Parallel Computing - Theory

Introduction to Parallel Computing

Ka-Chun Wong, Department of Computer Science, City University of Hong Kong
Blaise Barney, Lawrence Livermore National Laboratory

Outline

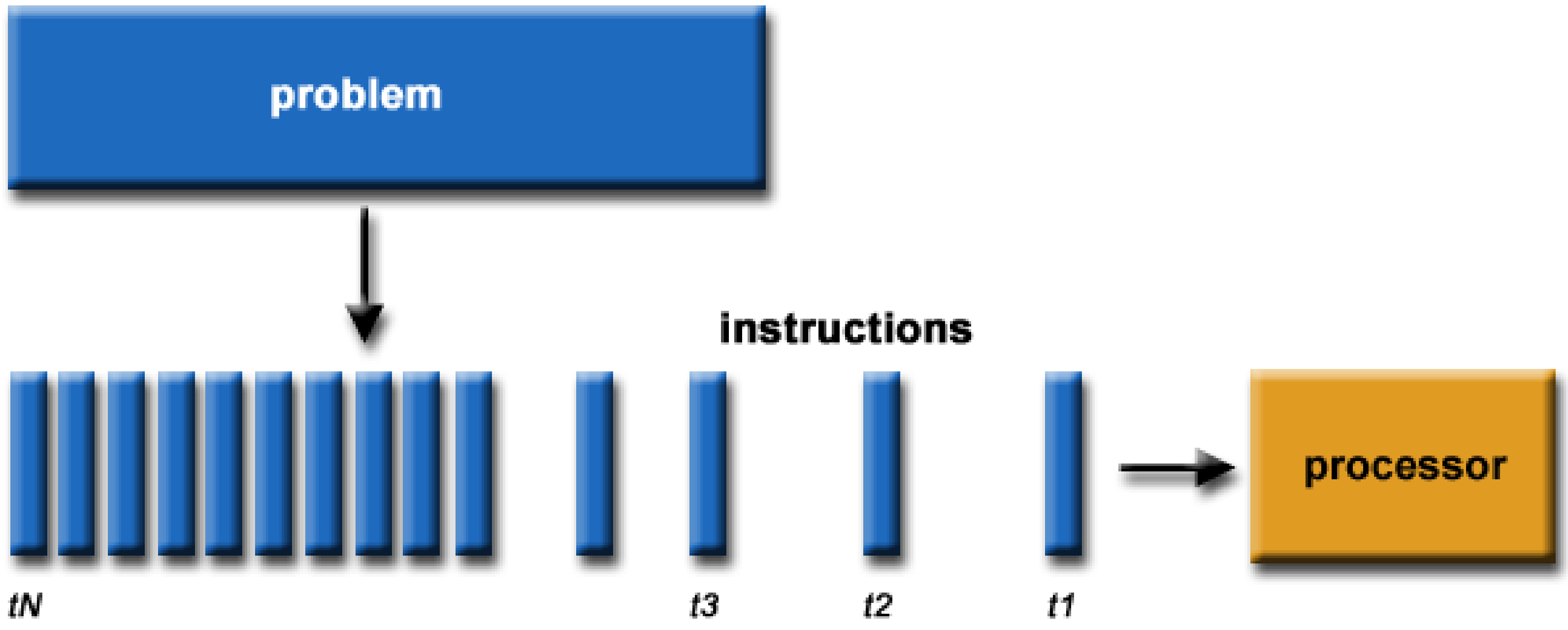
- Overview
- Concepts and Terminology
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Parallel Examples
- References and More Information

Overview

- **Serial Computing:**
- Traditionally, software has been written for ***serial*** computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time

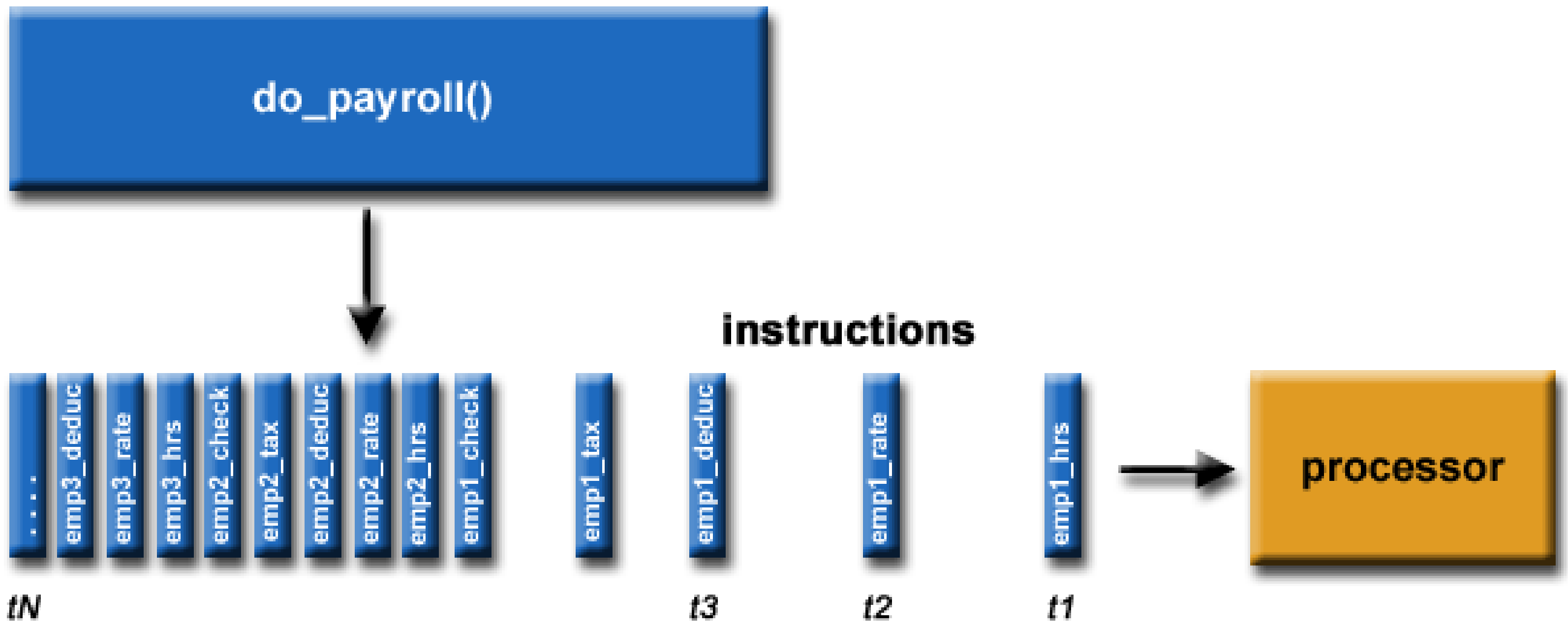
Overview

- **Serial Computing:**



Overview

- **Serial Computing:**

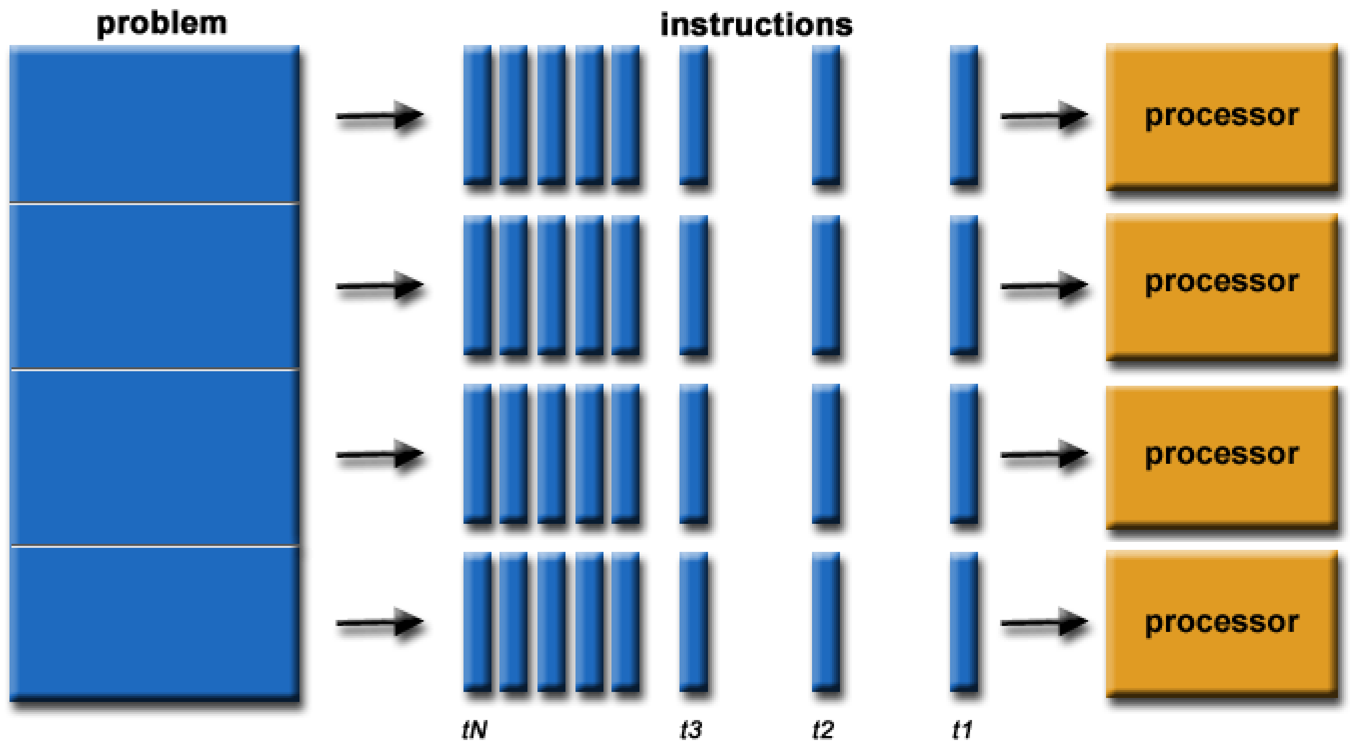


Overview

- **Parallel Computing:**
- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  - 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 processors
  - An overall control/coordination mechanism is employed

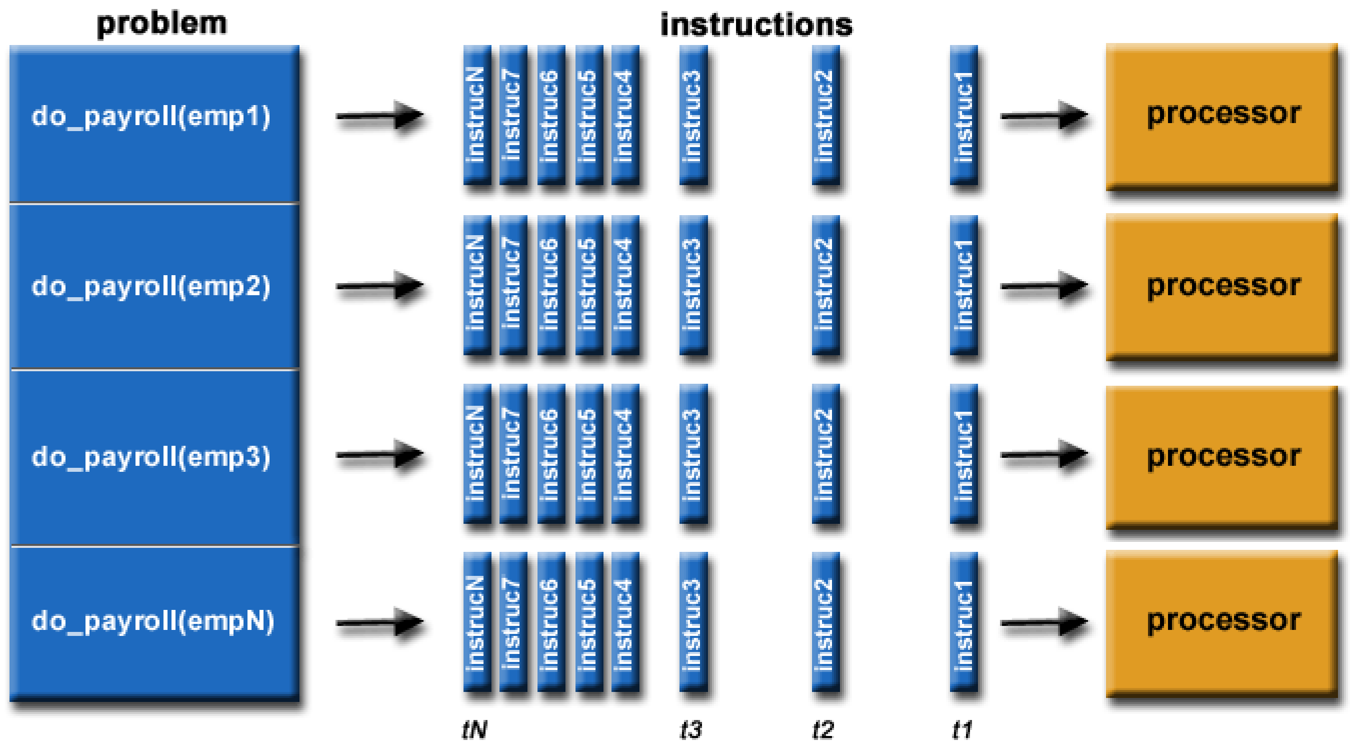
Overview

- **Parallel Computing:**



Overview

- **Parallel Computing:**



Overview

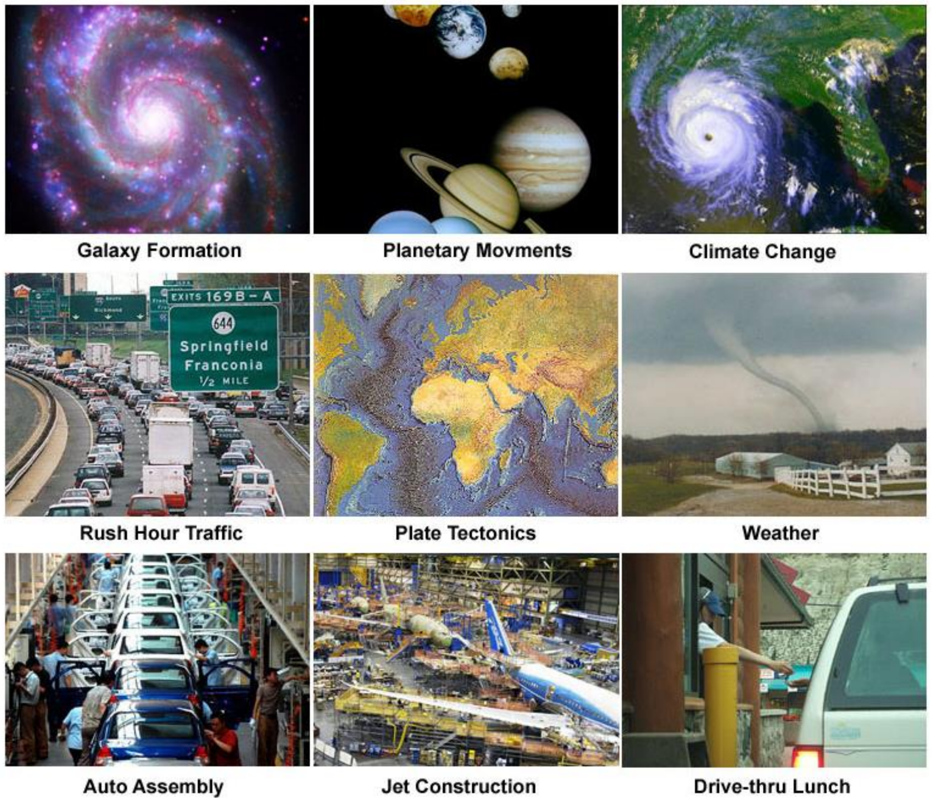
- **Parallel Computing:**
- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The computing resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

Overview

- **Why Use Parallel Computing?**
  - **The Real World is Massively Parallel:**
    - In the natural world, many complex and interrelated events are happening at the same time within a temporal sequence.
    - Compared to serial computing, parallel computing is much better suited for modeling, simulating, and understanding complex and real world phenomena.

