高性能计算II(B)

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

School of Computer and Control, University of Chinese Academy of Sciences

Key Lab of Big Data & Knowledge Management, Chinese Academy of Sciences

Parallel Computing

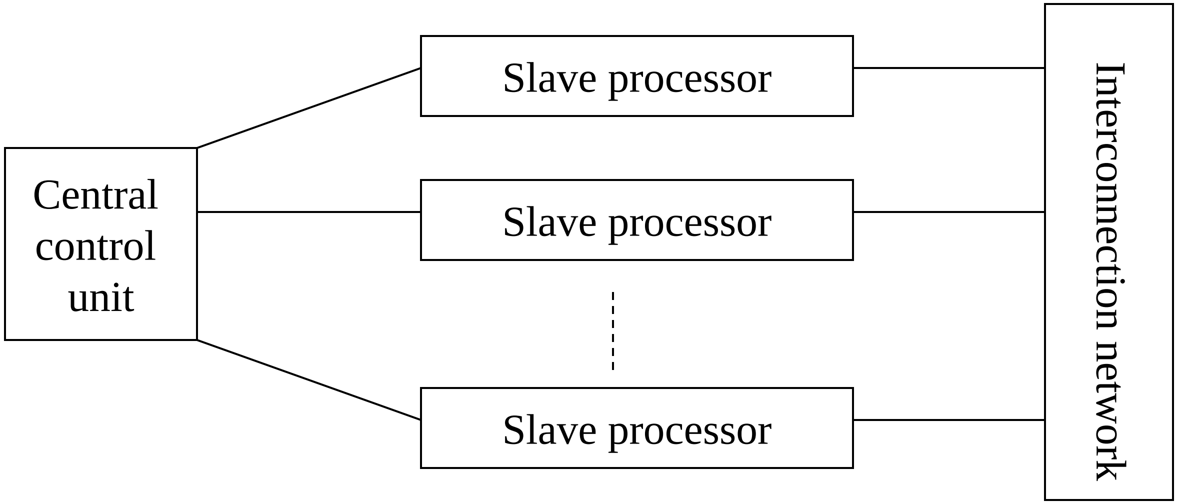
- Parallel computing platforms
- Parallel programming models
- Architecture of GPU
- Multi-threaded CPU computing vs. multi-threaded CUDA computing

Architectures

- Basic components of any architecture:
  - Processors and memory
  - Interconnect
- Logic classification based on:
  - Control mechanism
    - SISD
    - MISD
    - SIMD (Single Instruction Multiple Data stream)
    - MIMD (Multiple Instruction Multiple Data stream)
  - Address space organization
    - Shared Address Space
    - Distributed Address Space

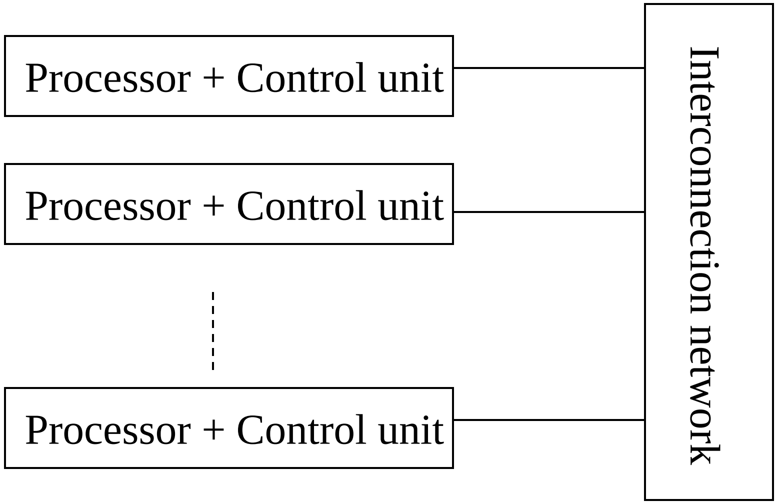
SIMD Architecture

- Multiple processors execute the same instruction
- Data that each processor sees may be different
- Individual processors can be turned on/off at each cycle ("masking")
- Examples: Illiac IV, Thinking Machines'CM-2, DAP, …
- Specialized architecture limits the popularity



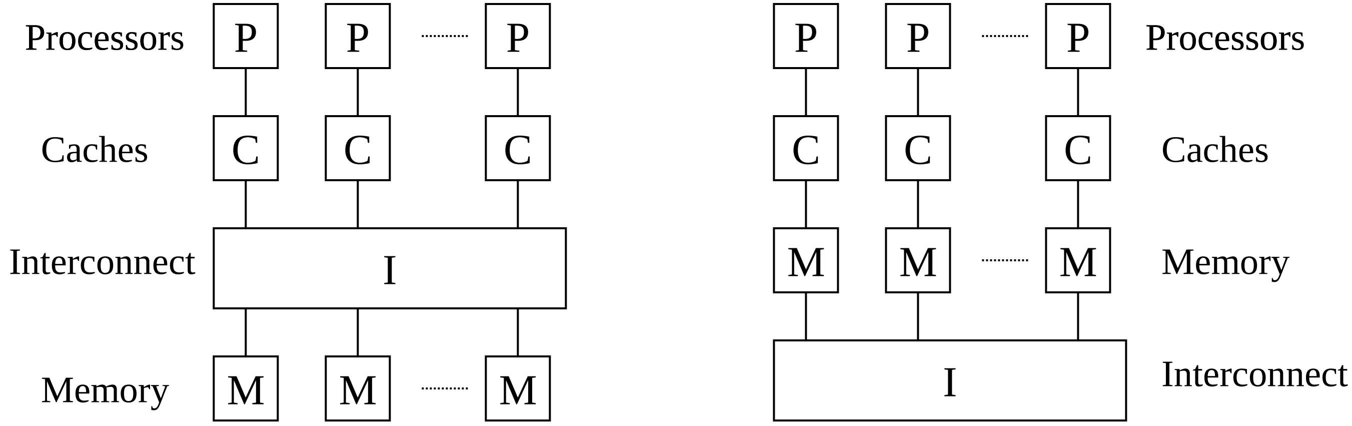
MIMD Architecture

- Each processor executes program independent of other processors
- Processors operate on separate data streams
- May have separate clocks
- Examples: IBM SP, Cray T3D & T3E, SGI Origin, Clusters, Sun Ultra Servers, …



Shared Address Space

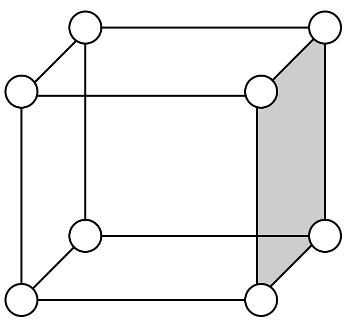
- All processors share a single global address space
- Single address space facilitates a simple programming model
- Examples: SGI Origin 2000, IBM SP2

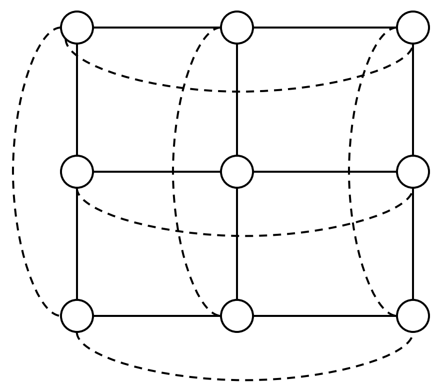


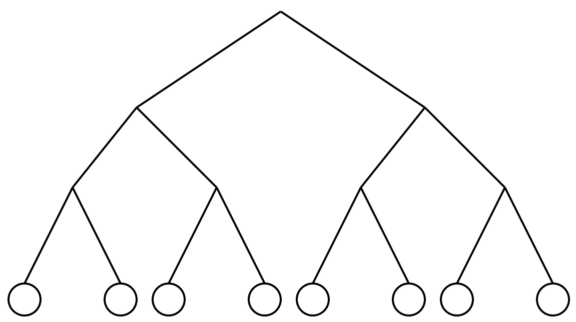
(a) UMA (b) NUMA

Static Interconnects

- Consist of point-to-point links between processors
- Examples: hypercube, mesh/torus, fat tree
- Two major characteristics:
  - Can make parallel system expansion easy
  - Some processors may be "closer" than others

