Introduction to Parallel Computing

Jay Boisseau
Texas Advanced Computing Center
July 14, 2008

(Pre-)TACC History

- Center for High Performance Computing (CHPC) created by UT System in 1986
- Massive downscaling, moved into UT Austin IT Department in early 90's
- Joined National Partnership for Advanced Computational Infrastructure led by San Diego Supercomputer Center (SDSC) in 1997
- Became TACC: new mission, vision, goals, and strategy in 2001
  - External reviews emphasized importance of HPC
  - VPR Office assumed ownership, set goal of world class

TACC's Growth and Leadership in Supercomputing

- TACC has 7 years of success leading to Ranger:
  - First terascale cluster in the NSF program (2003)
  - Joined NSF TeraGrid (2004)
  - Deployed (current) Lonestar (2006)
  - Growing R&D activities in HPC, Vis, and Grid Computing
- TACC now has the expertise, experience, passion, and partners to support world-class science

Ranger: What is it?

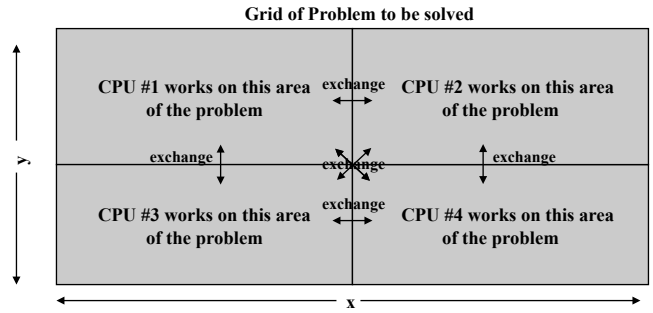
- Ranger is most powerful HPC system for open science
- Results from over 2 ½ years of initial planning and deployment efforts
- Funded by the National Science Foundation as part of a program to offer very high-end HPC for open science
- Oh yeah, it's a Texas-sized supercomputer

Ranger: The Big Numbers

- $59M Awarded by NSF in September 2006
- 579Teraflops (1/2 petaflop) peak performance
  - 1.7 petabytes disk
  - 123 terabytes memory
- 500 million processor hours per year
  - 200,000+ years of computational work over lifetime.

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
  - Processors can exchange information



Why Do Parallel Computing?

- To compute beyond the limits of single CPU systems
  - achieve more performance
  - utilize more memory
- Parallel computing allows one to:
  - solve problems that can't be solved in a reasonable time with a single processor
  - solve problems that don't fit on a single processor system (or a single server)
- So we can …
  - solve larger problems
  - solve problems faster
  - solve more problems/cases

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - load balancing
  - non-computational sections
- Other Considerations
  - time to develop/rewrite code
  - time to debug & optimize 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

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

  - Effect of multiple processors on speed up

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

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

Limit Cases of Amdahl's Law

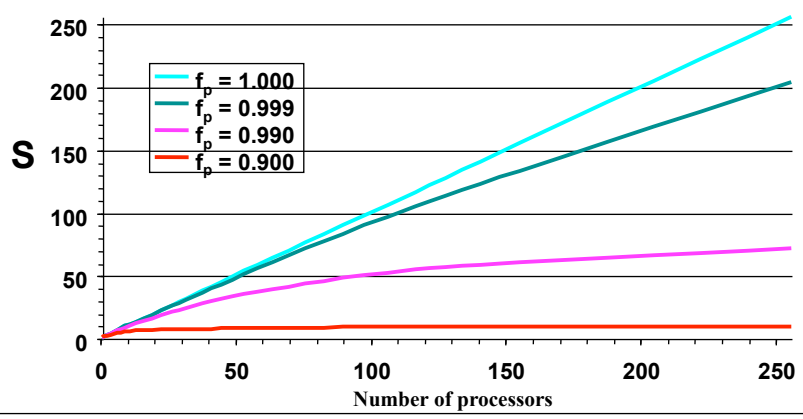
- Speed up formula:

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

  - 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.

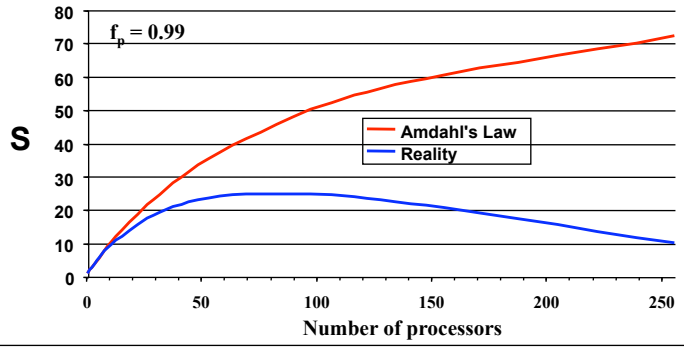
Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance.



Practical Limits: Amdahl's Law vs. Reality

Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for *communications*. In reality, communications will result in a further degradation of performance.



Practical Limits: Amdahl's Law vs. Reality

- 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 codes is difficult!
  - Load balance is important
  - Communication can limit parallel efficiency
  - Serial time can dominate
- Is it worth your time to parallelize your code?
  - Do the CPU, or time-to-solution, requirements justify parallelization?
  - Does the problem size need to be larger than you can currently run?
  - Will the code be used more than once?

