Parallel Computing: Overview

Jemmy Hu

SHARCNET
University of Waterloo

March 1, 2007

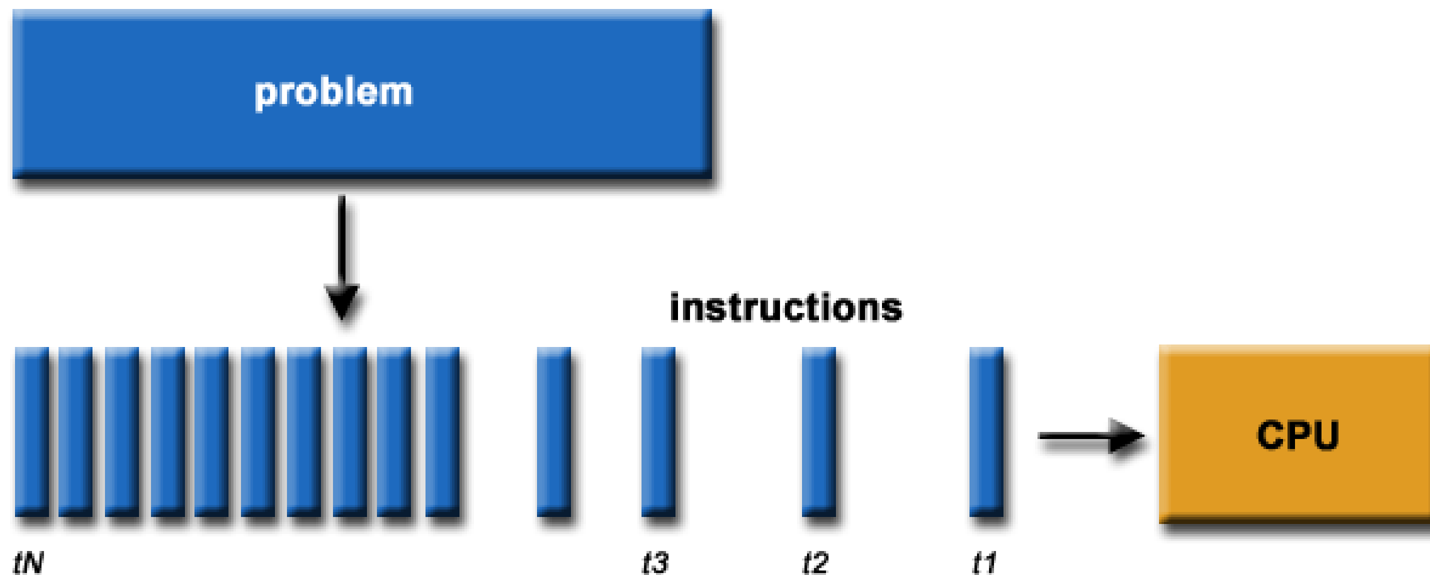
Contents

- **What is Parallel Computing?**
- **Why use Parallel Computing?**
- **Flynn's Classical Taxonomy**
- **Parallel Computer Memory Architectures**
- **Parallel Computing – problem faced**
- **Parallel Computing – software programming model**
  - MPI
  - OpenMP
- **Demo**

What is Parallel Computing?

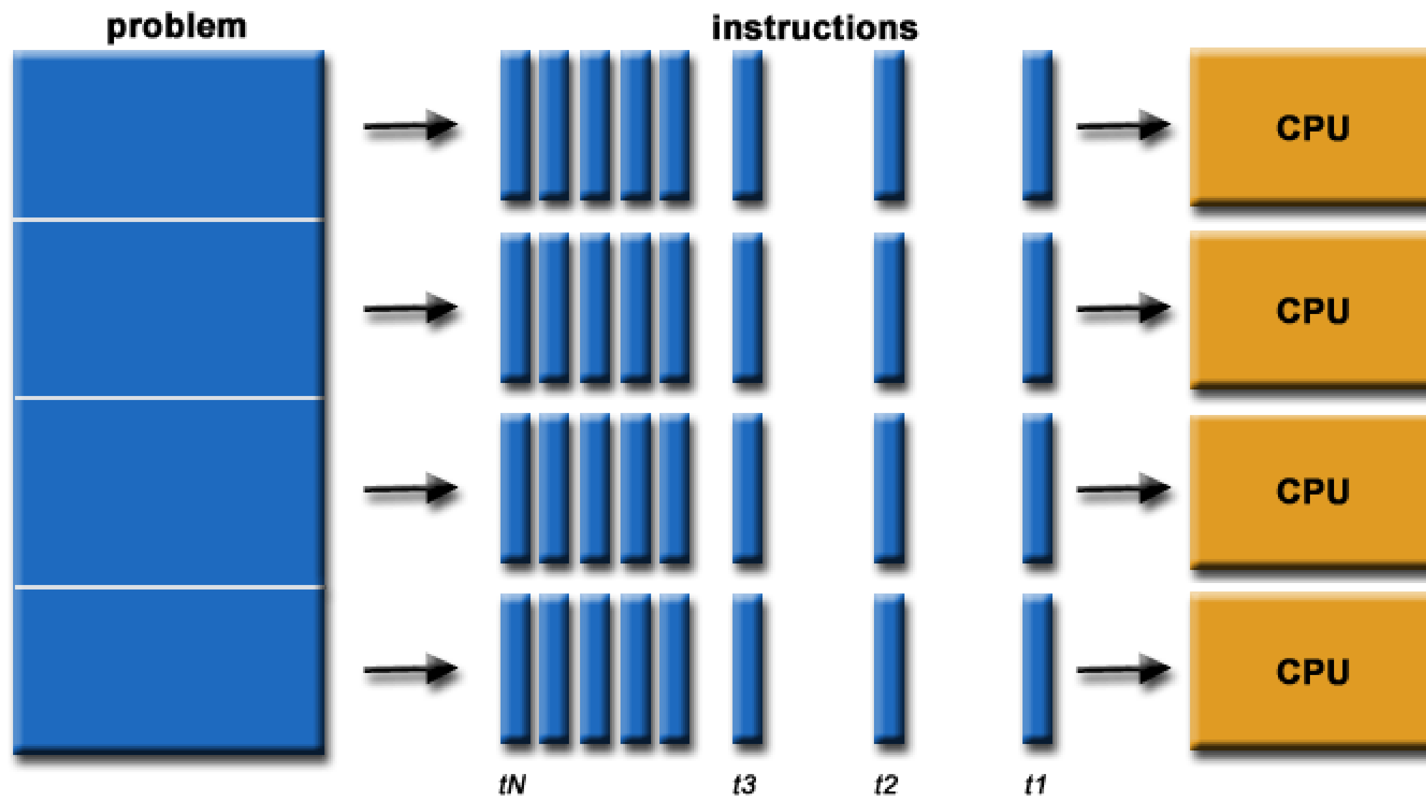
Traditionally, software has been written for ***serial*** computation:

- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.



In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Why use Parallel Computing?

- **The primary reasons for using parallel computing:**
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

- **Other reasons might include:**
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Flynn's Classical Taxonomy

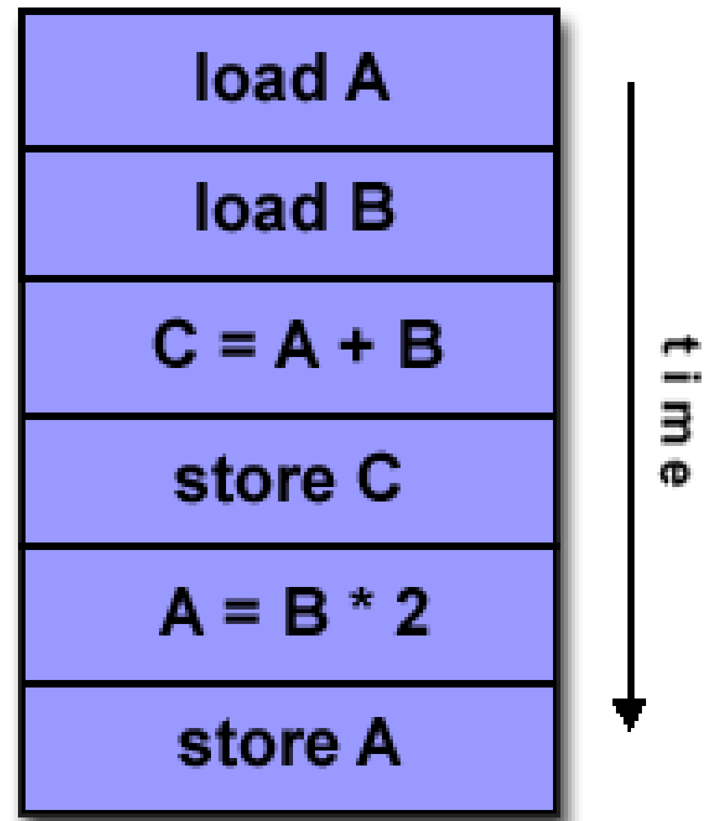
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.
- The matrix below defines the 4 possible classifications according to Flynn.

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

Single Instruction, Single Data (SISD):