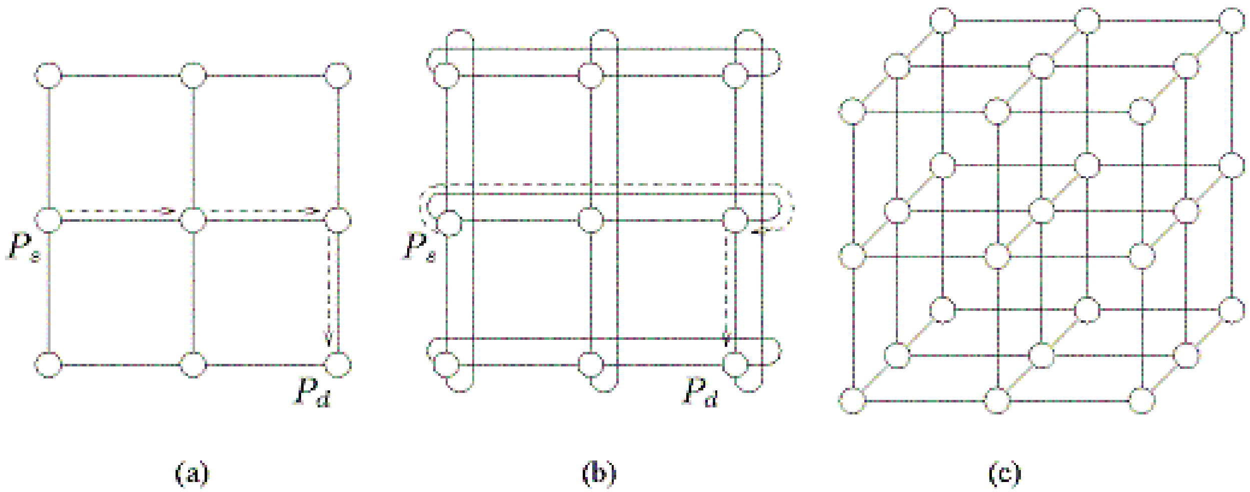






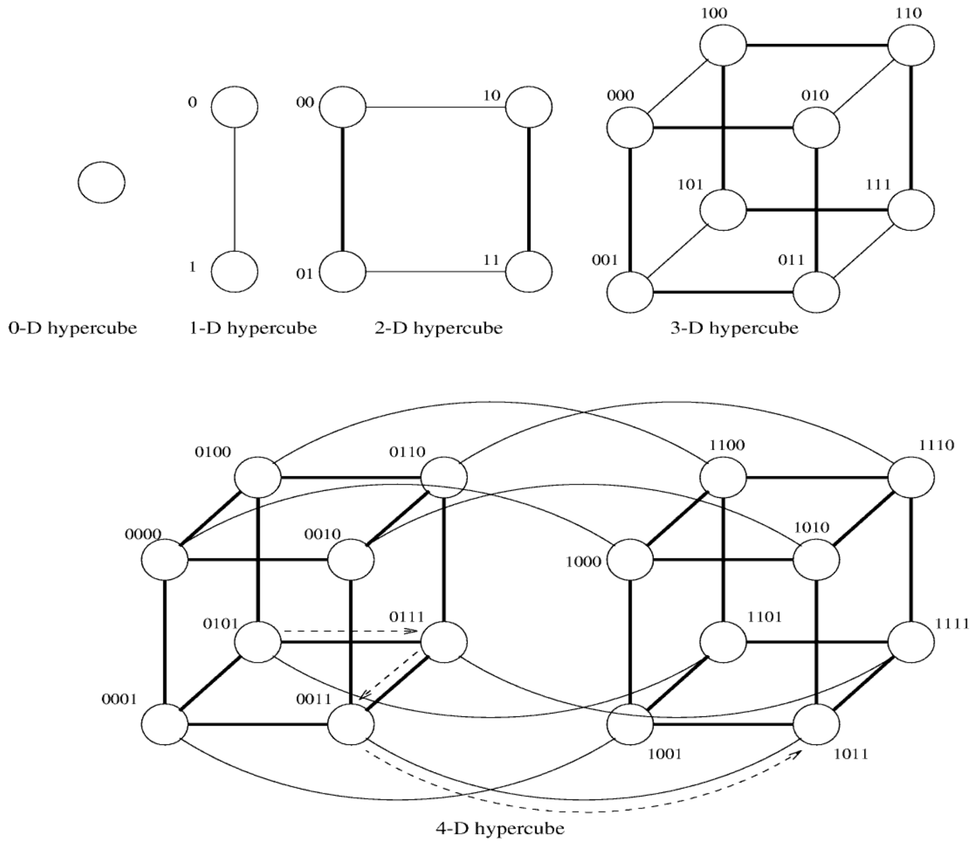
Multidimensional Meshes

- Each processor in a *d*-dimensional mesh is connected to *2d* other processors
- Data between processors routed via intermediate processors
- In practice, only 2 or 3 dimensional meshes are constructed
- Mesh with wrap around: Torus



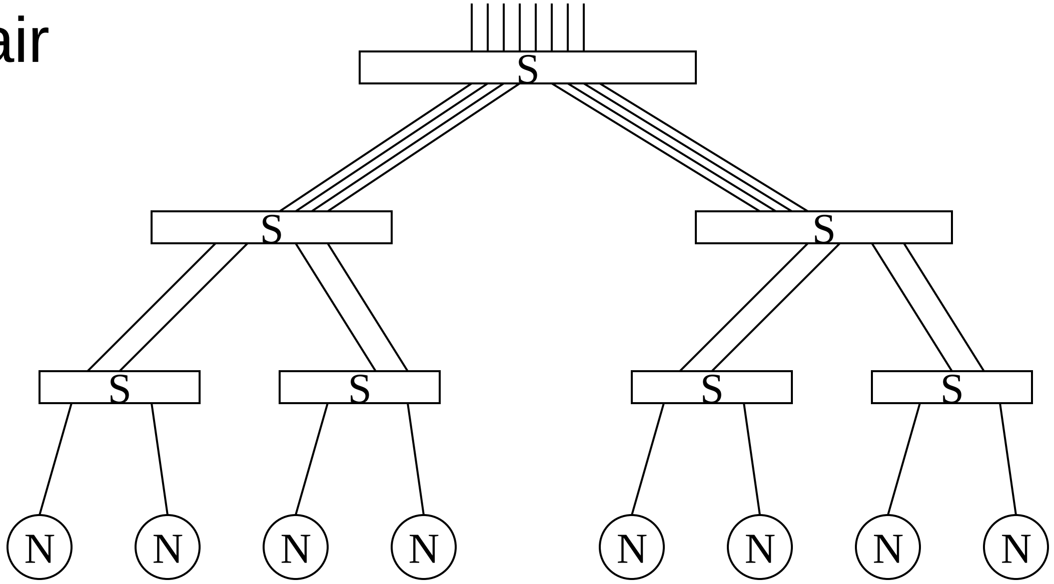
Hypercube

- A hypercube is a multidimensional mesh with exactly 2 processors in each dimension
- In a *d*-dimensional hypercube, each processor is connected with *d* other processors



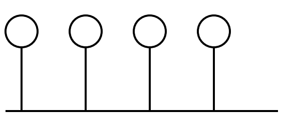
Fat Tree

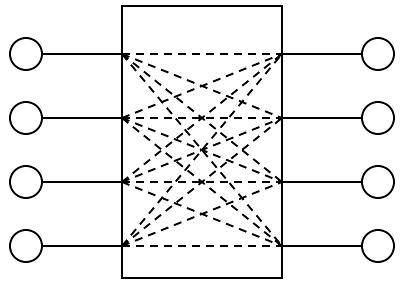
- A tree network is one where there is exactly one path between a pair of processors
- In a simple tree, the bandwidth on higher level links is shared among all processors below creating bottlenecks
- A fat tree solves the problem by using wider links at higher levels in the tree

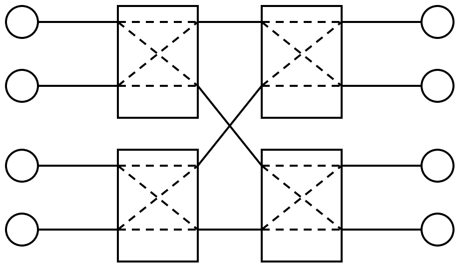


Dynamic Interconnects

- Paths are established as needed between processors
- Examples: bus based, crossbar, multistage networks
- Two major characteristics:
  - System expansion difficult
  - Processors are usually equidistant

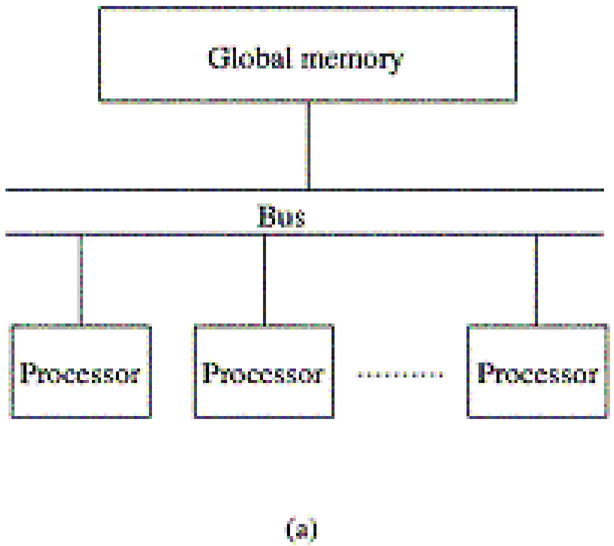


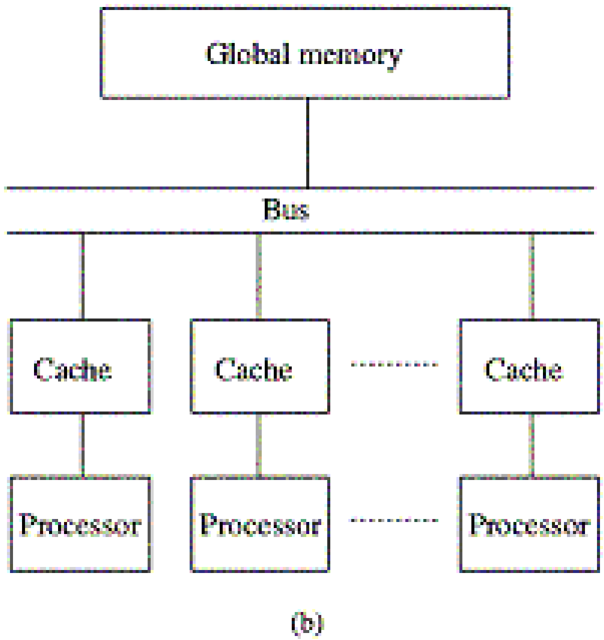




Bus Systems

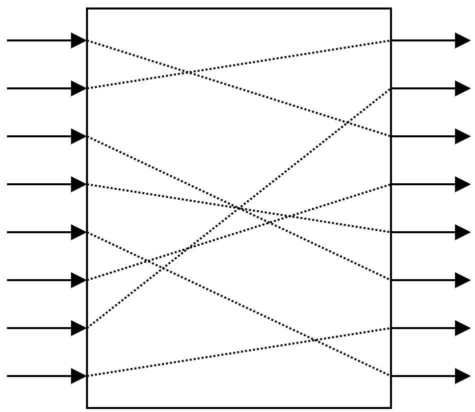
- Only one pair of processors or processor and memory can exchange data at a time
- Shared medium, good for broadcasting
- Bandwidth limitation, limited to dozens of nodes





Crossbars

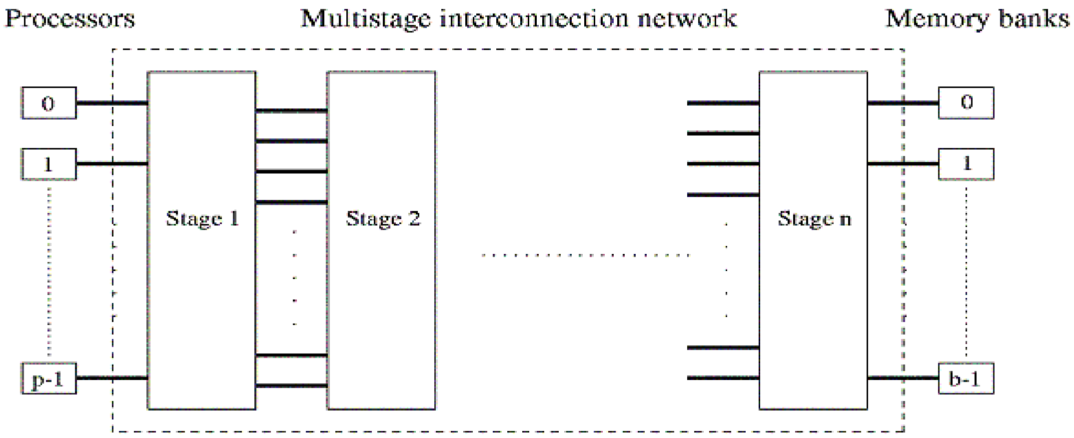
- Designed to increase the traffic between processors and memory
- Given N processors and an N-way interleaved memory, an N*N crossbar provides N simultaneous connections
- Can control crossbar settings to achieve any desired permutation



