical upper limit || speedup
- In reality, the situation is even worse than predicted by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Communications
  - I/O

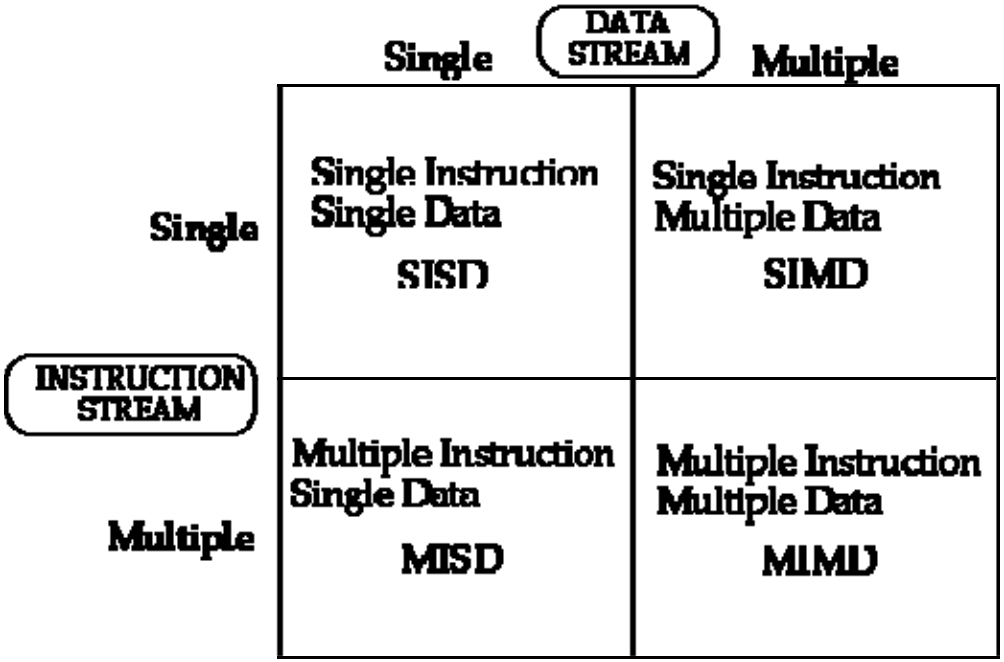


Other Considerations

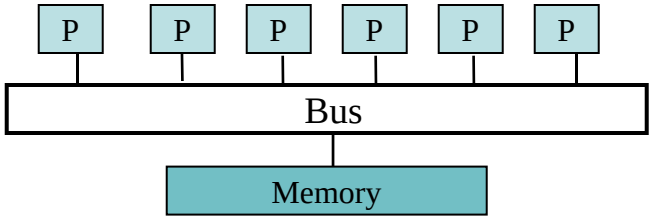
- Writing effective parallel applications is difficult!
  - Load balance is important
  - Communication can limit parallel efficiency
  - Serial time can dominate
- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization?
  - Will the code be used just once?

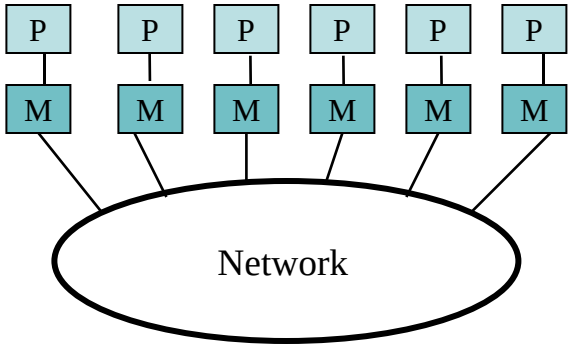
Taxonomy of Parallel Computers

- Flynn's taxonomy classifies parallel computers into four basic types
- Nearly all parallel machines are currently MIMD



Shared and Distributed Memory





Shared memory: single address space. All processors have access to a pool of shared memory. (examples: SGI Altix, IBM Power5 node)

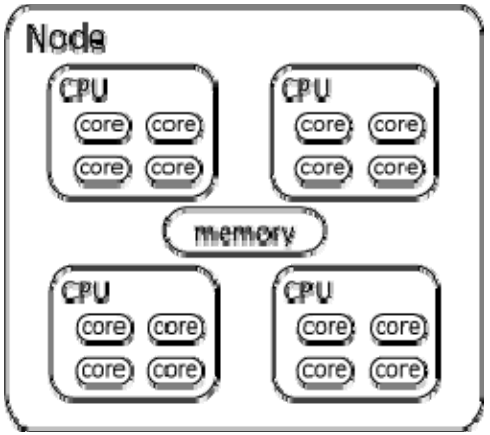
Methods of memory access :
- Bus
- Crossbar

Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors. (examples: Clusters)

Methods of memory access :
- various topological interconnects

Shared and Distributed Memory

- Cluster computers may or may not have global memory
- Although the memory is shared by all of the cores within a node, it is possible to run multiple executables on the same node by using virtual address space
- Another possibility is to run multithreaded tasks that use shared-memory programming to use multiple cores on a single node and distributed-memory programming for exploiting many nodes.