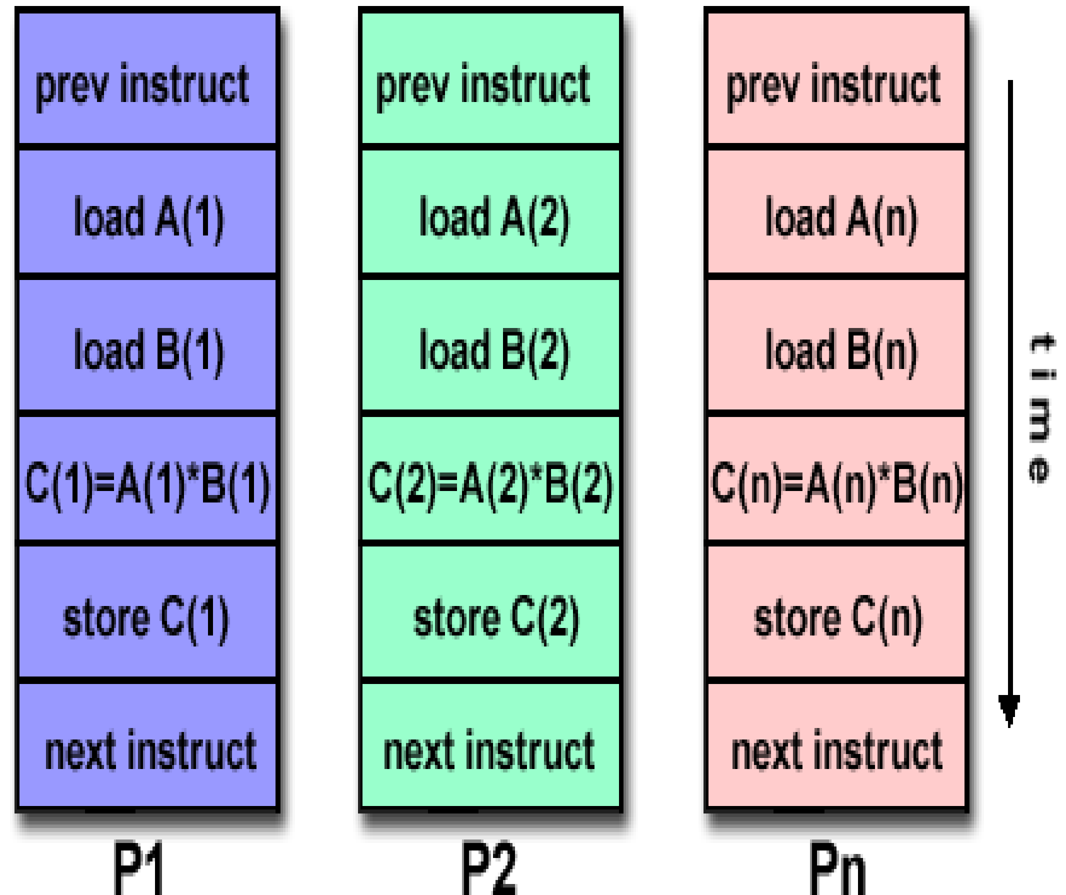
- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU

  workstations and mainframes



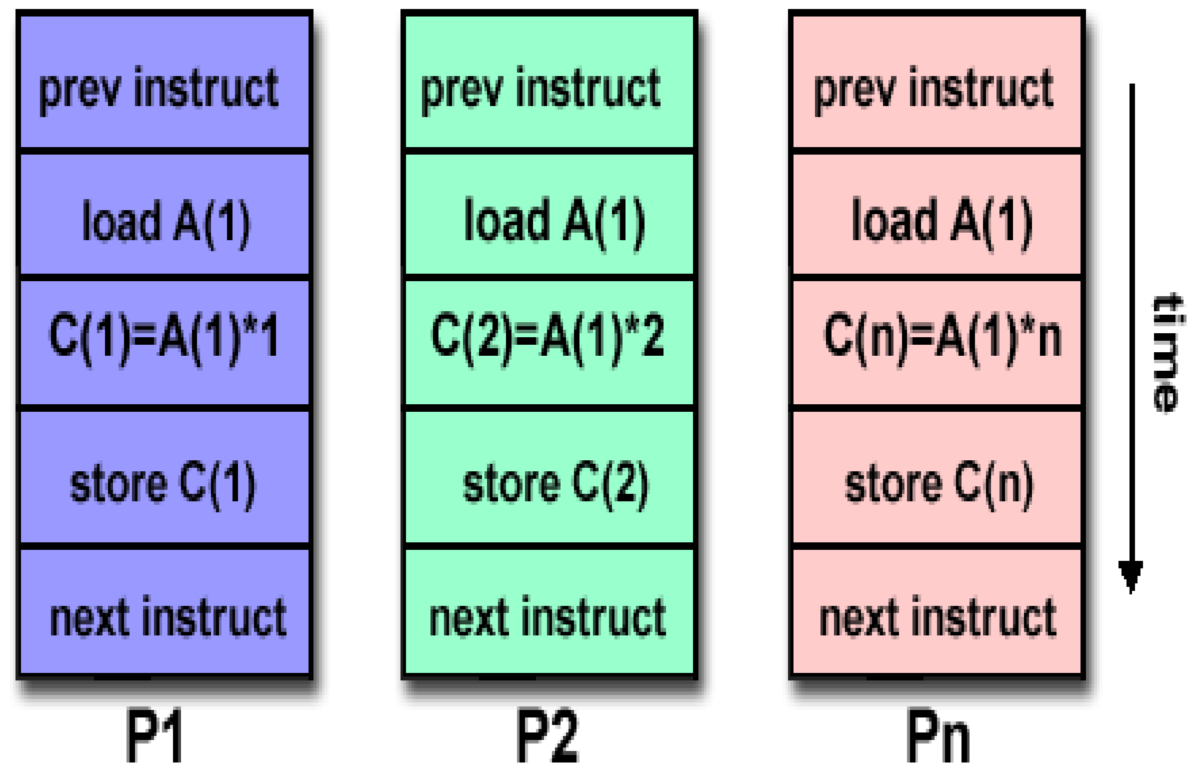
Single Instruction, Multiple Data (SIMD):

- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity,such as image processing.