An 8*8 crossbar switch (8!=40320 permutations)

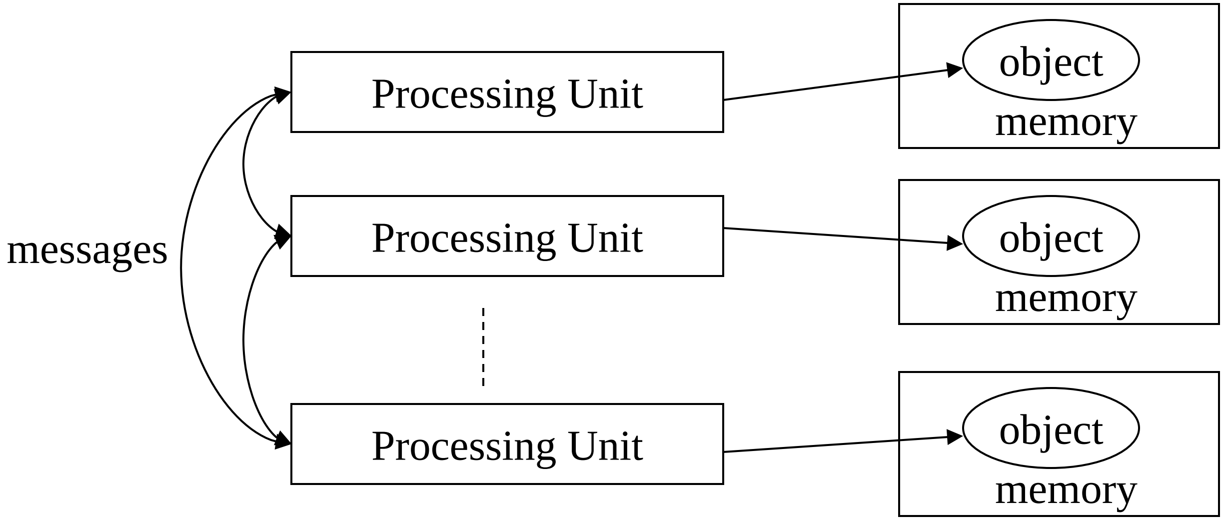
Multistage Network Systems

- Multistage networks use *Nlog(N)* switches
- Use multiple stages of switches to build interconnects
- Design choices in multistage networks:
    - Basic switch type and size
    - Number of stages
    - Connections between stages



Message Passing Platform (Distributed Address Space)

- "Shared nothing:" each processor has a private memory
- Processors can directly access only local data
- Each processing unit can be single processor or a multiprocessor
- Interaction between processors relies on message passing
- Example: clusters



Operating Systems

- Need to support tasks similar to serial OS like UNIX
  - Memory and process management
  - File system
  - Security
- Additional support needed
  - Job scheduling: assign tasks to idle processors, time sharing, space sharing
  - Parallel programming support: message passing, synchronization

Parallel Computing

- Parallel computing platforms
- Parallel programming models
- Architecture of GPU
- Multi-threaded CPU computing vs. multi-threaded CUDA computing

What is Parallel Programming?

- Parallel programming involves constructing or modifying a program for solving a given problem on a parallel machine
  - Start from a serial algorithm for solving the problem
- Goals of parallel programming – Performance!
- Effective parallel programming
  - *Key1:* Minimization of inter-processor synchronization costs
  - *Key 2*: Equal workload between processors

Alternatives to Parallel Programming

- Automatic parallelizing compiler
- Libraries
- Implicit parallel programming:
  - Programmer directives to help compiler analysis
  - Example OpenMP
- Explicit parallel programming
  - Easiest to support
  - Example MPI

Types of Parallelism

- Data parallelism:
  - Partition the data across processors
  - Each processor performs the same operations on its local data partitioning
- Task parallelism:
  - Assign independent modules to different processors
  - Each processor performs different operations

Data Dependence & Parallelization

- Consider the following loop of a C program

```
for (i=0;i<1000;i++)
  a[i]=b[i]+c[i]
```

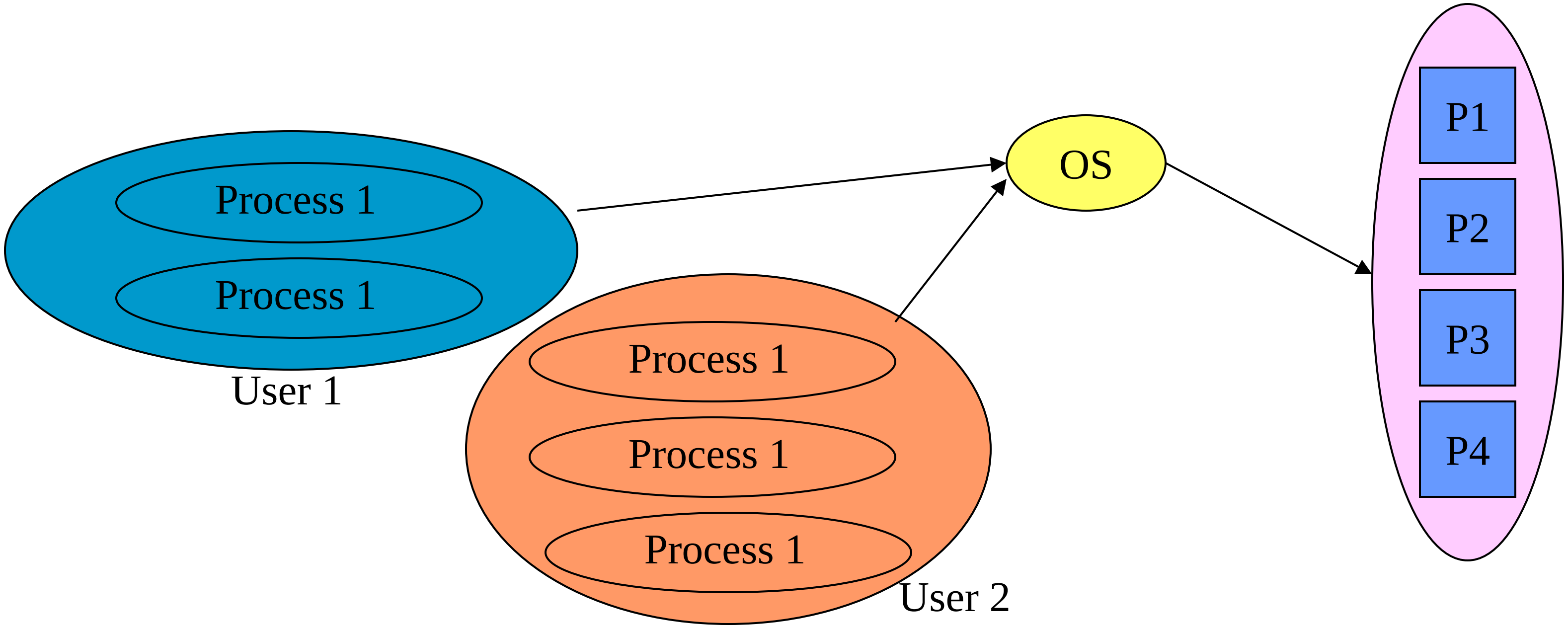
- If one unfolds the loops, the statements would be executed as follows:

```
a[0]=b[0]+c[0];
a[1]=b[1]+c[1];
…..
a[999]=b[999]+c[999];
```

- Can each iteration be executed in parallel?

Processes

- A process is a program along with all of its environment
  - How to communicate?
  - Who creates it?
  - Who map processes to physical processors?

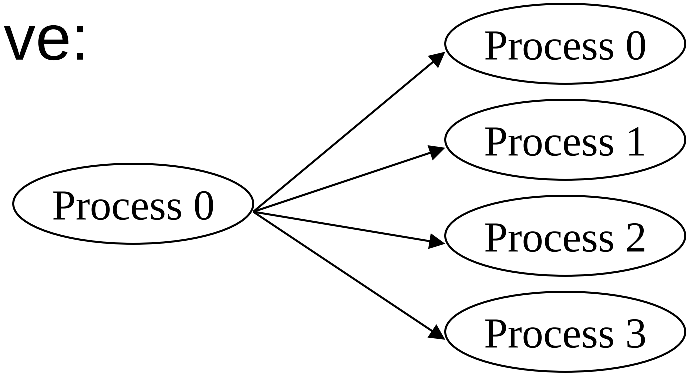


Process Creation

- Use a process creation primitive:

  m_set_procs(nproc);

  m_fork(func,[args]);



- Each child process is an exact copy of its parent
- All the variables associated with the parent are private for the parent and each child unless explicitly made shared
- The parent had ID=0, children share the other *nproc-1* IDs

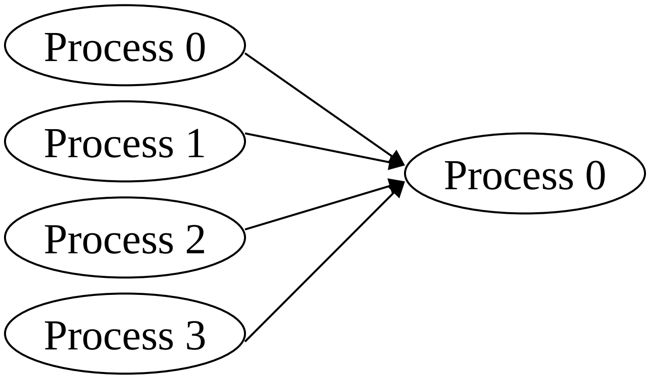
  id=m_get_myid();

  nprocs=m_get_procs();