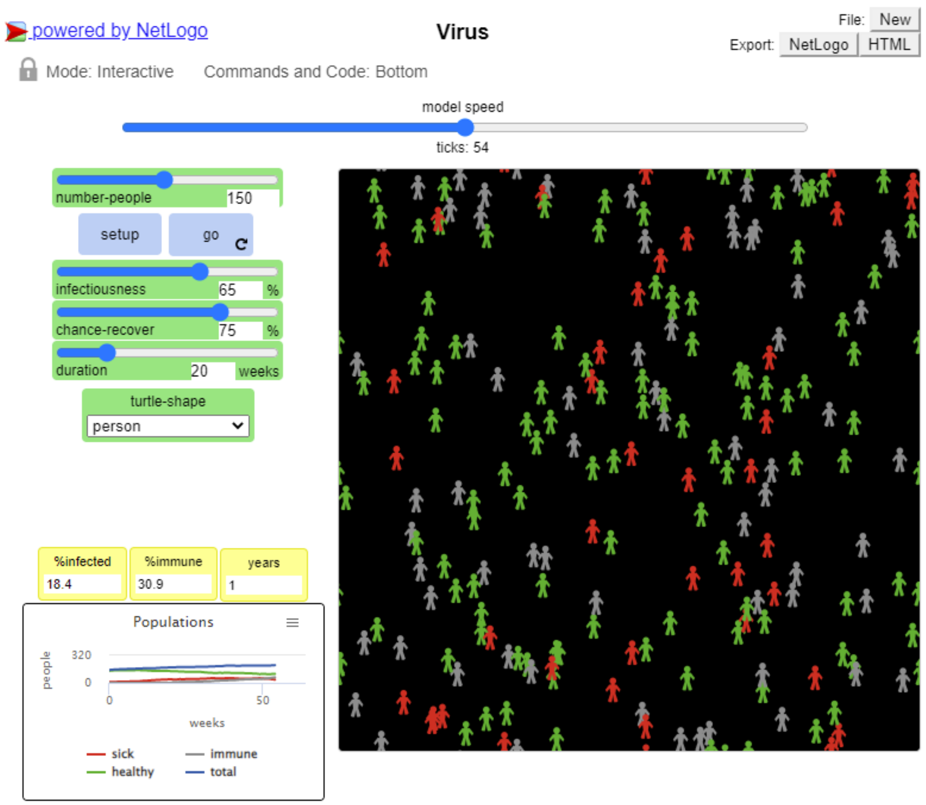
Overview

- **The Real World is Massively Parallel:**
- **For example, imagine modeling these:**



Overview

- **The Real World is Massively Parallel:**
- **For example,**



ticks: 54

number-people 150

infectiousness 65 %

chance-recover 75 %

duration 20 weeks

turtle-shape: person

%infected	%immune	years
18.4	30.9	1

— sick — immune — healthy — total

Netlogo:
https://ccl.northwestern.edu/netlogo/

Overview

- **Why Use Parallel Computing?**

•**SAVE TIME AND/OR MONEY:**

- In theory, throwing more resources at a task will shorten its time to completion with potential cost savings.
- Parallel computers can be built from cheap and commodity components.

Overview

- **Why Use Parallel Computing?**

  •**SOLVE LARGE / COMPLEX PROBLEMS:**

    •Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

    •Examples:

      •"Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.

      • Web search engines/databases processing millions of transactions every second

Overview

- **Why Use Parallel Computing?**

  •**PROVIDE CONCURRENCY:**

  •A single computing resource can only do one thing at a time. Multiple computing resources can do many things simultaneously.

  •Example

  • Collaborative Networks provide a global venue where people from around the world can meet and conduct work "virtually".

Overview

- **Why Use Parallel Computing?**

  •**TAKE ADVANTAGE OF NON-LOCAL RESOURCES:**

  •Using computing resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient. Two examples below, each of which has over 1.7 million contributors globally (May 2018):

  •Example: SETI@home (setiathome.berkeley.edu)

  •Example: Folding@home (https://en.wikipedia.org/wiki/Folding@home)

Overview

- **Why Use Parallel Computing?**

  - **MAKE USE OF UNDERLYING HARDWARE:**

    •Modern computers, even laptops, are parallel in architecture with multiple processors/cores.

    •Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.

    •In most cases, serial programs run on modern computers "waste" potential computing power.

Source: Top500.org

Overview

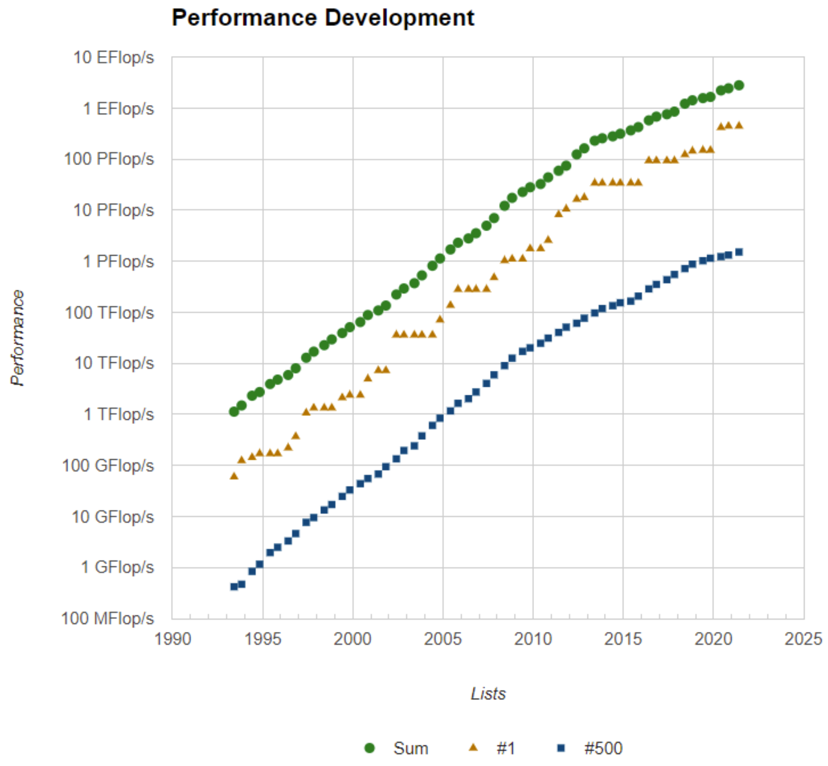
- The Future

•During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.

•In this same time period, there has been a greater than **500,000x** increase in supercomputer performance, with no end currently in sight.

•***The race is already on for Exascale Computing!***

• **Exaflop = 10^{18} calculations per sec**



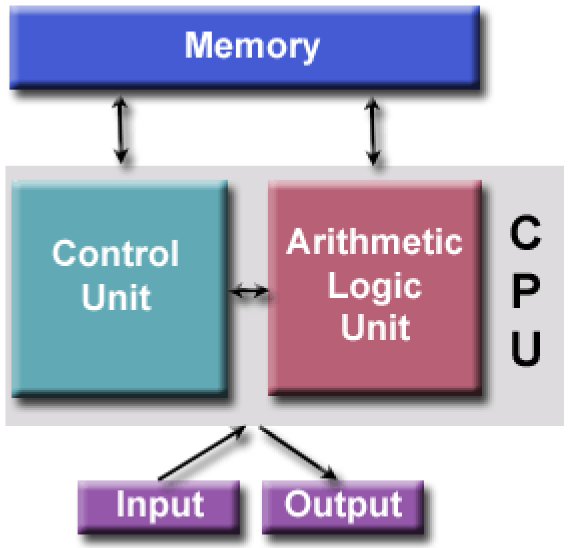
Concepts and Terminology

- **von Neumann Architecture**

- Named after the Hungarian mathematician/genius John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Also known as "stored-program computer" - both program instructions and data are kept in electronic memory. It is different from earlier computers which were programmed through "hard wiring".

Concepts and Terminology

von Neumann Architecture

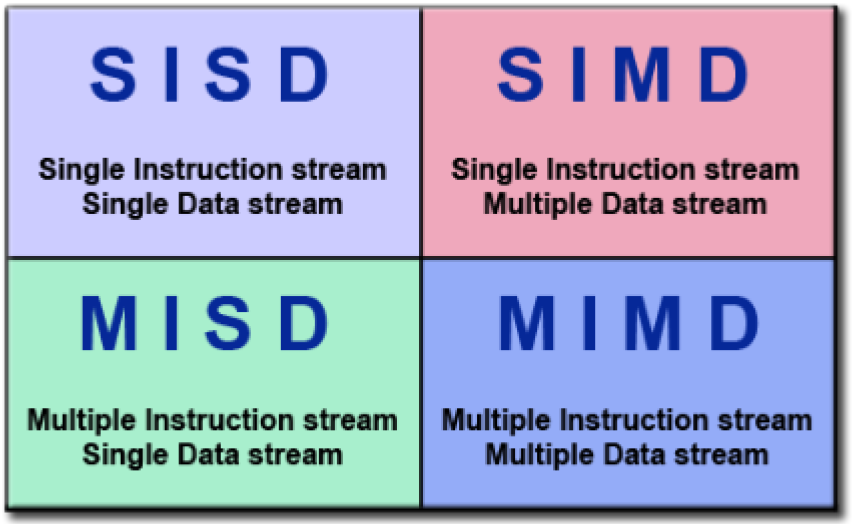


- Comprised of four main components:
  - Memory
  - Control Unit
  - Arithmetic Logic Unit
  - Input/Output
- Read/write, random access memory is used to store both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator

John von Neumann circa 1940s (Source: LANL archives)

Concepts and Terminology

- **Flynn's Classical Taxonomy**



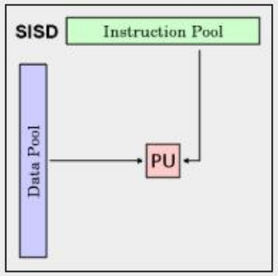
S I S D Single Instruction stream Single Data stream	**S I M D** Single Instruction stream Multiple Data stream
M I S D Multiple Instruction stream Single Data stream	**M I M D** Multiple Instruction stream Multiple Data stream

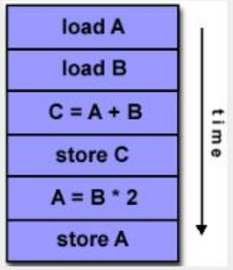
- There are different ways to classify parallel computers.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

Concepts and Terminology

- **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 type of computer
•Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.





UNIVAC1

IBM 360

CRAY1

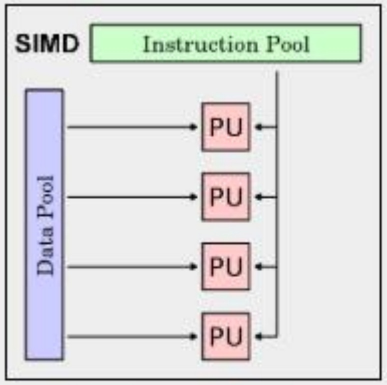
Concepts and Terminology

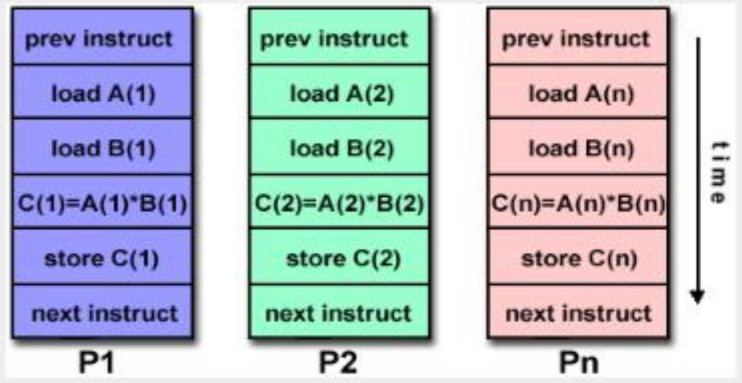
- **Single Instruction, Multiple Data (SIMD)**

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
Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
Synchronous (lockstep) and deterministic execution
Two varieties: Processor Arrays and Vector Pipelines
Examples:
  Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820
  Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

Concepts and Terminology

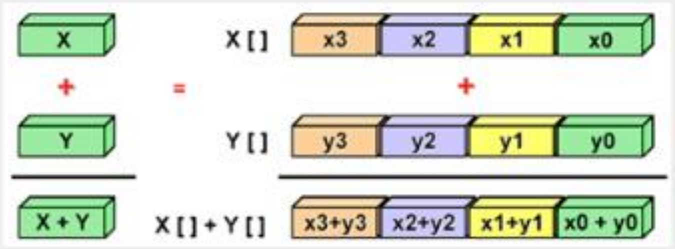
- **Single Instruction, Multiple Data (SIMD)**





ILLIAC IV

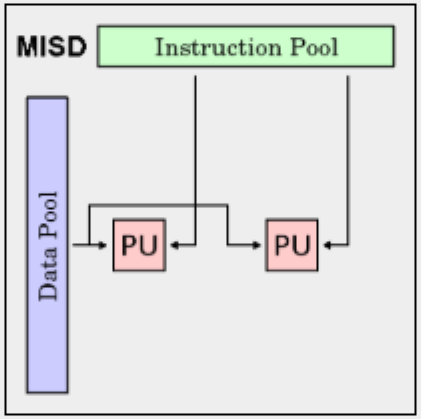
MasPar

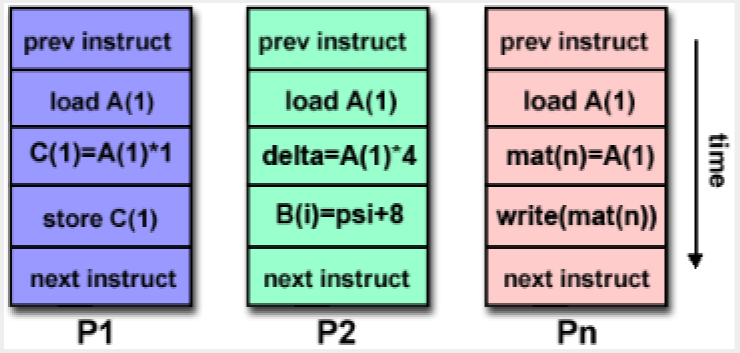


Concepts and Terminology

- **Multiple Instruction, Single Data (MISD)**

A type of parallel computer
Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
Single Data: A single data stream is fed into multiple processing units.
Few (if any) actual examples of this class of parallel computer have ever existed.
Some conceivable uses might be:
    multiple frequency filters operating on a single signal stream
    multiple cryptography algorithms attempting to crack a single coded message.





