Introduction to Parallel Programming

Linda Woodard
CAC
19 May 2010

Introduction to Parallel Computing on Ranger

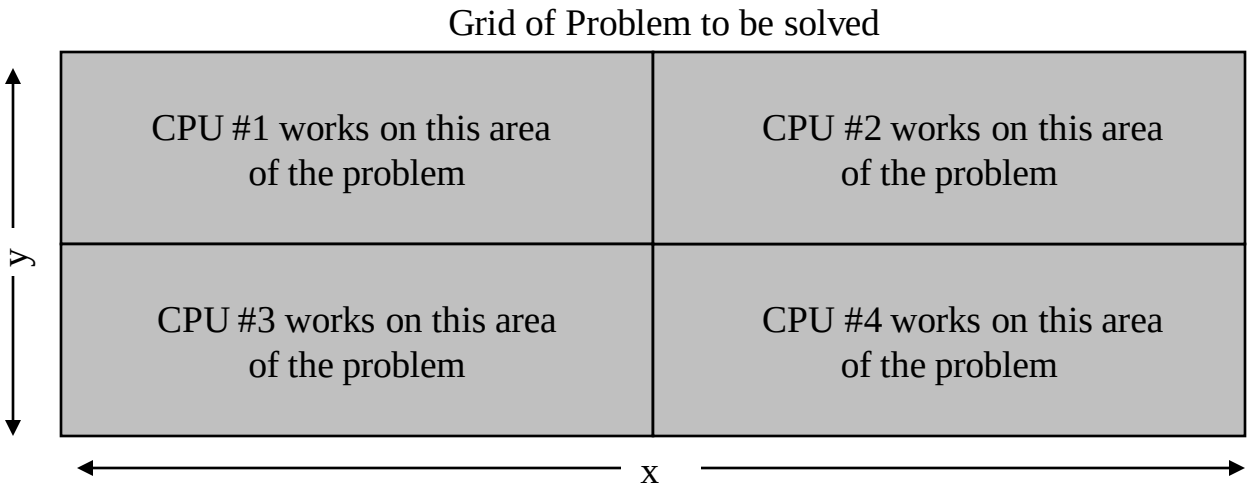
What is Parallel Programming?

Using more than one processor or computer to complete a task

– Each processor works on its section of the problem (functional parallelism)

– Each processor works on its section of the data (data parallelism)

– Processors can exchange information



Forest Inventory with Ranger Bob

Why Do Parallel Programming?

- 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

Forest Inventory with Ranger Bob








Terminology (1)

- **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.

Terminology (2)

- **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 someway as a single resource.

- **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.

Limits of Parallel Computing

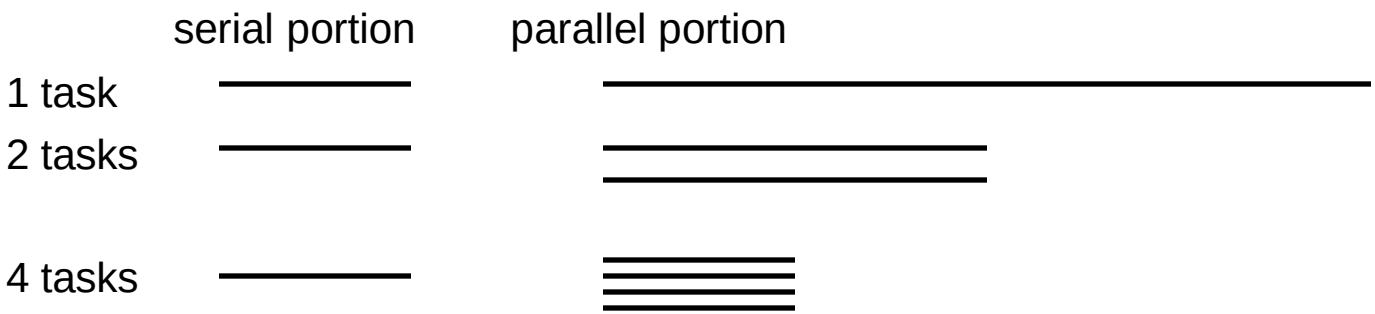
- Theoretical Upper Limits
  - Amdahl's Law