Other Considerations

- Writing effective parallel applications is difficult!
  - Load balance is important
  - Communication can limit parallel efficiency
  - Serial time can dominate
- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization?
  - Will the code be used just once?

Programming Parallel Computers

- Programming single-processor systems is (relatively) easy due to:
  - single thread of execution
  - single address space
- *Programming shared memory systems* can benefit from the single address space
- *Programming distributed memory systems* is the most difficult due to multiple address spaces and need to access remote data

Data vs Functional Parallelism

- Partition by task (functional parallelism)
  - Each process performs a different "function" or executes a different code section
  - First identify functions, then look at the data requirements
  - Commonly programmed with message-passing libraries
- Partition by data (data parallelism)
  - Each process does the same work on a unique piece of data
  - First divide the data. Each process then becomes responsible for whatever work is needed to process its data.
  - Data placement is an essential part of a data-parallel algorithm
  - Probably more scalable than functional parallelism
  - Can be programmed at a high level with OpenMP, or at a lower level (subroutine calls) using a message-passing library like MPI, depending on the machine.

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
```

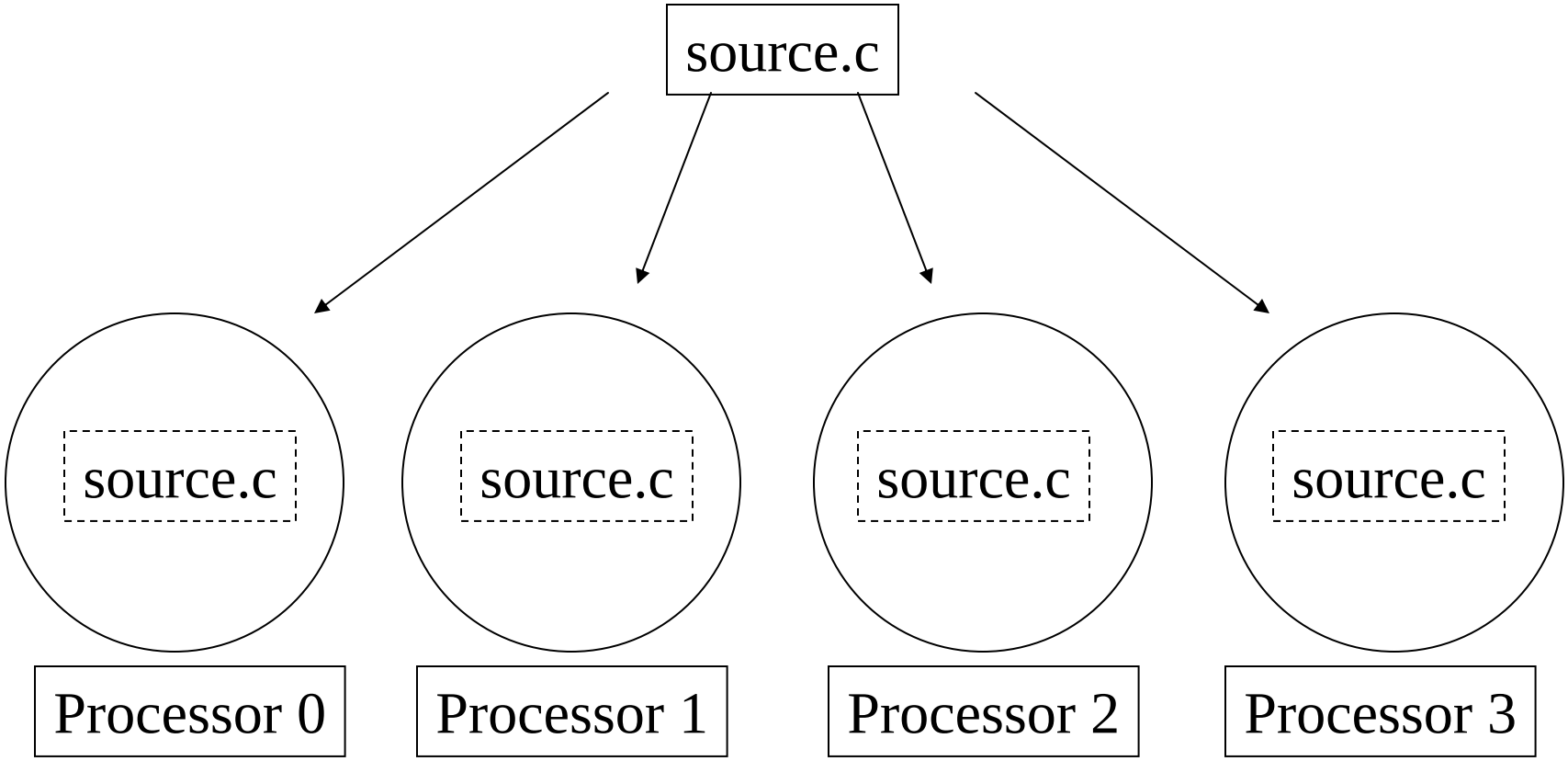