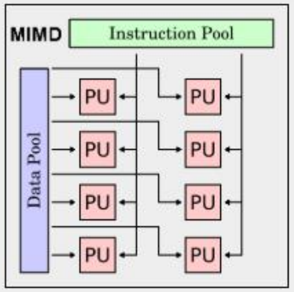
Concepts and Terminology

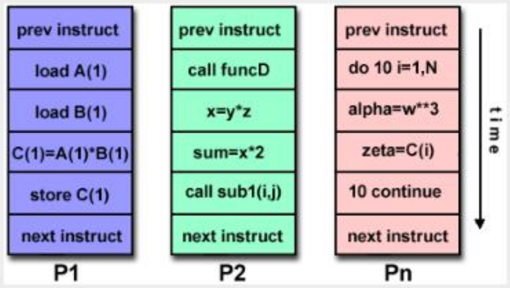
- **Multiple Instruction, Multiple Data (MIMD)**

A type of parallel computer
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
Currently, most modern supercomputers fall into this category.
Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
Note: many MIMD architectures also include SIMD execution sub-components

Concepts and Terminology

- **Multiple Instruction, Multiple Data (MIMD)**





IBM POWER5

HP/Compaq Alphaserver

Intel IA32

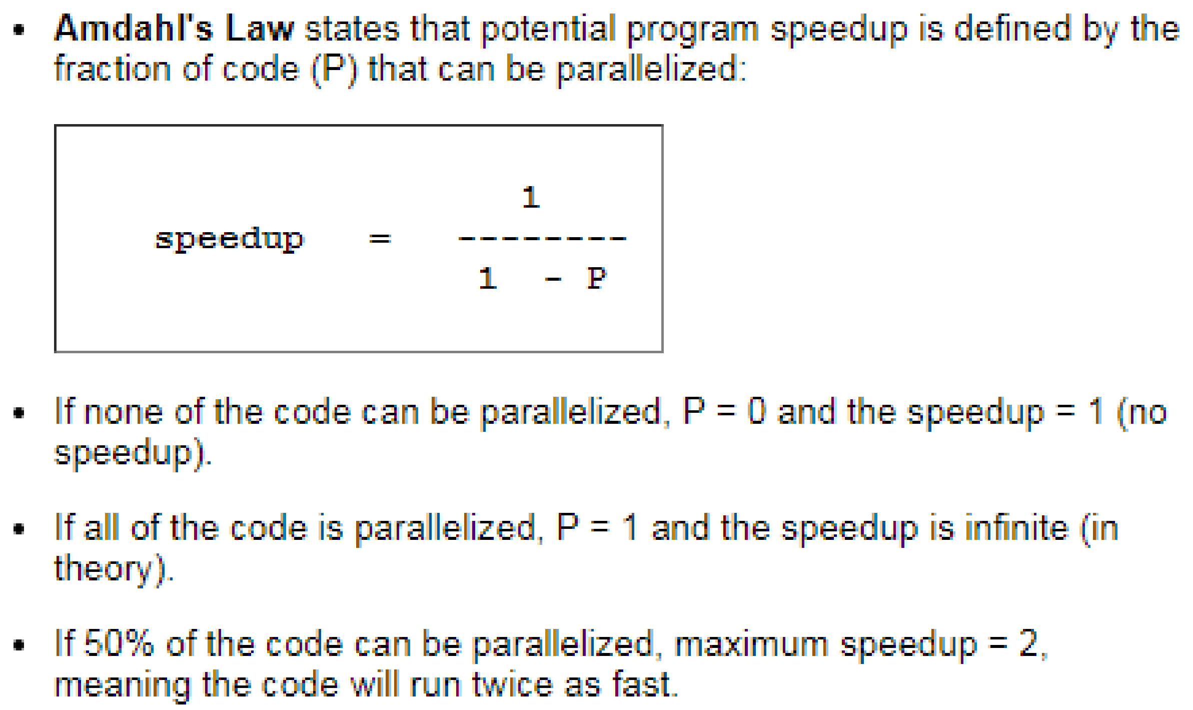
AMD Opteron

Cray XT3

IBM BG/L

Concepts and Terminology

- **Limits and Costs of Parallel Computing**



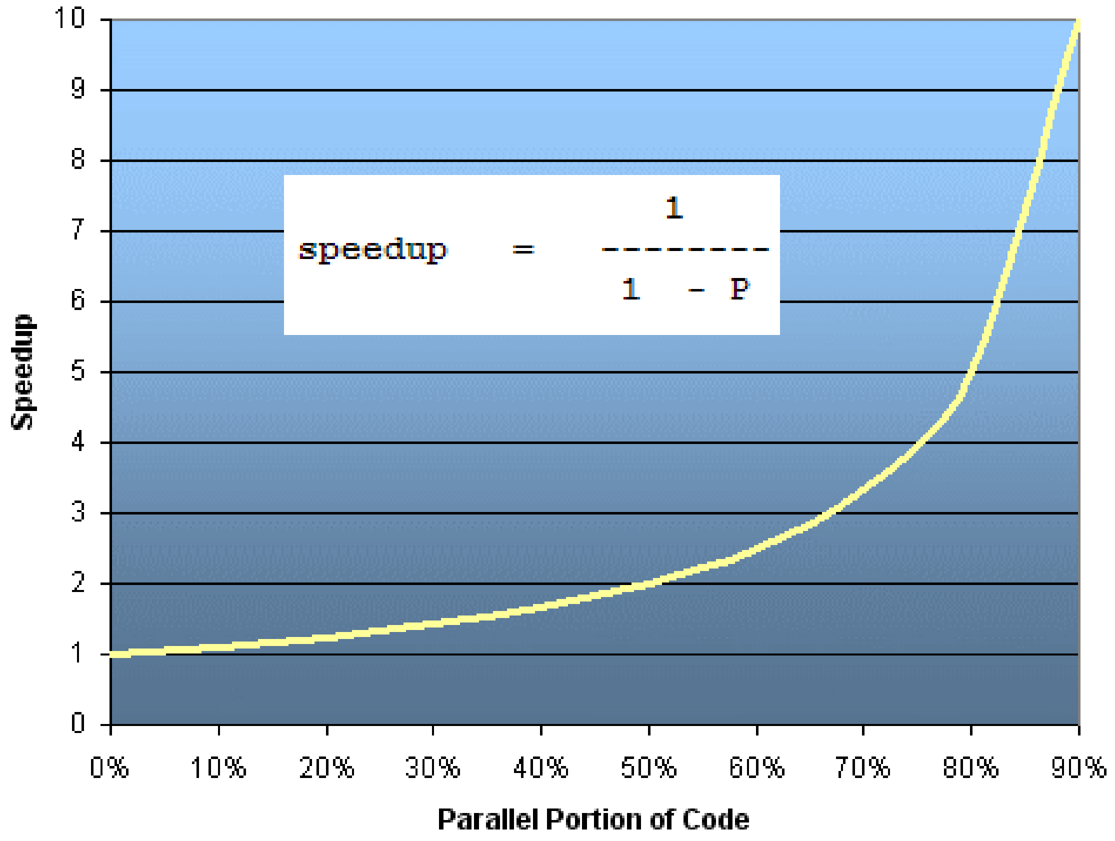
- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

```
                           1
    speedup   =    --------
                     1   - P
```

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Concepts and Terminology

- **Limits and Costs of Parallel Computing**



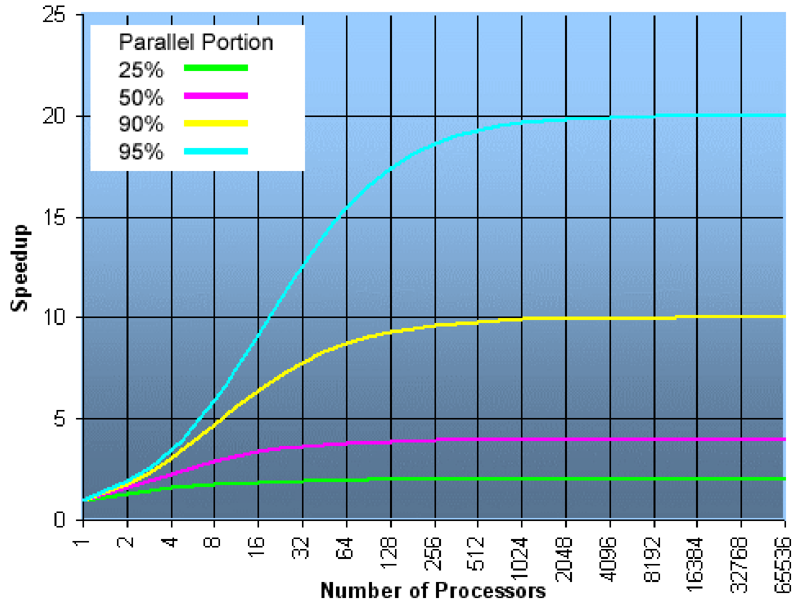
Concepts and Terminology

Limits and Costs of Parallel Computing

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

where P = parallel fraction, N = number of processors and S = serial fraction.



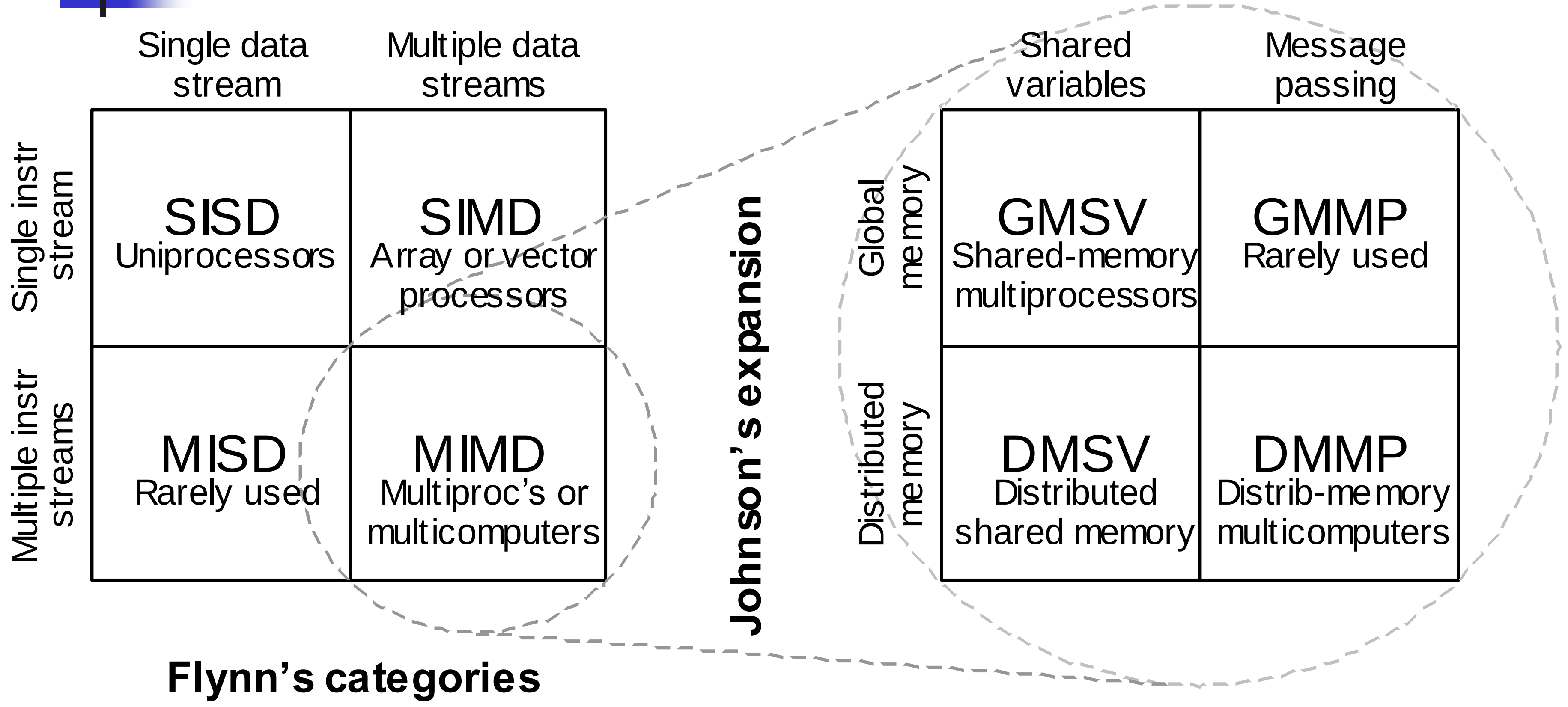
Concepts and Terminology

- **Limits and Costs of Parallel Computing**
  - **Complexity:**
    - In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
    - Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.

Concepts and Terminology

- **Limits and Costs of Parallel Computing**
  - **Resource Requirements:**
    - The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
    - For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Parallel Computer Memory Architectures



The Flynn-Johnson classification of computer systems

Parallel Computer Memory Architectures

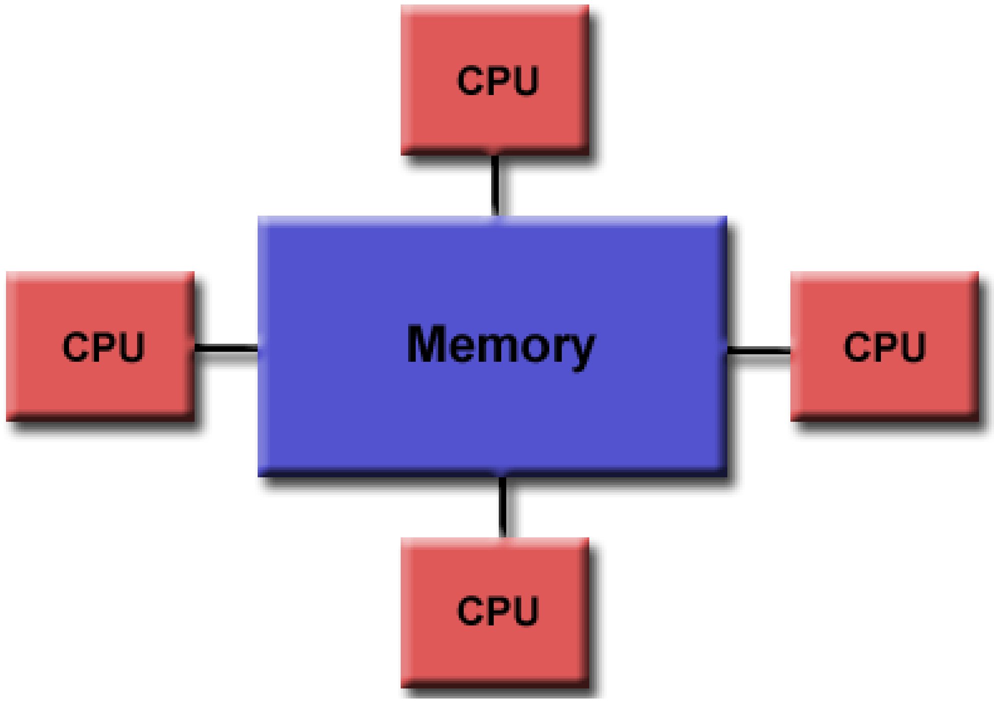
- **Shared Memory:**
  - **General Characteristics:**
    - Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
    - Multiple processors can operate independently but share the same memory resources.
    - Changes in a memory location effected by one processor are visible to all other processors.
    - Historically, shared memory machines have been classified as UMA and NUMA, based upon memory access times.

