

## Outline

- Announcements
  - HWII due Today
  - HWIII due Monday
  - Wed. and Fri. in 484 Rhodes
- Parallel Computers
- Types of Parallelism
- Message Passing with MPI
- Parallelizing RAD1D

---

---

---

---

---

---

---

---

## Basic Idea of Parallel Computing

- If you have N computers
  - Divide your problem into N pieces
  - Give each piece to its own computer
  - Solve your problem N-times faster

---

---

---

---

---

---

---

---

## Parallel Computers

- 2-4 processor workstations are becoming common
- Sun, SGI make systems with up to 32 processors
- Research systems:
  - Up to 9632 processors!

---

---

---

---

---

---

---

---

## Parallel Computers

- Symmetric Multi-Processors (SMP)
  - Several processors sharing same memory
  - Ex: 2x Dell Workstation, 32x SGI Origin
- Distributed Memory
  - Several independent machines linked by a network
  - Ex: "Beowulf" cluster
- Hybrid Systems
  - Several machines linked over network, each machine is SMP
  - Ex: Cornell's Velocity, ASCI White

---

---

---

---

---

---

---

---

## Using Parallel Computers: Easy Way

- If your problem involves applying a simple model to several different data sets
  - Distribute data among machines
  - Run the program
  - If it takes T seconds to run on 1 machine
    - Your problem will run in T/N seconds plus time to distribute data (D)

---

---

---

---

---

---

---

---

## Speed-Up

- Parallel performance is commonly measured as speed-up:
  - $S(N) = T_1 / T_N$
  - Ideal case:  $T_N = T_1 / N$ , then  $S(N) = N$
  - Easy parallelism
    - $T_N = T_1 / N + D$
    - If D is small, S(N) is very close to N

---

---

---

---

---

---

---

---

## Hard Parallelism

- You have a single very large problem
  - large in memory or time
- To parallelize:
  - you must divide your problem (data or program) into pieces
  - But, pieces are not independent
    - computers must share data
    - you must determine what data to share and when

---

---

---

---

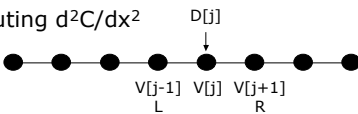
---

---

---

---

## Example:

- Computing  $d^2C/dx^2$ 

- ```
for(j=0;j<M;j++){
  - L=V[j-1] or V[M-1]
  - R=V[j+1] or V[0]
  - D[j]=(R-2V[j]+L)/dx/dx
}
```

---

---

---

---

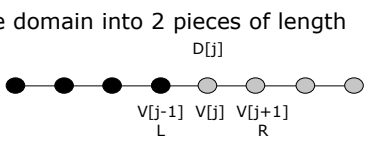
---

---

---

---

## Computing $d^2C/dx^2$

- Divide domain into 2 pieces of length  $M/2$ 

- ```
for(j=0;j<M/2-1;j++){
  - L=V[j-1] or V[M/2-1] on neighbor
  - R=V[j+1] or V[0] on neighbor
  - D[j]=(R-2V[j]+L)/dx/dx
}
```

---

---

---

---

---

---

---

---

## Computing $d^2C/dx^2$

- $T_N = T_1/N + 2*N*C$ , where  $C$ =time to send a double
- $S(N) = T_1/(T_1/T_N + 2NC)$
- $= N/(1 + 2N^2C/T_1)$
- Let  $T_1 = M\alpha C$ ,  $M$ =grid points,  $\alpha$ =compute/communicate
- $S(N) = N/(1 + 2N^2/(M\alpha))$

---

---

---

---

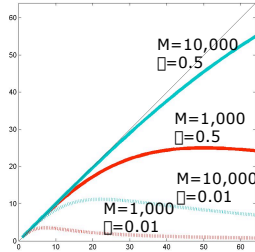
---

---

---

---

## Amdahl's Law of Parallel Processing



- Increasing  $N$  will reduce speed-up
  - There's a limit to parallel performance
- Increasing  $M$  will improve speed-up
  - Parallel is more attractive for big problems
- Increasing  $\alpha$  will improve speed-up
  - Hardware is critical