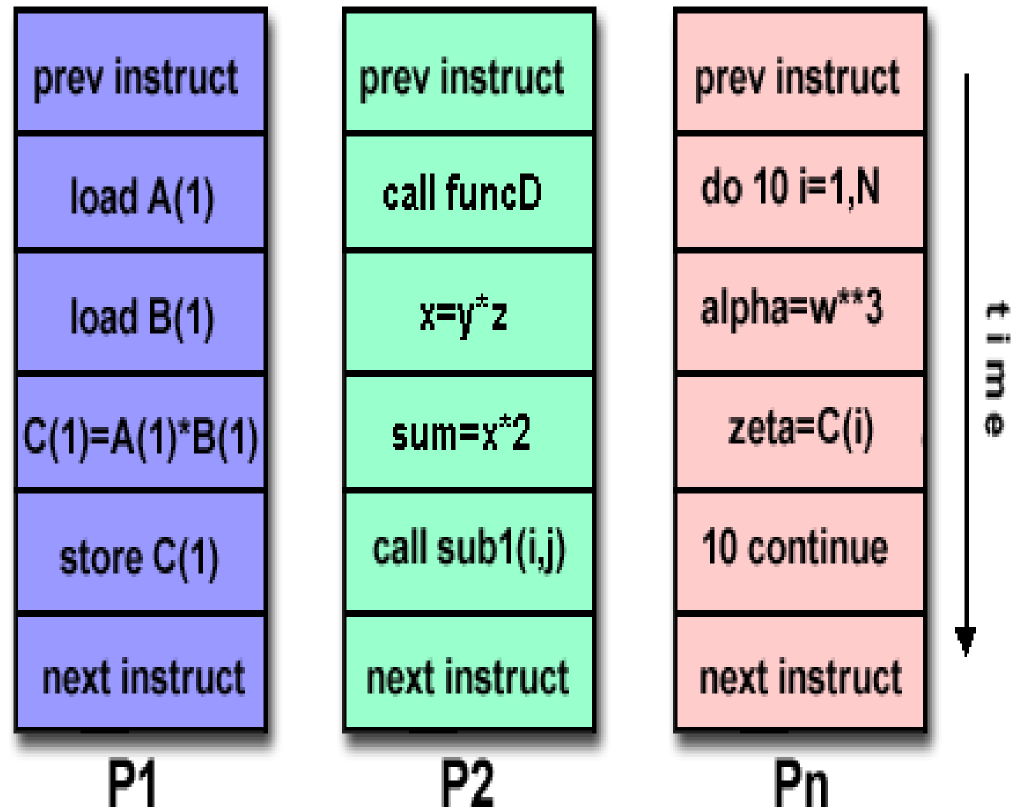
Multiple Instruction, Single Data (MISD):

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).



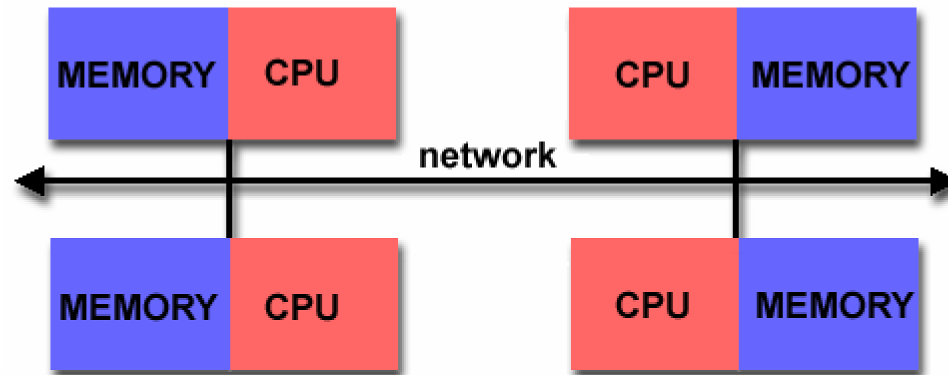
Multiple Instruction, Multiple Data (MIMD):

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



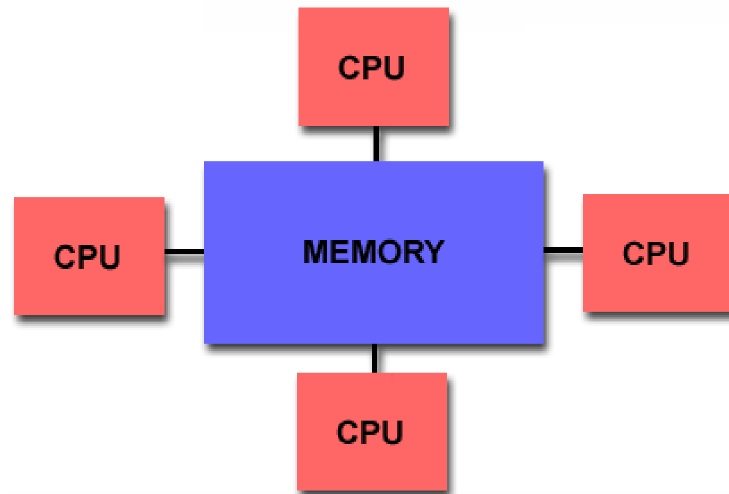
Parallel Computer Memory Architectures

Distributed Memory (Beowulf Cluster)



- A communication network to connect inter-processor memory
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