Parallel Computer Memory Architectures

- **Shared Memory:**
  - **Uniform Memory Access (UMA):**
    - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
    - Identical processors
    - Equal access and access times to memory
    - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Parallel Computer Memory Architectures

- **Shared Memory:**
  - **Uniform Memory Access (UMA):**

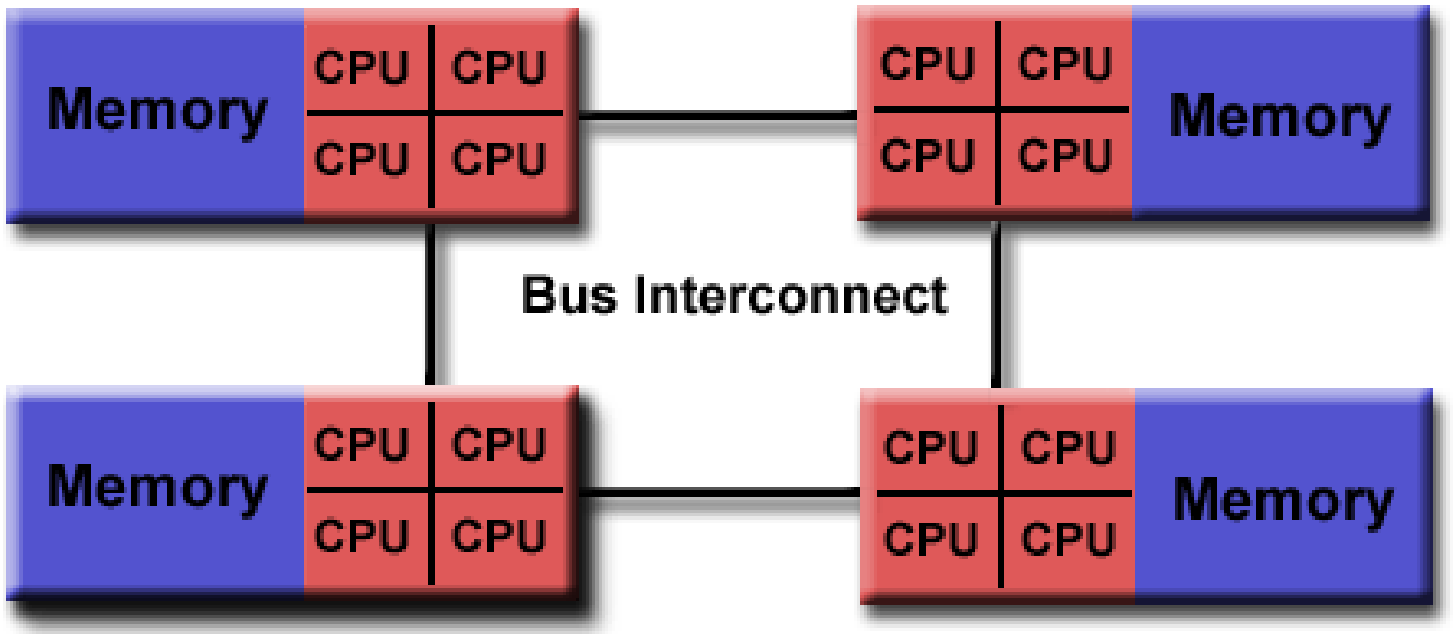


Parallel Computer Memory Architectures

- **Shared Memory:**
  - **Non-Uniform Memory Access (NUMA):**
    - Often made by physically linking two or more SMPs
    - One SMP can directly access memory of another SMP
    - Not all processors have equal access time to all memories
    - Memory access across link is slower
    - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Parallel Computer Memory Architectures

- **Shared Memory:**
  - **Non-Uniform Memory Access (NUMA):**



Parallel Computer Memory Architectures

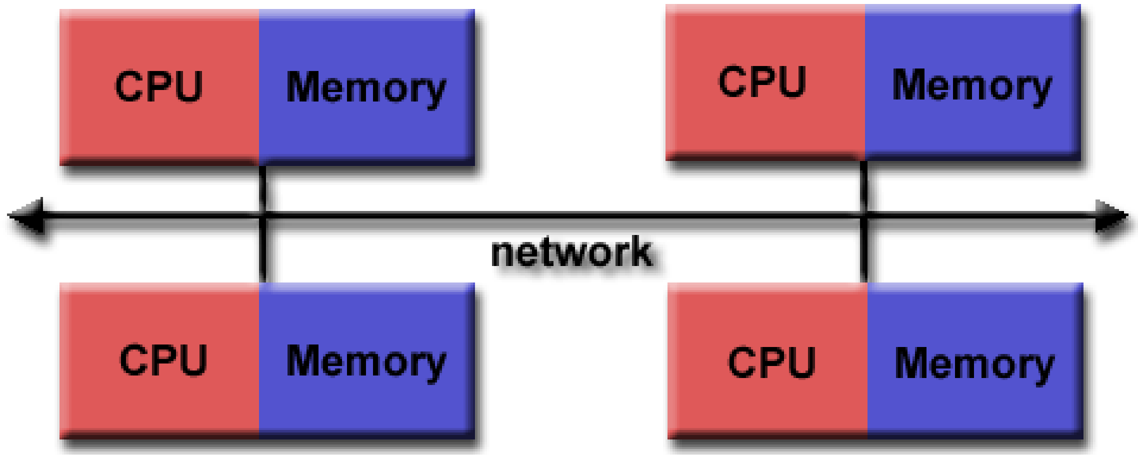
- **Shared Memory:**
  - **Advantages:**
    - Global address space provides a user-friendly programming perspective to memory
    - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Parallel Computer Memory Architectures

- **Shared Memory:**
  - **Disadvantages:**
    - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
    - Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Parallel Computer Memory Architectures

- **Distributed Memory:**
  - Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.



Parallel Computer Memory Architectures

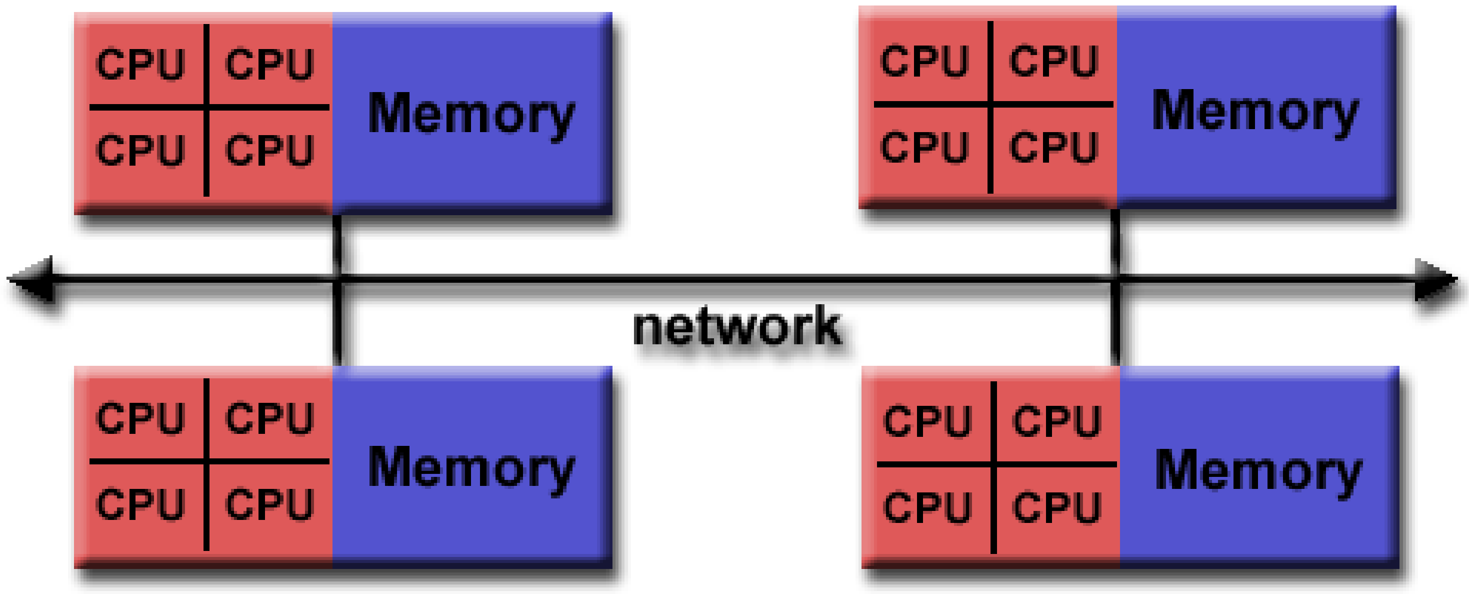
- **Distributed Memory:**
  - **Advantages:**
    - Memory is scalable with the number of processors. The number of processors and the size of memory are increased proportionately.
    - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
    - Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Parallel Computer Memory Architectures

- **Distributed Memory:**
  - **Disadvantages:**
    - The programmer is responsible for many of the details associated with data communication between processors.
    - It may be difficult to map existing data structures, based on global memory, to the distributed memory organization.
    - Non-uniform memory access times - data residing on a remote node takes longer to access than local data.

Parallel Computer Memory Architectures

- **Hybrid Distributed-Shared Memory:**
  - The largest and fastest computers in the world today employ both shared and distributed memory architectures.



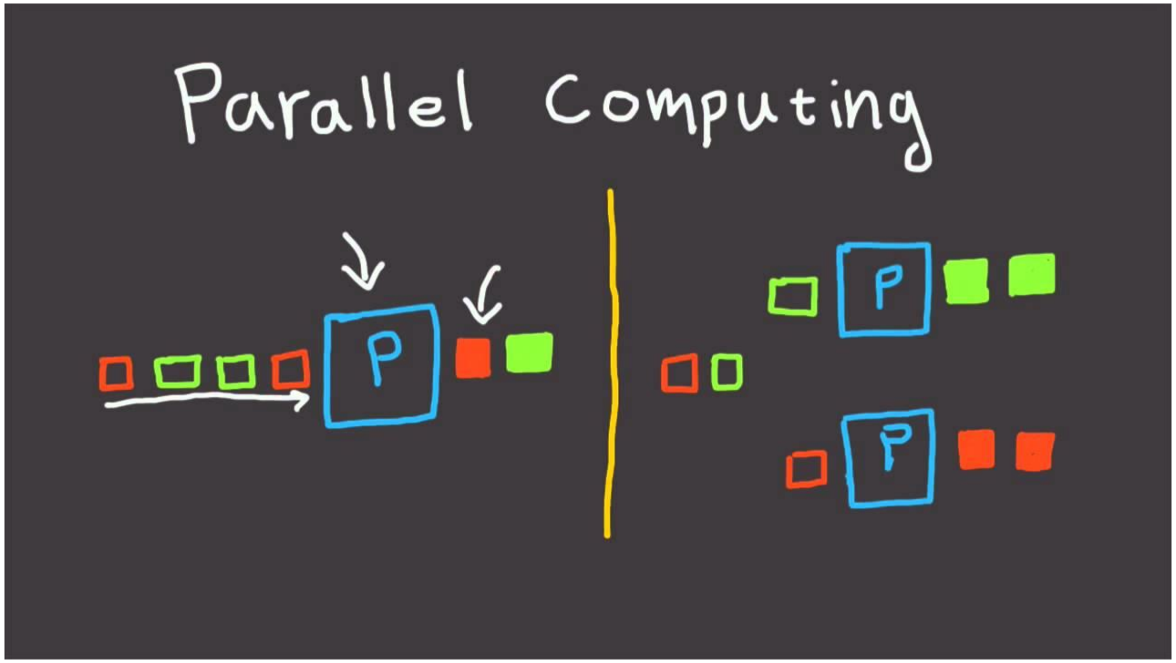
Parallel Computer Memory Architectures

- **Hybrid Distributed-Shared Memory:**
  - The shared memory component can be shared memory units and/or graphics processing units (GPU).
  - The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine 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.

Parallel Computer Memory Architectures

- **Hybrid Distributed-Shared Memory:**
  - Advantages and Disadvantages:
    - Whatever is common to both shared and distributed memory architectures.
    - Increased scalability is an important advantage
    - Increased programmer complexity is an important disadvantage

Lecture Break

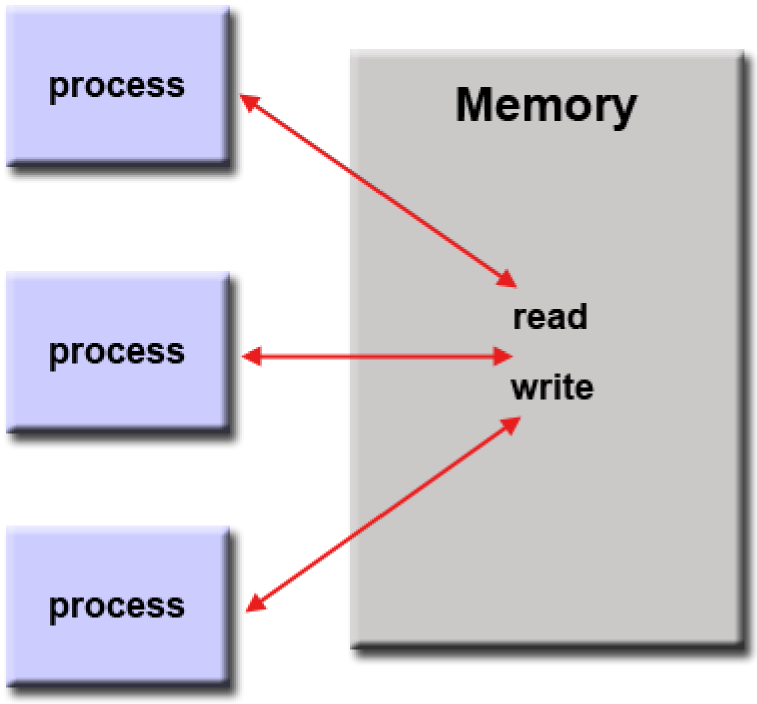


https://www.youtube.com/watch?v=q7sgzDH1cR8

Parallel Programming Models

- **Shared Memory Model (without threads)**

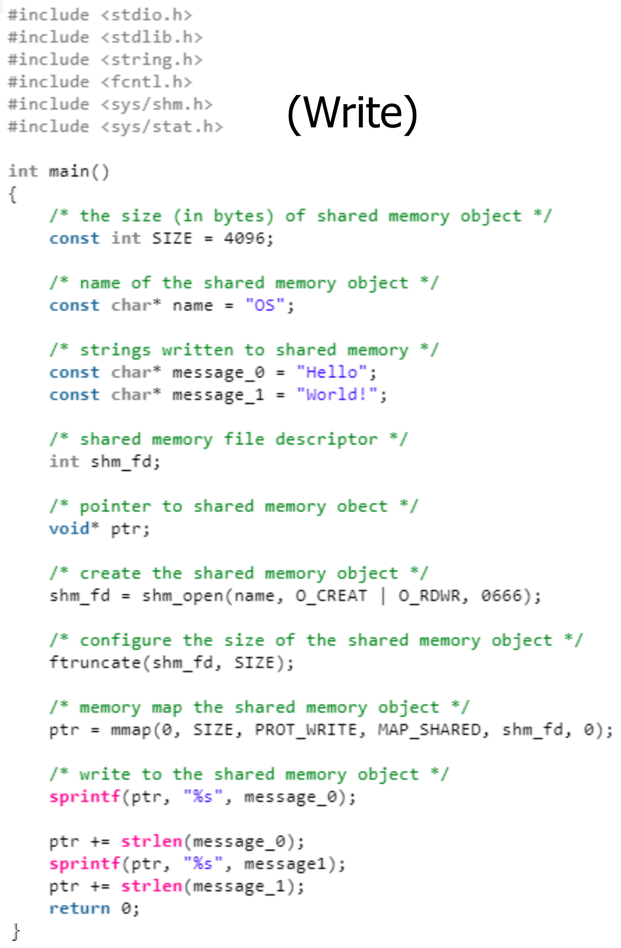
In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.