- Practical Limits
  - Load balancing (waiting)
  - Conflicts (accesses to shared memory)
  - Communications
  - I/O (file system access)

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

$$t\,n = (f\,p / N + f\,s\,)t\,1$$

  - Where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors
    - *t 1* = time to run on one processor

Limit Cases of Amdahl's Law

- Speed up formula:

$$S = 1 / (f_s + f_p / N)$$

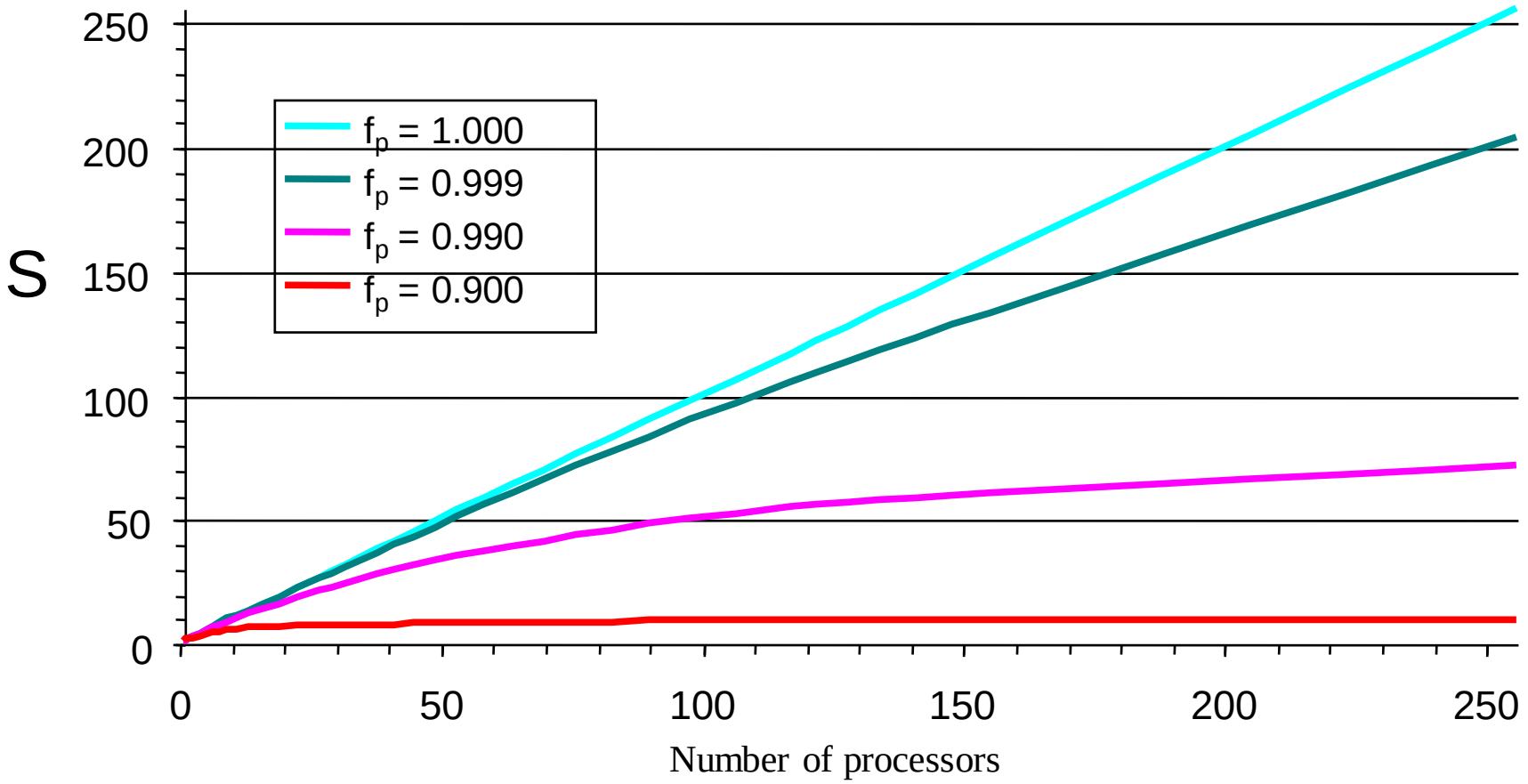
Where