Process Destruction

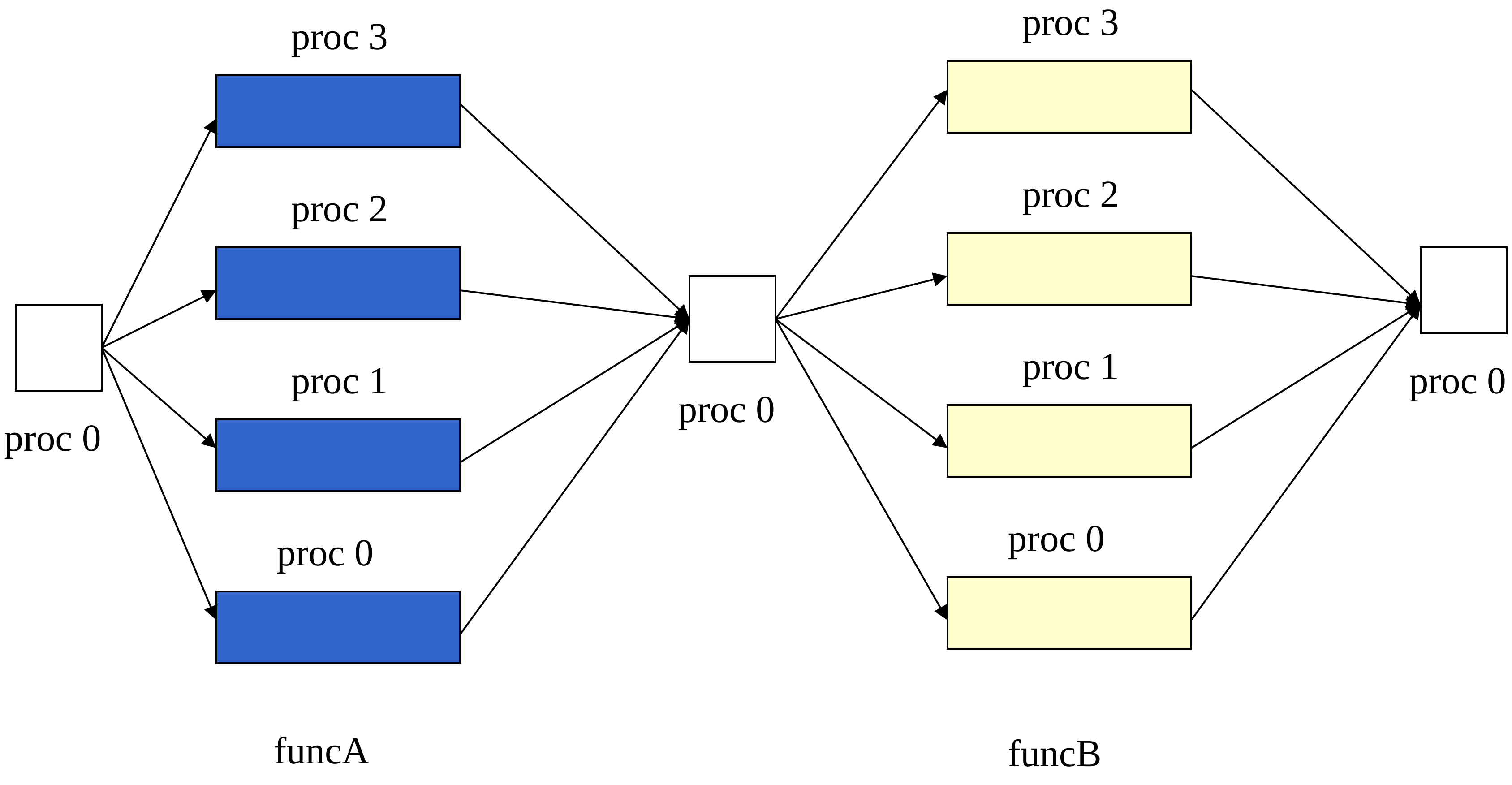
- All child processes are terminated; only the parent remains
- The parent waits for all child processes to terminated before returning
- Use a process destruction primitive

m_kill_procs();

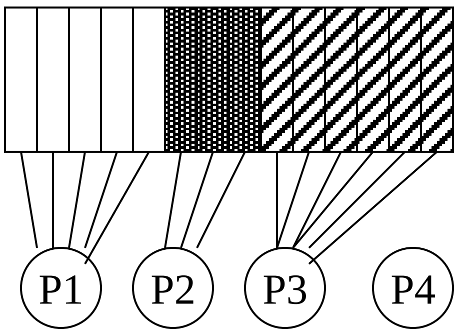


Parallel Program Model

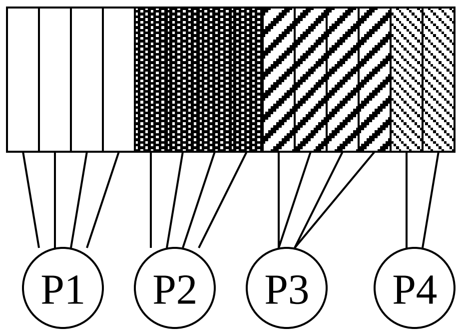
- Processes executing in a data parallel fashion



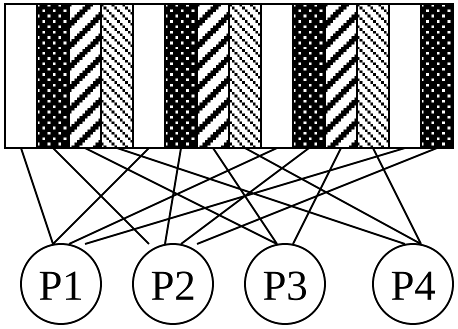
Loop Scheduling



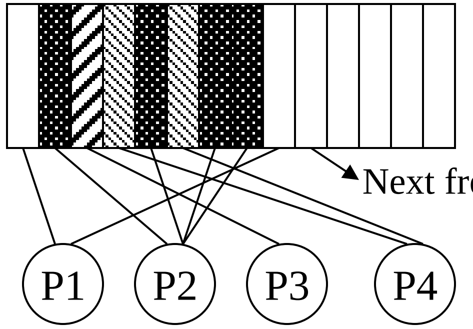
Prescheduling



Static blocked



Static interleaved



Dynamic

Prescheduling

- Assign work using a schedule array

```
float *a, *b, *c;
int low[ ] = {0,50,80}
int high[ ] = {50,80,120}

void parallel_func( ) {
        int id, i;
        id = m_get_myid( );
        for (i=low[id]; i<high[id]; i++)
                a[i] = b[i] + c[i];
}
```

Static Blocked Scheduling

- Assign a contiguous chunk of iterations based on process ID

```
float *a, *b, *c;
void parallel_func( ) {
    int id, i, nprocs;
    int low, high;
    id = m_get_myid( );
    nprocs = m_get_numprocs( );
    low = id*100 / nprocs;
    high = (id+1)*100 / nprocs;
    for (i=low; i<high; i++)
        a[i] = b[i] + c[i];
} /* end of parallel_func */
```

Static Interleaved Scheduling

- Assign iterations in a round-robin way based on process ID

```
float *a, *b, *c;
void parallel_func( ) {
    int id, i, nprocs;
    id = m_get_myid( );
    nprocs = m_get_numprocs( );
    for (i=id; i<100; i+=nprocs)
        a[i] = b[i] + c[i];
}
```

Dynamic Scheduling

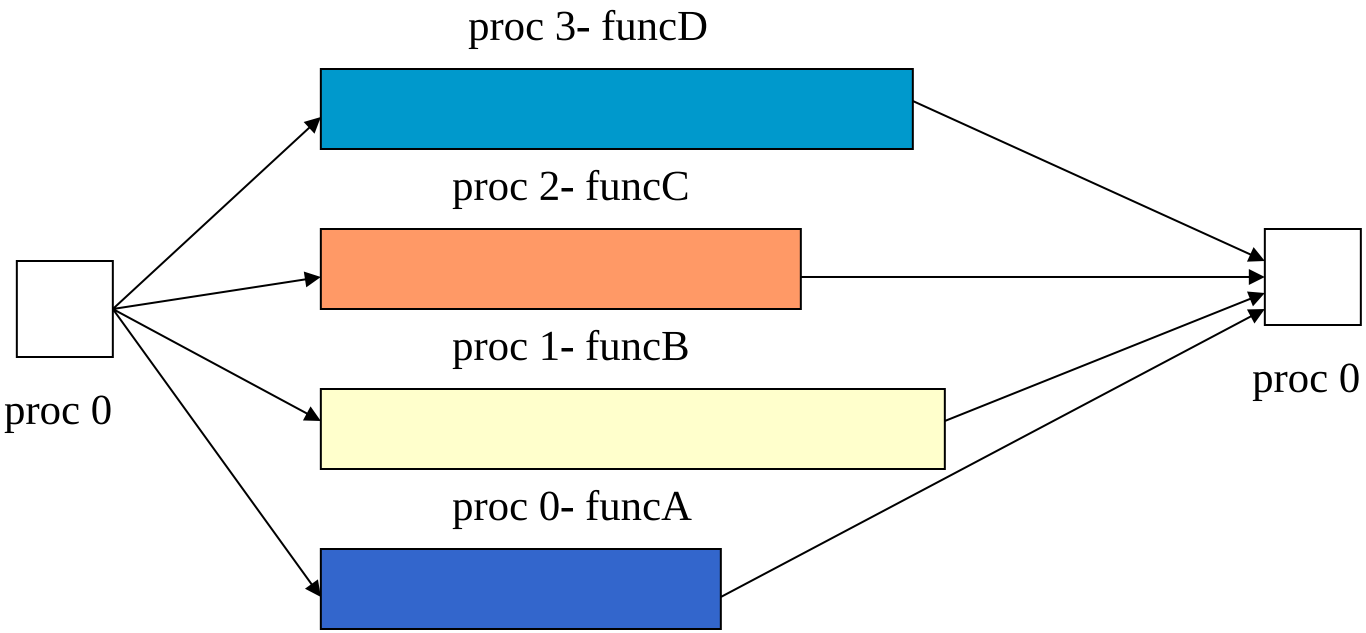
- Self scheduling: processes execute iterations using a shared counter

```
float *a, *b, *c;
int i;

i = 0;

void parallel_func( ) {
    int i1, i2;
    int chunk=2;
```

```
while (i1<100) {
    m_lock( );
        i1 = i;
        i += chunk;
    m_unlock( );
    for (i2=i1; i2<min(i1+chunk,100); i2++)
        a[i2] = b[i2] + c[i2];
} /* end of while */
} /* end of parallel_func */
```