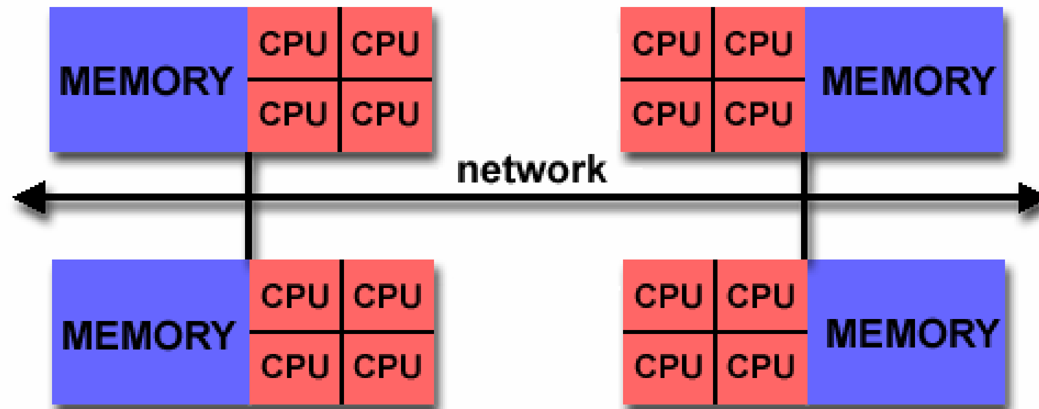
Shared Memory (SMP solution)



- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Hybrid Distributed-Shared Memory (Cluster solution)



- Employ both shared and distributed memory architectures
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

Parallel Computing – problem faced

- **Hardware**
  - in order to facilitate processors working together they must be able to communicate
  - interconnect hardware is complex
    - sharing memory is easy to say, harder to realize as system scales
    - communication over any kind of network is still painfully slow compared to bus speed --- overhead can be significant
- **Software**
  - parallel algorithms are actually fairly well understood
  - the realization of algorithms in software is non-trivial
  - compilers
    - automated parallelism is difficult
  - design
    - portability and power are typically at odds with each other

Parallel Computing – software programming model

- **Distributed memory systems**
  - For processors to share data, the programmer must explicitly arrange for communication -**“Message Passing”**
  - Message passing libraries:
    - **MPI (“Message Passing Interface”)**
    - PVM (“Parallel Virtual Machine”)

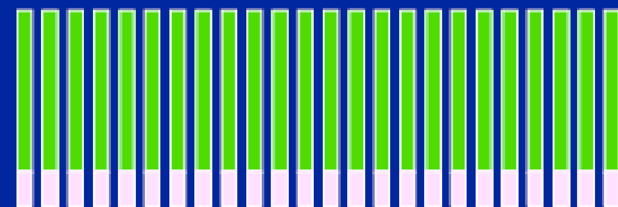
- **Shared memory systems**
  - “Thread” based programming (pthread, …)
  - **Compiler directives (OpenMP)**
  - Can also do explicit message passing, of course

Parallel Computing:

Writing a parallel application.

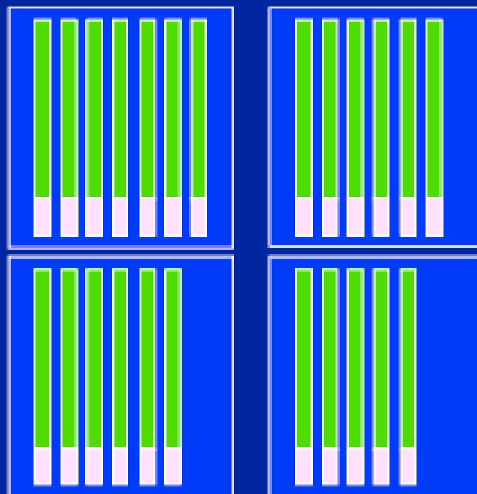
Original Problem

Decompose into tasks



Tasks, shared and local data

Group onto execution units.



Units of execution + new shared data for extracted dependencies

Code with a parallel Prog. Env.

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate(tmp);
  }
}
```

Corresponding source code

MPI

Message Passing Interface

http://www-unix.mcs.anl.gov/mpi/index.htm

http://www.mpi-forum.org/

MPI: What is it?

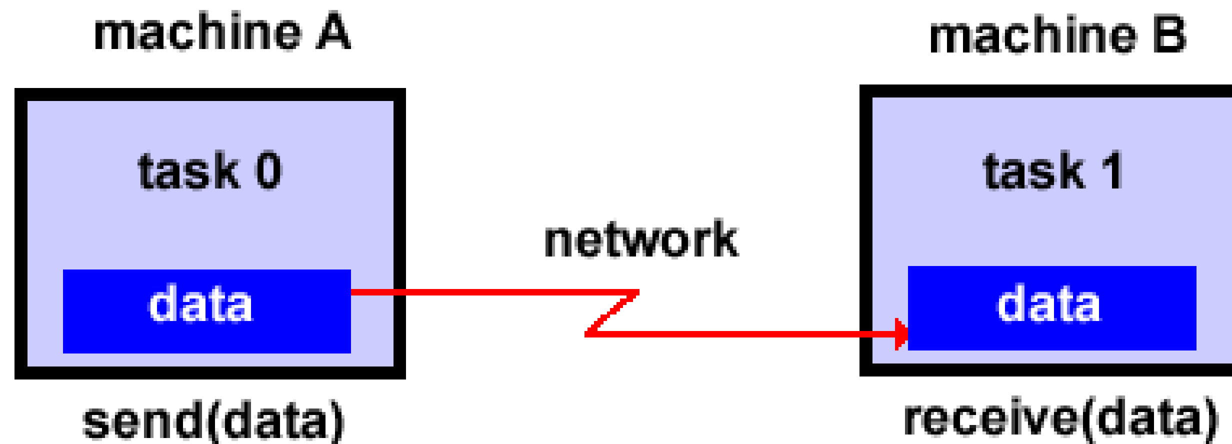
- Library providing message passing support for parallel/distributed applications
  - not a language: collection of subroutines (Fortran), functions/macros (C)
  - explicit communication between processes

- Advantages
  - standardized
  - scalability generally good
  - memory is local to a process (debugging/performance)

- Disadvantages
  - more complex than implicit techniques
  - communication overhead

Message passing Model

- **The message passing model demonstrates the following characteristics:**
  - A set of tasks that use their own local memory during computation. multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



MPI Programming Basics

- **Basic functionality, the ability to:**
  - Start Processes
  - Send Messages
  - Receive Messages
  - Synchronize
- **Core Functions**
  - MPI_Init()
  - MPI_Finalize()
  - MPI_Comm_rank()
  - MPI_Comm_size()
  - MPI_Send()
  - MPI_Recv()
  - MPI_BCAST()
  - MPI_REDUCE()

MPI library

- **Include MPI header file**
  - C
    include “mpi.h”
  - C++
    - include “mpiCC.h” or “mpi++.h”
  - Fortran
    - include “mpif.h”

- **Compile with MPI library**
  - mpicc
  - mpicxx or mpiCC
  - mpif77 or mpif90

Example in C: MPI *Hello, world!*

