Programming Parallel Computers

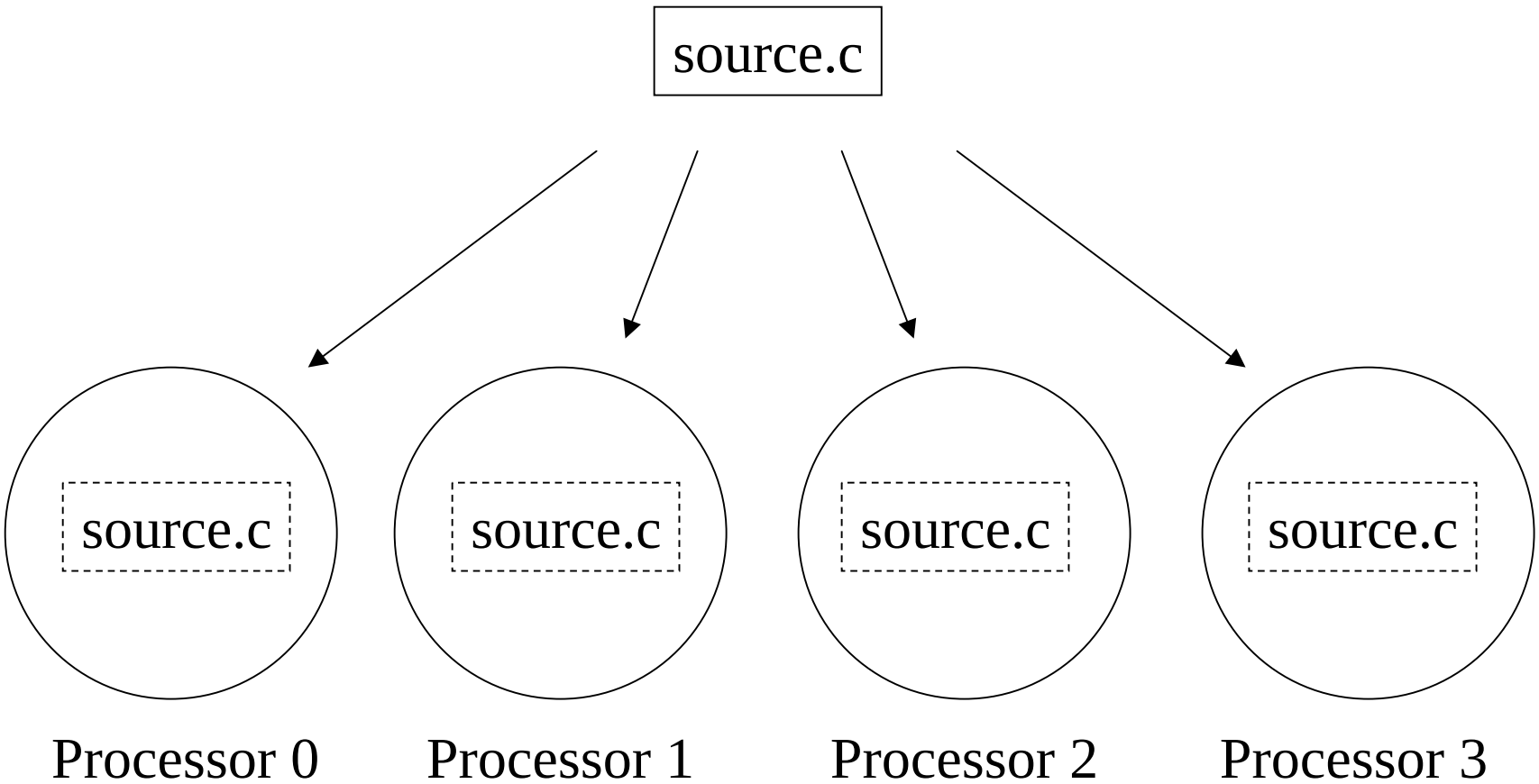
- **Programming single-processor systems is (relatively) easy due to:**
  - single thread of execution
  - single address space
- **Programming for shared memory systems can be benefit from the single address space**
- **Programming for distributed memory systems is the most difficult to due multiple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

