

HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

MTH 410/510

OUTLINE

- Fundamentals of high-performance computing (HPC)
- Basic theory and concepts of parallel programming
- Design and implementation of numerical algorithms using MPI
 - quadrature methods, linear algebra, solution to PDEs
- Applications to computationally intensive problems
 - Examples and applications in applied sciences
- Access to a modern Linux cluster - hands-on experience
 - Intel quad-core Xeon dual processor
- A unique experience at PSU

Course webpage: <http://www.mth.pdx.edu/~daescu/hpc.html>

E-mail: daescu@pdx.edu

Textbook: Parallel Programming with MPI. P.S. Pacheco

Grading: Homework assignments - 60%; Final project - 40%

Motivation

- Analysis and prediction of the behavior of complex systems that are beyond the reach of the experimental research have become feasible through numerical modeling and simulation.
- Scientific computing has gained worldwide recognition as a major component of the scientific research that is equally important to experiment and theory.
- Computational modeling and computer-based simulation are central to progress in applied sciences with applications ranging from molecular biology and nanotechnology to environmental finance and climate change.
 - Example: How much computing is needed to predict tomorrow's weather?



It's a big world!

So we need to "discretize it":

- longitude: 360° at $1/4$ degree resolution: 1440 grid points
- latitude: 180° at $1/4$ degree resolution: 720 grid points
- altitude: 10 – 20 km about 100 grid points
- variables: wind, temperature, pressure, humidity

Total size of the state: $1440 \times 720 \times 100 \times 6 \sim 10^9$

Assume a time step of 1 minute - for 24 hours about 1500 time steps

Assume 100 floating point operations per grid point

Total computational load $\sim 10^{14}$ operations

Assume 10^9 calculations per second

Total CPU required: ~ 26 hours (more than one day!)

Parallel (super) computers: a computer or collection of computers with multiple processors that can work together on solving a problem.

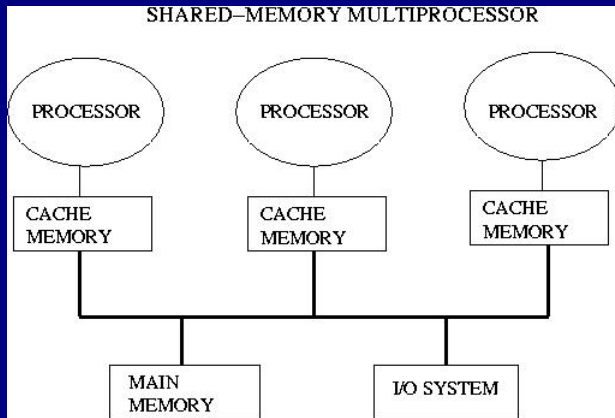
A process is an instance of a program or a subprogram that is executed autonomously on a physical processor.

A program is parallel if, at any time during its execution, it can comprise more than one process.

Processes can be created and destroyed and their interaction must be coordinated.

- Shared Memory Systems (SMP)

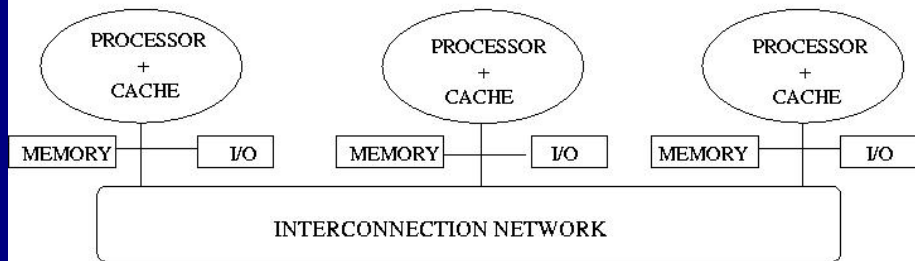
- 1 Memory and resources are shared by multiple CPUs.
- 2 Any memory update is seen by all processors.
- 3 Example: multicore CPUs



- Distributed Memory Systems (DMS)

- 1 computing units connected in a network
- 2 each processor sees only its memory
- 3 explicit messages are used to pass information among processors

DISTRIBUTED-MEMORY ARCHITECTURE



- Distributed Shared Memory Systems (DSMS)
 - 1 Combination of shared and distributed memory systems
 - 2 Example: Beowulf cluster

HPC: Multiple processors connected in a single computing system.