Parallel Program Model

- Processes executing in a function/task parallel fashion

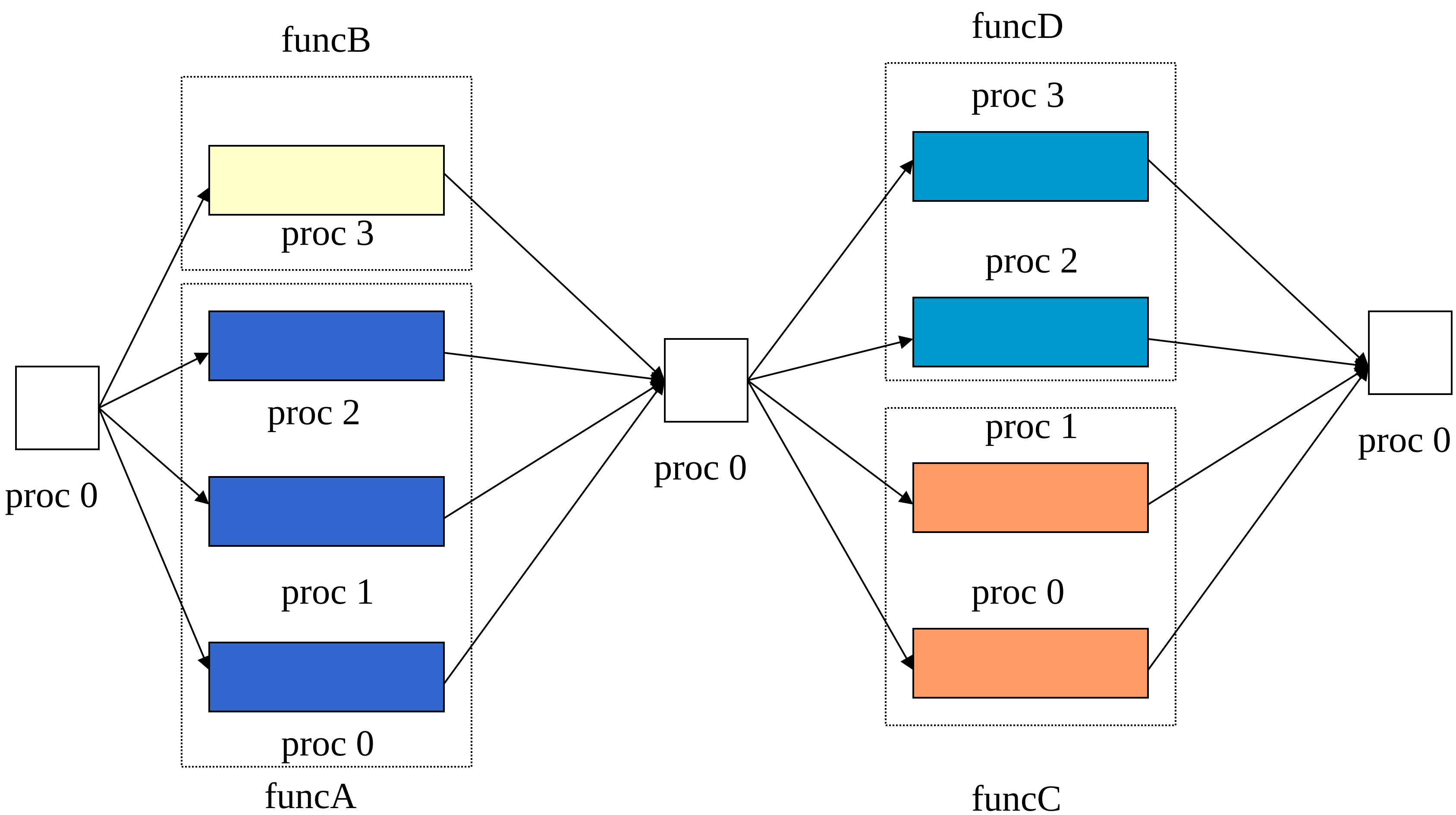


Parallel Program Model

```
main() {
            m_set_procs(4);
            m_fork(func);
            m_kill_procs();
}
void func() {
Int id;
       id = m_get_myid();
       switch (id) {
            case 0: funcA();
                        break;
            case 1: funcB();
                        break;
            case 2: funcC();
                        break;
            case 3: funcD();
                        break;
       }
}
```

Parallel Program Model

- Processes executing in a function & data parallel fashion



Parallel Program Model

```
main() {
    m_set_procs(4);
    m_fork(func1);
    m_park_procs();
        …..
    m_rele_procs();
    func2();
    m_kill_procs();
}
```

```
void func1() {
    switch (id) {
        case 0,1,2: funcA();
                    break;
        case 3: funcB();
                break;
    }
}

void func2() {
    switch (id) {
        case 0,1: funcC();
                  break;
        case 2,3: funcD();
                  break;
    }
}
```

Performance of Parallel Programs

- T = execution time of the best serial algorithm
- T_p = execution time of the parallel algorithm with p processors
- Speedup, $S_p = T / T_p$
- Efficiency $E_p = S_p / p$
- However, E_p is realistically never 100% due to:
  - Synchronization and communication cost across processors
  - Less-than-perfect load balancing among processors
  - Assume also that α represents the fraction that cannot be parallelized, i.e., the serial fraction
  - Then, using p processors in parallel and assuming perfect speedups in the parallelized portion, the parallel execution time,

$$T_p = T\,(\alpha + (1 - \alpha) / p)$$

Example

```
float sum,sum0,sum1;
main() {
    m_set_procs(2);
    m_fork(parallel_func);
    m_kill_procs();
    sum=sum0+sum1;
    printf("total sum is %lf\n",sum);
}
void parallel_func(){
    int id;
    id=m_get_myid();

    if (id==0) sum0=1.0+2.0;
    else sum1=3.0+4.0;
}
```

Example

```
float total_sum;
main() {
    total_sum=0.0;
    m_set_procs(2);
    m_fork(parallel_func);
    m_kill_procs();
    printf("total sum is %f\n",total_sum);
}
void parallel_func() {
    int id;
    float  partial_sum;
     id=m_get_myid();

    if (id==0) partial_sum=1.0+2.0;
    else partial_sum=3.0+4.0;

    total_sum+=partial_sum;
}
```

Example

Possible outputs?

Total sum is 10.0
(ts = 0;
P0: ts = ts(0) + 3;
P1: ts = ts(3) + 7;

Total sum is 7.0
(ts = 0;
P0: ts = ts(0) + 3;
P1: ts = ts(0) + 7;

Total sum is 3.0
(ts = 0;
P1: ts = ts(0) + 7;
P0: ts = ts(0) + 3;

Contention

- Random process scheduling creates problems with respect to modifying shared variables
- The final values of a shared variable can differ from run to run depending on the order in which it was modified
- Root of this contention problem is the simultaneous accessing of a shared variable
- Solution ?

Restrict the access to shared variables

Variable Locks

- Locks are used to provide exclusive access for the modification of a variable

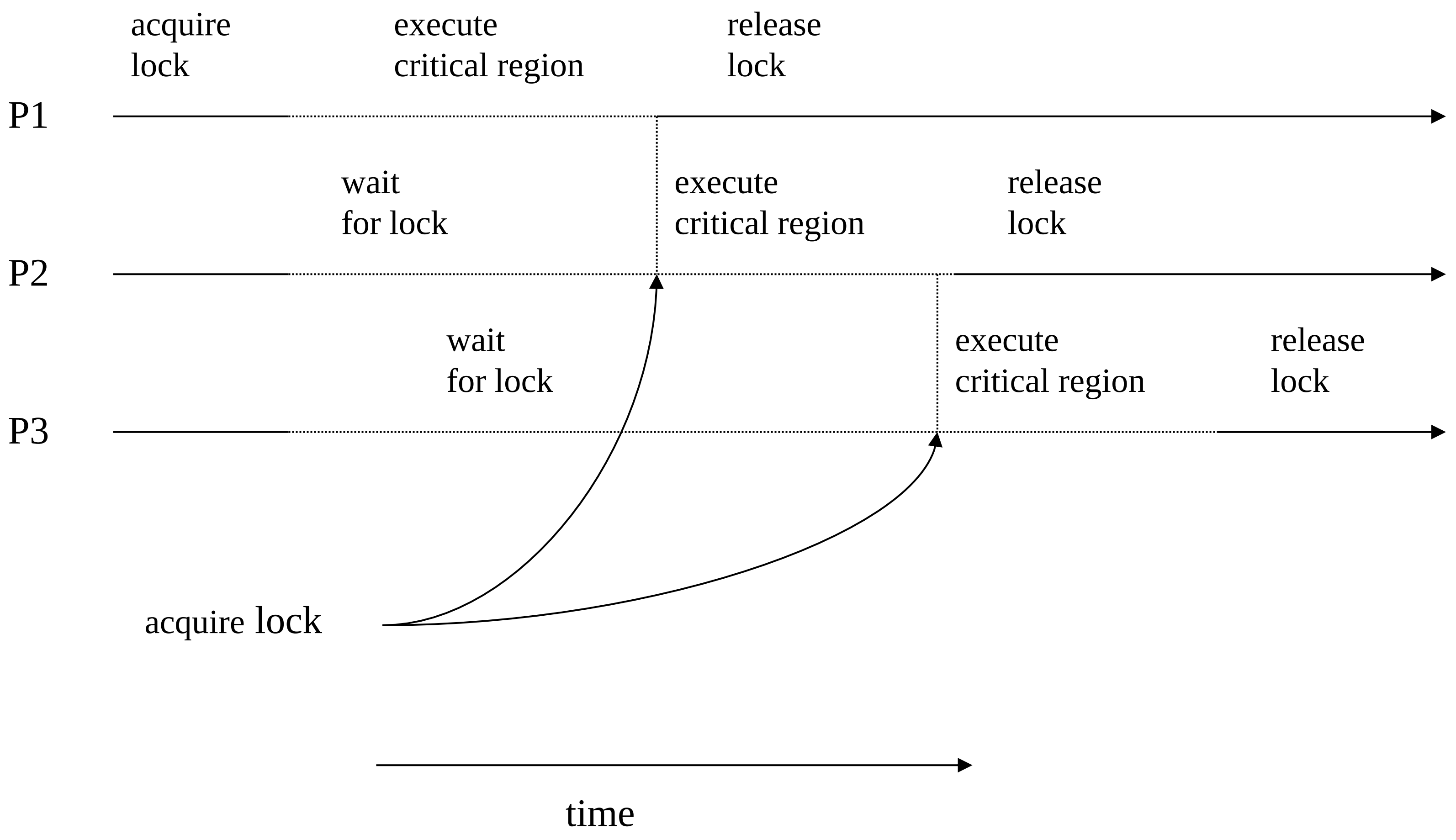
- A lock variable is to be created for every shared variable that needs exclusive access:
  - It must be shared among all processes that need to use it
  - It can either be in a “locked” or “unlocked” state

Variable Locks

- The steps to be followed for exclusive access using locks are:
  - Acquire the lock for a variable
  - Modify the variable
  - Release the lock for the variable