CPU B

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

Single Program, Multiple Data (SPMD)

- SPMD: 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 are started simultaneously and communicate and sync with each other periodically
- MPMD: more general, and possible in hardware, but no system/programming software enables it

Single Program, Multiple Data (SPMD)



Shared Memory vs. Distributed Memory

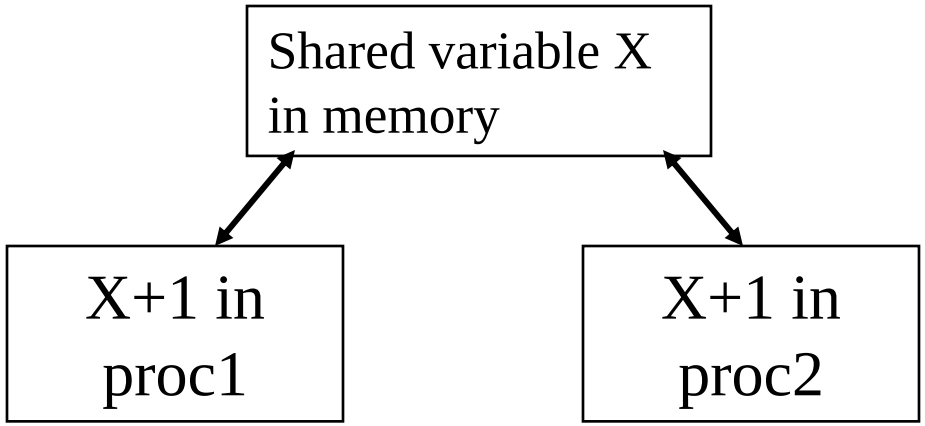
- Tools can be developed to make any system appear to look like a different kind of system
  - distributed memory systems can be programmed as if they have shared memory, and vice versa
  - such tools do not produce the most efficient code, but might enable portability
- HOWEVER, the most natural way to program any machine is to use tools & languages that express the algorithm explicitly for the architecture.

Shared Memory Programming: OpenMP

- Shared memory systems (SMPs, cc-NUMAs) have a single address space:
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, 'SIMD' kinds of codes
  - OpenMP is the new 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



OpenMP Example #1: Parallel Loop

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

- 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
```

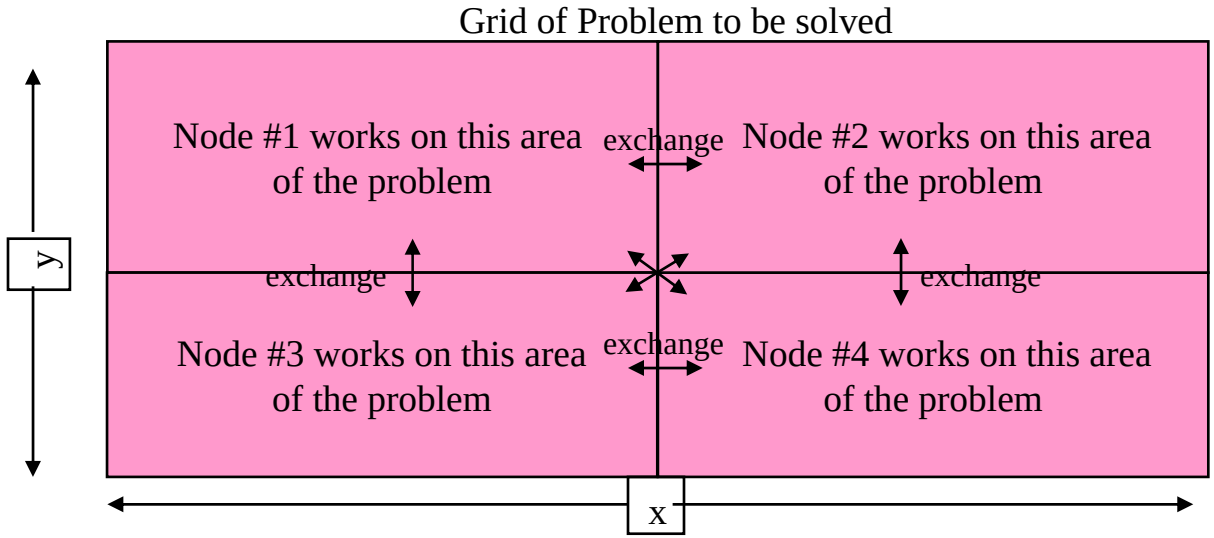
- In this loop, each processor needs its own private copy of the variable TEMP. If TEMP were shared, the result would be unpredictable since multiple processors would be writing to the same memory location.