```
#include <stdio.h>
#include “mpi.h”

int main(int argc, char *argv[])
{
   int rank, size;

   MPI_Init(&argc, &argv);      /* starts MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
   MPI_Comm_size(MPI_COMM_WORLD, &size);     /* get number of processes */

   printf(“Hello, world! from process %d of %d\n”, rank, size);

   MPI_Finalize();       /* end of mpi*/

   return(0);
}
```

Example in C++: MPI *Hello, world!*

```
#include <iostream.h>
#include "mpi++.h"

int main(int argc, char *argv[])
{
    MPI::Init(argc, argv);

     int rank = MPI::COMM_WORLD.Get_rank();
    int size = MPI::COMM_WORLD.Get_size();

    cout << "Hello World! From process " << rank << " of " << size << endl;

    MPI::Finalize();
    return(0);
}
```

Compile and Run result on watsci1

- **Compile**

  mpicc -o helloworld_mpi helloworld_mpi.c
- **Submit a mpi job**

  qsub helloworld_mpi.run
- **Run Result (use 4 cpus)**

  Hello, world! from process 0 of 4

  Hello, world! from process 1 of 4

  Hello, world! from process 2 of 4

  Hello, world! from process 3 of 4

helloworld_mpi.run is:

```
#!/bin/bash
#PBS -l nodes=1:ppn=4
#PBS -o helloworld_mpi.log_1_4
#PBS -j oe
#
cd /home/jemmyhu/jeffclass/mpi/
mpirun -np 4 -machinefile $PBS_NODEFILE ./helloworld_mpi
```

OpenMP: Concepts

Open specifications for Multi Processing

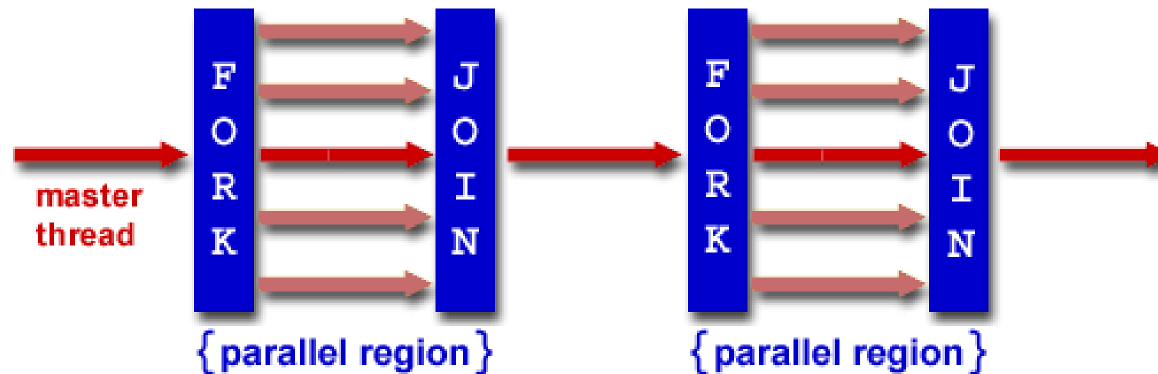
http://www.openmp.org

OpenMP: What is it?

- An Application Program Interface (API) that may be used to explicitly direct **multi-threaded, shared memory parallelism**
- Using **compiler directives**, **library routines** and **environment variables** to automatically generate threaded (or multi-process) code that can run in a concurrent or parallel environment.
- **Portable:**
  - The API is specified for C/C++ and Fortran
  - Multiple platforms have been implemented including most Unix platforms and Windows NT
- **Standardized:** Jointly defined and endorsed by a group of major computer hardware and software vendors

OpenMP: Fork-Join Model

- OpenMP uses the fork-join model of parallel execution:



FORK: the master thread then creates a ***team*** of parallel threads
The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

C / C++ - General Code Structure

```
#include <omp.h>

main () {
  int var1, var2, var3;

  Serial code
  . . .