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

Case:

1. $f_s = 0$, $f_p = 1$, then S = N
2. N → infinity: $S = 1/f_s$; if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors.

Ilustration of Amdahl's Law



More 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.

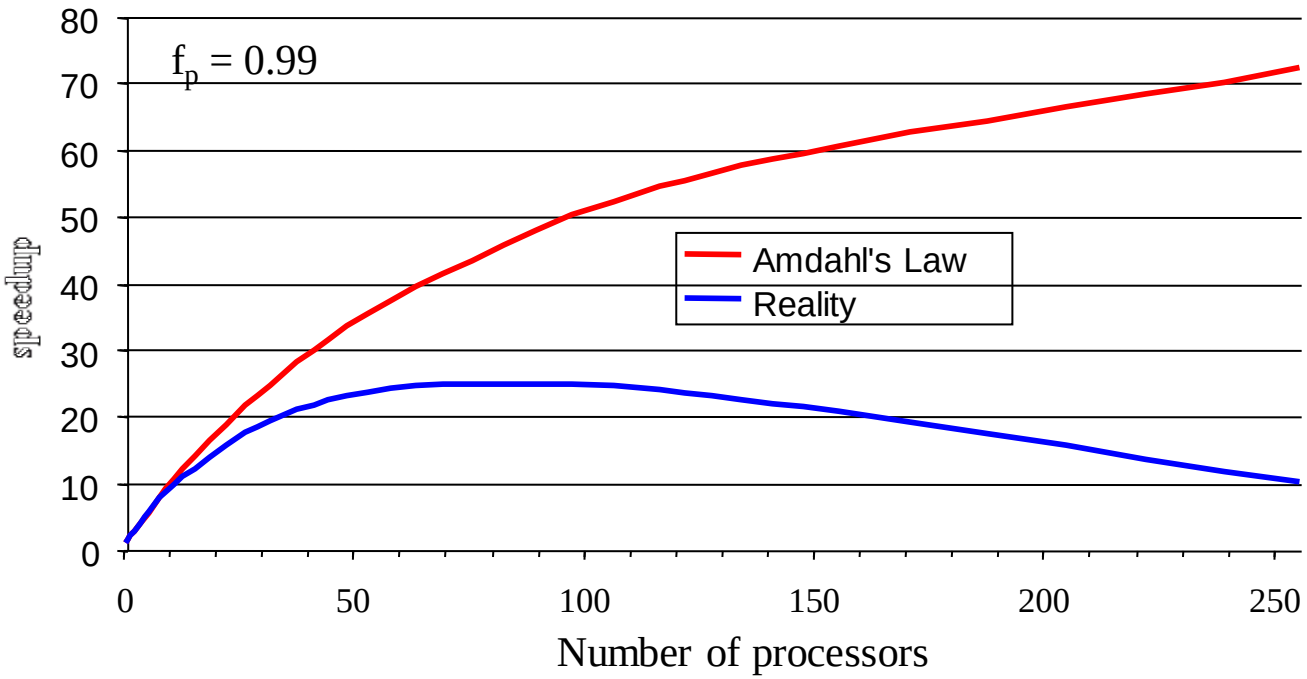
- **parallel overhead**: the amount of time required to coordinate parallel tasks, as opposed to doing useful work, including time to start and terminate tasks, communication, move data.