Computer cluster - a group of computers connected to each other

- 1 improve performance over that provided by a single computer
- 2 flexibility and power at low price
- 3 dominate the world of supercomputing



23x2 Quad Core 5320 Processor 2X4MB Cache, 1.86GHz, Xeon
1066MHz. Memory: 8GB (4X2GB)

Measuring performance: X is n times faster than Y

$$\frac{\text{Execution time Y}}{\text{Execution time X}} = n$$

CPU (Central Processing Unit) time: time used in processing CPU instructions

- User CPU time - CPU time spent in the program
- System CPU time - CPU time spent in the operating system

Time command applied to a program execution

```
90.7u 12.9s 2 : 39 65%
```

User CPU time is 90.7 seconds, system CPU time is 12.9 seconds

Elapsed time is 2 minutes and 39 seconds

Percentage of elapsed time that is CPU time is $(90.7 + 12.9)/159 = 65\%$

(while you were busy thinking your opponent makes a move...)

Amdahl's Law

Speedup that can be gained by an enhancement to a task

$$\text{Speedup} = \frac{\text{CPU of entire task without the enhancement}}{\text{CPU of entire task with enhancement}}$$

Factors:

- the fraction F of the CPU time in the original task that can be converted to take advantage of the enhancement
- the speedup S of the enhanced mode versus the original mode

$$CPU_{new} = CPU_{old} \times \left((1 - F) + \frac{F}{S} \right)$$

Q: What is the speedup gained by using an enhancement that runs 10 times faster than the original but can be used only 40% of the time?

Answer: $F = 0.4$, $S = 10$.

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{10}} = 1.56$$

Challenges of parallel processing

- limited parallelism available in the program: $x_{k+1} = f(x_k)$
 - 1 software and new algorithms
- long latency remote communication
 - 1 hardware - cache shared data
 - 2 software - minimize communication, improve data locality

Q: Suppose that you want to achieve a speedup of 50 with 100 processors. What fraction of the original computation can be sequential?

Answer (Amdahl's law):

$$n = \frac{1}{(1 - F) + \frac{F}{S}} \Rightarrow F = \frac{1 - \frac{1}{n}}{1 - \frac{1}{S}} = \frac{49}{50} \times \frac{100}{99} \approx 0.9899$$

Only about 1% of the original code may be sequential!

Q: Suppose that a cache memory is 10 times faster than the main memory and suppose that a cache can be used 90% of the time. How much speedup may be gained by using a cache?

Answer:

$$n = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = \frac{1}{0.19} \approx 5.3$$

Consider the matrix addition $\mathbf{A} + \mathbf{B} = \mathbf{C}$.

Algorithm 1

```
for i=1:n
  for j=1:n
    C(i,j) = A(i,j) + B(i,j)
  end
end
```

Algorithm 2

```
for j=1:n
  for i=1:n
    C(i,j) = A(i,j) + B(i,j)
  end
end
```

Q: what is the difference between these algorithms when executed?

Code optimization:

SAXPY, Basic Linear Algebra Subprograms: $y = ax + y$

```
DO I=1,N
    Y(I) = A*X(I)+Y(I)
END DO

M = MOD(N,4)
IF (M.EQ.0) GO TO 40
DO 30 I = 1,M
    SY(I) = SY(I) + SA*SX(I)
30 CONTINUE
    IF (N.LT.4) RETURN
40 MP1 = M + 1
    DO 50 I = MP1,N,4
        SY(I) = SY(I) + SA*SX(I)
        SY(I+1) = SY(I+1) + SA*SX(I+1)
        SY(I+2) = SY(I+2) + SA*SX(I+2)
        SY(I+3) = SY(I+3) + SA*SX(I+3)
    50 CONTINUE
```

Q: which algorithm is better?

Load balancing: An execution is no faster than the slowest process.