In C,

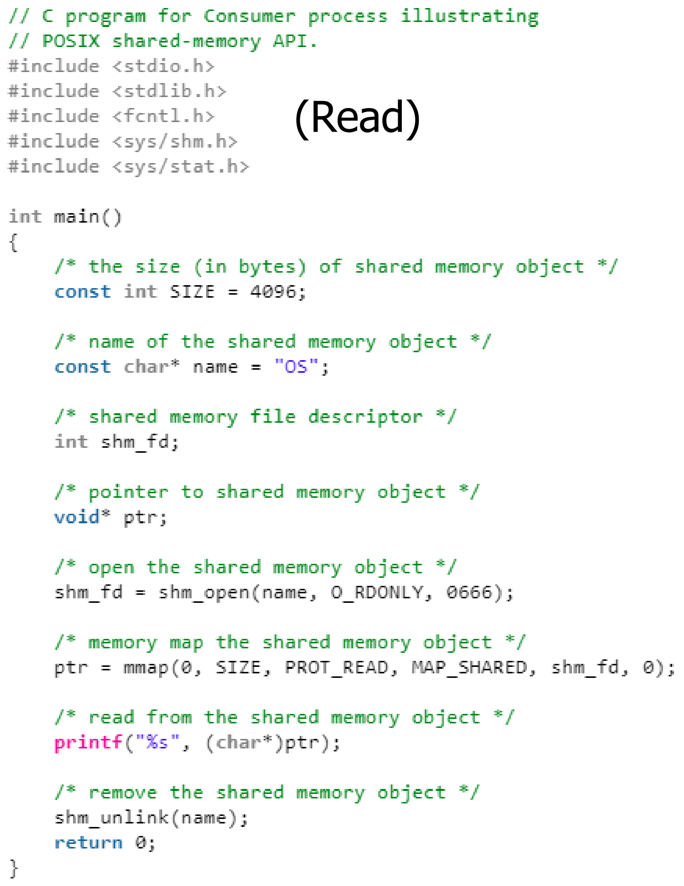
POSIX Shared-Memory API

(Write)



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char* name = "OS";

    /* strings written to shared memory */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;

    /* pointer to shared memory obect */
    void* ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);

    ptr += strlen(message_0);
    sprintf(ptr, "%s", message1);
    ptr += strlen(message_1);
    return 0;
}
```

(Read)



```c
// C program for Consumer process illustrating
// POSIX shared-memory API.
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char* name = "OS";

    /* shared memory file descriptor */
    int shm_fd;

    /* pointer to shared memory object */
    void* ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char*)ptr);

    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

https://www.geeksforgeeks.org/posix-shared-memory-api/

Parallel Programming Models

- **Shared Memory Model (without threads)**
  - An <u>advantage</u> of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.

Parallel Programming Models

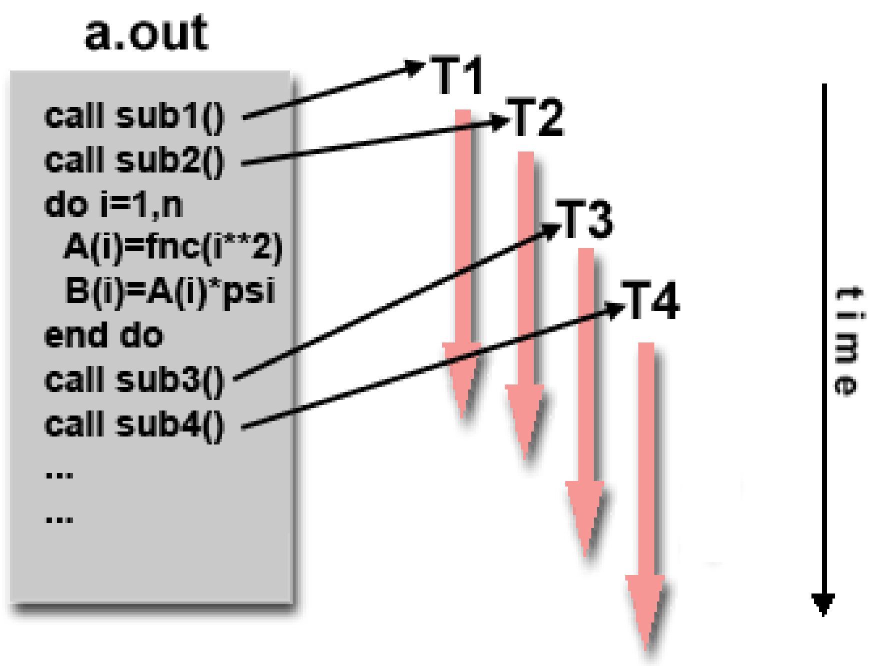
- **Shared Memory Model (without threads)**
  - An important <u>disadvantage</u> in terms of performance is that it becomes more difficult to understand and manage ***data locality***:
    - Keeping data local to the process that works on it conserves memory accesses, cache refreshes, and bus traffic that occurs when multiple processes use the same data.
    - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

Parallel Programming Models

Shared Memory Model (with threads)

•This programming model is a type of shared memory programming.
•In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.

For example:
•The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.

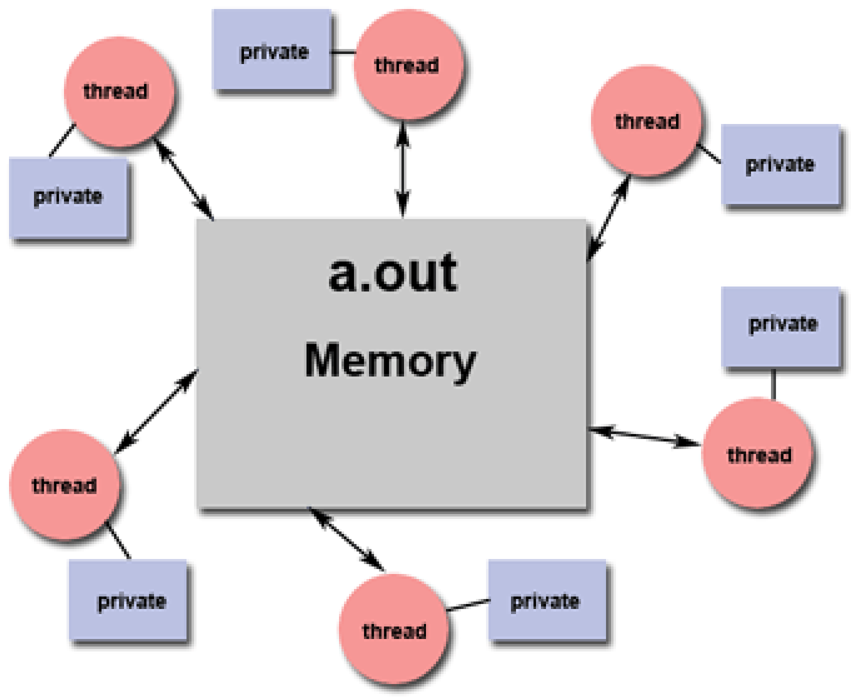


```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```

Parallel Programming Models

- **Shared Memory Model (with threads)**

•Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
•Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.



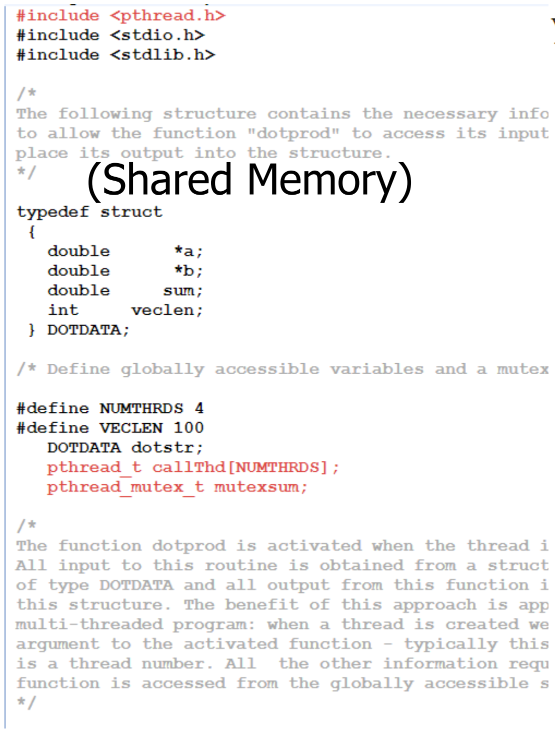
In C,

POSIX Threads Programming

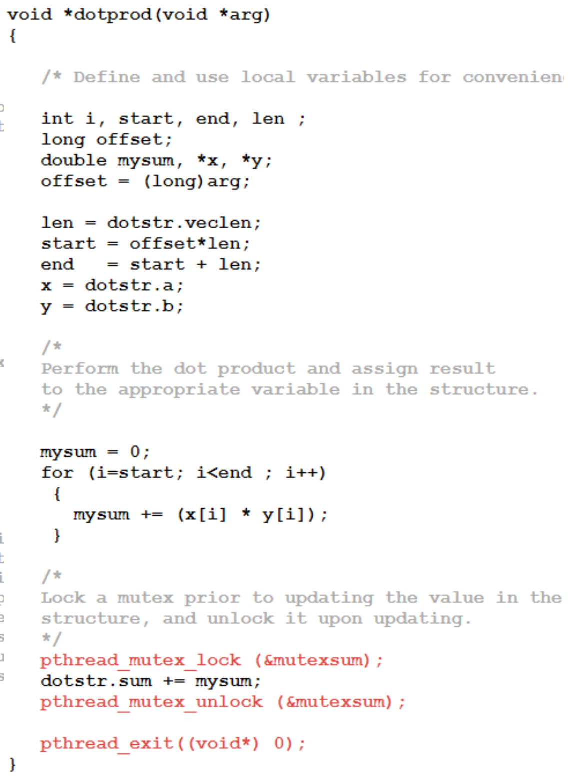
(Main)

(Thread Function)

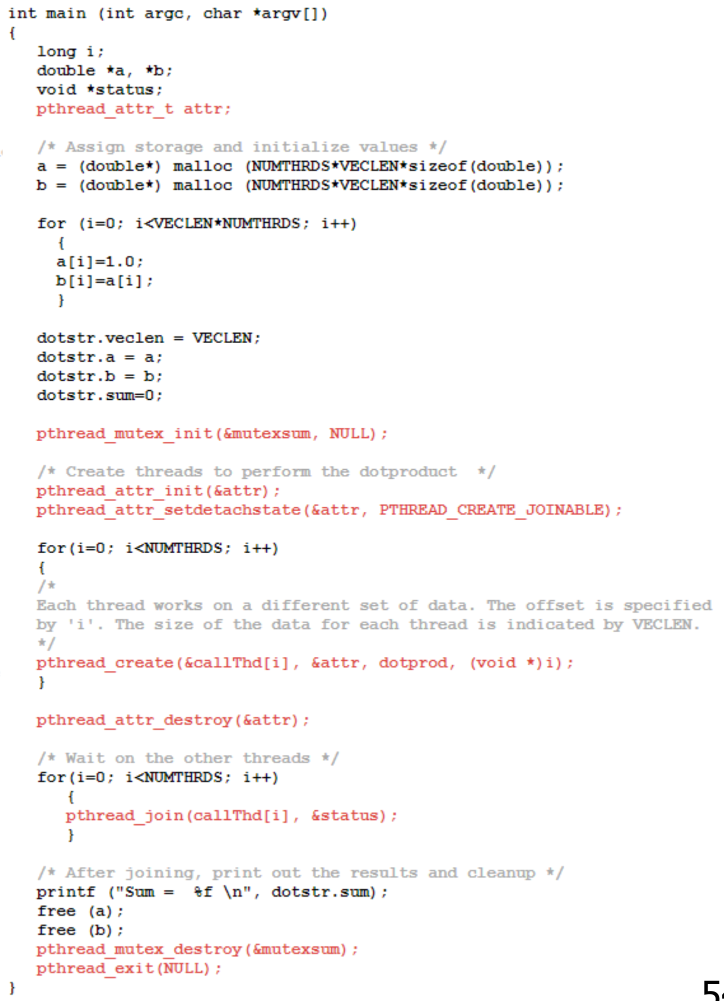
(Shared Memory)



```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
The following structure contains the necessary info
to allow the function "dotprod" to access its input
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

/*
The function dotprod is activated when the thread i
All input to this routine is obtained from a struct
of type DOTDATA and all output from this function i
this structure. The benefit of this approach is app
multi-threaded program: when a thread is created we
argument to the activated function - typically this
is a thread number. All  the other information requ
function is accessed from the globally accessible s
*/
```



```
void *dotprod(void *arg)
{

   /* Define and use local variables for convenien

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   Lock a mutex prior to updating the value in the
   structure, and unlock it upon updating.
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}
```



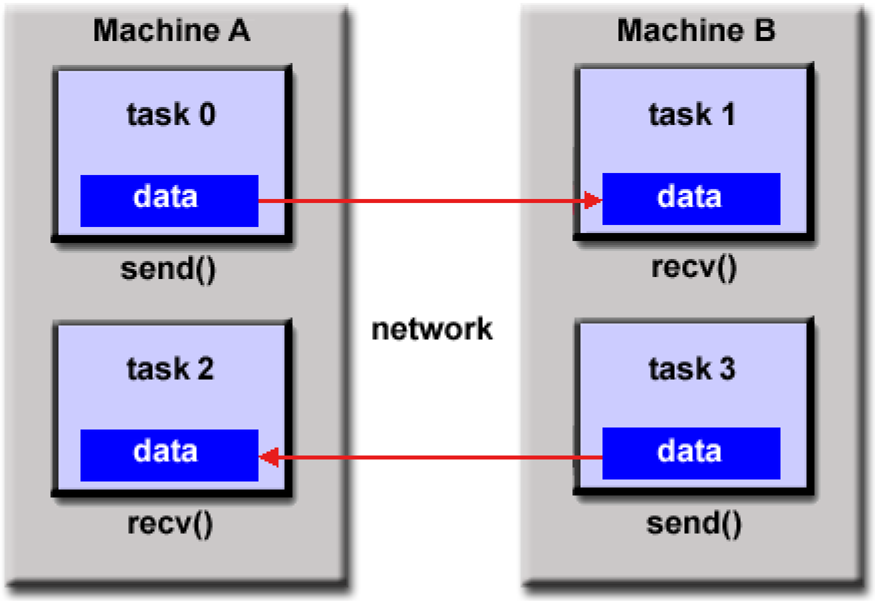
```
int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
     {
      a[i]=1.0;
      b[i]=a[i];
     }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the dotproduct  */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(i=0; i<NUMTHRDS; i++)
   {
   /*
   Each thread works on a different set of data. The offset is specified
   by 'i'. The size of the data for each thread is indicated by VECLEN.
   */
   pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Wait on the other threads */
   for(i=0; i<NUMTHRDS; i++)
     {
     pthread_join(callThd[i], &status);
     }