Distributed Memory Programming: MPI

- Distributed memory systems have separate address spaces for each processor
  - Local memory accessed faster than remote memory
  - Data must be manually decomposed
  - MPI is the standard for distributed memory programming (library of subprogram calls)
  - Older message passing libraries include PVM and P4; all vendors have native libraries such as SHMEM (T3E) and LAPI (IBM)

Data Decomposition

- For distributed memory systems, the 'whole' grid or sum of particles is decomposed to the individual nodes
  - Each node works on its section of the problem
  - Nodes can exchange information



MPI Example

```
#include <. . . .>
#include "mpi.h"
main(int argc, char **argv)
{
   char message[20];
   int i, rank, size, type = 99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      strcpy(message, "Hello, world");
      for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
   }
   else
      MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
   printf( "Message from process = %d : %.13s\n", rank,message);
   MPI_Finalize();
}
```

MPI Example

```
#include <. . . .>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, type = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
  printf( "Message from process = %d : %.13s\n", rank,message);
  MPI_Finalize();
}
```

Initialize MPI environment

An implementation may also use this call as a mechanism for making the usual argc and argv command-line arguments from “main” available to all tasks (C language only).

Close MPI environment

MPI Example

```
#include <. . . .>
#include "mpi.h"
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, type = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
  printf( "Message from process = %d : %.13s\n", rank,message);
  MPI_Finalize();
}
```

Returns number of Processes
This, like nearly all other MPI functions, must be called after MPI_Init and before MPI_Finalize. Input is the name of a communicator (MPI_COMM_WORLD is the default communicator) and output is the size of that communicator.

Returns this process' number, or rank
Input is again the name of a communicator and the output is the rank of this process in that communicator.

MPI Example

```
#include <. . . .>
#include "mpi.h"
main(int argc, char **argv)
{
   char message[20];
   int i, rank, size, type = 99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      strcpy(message, "Hello, world");
      for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
   }
   else
      MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
   printf( "Message from process = %d : %.13s\n", rank,message);
   MPI_Finalize();
}
```

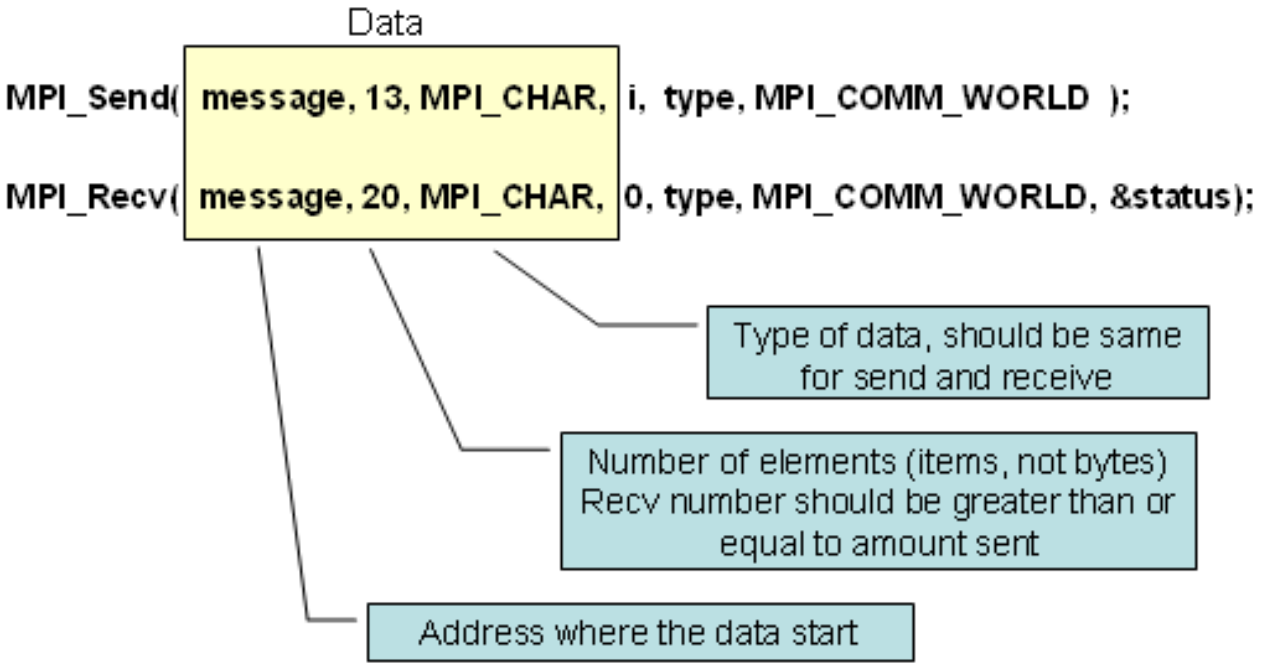
Send a message
Blocking send of data in the buffer.