  Beginning of parallel section. Fork a team of threads.
  Specify variable scoping
 #pragma omp parallel private(var1, var2) shared(var3)
 {
          Parallel section executed by all threads
          . . .
          All threads join master thread and disband
 }

 Resume serial code
 . . .
}
```

OpenMP: compiler

Compiler	OpenMP flag
Intel (icc, ifort)	**-openmp**
Pathscale (pathcc, pathf90)	**-openmp**
PGI (pgcc, pgf77, pgf90)	**-mp**
GNU (gcc4, gfortran)	**-fopenmp**

Examples:

gcc4 –fopenmp –o myopenmp.exe myopenmp.c

icc –openmp –o hello_openmp hello_openmp.c

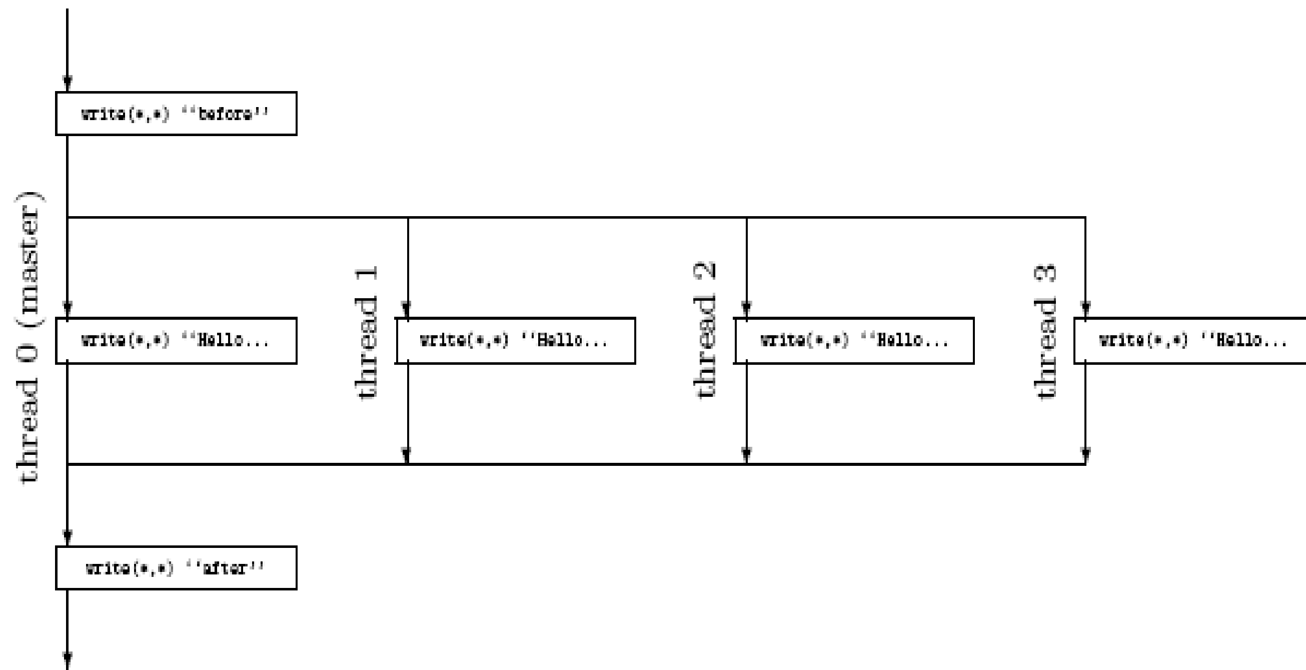
OpenMP: simplest example

```
program hello
  write(*,*) "before"
  !$omp parallel
     write(*,*) "Hello, parallel world!"
  !$omp end parallel
  write(*,*) "after"
end program
```

Before

Hello, parallel world!
Hello, parallel world!
Hello, parallel world!
Hello, parallel world!

after



OpenMP example-1: hello world in C

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        printf("Hello World from thread %d\n", id);
        #pragma omp barrier
        if ( id == 0 ) {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n",nthreads);
        }
    }
    return 0;
}
```

Compile and Run result on watsci1

```
[jemmyhu@watsci1 my_runs]$ gcc4 -fopenmp -o helloworld helloworld.c

[jemmyhu@watsci1 my_runs]$ export OMP_NUM_THREADS=4
[jemmyhu@watsci1 my_runs]$ ./helloworld
Hello World from thread 3
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
There are 4 threads
[jemmyhu@watsci1 my_runs]$
```