   /* After joining, print out the results and cleanup */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

https://computing.llnl.gov/tutorials/pthreads/#MutexCreation

Parallel Programming Models

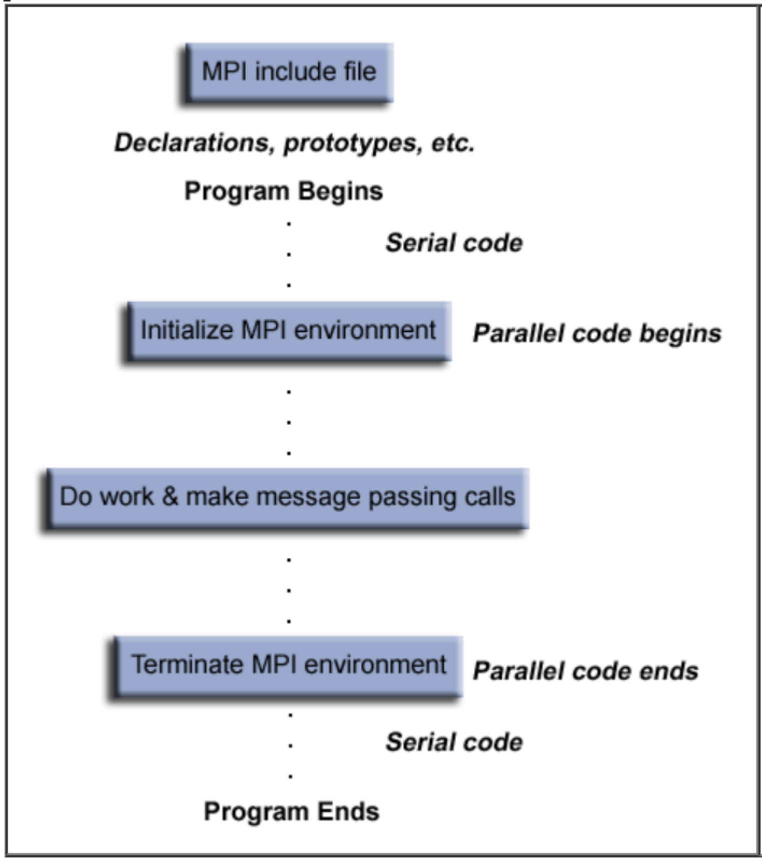
Distributed Memory Model

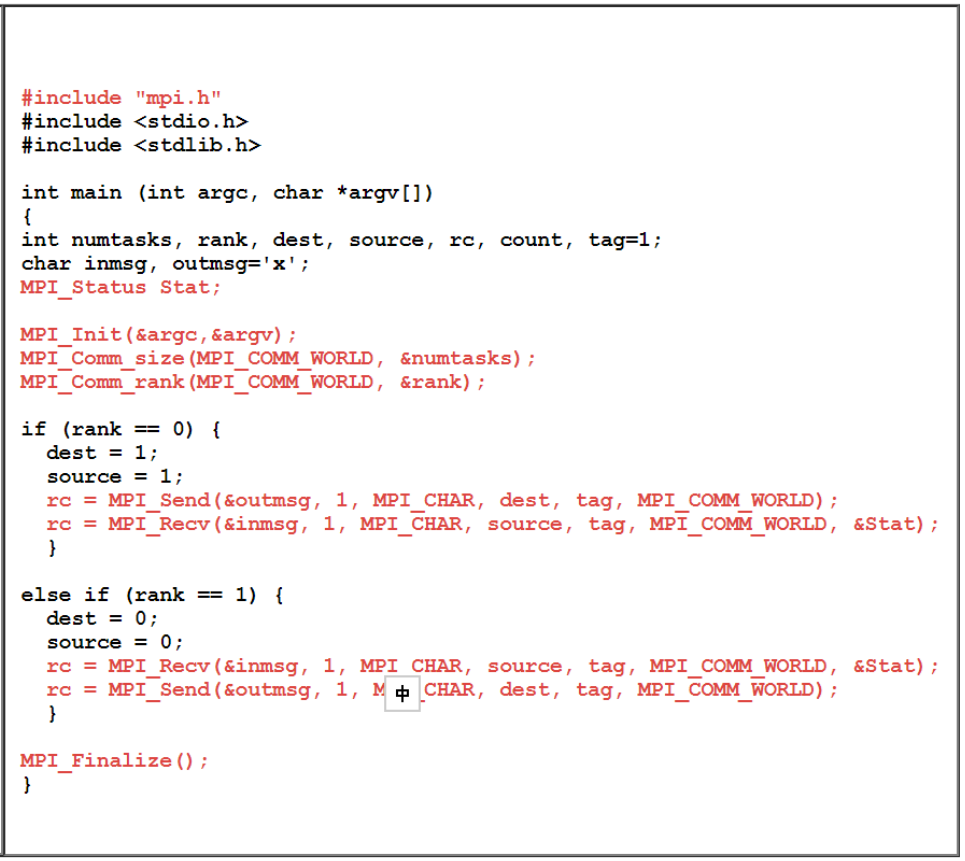
•This model demonstrates the following characteristics:
•A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
•Tasks exchange data through communications by sending and receiving messages.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



In C,

Message Passing Interface (MPI)


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

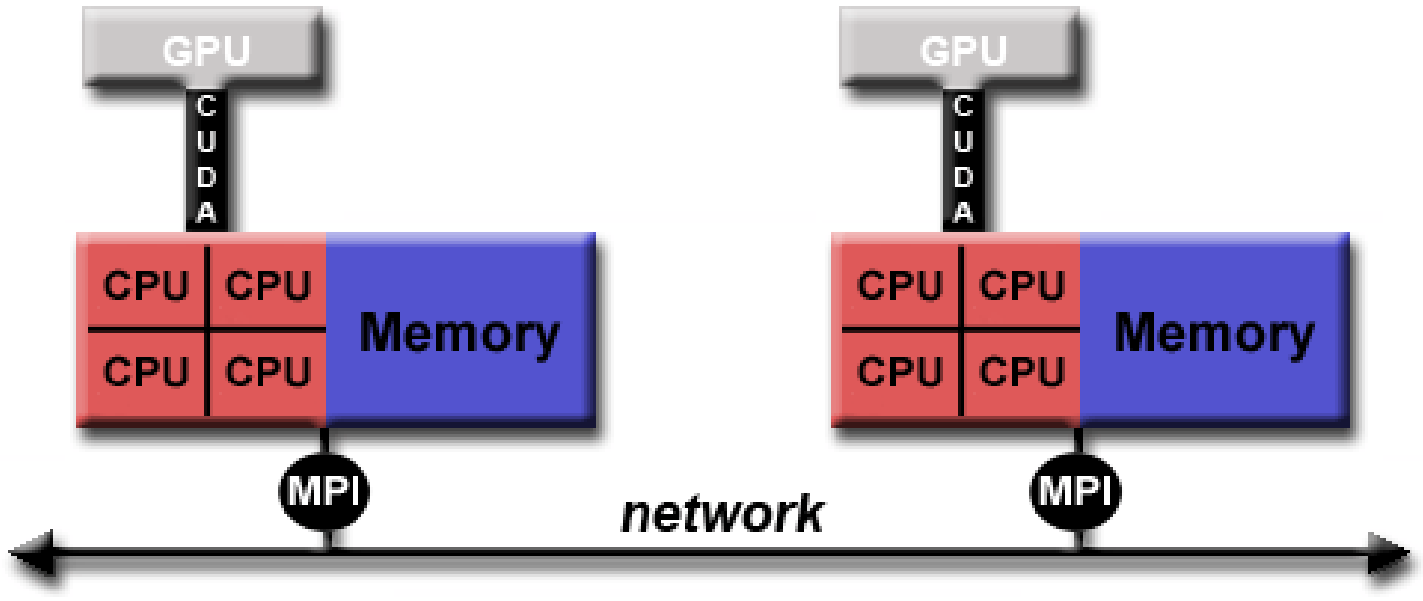


```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Parallel Programming Models

- **Hybrid Model**



Designing Parallel Programs

- **Automatic vs. Manual Parallelization**
  - Designing and developing parallel programs has characteristically been a very manual process. The programmer is typically responsible for both identifying and actually implementing parallelism.
  - Very often, manually developing parallel codes is time-consuming, complex, error-prone and ***iterative***.
  - For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs. The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

Designing Parallel Programs

- **Automatic vs. Manual Parallelization**
  - **Fully Automatic**
    - The compiler analyzes the source code and identifies opportunities for parallelism.
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops are the most frequent target for automatic parallelization.

Designing Parallel Programs

- **Automatic vs. Manual Parallelization**
  - **Programmer Directed**
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.
  - The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP).

OpenMP Example

```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

Designing Parallel Programs

- **Problem Understanding**
  - Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
  - Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

Designing Parallel Programs

- **Problem Understanding**

Example of an easy to parallelize problem:

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

Designing Parallel Programs

- **Problem Understanding**

Example of a problem with little-to-no parallelism:

Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:

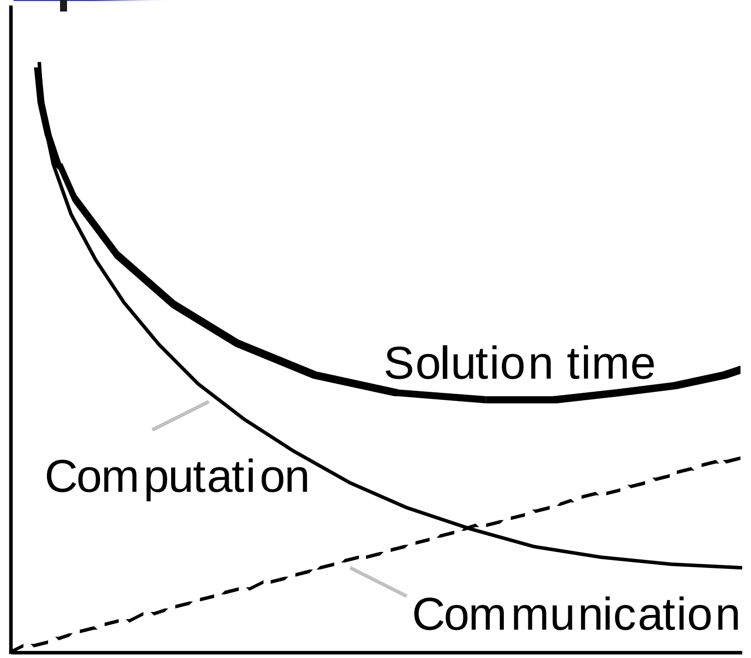
F(n) = F(n-1) + F(n-2)

The calculation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed first.

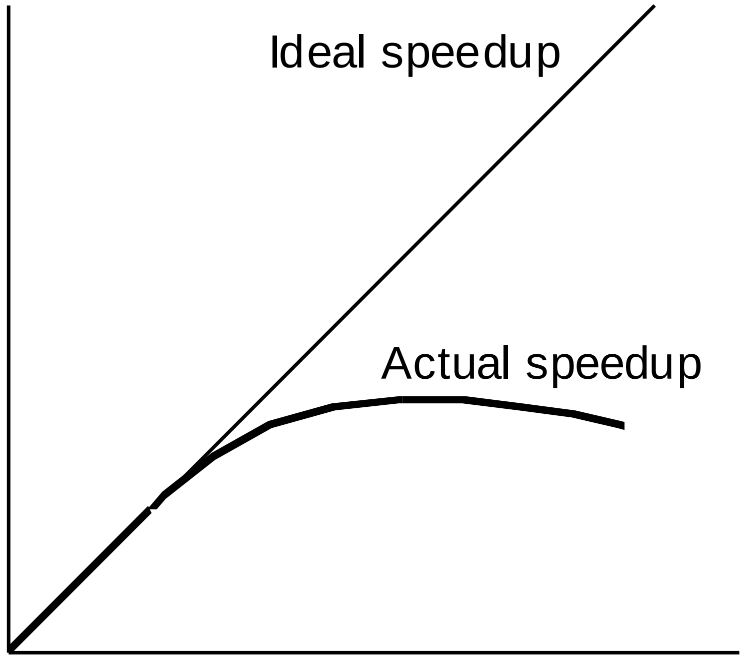
Designing Parallel Programs

- **Problem Understanding**
  - Identify the program's ***hotspots.***
  - Identify ***bottlenecks*** in the program.
  - Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.
  - Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Communication Overhead



Number of processors



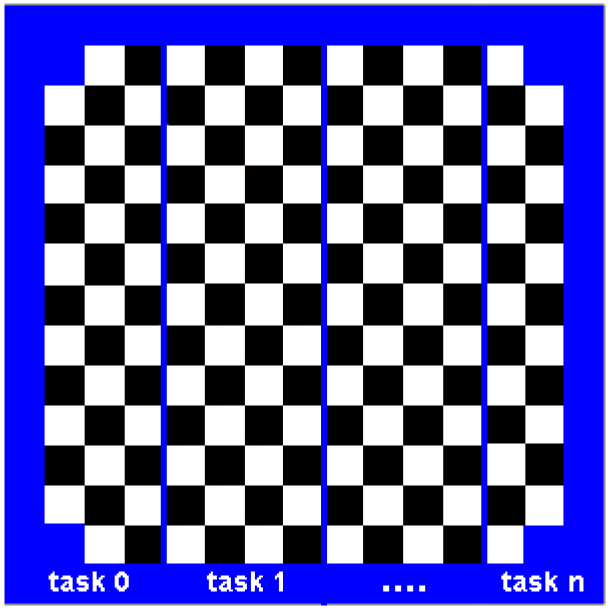
Number of processors

Trade-off between communication time and computation time in the data-parallel realization of "the sieve of Eratosthenes" algorithm.