Receive a message
Blocking receive of data into the buffer.

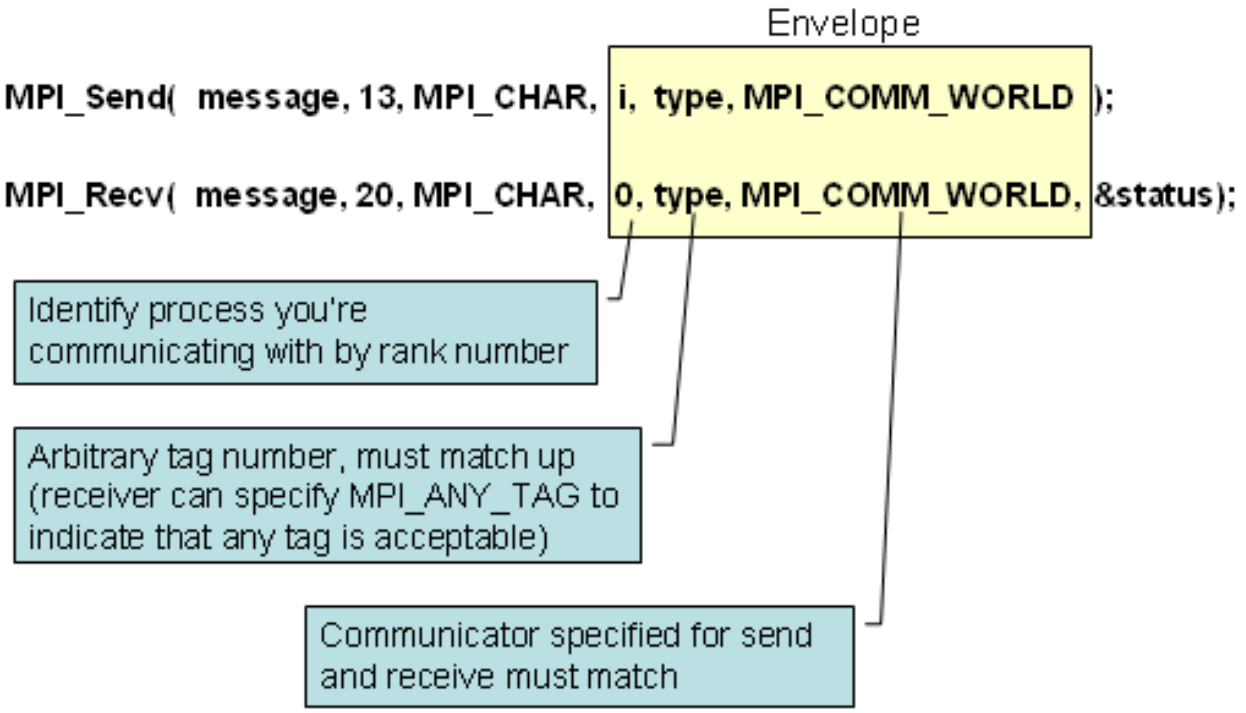
MPI: Sends and Receives

- Real MPI programs must send and receive data between the processors (communication)
- The most basic calls in MPI (besides the initialization, rank/size, and 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



Message Passing Communication



Questions?