MPI vs. OpenMP

http://www.science.uwaterloo.ca/~j8hu/Teaching_Cluster/mpi+smp.swf

Example: Calculating π

- **Numerical integration**

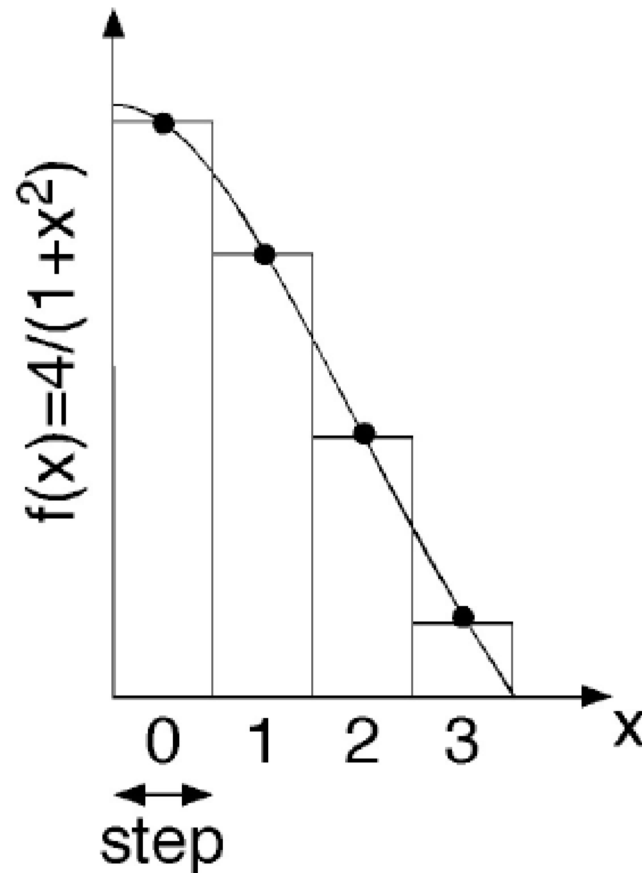
$$\int_0^1 \frac{4}{1+x^2}dx = \pi$$

- **Discretization:**

  $\Delta = 1/N$: `step = 1/NBIN`
  $x_i = (i+0.5)\Delta \; (i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



OpenMP Program: omp_pi_critical.c

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  double step,sum=0.0,pi;
  step = 1.0/NBIN;
#pragma omp parallel
  {
    int nthreads,tid,i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
#pragma omp critical
      sum += 4.0/(1.0+x*x);
    }
  }
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```

Shared variables

Private (local) variables

This has to be atomic

pi – OpenMP version

```
#include <stdio.h>
#include <omp.h>              /* OpenMP header file*/
#define NUM_STEPS 100000000
int main(int argc, char *argv[]) {
   int i, nthreads;
   double x, pi, sum = 0.0;
   double step = 1.0/(double) NUM_STEPS;
   start_time = omp_get_wtime();
   #pragma omp parallel
   {
      nthreads = omp_get_num_threads();
      #pragma omp for private(x) reduction(+:sum) schedule(runtime)
      for (i=0; i < NUM_STEPS; ++i) {
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
      #pragma omp master
      {
         pi = step * sum;
      }
   }
  end_time = omp_get_wtime();
   printf("parallel program results with %d threads:\n", nthreads);
   printf("pi = %g  (%17.15f)\n",pi, pi);
   printf("time to compute = %g seconds\n", end_time - start_time);
   return 0;
}
```

Compile and Run on Sharcnet

Compile on whale

```
cc -openmp  –o  pi-par-omp  pi-par-omp.c
```

Run results on whale:

Serial (1 cpu):

```
sequential program results:
pi = 3.14159  (3.141592653590022)
time to compute = 0.858833 seconds
```

Parallel (4 cpus):

```
parallel program results with 4 threads:
pi = 3.14159  (3.141592653589683)
time to compute = 0.201562 seconds
```

/home/jemmyhu/Courses/CES706/openmp/C/pi/timing

Watsci1 does not work well for openmp timing currently:
/home/jemmyhu/jeffclass/openmp/my_runs

pi – MPI version

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>                /* MPI header file */
#define NUM_STEPS 100000000

int main(int argc, char *argv[]) {
    int nprocs;
    int myid;
    double start_time, end_time;
    int i;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;

    /* initialize for MPI */
   MPI_Init(&argc, &argv);       /* starts MPI */

    /* get number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    /* get this process's number (ranges from 0 to nprocs - 1) */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
/* record start time */
    start_time = MPI_Wtime();

    /* do computation */
    for (i=myid; i < NUM_STEPS; i += nprocs) {      /* changed */
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    sum = step * sum;                              /* changed */
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

    /* record end time */
    end_time = MPI_Wtime();

    /* print results */
    if (myid == 0) {
        printf("parallel program results with %d processes:\n", nprocs);
        printf("pi = %g  (%17.15f)\n",pi, pi);
        printf("time to compute = %g seconds\n", end_time - start_time);
    }

    /* clean up for MPI */
    MPI_Finalize();

    return 0;
}
```

Compile and Run on watsci1

Compile on watsci1

```
mpicc  –o  pi-par-mpi  pi-par-mpi.c
```

Run results on watsci1:

Serial (1 cpu):

```
sequential program results:
pi = 3.14159  (3.141592653590426)
time to compute = 0.867188 seconds
```

Parallel (4 cpus):

```
parallel program results with 4 processes:
pi = 3.14159  (3.141592653590217)
time to compute = 0.289062 seconds
```

/home/jemmyhu/jeffclass/mpi/my_runs

Demo/Hand-on exercise

- logon to watsci1
- cp files (codes) from
  /home/jemmyhu/jeffclass/mpi/source
  /home/jemmyhu/jeffclass/openmp/source
- compile and run on watsci1

References

- http://www.llnl.gov/computing/tutorials/parallel_comp/
- http://cacs.usc.edu/education/cs596/06OMP.pdf