- **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).

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

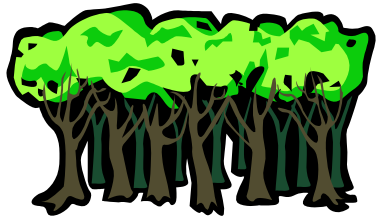
S



Forest Inventory with Ranger Bob





Is it really worth it to go Parallel?

- 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? Is your problem really `large'?
  - Is there a library that does what you need (parallel FFT, linear system solving)
  - Will the code be used more than once?

Types of Parallel Computers (Flynn's taxonomy)

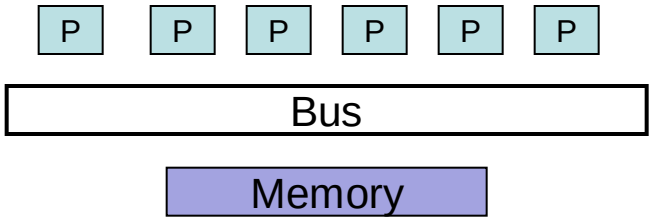
Instruction Stream \ Data Stream	Single	Multiple
Single	Single Instruction Single Data SISD	Single Instruction Multiple Data SIMD
Multiple	Multiple Instruction Single Data MISD	Multiple Instruction Multiple Data MIMD

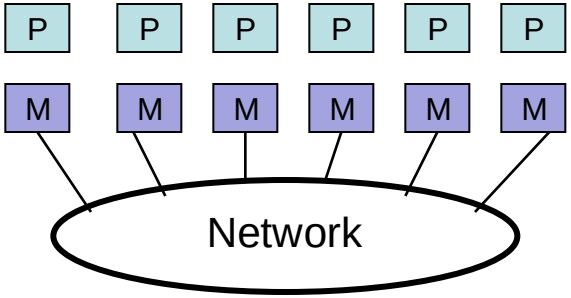
Types of Parallel Computers (Memory Model)

- Nearly all parallel machines these days are multiple instruction, multiple data (MIMD)

- A much more useful way to classify modern parallel computers is by their memory model
  - shared memory
  - distributed memory