- The use of locks sequentializes execution of program sections and hurts performance

Illustration of Locks



Example

```
float total_sum;
void parallel_func() {
    int id;
    float  partial_sum;
    id=m_get_myid();

    if (id==0) partial_sum=1.0+2.0;
    else partial_sum=3.0+4.0;

    m_lock();
    total_sum+=partial_sum;
    m_unlock();
}
```

Race Conditions

```
float total_sum;
void parallel_func() {
        int id;
        float partial_sum;
        id=m_get_myid();

        if (id==0) partial_sum=1.0+2.0;
        else partial_sum=3.0+4.0;

        m_lock();
        total_sum+=partial_sum;
        m_unlock();

        average=total_sum/4.0;
        printf("%d average is %f\n",id, average);
}
```

Race Conditions

Possible output

0 average is 2.5 1 average is 2.5	1 average is 1.75 0 average is 2.5	0 average is 0.75 1 average is 2.5

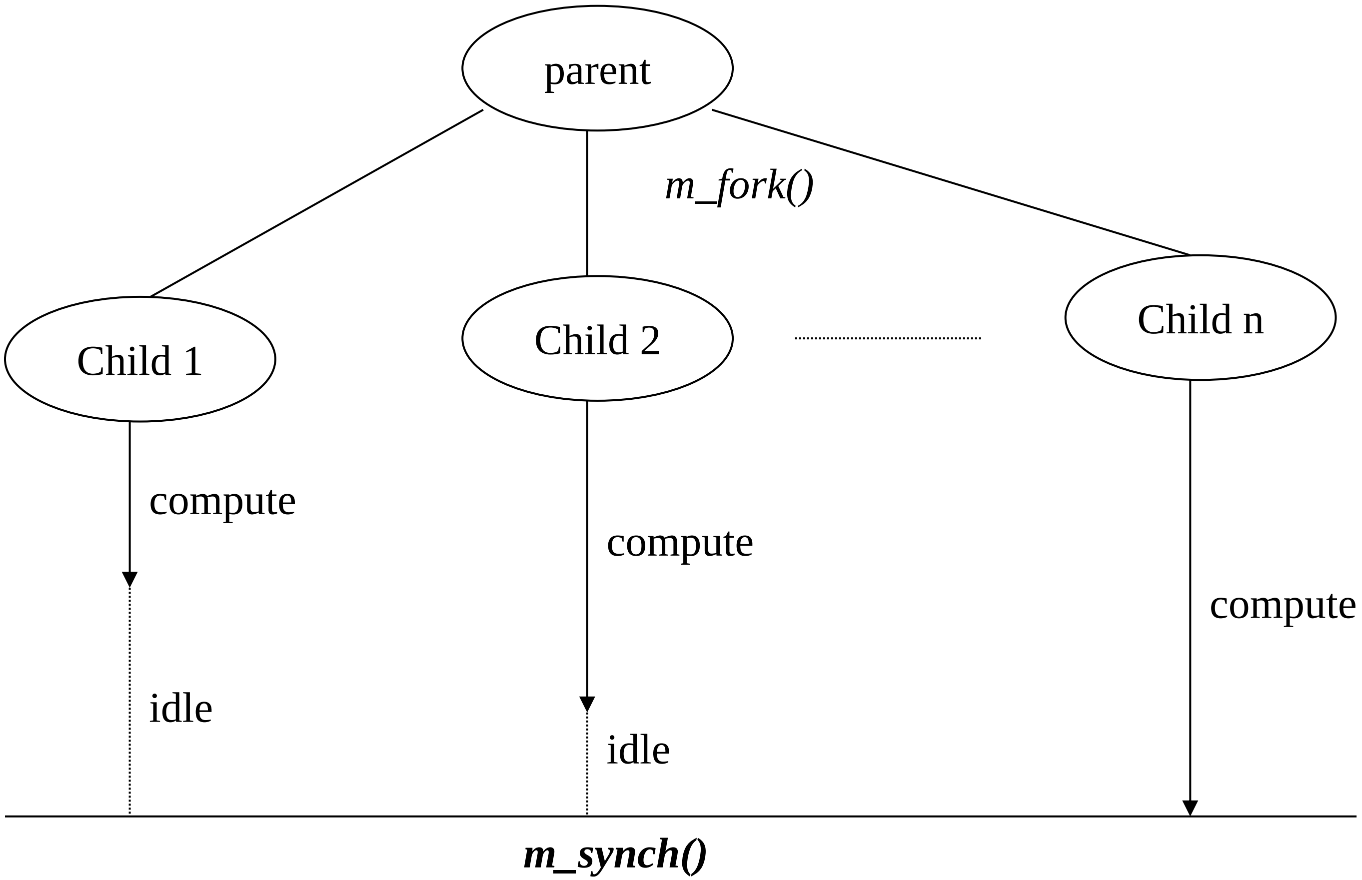
Barrier

- A race condition exists when?

The results of a parallel program depend on the relative execution speed of processes

- Barriers enable processes to synchronize with each other and help avoid race conditions
- When a process enters a barrier, it waits for all other processes involved to reach the barrier before continuing
- Barriers cause performance degradation and therefore must be used carefully

Illustration of Barrier



Barriers

```
float total_sum;
void parallel_func() {
        int id;
        float partial_sum;
           id=m_get_myid();
        if (id==0) partial_sum=1.0+2.0;
        else partial_sum=3.0+4.0;
        m_lock();
        total_sum+=partial_sum;
        m_unlock();

        m_synch( );

        average=total_sum/4.0;
        printf("%d average is %f\n",id, average);
}
```

Process Summary

- Contention
  - Locks for exclusive access to shared variables

- Race conditions
  - Barriers for synchronization

Loop Parallelization

- We will look at some simple and commonly occurring code fragments and consider how to parallelize them
- Techniques involve to remove dependencies
  - Local variables to remove dependencies
  - Code transformations
  - Use of particular scheduling techniques

Local Variables

- Code Fragment

```
float *a, *b, x;
int n;

for (i=0 ;i<n; i++) {
      x = a[i]*3;
      b[i] = x*b[i];
  }
```

Local Variables

```
float *a, *b;
int n;

void parallel_func(n) {
      int i, id, nprocs;
      float x;

      id = me_get_myid();
      nprocs = m_get_numprocs();
      for (i=id ;i<n;i+=nprocs) {
         x=a[i]*3;
         b[i]=x*b[i];
      }
}
```

Loop-Carried Variables

- Code fragment

```
indx=0;
for (i=0;i<n;i++) {
   indx+=i;
   a[i]=b[indx];
}
```

Loop-Carried Variables

```
float *a, *b;
int n;

void parallel_func(n) {
      int i, id, nprocs, indx;

      id = me_get_myid();
      nprocs = m_get_numprocs();
      for (i=id ;i<n;i+=nprocs) {
         indx = (i*(i+1))/2;
         a[i]=b[indx];
      }
}
```

Indirect Indexing

- Code fragment

```
for (i=0;i<n;i++) {
   ix=indexx[i];
   iix=ixoffset[ix];
   total[iix]=total[iix]+delta;
}
```

Indirect Indexing

```
for (i=0;i<n;i++) {
    ix=indexx[i];
    iix [i]=ixoffset[ix];
}
for (i=0;i<n;i++) {
    total[iix[i]]=total[iix[i]]+delta;
}
```

Loop Reordering

- Code fragment

```
for (k=0;k<n;k++) {
   for (i=0;i<n;i++) {
          for (j=0;j<n;j++) {
            a[i][j]+=b[i][k]+c[k][j];
         }
   }
}
```

Loop Reordering

```
float *a,*b,*c;
void parallel_func() {
    int i, id, nprocs;

    id = m_get_myid();
    nprocs = m_get_numprocs();
    for (i=id; i<n; i+=nprocs)
        for (k=0; k<n; k++)
            for(j=0; j<0; j++)
                a[i][j]+=b[i][k]+c[k][j];
}
```

Processes vs. Threads

- Process
  - a process may have multiple threads
  - heavyweight
  - allocated a private memory region
- Thread
  - share memory region with its parent
  - lightweight
  - share CPU time slots

Parallel Program Model Summary

- Data parallel programming dominates
- Loops in programs are the source of data parallelism
- Exploitation of parallelism involves sharing work in loops among processes
- Have to use appropriate scheduling techniques for optimal work sharing
- Parallelism in loops not always straightforward to find due to dependence
- Have to perform some transformations to expose parallelism

OpenMP

- A standard for directive based parallel programming
- Higher level than Pthreads, no threads manipulating
- API used in FORTRAN, C, C++
- Support for concurrency, synchronization, etc.

OpenMP Programming Model

- #pragma omp directive [clause list]
- #pragma omp parallel [clause list]

  /*structured block*/
- Example

```
int a, b;
main() {
    //serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
            //parallel segment
    }
    //serial segment
}
```

Message Passing Interface (MPI)

- MPI – standard for explicit message passing (C and Fortran)
- Portable
- Widely used, requires minimal underlying hardware knowledge
- Developed by a group of researchers
- Support almost all hardware vendors

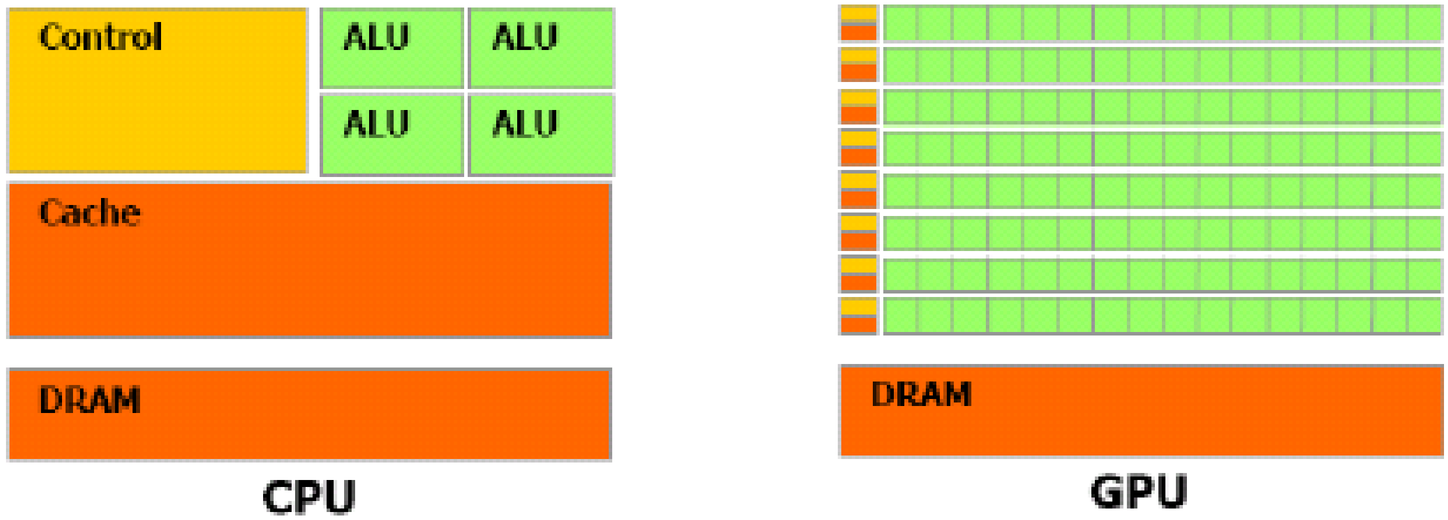
MPI Primitives

- MPI has over 125 functions, but need to know only 6 to get started on writing real applications
  - MPI_Init: initiate an MPI computation
  - MPI_Finalize: terminate an MPI computation
  - MPI_Comm_Size: determine number of processes
  - MPI_Comm_Rank: determine my process identifier
  - MPI_Send: send a message
  - MPI_Recv: receive a message