Designing Parallel Programs

- **Communications**
  - The need for communications between tasks depends upon your problem:

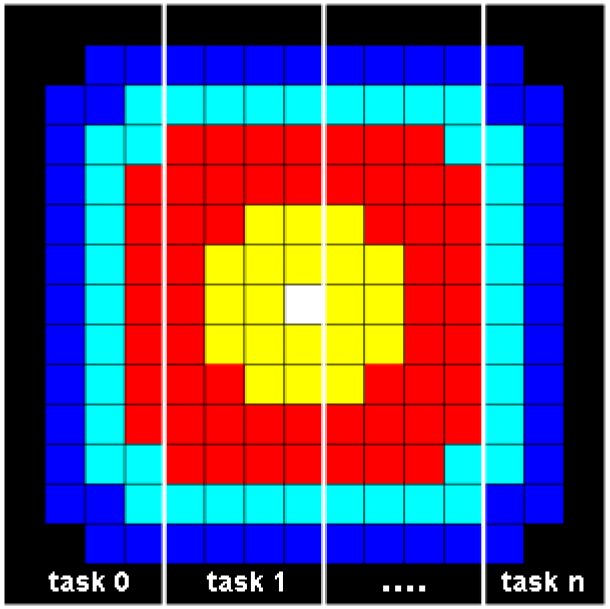
•**You DON'T need communications:**
•Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. These types of problems are often called ***embarrassingly parallel*** - little or no communications are required.
•For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.



Designing Parallel Programs

- **Communications**
  - The need for communications between tasks depends upon your problem:

•**You DO need communications:**
•Most parallel applications are not quite so simple, and do require tasks to share data with each other.
•For example, a 2-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.



Designing Parallel Programs

- **Communications**
  - **Factors to Consider:**
    - Communication overhead
    - Latency vs. Bandwidth
    - Visibility of communications
    - Synchronous vs. asynchronous communications
    - Scope of communications
    - Communication hardware setup

Designing Parallel Programs

- **Data Dependencies**
  - **Definition:**
    - A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
    - A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
    - Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

Designing Parallel Programs

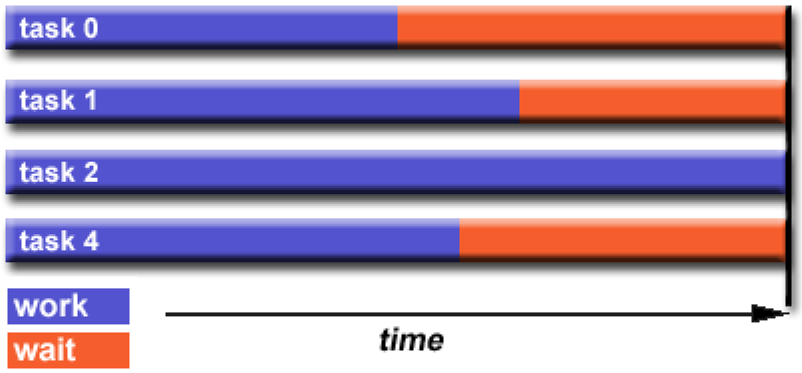
- **Data Dependencies**
  - **Example:**

    ```
    DO J = MYSTART,MYEND
       A(J) = A(J-1) * 2.0
    END DO
    ```

    - The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
    - If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
    - ***Distributed memory architecture*** - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation;
    - ***Shared memory architecture*** - task 2 must read A(J-1) after task 1 updates it

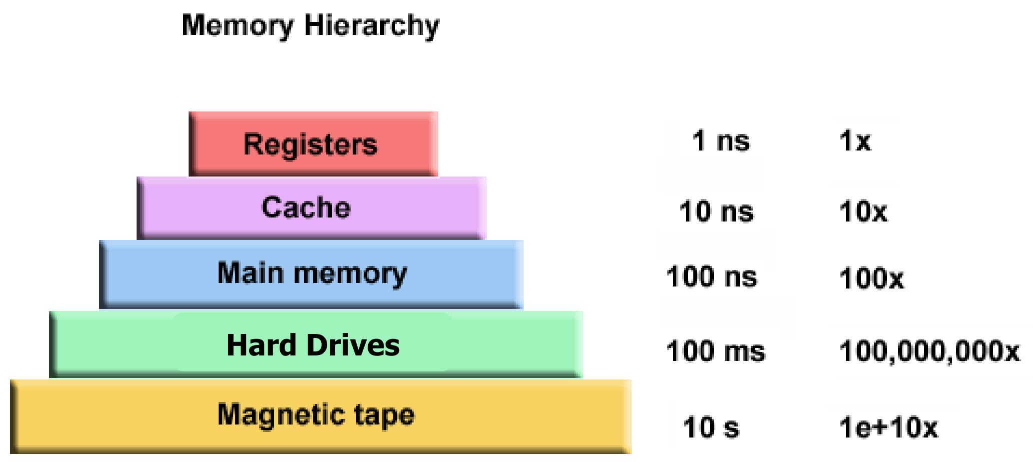
Designing Parallel Programs

- **Load Balancing**
  - Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
  - Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

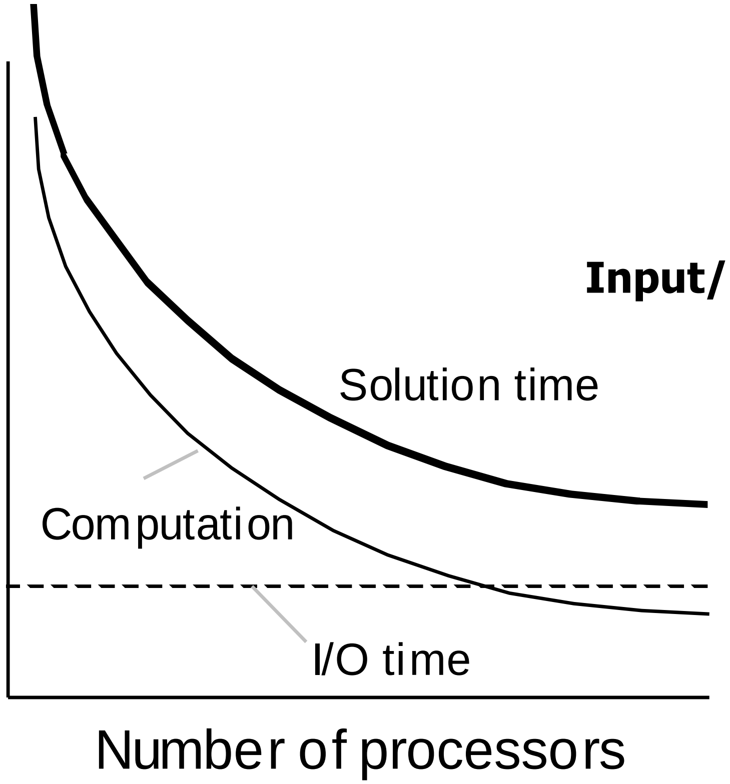


Designing Parallel Programs

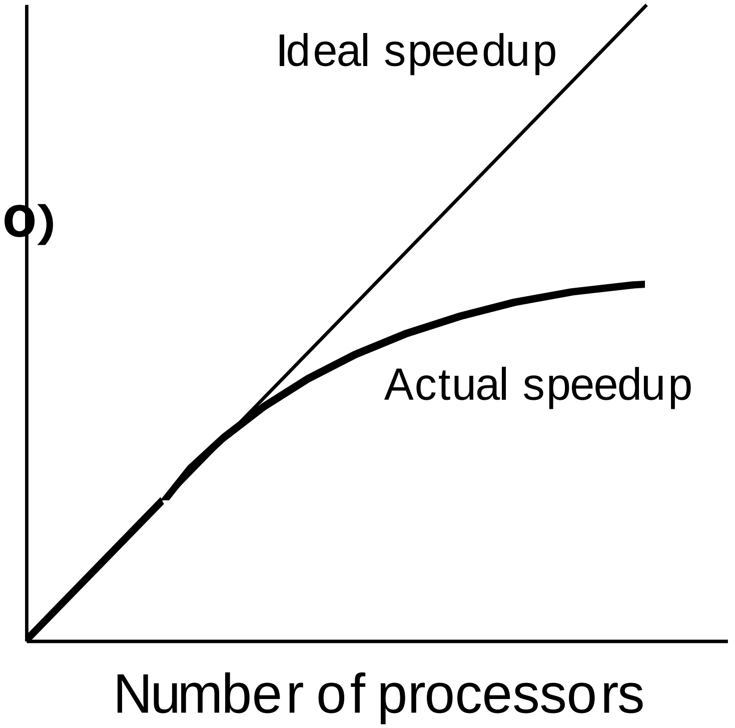
- **Input/Output (I/O)**
  - I/O operations are generally regarded as inhibitors to parallelism.
  - I/O operations require orders of magnitude more time than memory operations.



Another Reason for Sublinear Speedup: I/O Overhead



Input/Output (I/O)



Effect of a constant I/O time on the data-parallel realization of the sieve of Eratosthenes.

Designing Parallel Programs

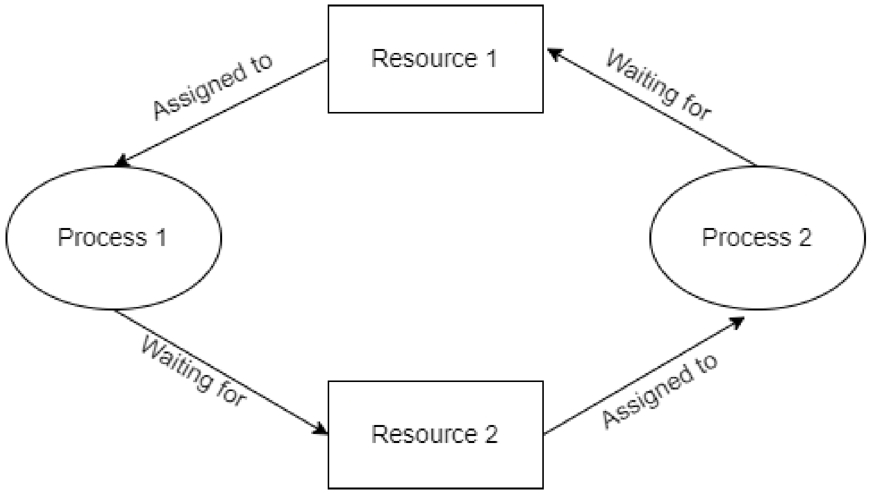
In Short,

- **Automatic vs. Manual Parallelization**
- **Problem Understanding**
- **Communication Overhead**
- **Data Dependencies**
- **Load Balancing**
- **Input/Output (I/O)**

Deadlocks

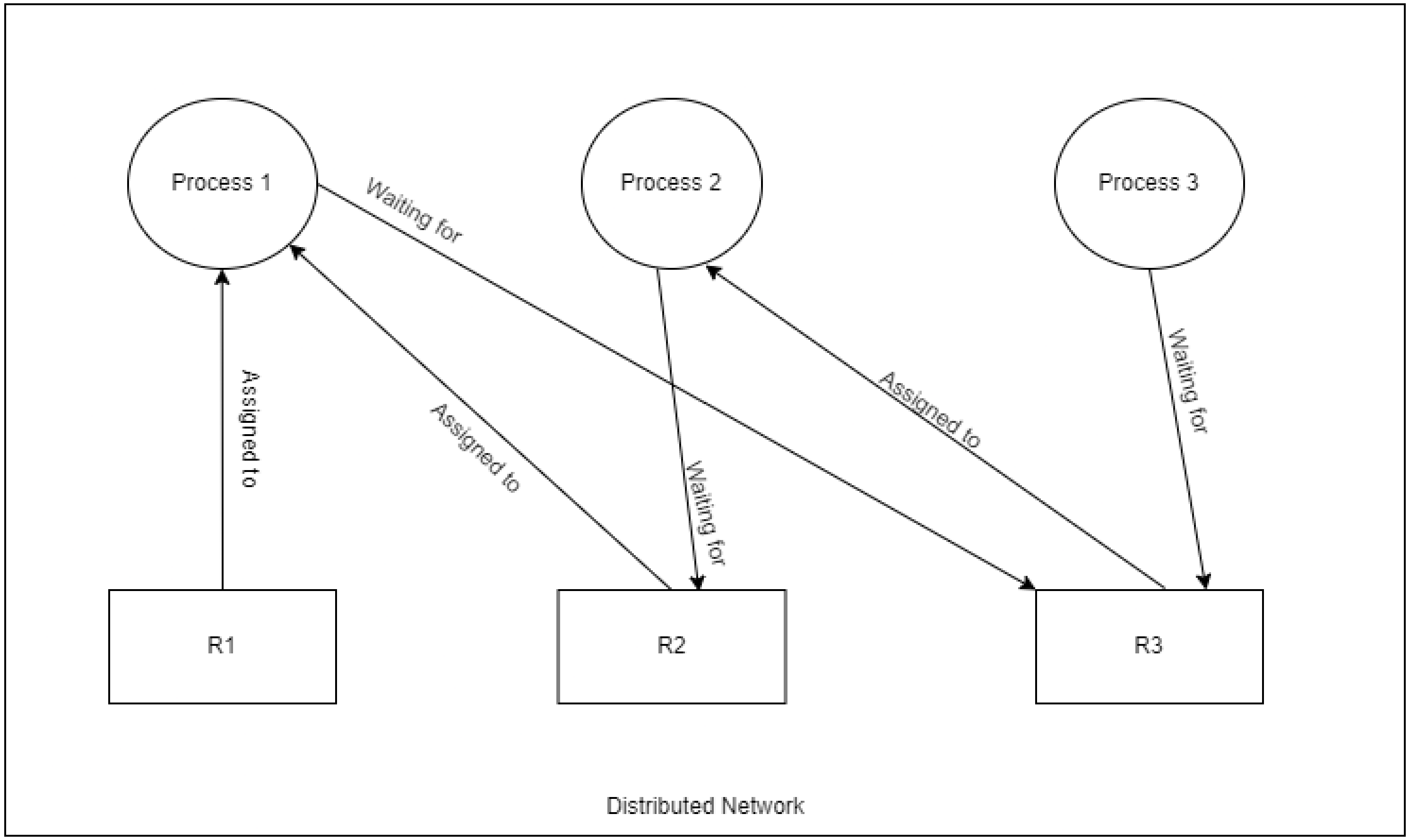
A Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource occupied by some other process. When this situation arises, it is known as Deadlock. There are 2 types of deadlocks:

1. Resource Deadlock
2. Communication Deadlock



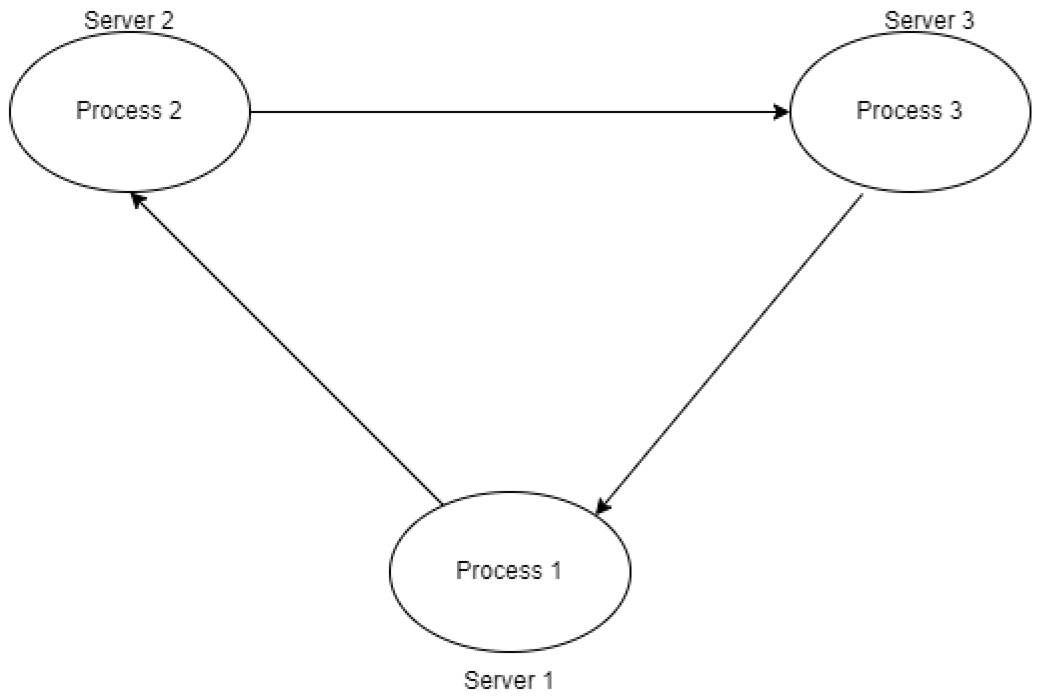
https://www.geeksforgeeks.org/distributed-system-types-of-distributed-deadlock

Resource Deadlock



https://www.geeksforgeeks.org/distributed-system-types-of-distributed-deadlock

Communication Deadlock



https://www.geeksforgeeks.org/distributed-system-types-of-distributed-deadlock

Solutions for deadlocks

- **Avoidance:** Resources are carefully allocated to avoid deadlocks.
- **Prevention:** Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.
- **Detection and recovery:** Deadlocks are allowed to occur and a detection algorithm is used to detect them. After a deadlock is detected, it is resolved by certain means.