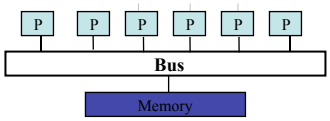
Types of Parallel Computers

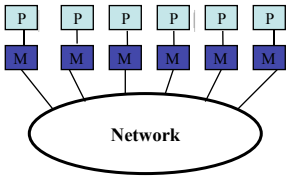
- Until recently, Flynn's taxonomy was commonly used to classify parallel computers into one of four basic types:
  - Single instruction, single data (SISD): single scalar processor
  - Single instruction, multiple data (SIMD): “array processors,” Connection Machine, MasPar
  - Multiple instruction, single data (MISD): various special purpose machines
  - Multiple instruction, multiple data (MIMD): Nearly all parallel machines these days

Types of Parallel Computers

- However, since the MIMD model "won," a much more useful way to classify modern parallel computers is by their memory model
  - *shared* memory
  - *distributed* memory

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

Reasons for Each System

- **SMPs**: easy to build, easy to program, good price-performance for small numbers of processors; predictable performance due to UMA
- **cc-NUMAs (Distributed Shared memory machines)** : enables larger number of processors and shared memory address space than SMPs while still being easy to program, but harder and more expensive to build
- **Distributed memory MPPs and clusters**: easy to build and to scale to large numbers of processors, but hard to program and to achieve good performance
- **Multi-tiered/hybrid/CLUMPS**: combines best (worst?) of all worlds… but maximum scalability!

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

Programming Parallel Computers

- 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

Types of Parallelism: Two Extremes

- Data parallelism
  - Each processor performs the same task on different data
- Task parallelism
  - Each processor performs a different task
- Most applications fall somewhere on the continuum between these two extremes

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
```

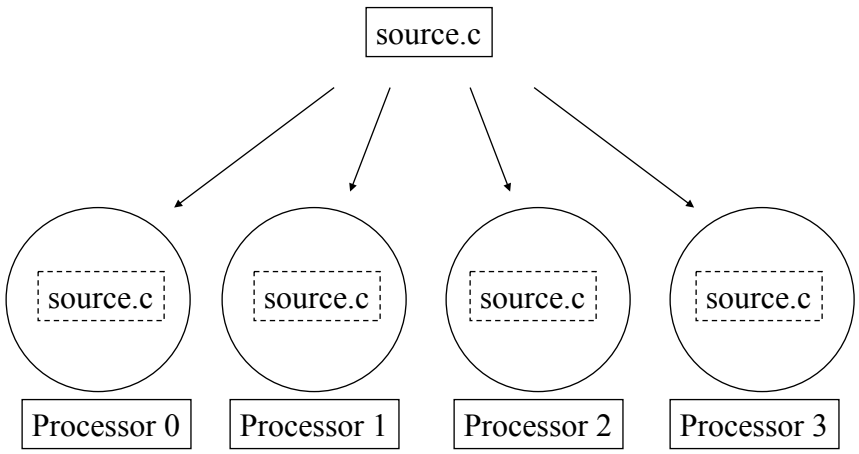
CPU B

```
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 HPC system/programming software generally enables it

SPMD Programming Model



Shared Memory vs. Distributed Memory

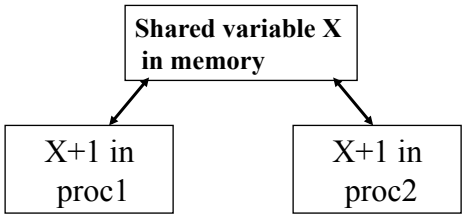
- Tools can be developed to make any system appear to look like a different kind of system
  - distributed memory systems can be programmed as if they have shared memory, and vice versa
  - such tools do not produce the most efficient code, but might enable portability
- However, the most natural way 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 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
```

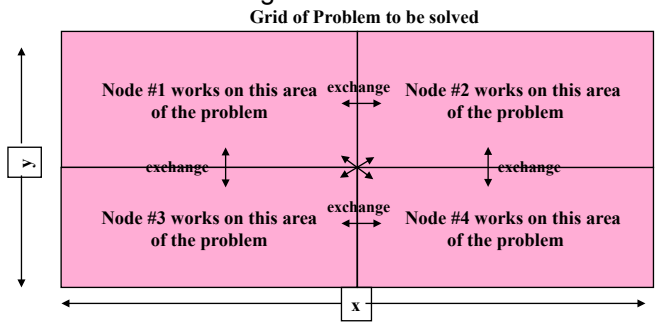
- In this loop, each processor needs its own private copy of the variable TEMP. If TEMP were shared, the result would be unpredictable since multiple processors would be writing to the same memory location.

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



Typical Data Decomposition

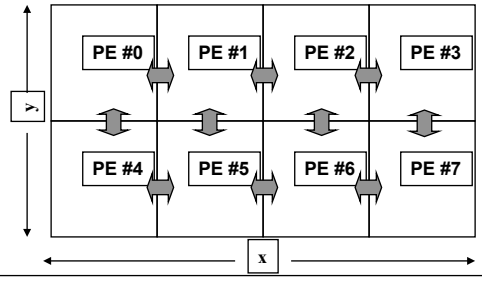
- Example: integrate 2-D propagation problem:

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> /* the mpi include file */
int main(int argc, char *argv[])
{
  /* 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

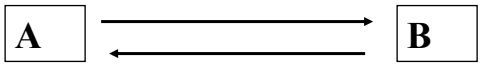
```
#include
#include "mpi.h"

int main(int argc, char *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," of ",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 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

- 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

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();
}
```


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