Parallel Computing

- Parallel computing platforms
- Parallel programming models
- Architecture of GPU
- Multi-threaded CPU computing vs. multi-threaded CUDA computing

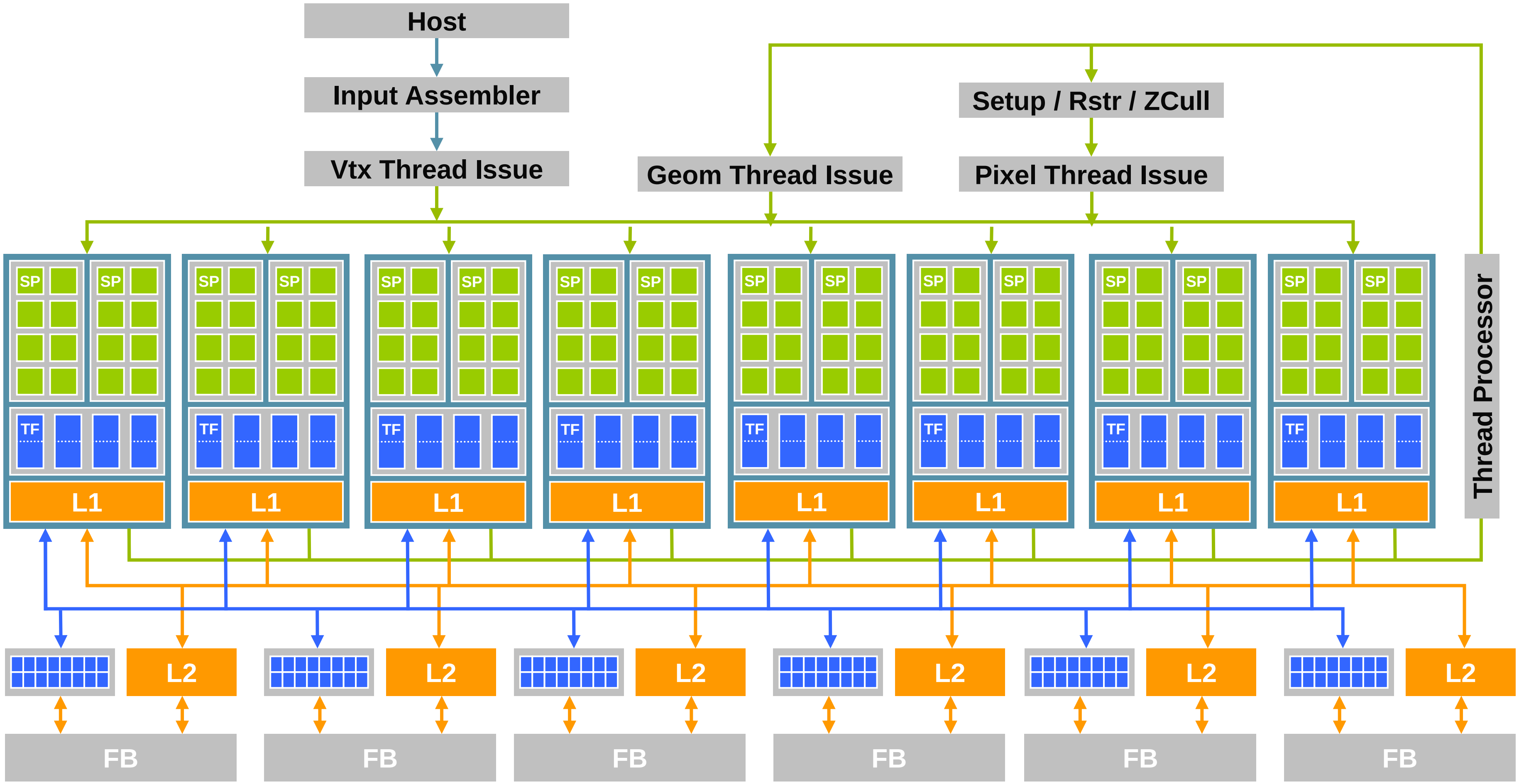
GPU Design Philosophy



- Difference between GPU and CPU
  - More transistors for data processing
  - Many-core (hundreds of cores)

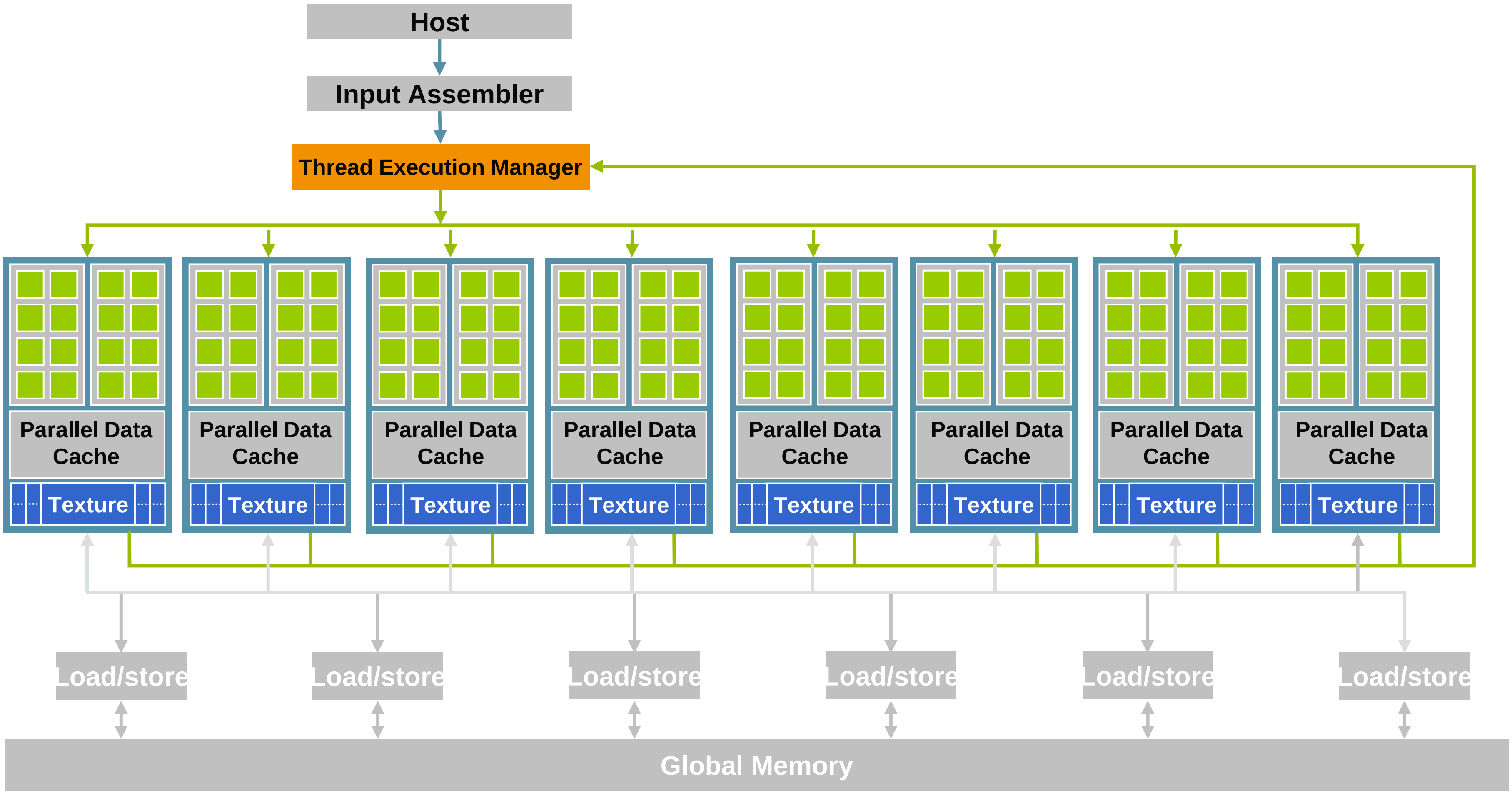
Nvidia's GeForce 8800

G80 – Graphics Mode



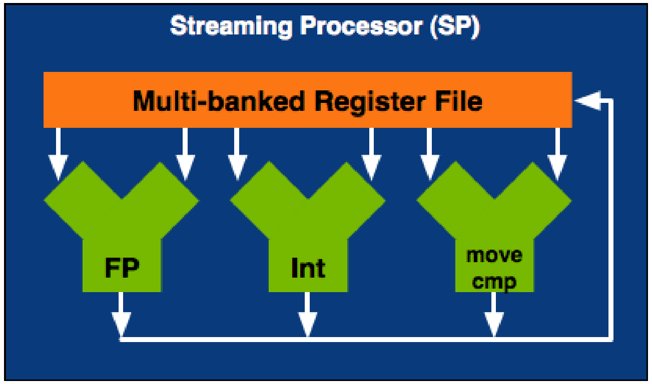
Block Diagram of the GeForce 8800

G80 CUDA Mode



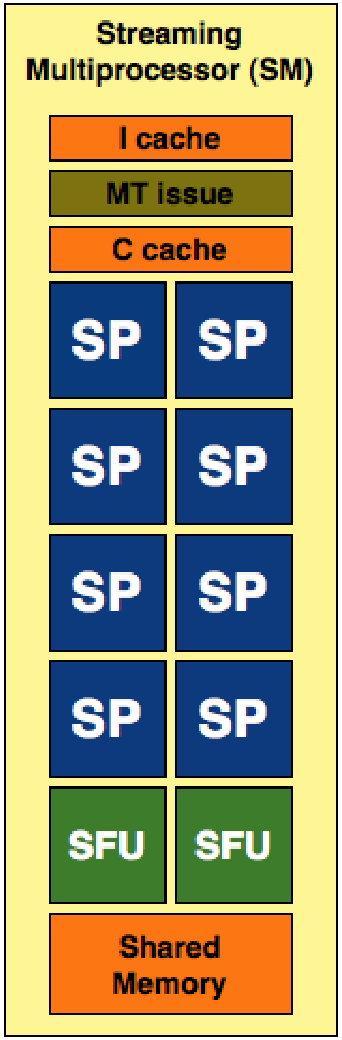
Streaming Processor (SP)