- **Both parallel systems (shared memory and distributed memory) offer ability to perform independent operations on different data (MIMD) and implement task parallelism**
- **Both can be programmed in a data parallel, SIMD fashion**

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 synch with each other periodically
- **MPMD: more general, and possible in hardware, but no system/programming software enables it**

SPMD Programming Model



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

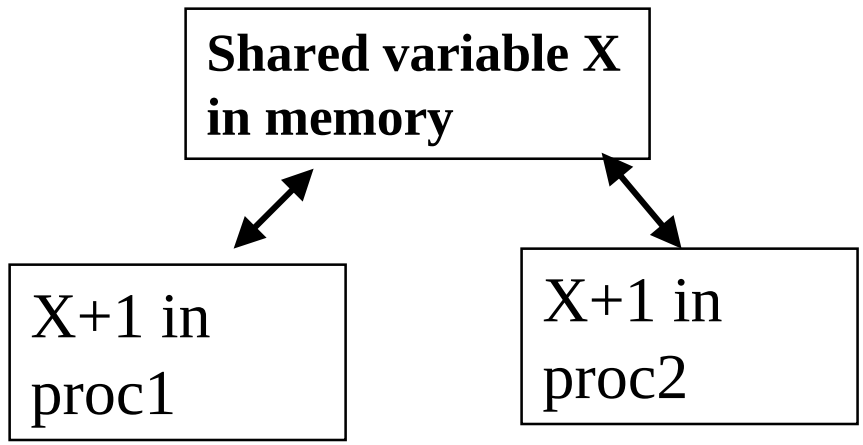
- **Shared memory systems (SMPs, cc-NUMAs) have a single address space:**
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
  - OpenMP is the new standard for shared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

Process 1 and 2

1) **read X**
2) **compute X+1**
3) **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.

OpenMP Example #3: Reduction variables

```
ASUM = 0.0
APROD = 1.0
!$OMP PARALLEL DO REDUCTION(+:ASUM)
REDUCTION(*:APROD)
do I=1,n
      ASUM = ASUM + A(I)
  APROD = APROD * A(I)
enddo
!$OMP END PARALLEL DO
```

- Variables used in collective operations over the elements of an array can be labeled as REDUCTION variables.

- **Each processor has its own copy of ASUM and APROD. After the parallel work is finished, the master processor collects the values generated by each processor and performs global reduction.**

- **More on OpenMP coming in a few weeks...**

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 new 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)

MPI Example #1

- **Every MPI program needs these:**

```
#include <mpi.h> /* the mpi include file */
/* 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

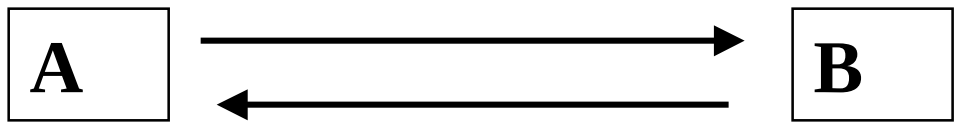
```
#include
#include "mpi.h”

int main(argc,argv)
int argc;
char *argv[];
{
     int myid, numprocs;
     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\n",myid);
     printf("Numprocs is %d\n",numprocs);
     MPI_Finalize();
}
```

MPI: Sends and Receives

- **Real 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 program communicate by passing messages**



- **Basic message passing primitives**
- **Send (parameters list)**
- **Receive (parameter list)**
- **Parameters depend on the library used**

Flavors of message passing

- **Synchronous used for routines that return when the message transfer is complete**
- **Synchronous send waits until the complete message can be accepted by the receiving process before sending the message (send suspends until receive)**
- **Synchronous receive will wait until the message it is expecting arrives (receive suspends until message sent)**
- **Also called blocking**

Nonblocking message passing

- **Nonblocking sends (or receive) return whether or not the message has been received (sent)**
- **If receiving processor is not ready, message may wait in a buffer**

MPI Example #3: Send/Receive

```
#include "mpi.h"
/*************************************************************
This is a simple send/receive program in MPI
*************************************************************/
int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs, 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();
}
```

- **More on MPI coming in several weeks...**

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 MPI only might be possible