Consider the problem of evaluating 11!

If there are three workers (students), what is a fair work balancing?

We want to assign the same amount of work to each processor.

Data mapping:

- block mapping:

$$\{a_0, a_1, a_2\} \rightarrow q_1, \{a_3, a_4, a_5\} \rightarrow q_2, \{a_6, a_7, a_8\} \rightarrow q_3$$

- cyclic mapping

$$\{a_0, a_3, a_6\} \rightarrow q_1, \{a_1, a_4, a_7\} \rightarrow q_2, \{a_2, a_5, a_8\} \rightarrow q_3$$

- block-cyclic mapping

Designing Parallel Programs

Understand the Problem and the Program

- Understand the serial version of the problem
- Can the problem actually be parallelized?
 - 1 Parallelizable Problem: Matrix-vector multiplication: $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$
 - 2 Non-parallelizable Problem: Fibonacci series: $\mathbf{x}_{k+2} = \mathbf{x}_{k+1} + \mathbf{x}_k$
- Understand program's dependence graph:

$$A = B \cdot C \quad (\text{T1})$$

Data dependencies

$$D = A \cdot E + 1 \quad (\text{T2})$$

There is a *flow dependence*:

task $T2$ is dependent on task $T1$ if at least one variable modified by $T1$ is used as input to $T2$.

Control dependencies:

```
a = b · c      (T1)
if a > 0      % control instruction
    d = e · f  (T2)
end if
```

The order of execution can not be a priori determined!

- Two tasks $T1$ and $T2$ can be executed in parallel if and only if they are independent.

Let

I_i - set of input variables to task i (domain)

O_i - set of output variables of task i (range)

- Bernstein's conditions (1966) for parallel tasks:

$$T_i \parallel T_j \iff I_i \cap O_j = \emptyset \quad I_j \cap O_i = \emptyset \quad O_i \cap O_j = \emptyset$$

- Notice that the parallelism relation is commutative but not transitive. We may have $T_i \parallel T_j$, $T_j \parallel T_k$, but $T_i \# T_k$.

Computational efficiency

- Identify the program's hotspots:
 - ① Identify where most of the CPU time is spent during the execution.

$$\mathbf{z} = \mathbf{x} + \mathbf{y} \quad \textit{versus} \quad \mathbf{C} = \mathbf{A} \cdot \mathbf{B}$$

- ② Focus on parallelizing the CPU intensive segments of the program.
- Break the problem into segments of work that can be distributed to multiple tasks.
 - A good partition divides both the computation associated with a problem and the data on which this computation operates.
 - ① Data partitioning (domain decomposition):

$$x[1 : 10] \rightarrow x_1[1 : 5]; x_2[6 : 10]$$

- ② Functional decomposition:

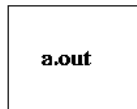
$$x(i + 1) = \textit{model}_x(x(i), u(i)), \quad u(i + 1) = \textit{model}_u(u(i))$$

Parallel Programming Models

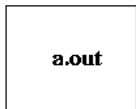
- Single program multiple data (SPMD)
 - 1 a single program is executed by all tasks simultaneously
 - 2 tasks can be executing the same or different instructions within the same program
 - 3 use controls to allow different processors to execute only those parts of the program they are designed to execute.
 - 4 all tasks may use different data
- Multiple program multiple data (MPMD)
 - 1 use multiple executable object files (programs)
 - 2 each task is executing the same or different *programs* as other tasks
 - 3 all tasks may use different data

Parallel Programming Models

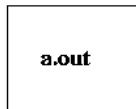
SINGLE PROGRAM MULTIPLE DATA (SPMD)



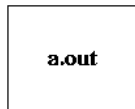
task 1



task 2



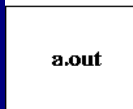
task 3



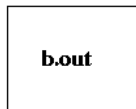
task 4

a single program is executed by all processors

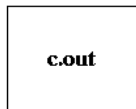
MULTIPLE PROGRAM MULTIPLE DATA (MPMD)



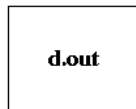
task 1



task 2



task 3



task 4

each processor executes different programs

The MASTER - SLAVE programming model

Distinguish between two types of processes: master and slave

1 Master Process:

- coordinates the tasks for the slave (workers) processes
- sends worker a task when requested
- collects results from workers

2 Slave Process:

- gets task from master process
- performs work (computation)
- sends results to master

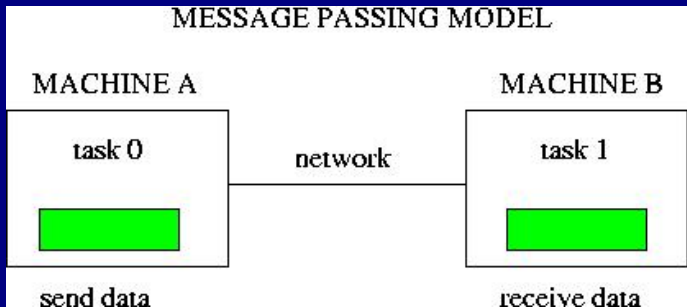
Worker processes do not know before runtime which portion of data they will handle or how many tasks they will perform.

Dynamic load balancing occurs at run time: faster tasks will get more work to do.

How to communicate?

The Message Passing Model

- The most commonly used method of programming distributed-memory systems
- Processes coordinate their activities by explicitly sending and receiving messages
- To implement message passing a library of subroutines is imbedded in the source code
- **A send operation must have a matching receive operation.**



The Message Passing Interface (MPI)

- MPI is a library of subroutines for handling communication and synchronization for programs running on parallel platforms
- First released in 1994 (MPI-1). Part 2 (MPI-2) was released in 1996. MPI is widely available, with both free available and vendor-supplied implementations.
- MPI was designed for high performance on parallel machines, targets distributed memory platforms
- MPI is portable and is defined for both Fortran and C/ C++
- MPI is now the "de facto" industry standard for message passing
- MPI programs usually follow a SPMD format
- Parallelism is explicit: user is responsible for implementing the MPI construct in the parallel code!

The Structure of a MPI Program

Minimum entries in a program to run an MPI job

- Include the MPI library in the program

C: #include "mpi.h"

Fortran: include 'mpif.h'

- Initialize MPI - no MPI functions may be called before this

C: MPI_Init(&argc, &argv);

Fortran: CALL MPI_INIT(IERROR)

- Exit MPI - no MPI functions may be called after this

C: MPI_FINALIZE()

Fortran: CALL MPI_FINALIZE(IERROR)

MPI FUNCTION FORMAT

- C (case sensitive):

```
error = MPI_Xxxxx(argument,...);  
MPI_Xxxxx(argument,...);
```

- Fortran (case unimportant):

```
CALL MPI_XXXXX(argument,...,IERROR)
```

- MPI assigns a rank (an integer number) to each process to identify it.
- The routine `MPI_COM_RANK` returns the rank of the calling process.
- The routine `MPI_COM_SIZE` returns the size or number of processes in the communicator.

COMMUNICATOR SIZE

How many processes are contained within a communicator?

C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
```

PROCESS RANK

Process ID number within the communicator (who am I?)

- Integer between 0 and (size-1)

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
```

FIRST MPI PROGRAM: "Hello world!"

```
program main
include 'mpif.h'
integer rank, size, ierr

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

print*, 'Hello world! I am processor ', rank, ' of ', size

call MPI_FINALIZE(ierr)
stop
end
```

```
#include "mpi.h"
#include "stdio.h"
int main(int argc, char *argv[ ])
{
int rank, nprocs;

MPI_Init(&argc,&argv);

MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

printf("Hello, world. I am processor %d of %d\n", rank, nprocs);
fflush(stdout);

MPI_Finalize();
return 0;
}
```

COMPILE AND RUN THE CODE

- Compilers: mpif77, mpif90, mpicc, mpic++
- Create executable:

```
mpif77 -o hello hello.f  
mpicc -o greetings greetings.c
```

- Run MPI code

```
mpirun -np 8 hello  
mpiexec -np 8 hello  
mpirun -np 8 greetings  
mpiexec -np 8 greetings
```