Shared and Distributed Memory Models





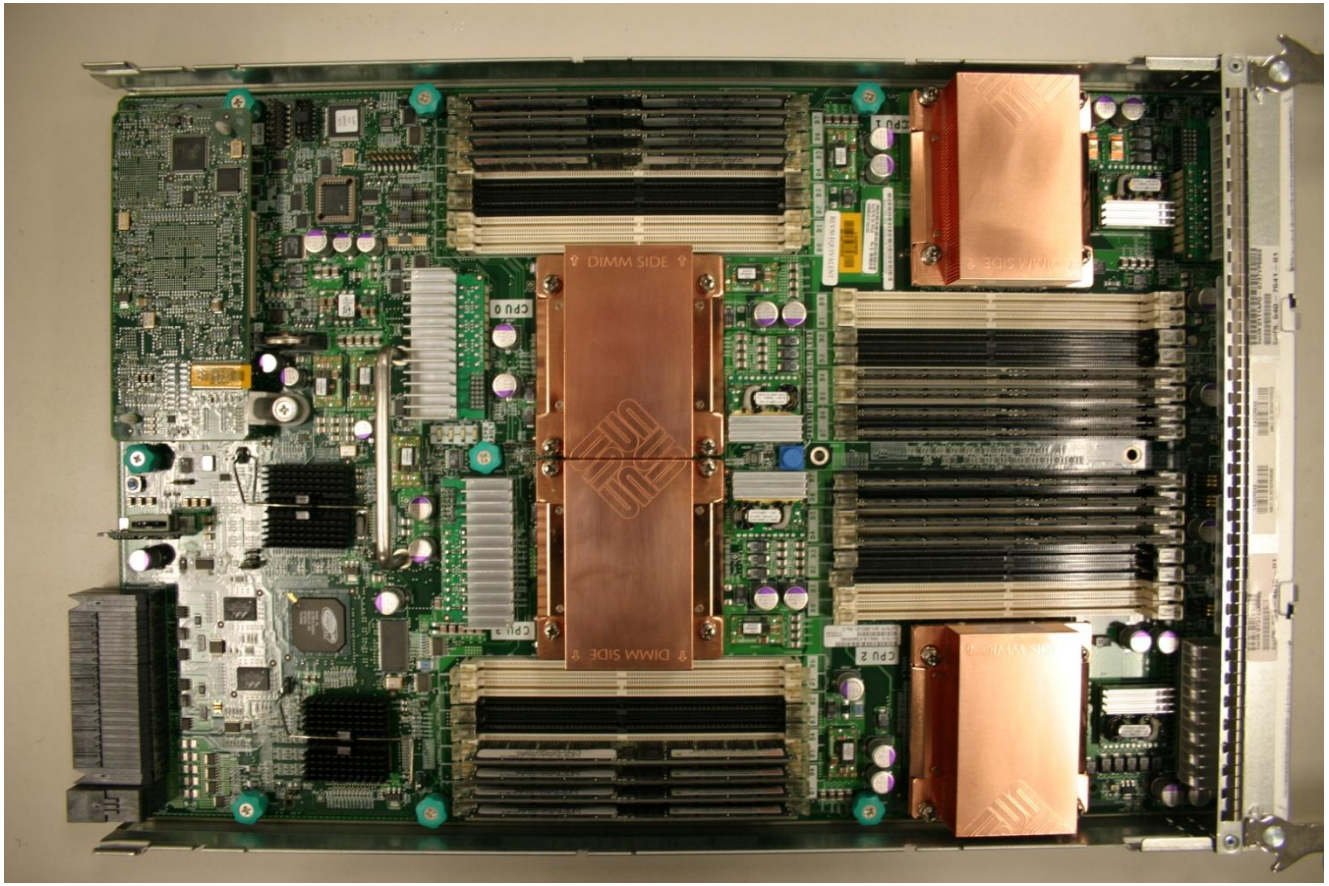
Shared memory: single address space. All processors have access to a pool of shared memory; easy to build and program, good price-performance for small numbers of processors; predictable performance due to UMA .(example: SGI Altix)

Methods of memory access :
- Bus
- Crossbar

Distributed memory: each processor has its own local memory. Must do message passing to exchange data between processors. cc-NUMA enables larger number of processors and shared memory address space than SMPs; still easy to program, but harder and more expensive to build. (example: Clusters)

Methods of memory access :
- various topological interconnects

Ranger



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 and languages that express the algorithm explicitly for the architecture.

Programming Parallel Computers

- Programming single-processor systems is (relatively) easy because they have a single thread of execution and a 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
- Both shared memory and distributed memory parallel computers can be programmed in a data parallel, SIMD fashion and they also can perform independent operations on different data (MIMD) and implement task parallelism.

Data vs. Functional Parallelism

- 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

  – Usually 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.

- 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

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 in two.

```
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
```

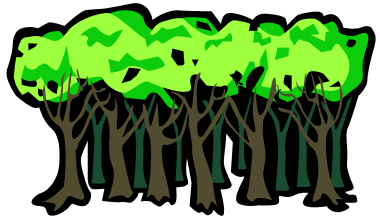
CPU B

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

Forest Inventory with Ranger Bob





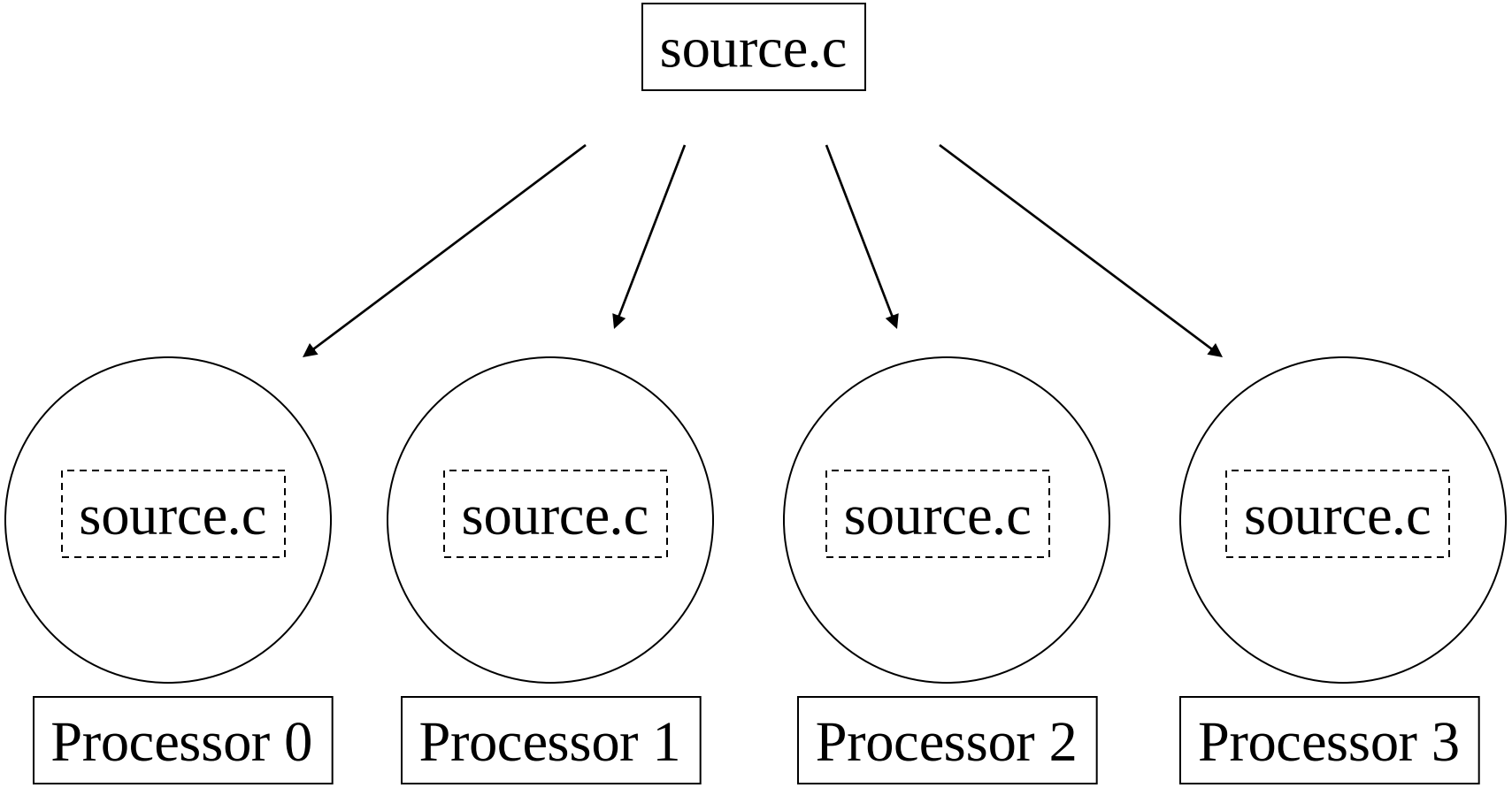



Single Program, Multiple Data (SPMD)

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

SPMD Programming Model



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
```

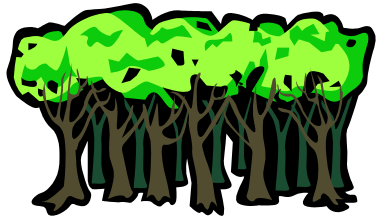
CPU B

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

Forest Inventory with Ranger Bob





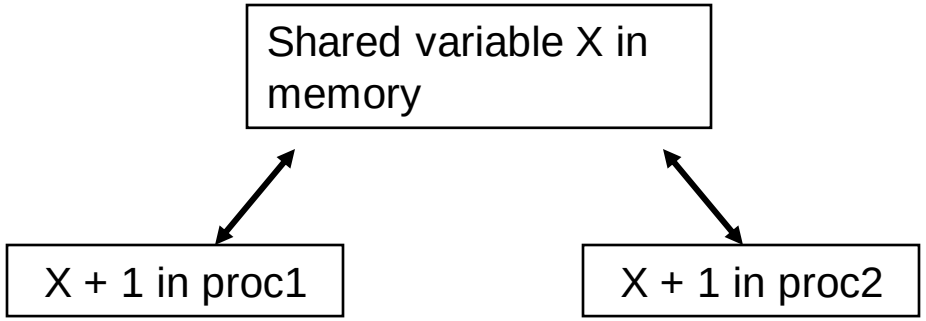
Shared Memory Programming: OpenMP

- Shared memory systems (SMPs and 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

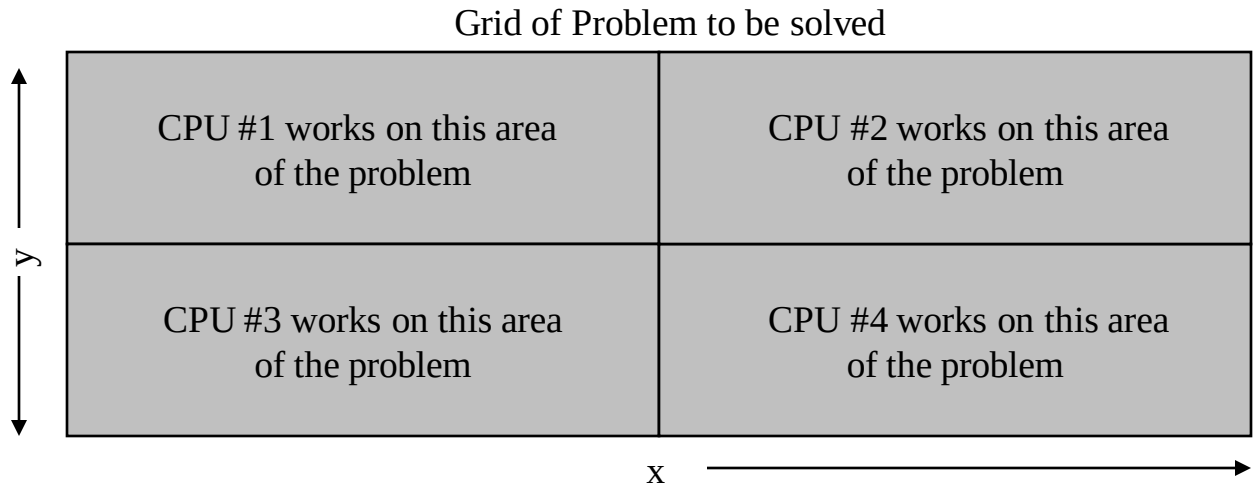
Distributed memory systems have separate address spaces for each processor

- Local memory accessed faster than remote memory
- Data must be manually decomposed
- MPI is the standard for distributed memory programming (library of subprogram calls)

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: Sends and Receives

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.

Programming Multi-tiered Systems

- Systems with multiple shared memory nodes are becoming common for reasons of economics and engineering.

- Memory is shared at the node level, distributed above that:
  - Applications can be written using OpenMP + MPI
  - Developing apps with only MPI usually possible