---

---

---

---

---

---

---

---

## Message Passing with MPI

- We need a way to share data
- Message Passing Interface (MPI)
  - library for communication
  - device independent
    - same MPI code will run on cluster or SMP
  - well-specified, so very portable
  - versions available for all systems
    - MPICH, LAM are free

---

---

---

---

---

---

---

---

## Using MPI

- First, set it up:
  - MPI\_Init(&argc, &argv);
    - Starts MPI
  - MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
    - Gets "rank" of this job--processor number
  - MPI\_Comm\_size(MPI\_COMM\_WORLD, &N);
    - Gets number of jobs

---

---

---

---

---

---

---

---

## Using MPI

- Point-to-Point Communication
  - MPI\_Send(ovec, 10, MPI\_DOUBLE, 1, tag, comm)
    - Sends 10 doubles in array ovec to process 1
    - tag is a message label
    - comm is a communicator--a subgroup of processes (MPI\_COMM\_WORLD is everyone)
  - MPI\_Recv(ivec, 10, MPI\_DOUBLE, 0, tag, comm, &status)
    - receives 10 doubles from process 0
    - status may contain error info

---

---

---

---

---

---

---

---

## Point-to-Point Communication

- For every send, there must be a receive
- Send waits until Recv is posted & vice-versa
  - Called blocking
- Allows for the possibility of deadlock
  - Recv's waiting for Sends that will never be posted

---

---

---

---

---

---

---

---

## Collective Communication

- Sometimes, you need to send data to all processes
- Ex: Computing sum of an array
  - Each processor computes the sum of its array
  - The partial sums must be added together and answer distributed to everyone
  - `MPI_Allreduce(&locsum, &allsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);`

---

---

---

---

---

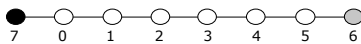
---

---

---

## Parallelizing RAD1D

- Divide domain among N processors
  - Each proc gets M/N grid points



Proc P #	Global #
0	M/N*P+0
1	M/N*P+1
:	:
M/N-1	M/N*(P+1)-1
M/N	M/N*(P+1)
M/N+1	M/N*(P)-1

Need a subroutine to get last two entries from neighbors

---

---

---

---

---

---

---

---

## Parallelizing RAD1D

- Each proc will get N and P
- Each proc will read inputs, taking what it needs
- Compute as usual ( $m=M/N$ ) updating arrays with communication routine as needed:

```

Update(C,m,P,N);
for(j=0;j<m;j++){
  left=j-1; if(j==0){left=m+1;}
  right=j+1;
  RHS[j]=C[j]+0.5*lambda[j]*(C[right]-C[left]);
}
Update(RHS,m,P,N);

```

---

---

---

---

---

---

---

---

## Parallelizing RAD1D

```
Update(Arr, m, P, N);
if(N is odd){Error}
left=P-1; if(left<0){left=N-1;};
right=P+1;if(right==N){right=0;};
if(P is odd)
  MPI_Send(&Arr[0], 1, MPI_DOUBLE, left, MPI_ANY_TAG,
  MPI_COMM_WORLD);
  MPI_Recv(&Arr[m+1], 1, MPI_DOUBLE, left, MPI_ANY_TAG,
  MPI_COMM_WORLD, &stat);
  MPI_Send(&Arr[m-1], 1, MPI_DOUBLE, right, MPI_ANY_TAG,
  MPI_COMM_WORLD);
  MPI_Recv(&Arr[m], 1, MPI_DOUBLE, right, MPI_ANY_TAG,
  MPI_COMM_WORLD, &stat);
else
  MPI_Recv(&Arr[m], 1, MPI_DOUBLE, right, MPI_ANY_TAG,
  MPI_COMM_WORLD, &stat);
  MPI_Send(&Arr[m-1], 1, MPI_DOUBLE, right, MPI_ANY_TAG,
  MPI_COMM_WORLD);
  MPI_Recv(&Arr[m+1], 1, MPI_DOUBLE, left, MPI_ANY_TAG,
  MPI_COMM_WORLD, &stat);
  MPI_Send(&Arr[0], 1, MPI_DOUBLE, left, MPI_ANY_TAG,
  MPI_COMM_WORLD);
```

---

---

---

---

---

---

---

---