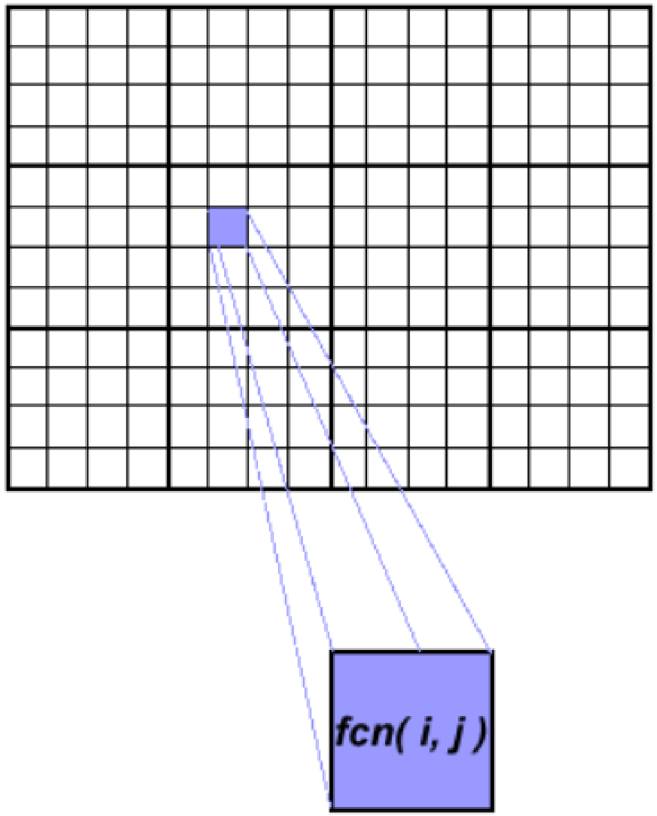
https://www.geeksforgeeks.org/distributed-system-types-of-distributed-deadlock

Parallel Examples

Array Processing

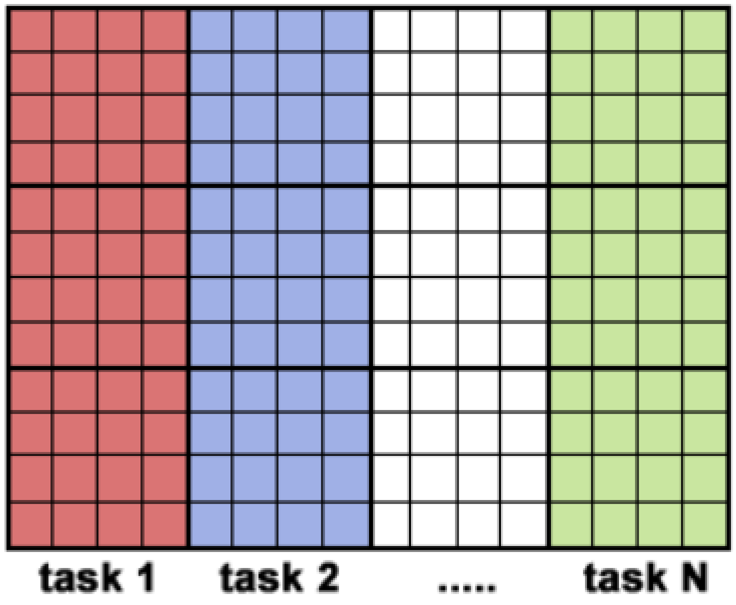
- This example demonstrates calculations on 2-dimensional array elements; a function is evaluated on each array element.
- The computation on each array element is independent from other array elements.
- The problem is computationally intensive.
- The serial program calculates one element at a time in sequential order.
- Serial code could be of the form:

```
do j = 1,n
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```



Parallel Examples

- The calculation of elements is independent of one another - leads to an embarrassingly parallel solution.
- Arrays elements are evenly distributed so that each process owns a portion of the array (subarray).
  - Distribution scheme is chosen for efficient memory access; e.g. unit stride (stride of 1) through the subarrays. Unit stride maximizes cache/memory usage.
  - Since it is desirable to have unit stride through the subarrays, the choice of a distribution scheme depends on the programming language. See the Block - Cyclic Distributions Diagram for the options.



- Independent calculation of array elements ensures there is no need for communication or synchronization between tasks.
- Since the amount of work is evenly distributed across processes, there should not be load balance concerns.

Parallel Examples

Solution 1

- After the array is distributed, each task executes the portion of the loop corresponding to the data it owns.
  For example, both Fortran (column-major) and C (row-major) block distributions are shown:

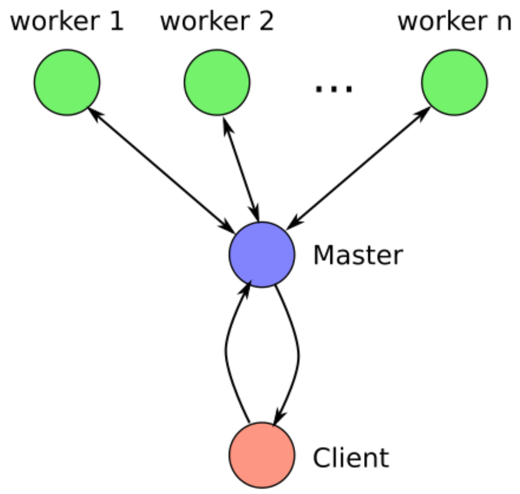
`do j = mystart, myend`
`  do i = 1, n`
`    a(i,j) = fcn(i,j)`
`  end do`
`end do`
	`for i (i = mystart; i < myend; i++) {`
	`  for j (j = 0; j < n; j++) {`
	`    a(i,j) = fcn(i,j);`
	`    }`
	`  }`

- Notice that only the outer loop variables are different from the serial solution.

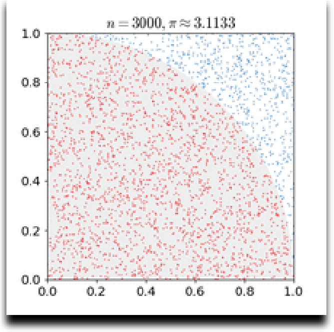
Parallel Examples

Solution 2

```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
    do i = 1,n
      a(i,j) = fcn(i,j)
    end do
  end do

  send MASTER results

endif
```



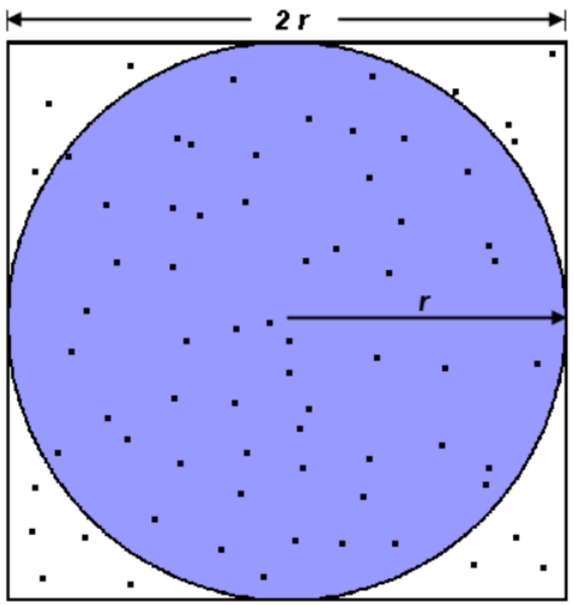
Parallel Examples



- The value of PI can be calculated in various ways. Consider the Monte Carlo method of approximating PI:
  - Inscribe a circle with radius **r** in a square with side length of **2*r***
  - The area of the circle is **Πr^2** and the area of the square is **$4r^2$**
  - The ratio of the area of the circle to the area of the square is:
    $\Pi r^2 / 4r^2 = \Pi / 4$
  - If you randomly generate **N** points inside the square, approximately
    N * Π / 4 of those points (**M**) should fall inside the circle.
  - **Π** is then approximated as:
    N * Π / 4 = M
    Π / 4 = M / N
    Π = 4 * M / N
  - Note that increasing the number of points generated improves the approximation.

- Serial pseudo code for this procedure:

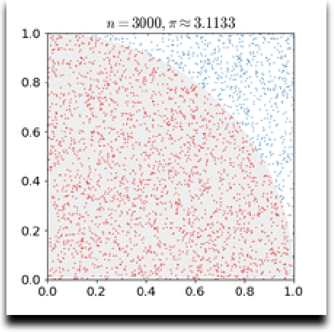
```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

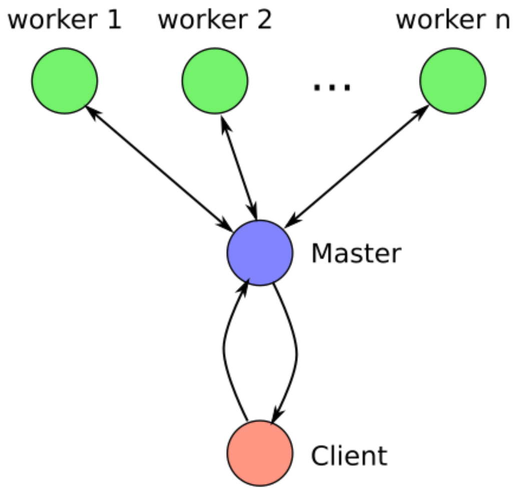
- The problem is computationally intensive - most of the time is spent executing the loop

Parallel Examples



Solution

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

  send to MASTER circle_count

endif
```



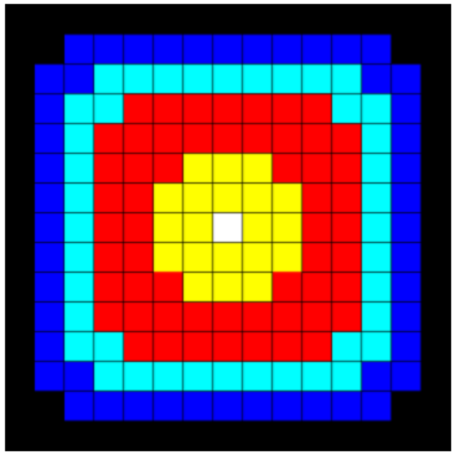
Parallel Examples

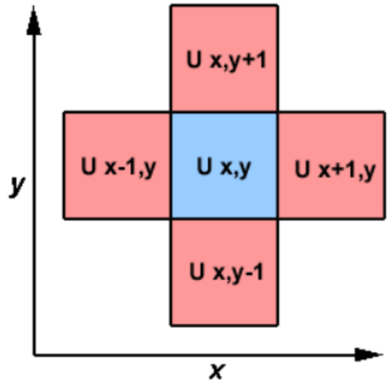
- Most problems in parallel computing require communication among the tasks. A number of common problems require communication with "neighbor" tasks.
- The 2-D heat equation describes the temperature change over time, given initial temperature distribution and boundary conditions.
- A finite differencing scheme is employed to solve the heat equation numerically on a square region.
  - The elements of a 2-dimensional array represent the temperature at points on the square.
  - The initial temperature is zero on the boundaries and high in the middle.
  - The boundary temperature is held at zero.
  - A time stepping algorithm is used.
- The calculation of an element is ***dependent*** upon neighbor element values:

$$U_{x,y} = U_{x,y}$$
$$+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy})$$
$$+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$

- A serial program would contain code like:

```
do iy = 2, ny - 1
  do ix = 2, nx - 1
    u2(ix, iy) =  u1(ix, iy)  +
        cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
        cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
  end do
end do
```

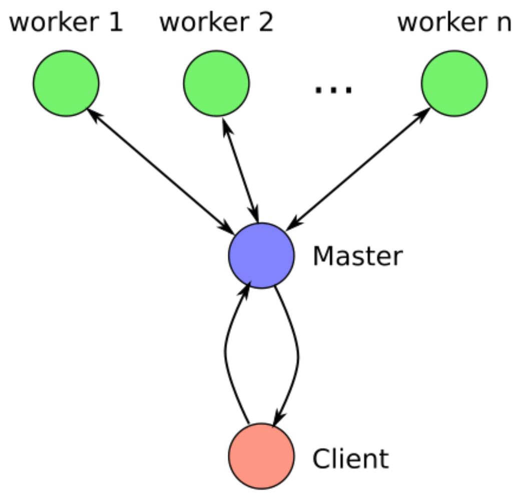




Parallel Examples

Solution

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray
  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  # Perform time steps
  do t = 1, nsteps
    update time
    send neighbors my border info
    receive from neighbors their border info
    update my portion of solution array

  end do

  send MASTER results

endif
```



Summary

- Overview
- Concepts and Terminology
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Deadlocks
- Parallel Examples
- References and More Information

References and More Information

•Author: Blaise Barney, Livermore Computing.
•A search on the WWW for "parallel programming" or "parallel computing" will yield a wide variety of information.
•Recommended reading:

- "Designing and Building Parallel Programs". Ian Foster.
  http://www.mcs.anl.gov/~itf/dbpp/
- "Introduction to Parallel Computing". Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar.
  http://www-users.cs.umn.edu/~karypis/parbook/
- "Overview of Recent Supercomputers". A.J. van der Steen, Jack Dongarra.
  OverviewRecentSupercomputers.2008.pdf

•Photos/Graphics have been created by the author, created by other LLNL employees, obtained from non-copyrighted, government or public domain (such as http://commons.wikimedia.org/) sources, or used with the permission of authors from other presentations and web pages.
•History: These materials have evolved from the following sources, some of which are no longer maintained or available:

- Tutorials developed for the Maui High Performance Computing Center's "SP Parallel Programming Workshop".
- Tutorials developed by the Cornell University Center for Advanced Computing (CAC), now available as Cornell Virtual Workshops at: https://cvw.cac.cornell.edu/topics.