- A fully pipelined, single-issue, in-order microprocessor
  - 2 ALUs and a FPU
  - Register file
  - 32-bit scalar processing
  - No cache



Streaming Multiprocessor (SM)

- An array of SPs
  - 8 streaming processors
  - 2 Special Function Units (SFU)
    - Transcendental operations (e.g. sin, cos) and interpolation
  - A 16KB read/write shared memory
    - Not a cache, but a software-managed data store
  - Multithreading issuing unit
    - Dispatch instructions
  - Instruction cache
  - Constant cache



Parallel Computing

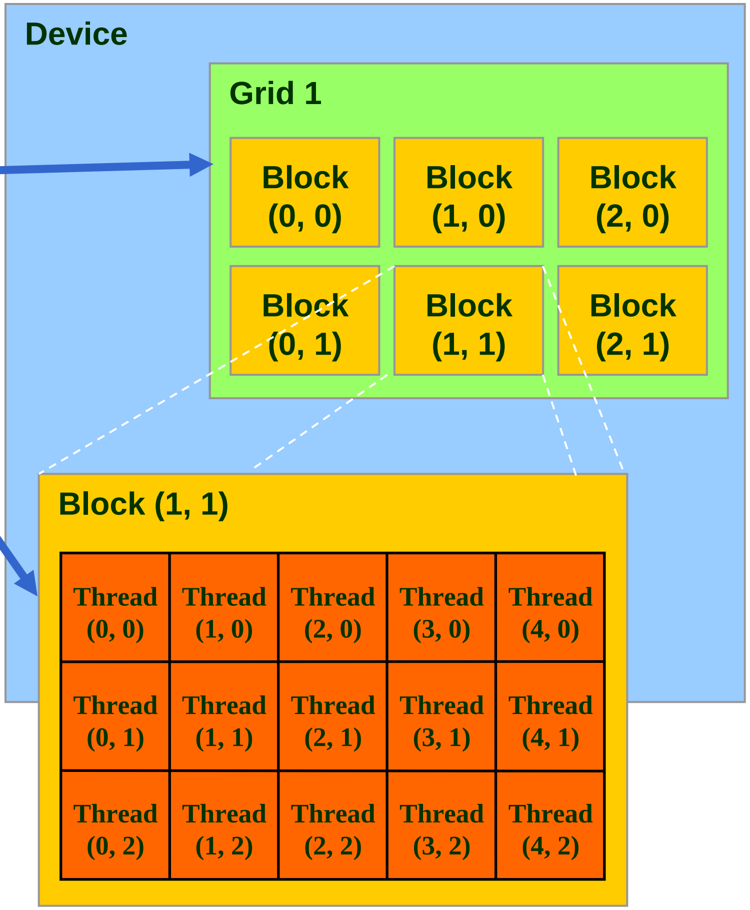
- Parallel computing platforms
- Parallel programming models
- Architecture of GPU
- Multi-threaded CPU computing vs. multi-threaded CUDA computing

CUDA Device

- A compute device
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Kernel — Data-parallel portions of an application which run on many threads

Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplify memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



CUDA

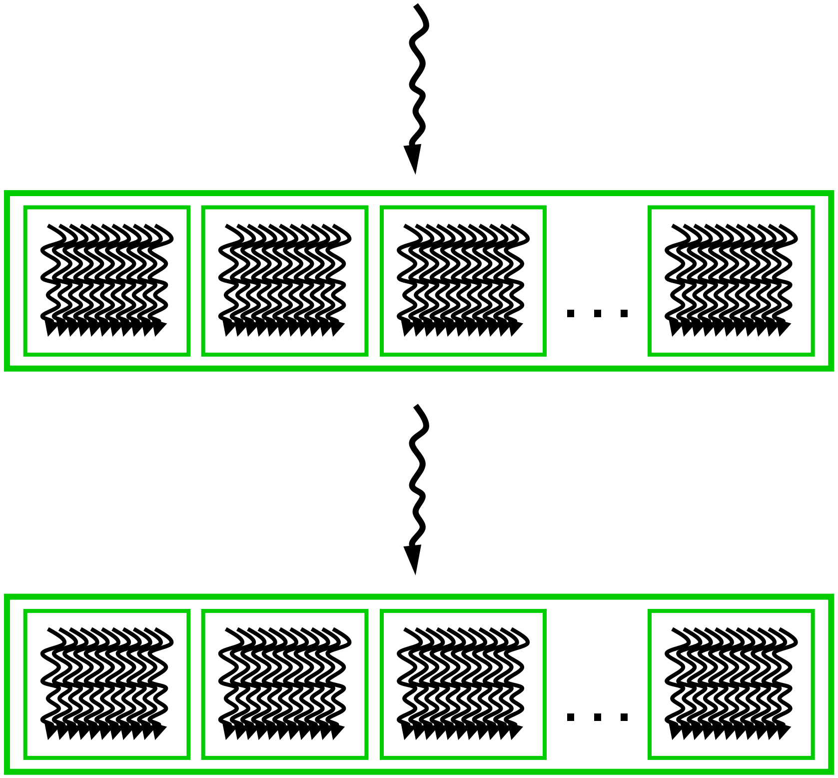
- Integrated host+device C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

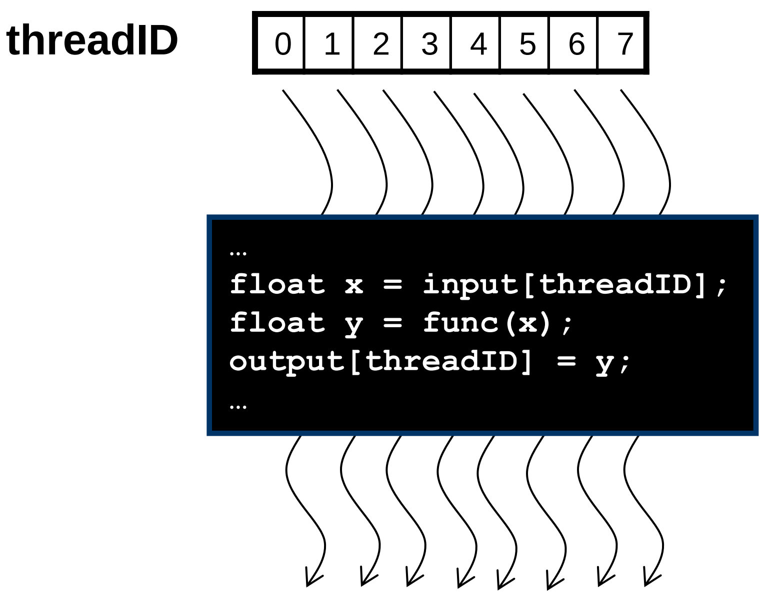
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



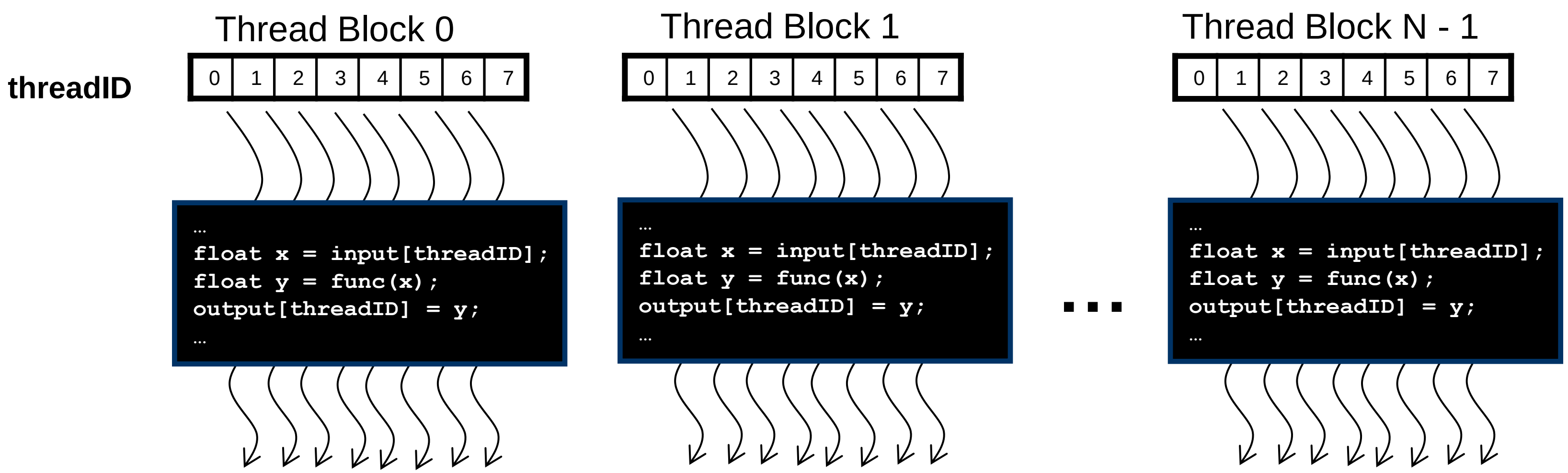
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Thread Blocks

- Divide thread array into multiple blocks
  - Threads within a block cooperate via *shared memory, atomic operations* and *barrier synchronization*
  - Threads in different blocks cannot cooperate



Differences Between GPU Threads and CPU Processes

- GPU threads execute on streaming processors (SPs)
  - CPU threads (processes) execute on processors
- GPU threads are extremely lightweight
  - Very little creation overhead
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few
- GPU good at applications of little communication between threads

GPU + CPU Cluster

- Host
  - C/C++
  - MPI primitives
  - One CPU core is explicitly coupled with a GPU
  - GPU kernel is launched by a CPU process

CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance