Introduction to Parallel & Distributed Computing

Programming Models

Lecture 2, Spring 2014

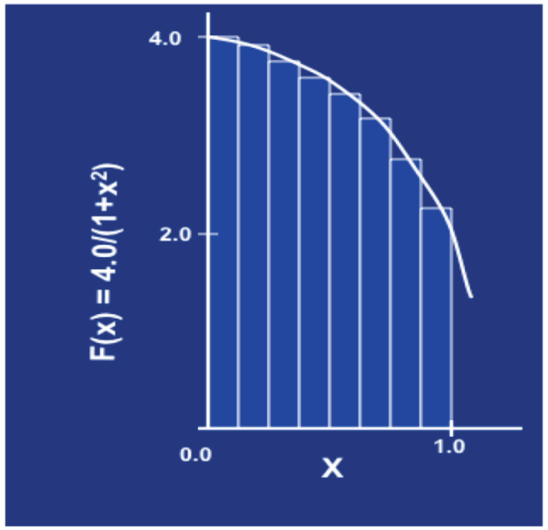
Instructor: 罗国杰

gluo@pku.edu.cn

Example: Compute π

- **Computing π**

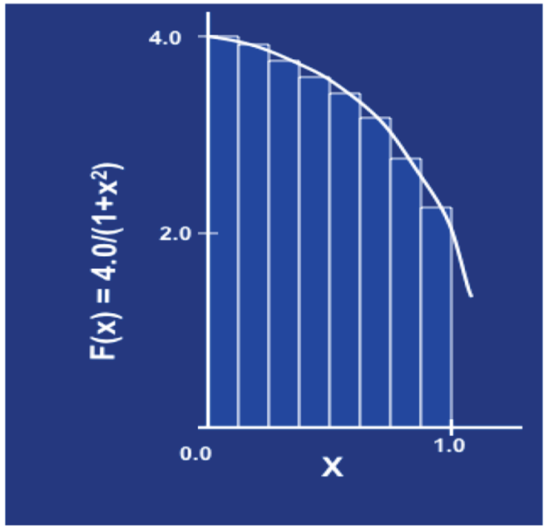
$$\left.\begin{aligned} \arctan(1) &= \pi/4 \\ \arctan(0) &= 0 \\ \frac{d}{dx}\arctan(x) &= 1/(1+x^2) \end{aligned}\right\} \Rightarrow \quad \pi = \int_0^1 \frac{4}{1+x^2}\,dx$$



Barbara Chapman, “A Guide to OpenMP,” 2010.

Example: Sequential Code

```
double compute_pi(int n) {
  double sum = 0.0;
  for (int i = 0; i < n; i++) {
    double x = (i + 0.5) / n;
    sum += 1.0 / (1.0 + x*x);
  }
  double pi = 4.0 * sum / n;
  return pi;
}
```



POSIX Thread Version (1/2)

```
// global variables
int thread_count;
int n;
double* local_sum;

// multithreaded version
double compute_pi () {
  thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
  local_sum = (double*) malloc (thread_count*sizeof(double));
  // parallel part
  for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL, thread_sum, (void*)thread);
  for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
  // sequential part
  double sum = 0.0;
  for (thread = 0; thread < thread_count; thread++)
   sum += local_sum[thread];
  double pi = 4.0 * sum / n;
  return pi;
}
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

POSIX Thread Version (2/2)

```
void* thread_sum(void* rank) {
   int my_rank = (int) rank;
   double my_sum = 0.0;
   // domain decomposition
   for (int i = my_rank; i < n; i += thread_count) {
     double x = (i + 0.5) / n;
     my_sum += 1.0 / (1.0 + x * x);
   }
   local_sum[my_rank] = my_sum;
   return NULL;
}
```

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

OpenMP Version

```
/*  compile as $> gcc -fopenmp -lm
 *  run as         $> OMP_NUM_THREADS=2 ./a.out
 */
double compute_pi(int n) {
   int i;
   double sum = 0.0;
#pragma omp parallel for reduction(+: sum) schedule(static)
   for (i = 0; i < n; ++i) {
      double x = (i + 0.5) / n;
      sum += 1.0 / (1.0 + x * x);
   }
   double pi = 4.0 * sum / n;
   return pi;
}
```

MPI Version (1/2)

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )  {
  int n, my_rank, numprocs, i;
  double mypi, pi, h, sum, x;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  while (1) {
    if (my_rank == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

MPI Version (2/2)

```
  if (n == 0)  break;
  else {
    sum = 0.0;
    for (i = my_rank; i <; i += numprocs) {
      x = (i + 0.5) / n;
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = sum / n;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                           MPI_COMM_WORLD);
    if (my_rank == 0) printf("pi is approximately %.16f\n", pi));
  }
 }
 MPI_Finalize();
 return 0;
}
```

OpenCL Version

```
/* to be executed on deivce */
__kernel void pi(const int  niters,  const float  step_size,
                  __local  float*  local_sums,  __global float*  partial_sums) {
   int num_wrk_items  = get_local_size(0);
   int local_id       = get_local_id(0);
   int group_id       = get_group_id(0);

   float x, accum = 0.0f;
   int i,istart,iend;

   istart = (group_id * num_wrk_items + local_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++) {
       x = (i+0.5f)*step_size;
       accum += 4.0f/(1.0f+x*x);
   }

   local_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);

   reduce(local_sums, partial_sums);
}

/* to be executed on host */
pi_res = 0.0f;
for (unsigned int i = 0; i < nwork_groups; i++) { pi_res += h_psum[i]; }
pi_res *= step_size;
```

Outline

- Parallel programming models and architectures
- Memory limitations

Architectural Trends

- **Greatest trend in VLSI is an increase in the exploited parallelism**
  - Up to 1985: bit-level parallelism: 4-bit -> 8-bit -> 16-bit
    - slows after 32 bits
    - adoption of 64-bit now under way
  - Mid 80s to mid 90s: instruction-level parallelism
    - pipelining and simple instruction sets (RISC)
    - on-chip caches and functional units => superscalar execution
    - greater sophistication: out of order execution
  - Nowadays:
    - hyper-threading
    - multi-core

Evolution of Architectural Models

- Historically (1970s – early 1990s), each parallel machine was unique, along with its programming model and language
  - Architecture = prog. model + comm. abstraction + machine organization
- Throw away software & start over with each new kind of machine
  - Dead Supercomputer Society
    - http://www.paralogos.com/DeadSuper/
- Nowadays we separate the programming model from the underlying parallel machine architecture
  - 3 or 4 dominant programming models
    - Dominant: shared address space, message passing, data parallel
    - Others: data flow, systolic arrays

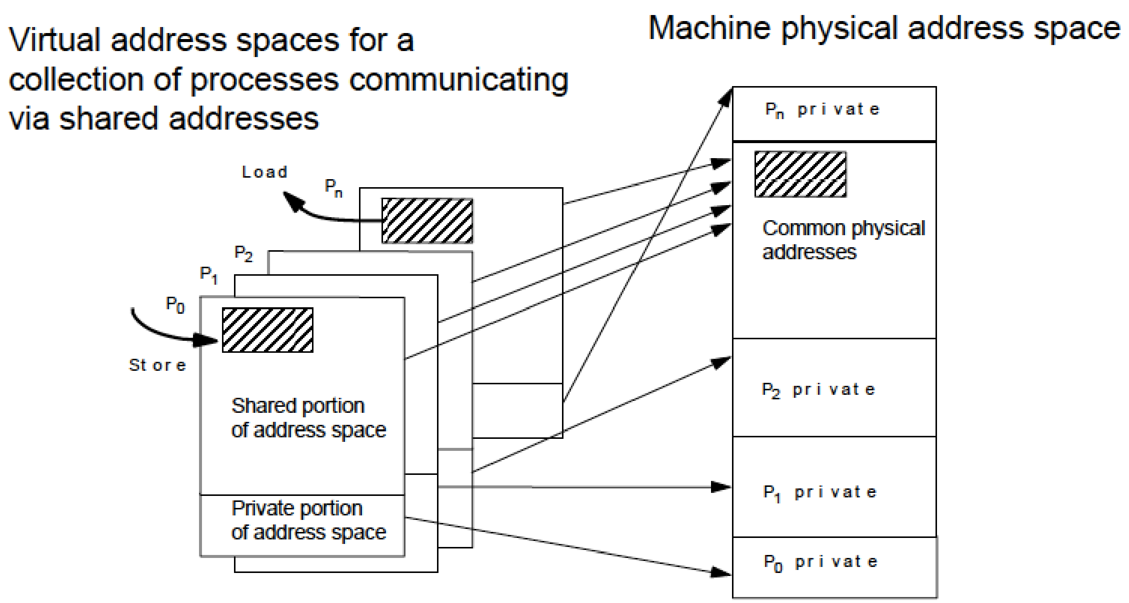
Programming Model for Various Architectures

- **Programming models specify communication and synchronization**
  - **Multiprogramming: no communication/synchronization**
  - **Shared address space: like bulletin board**
  - **Message passing: like phone calls**
  - **Data parallel: more regimented, global actions on data**
- **Communication abstraction: primitives for implementing the model**
  - **Play the role like the instruction set in a uniprocessor computer**
  - **Supported by HW, by OS, or by user-level software**
- **Programming models are the abstraction presented to programmers**
  - **Write portably correct code that runs on many machines**
  - **Write fast code requires tuning for the architecture**
    - **Not always worthy of it – sometimes programmer time is more precious**

Aspects of a Parallel Programming Model

- **Control**
  - How is parallelism created?
  - In what order should operations take place?
  - How are different threads of control synchronized?
- **Naming**
  - What data is private vs. shared?
  - How is shared data accessed?
- **Operations**
  - What operations are atomic?
- **Cost**
  - How do we account for the cost of operations?

Programming Models: Shared Address Space (SAS)



Virtual address spaces for a collection of processes communicating via shared addresses
Machine physical address space
Load
P_n
P_2
P_1
P_0
Store
Shared portion of address space
Private portion of address space
P_n private
Common physical addresses
P_2 private
P_1 private
P_0 private

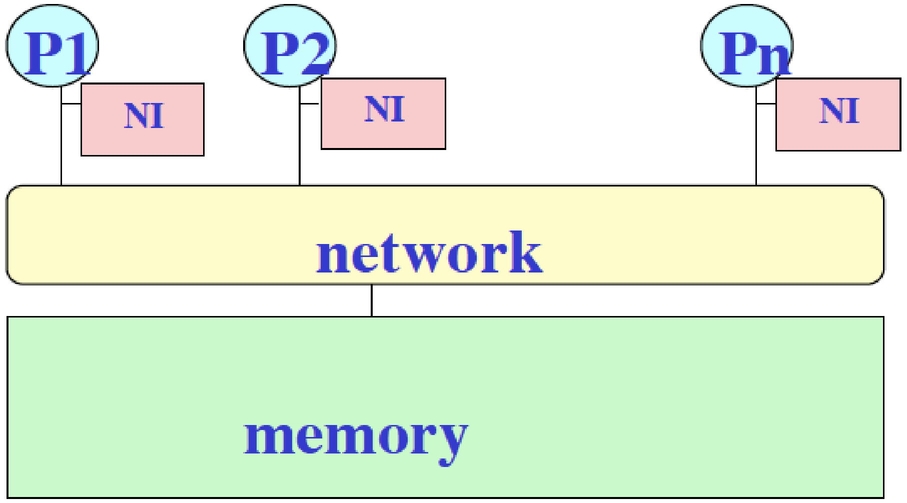
- **Programming model**
  - **Process: virtual address space plus one or more threads of control**
  - **Portions of address spaces of processes are shared**
  - **Writes to shared address visible to all threads (in other processes as well)**
- **Natural extension of uniprocess model**
  - **conventional memory operations for communication**
  - **special atomic operations for synchronization**

SAS Machine Architecture (1/2)

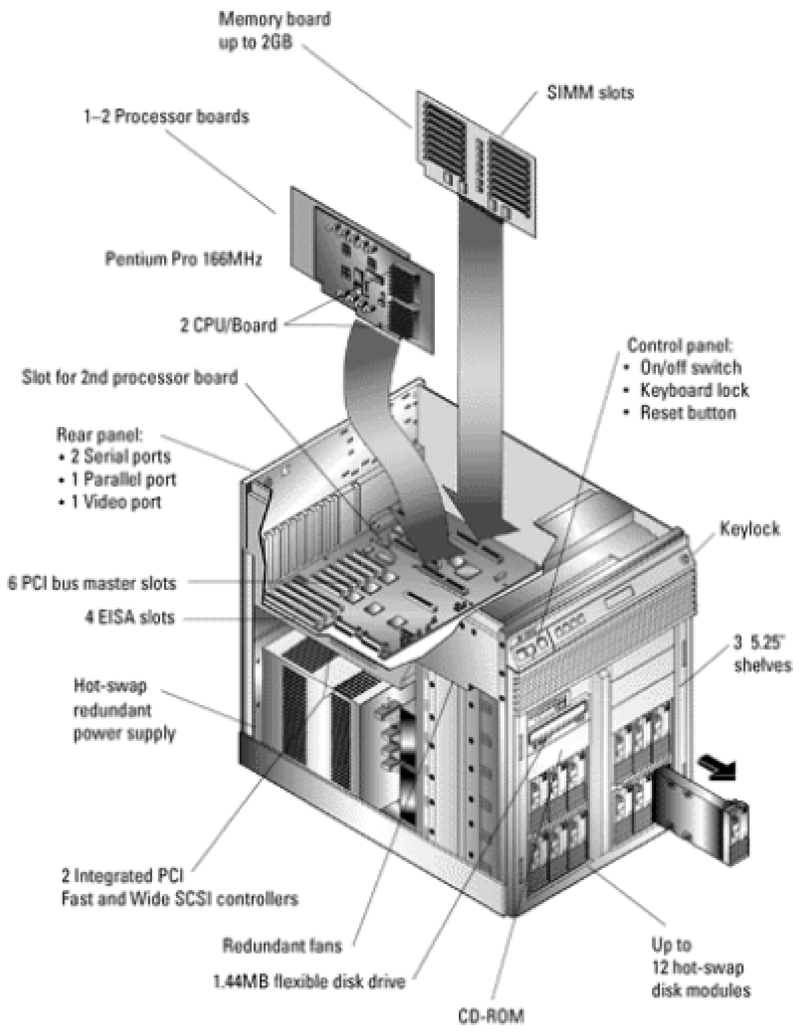
- **Motivation: programming convenience**
  - **Location transparency**
    - **Any processor can directly reference any memory location**
    - **Communication occurs implicitly as result of loads and stores**
  - **Extended from time-sharing on uni-processors**
    - **Processes can run on different processors**
    - **Improved throughput on multi-programmed workloads**
- **Communication hardware also natural extension of uniprocessor**
  - **Addition of processors similar to memory modules, I/O controllers**

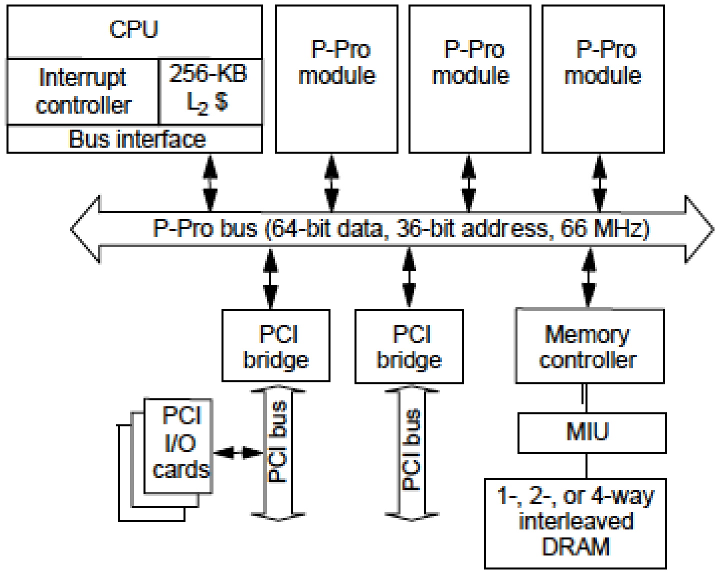
SAS Machine Architecture (2/2)

- **One representative architecture: SMP**
  - Used to mean *Symmetric MultiProcessor*
    - All CPUs had equal capabilities in every area, e.g., in terms of I/O as well as memory access
  - Evolved to mean *Shared Memory Processor*
    - Non-message-passing machines (included crossbar as well as bus based systems)
  - Now it tends to refer *to bus-based shared memory machines*
    - Small scale: < 32 processors typically



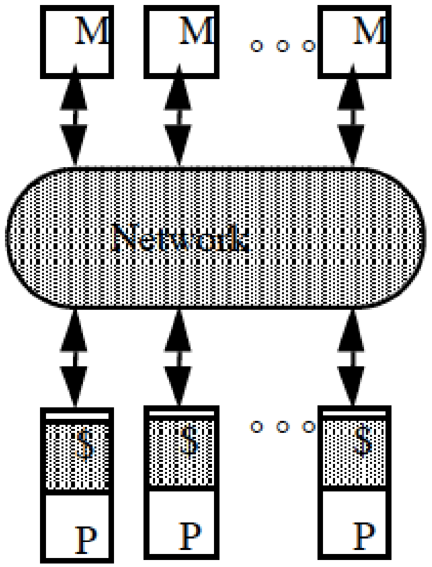
Example: Intel Pentium Pro Quad





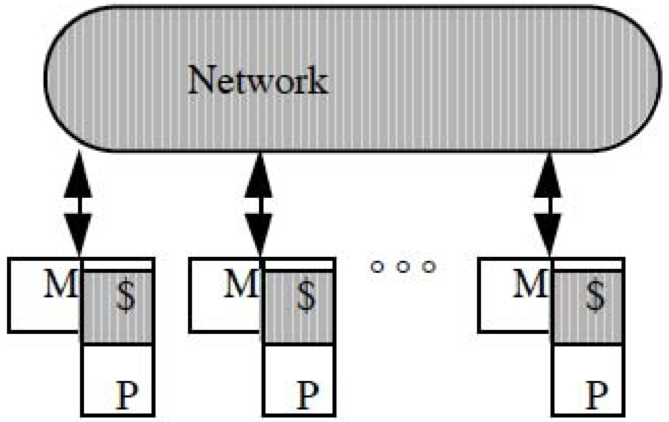
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and high bandwidth

Scaling Up: More SAS Machine Architectures (1/2)



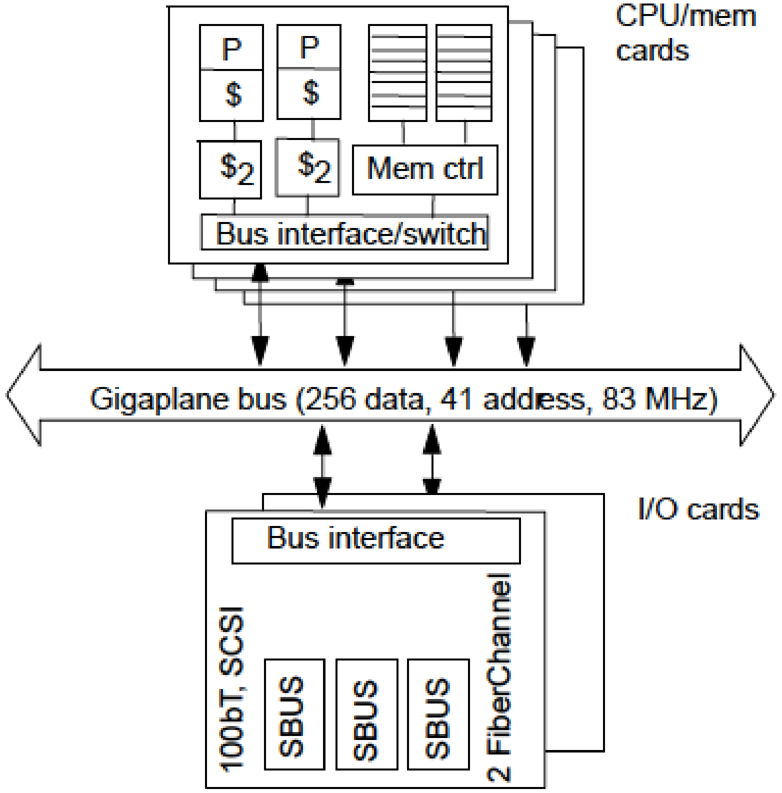
- **Dance-hall**
  - **Problem: interconnect cost (crossbar) or bandwidth (bus)**
  - **Solution: scalable interconnect network -> bandwidth scalable**
  - **Latencies to memory uniform, but uniformly large (Uniform Memory Access, UMA)**
  - **Caching is key: coherence problem**

Scaling Up: More SAS Machine Architectures (2/2)



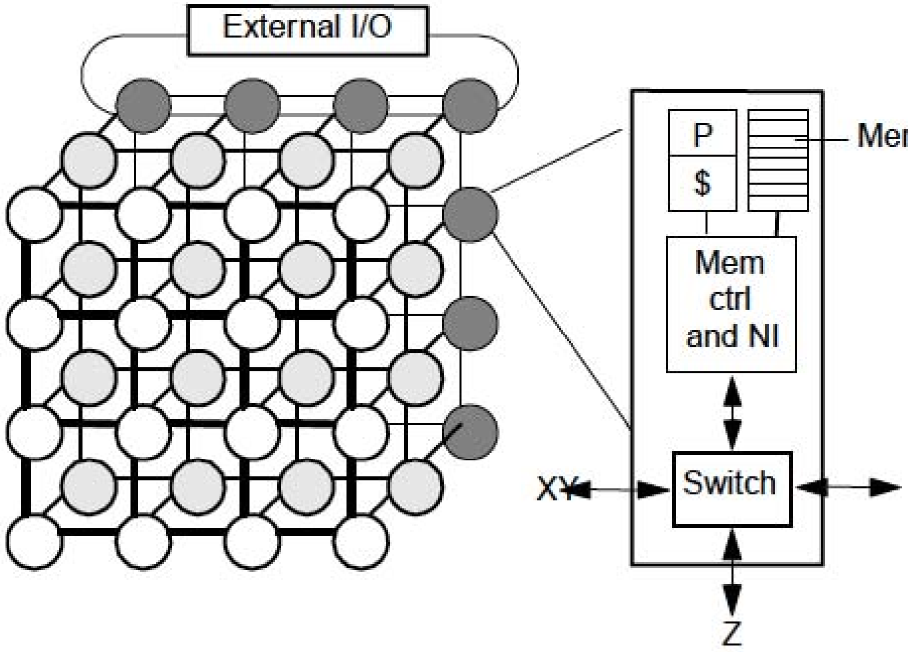
- **Distributed shared memory (DSM) or non-uniform memory access (NUMA)**
  - **Non-uniform time for the access to data in local memory and remote memory**
  - **Caching of non-local data is key**
    - **coherence cost**

Example: SUN Enterprise



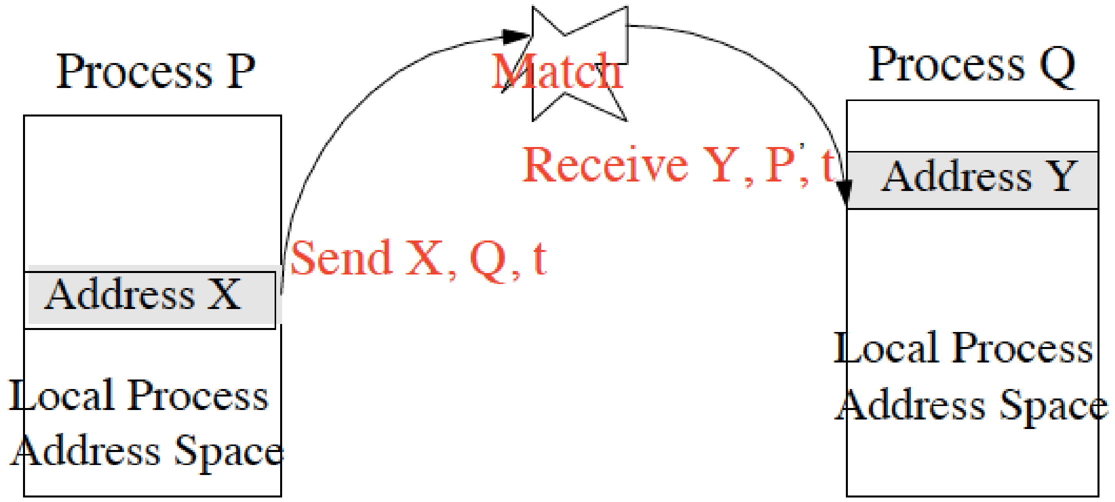
- 16 cards of either type: processors + memory, or I/O
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus

Example: Cray T3E



- Scale up to 1024 processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)

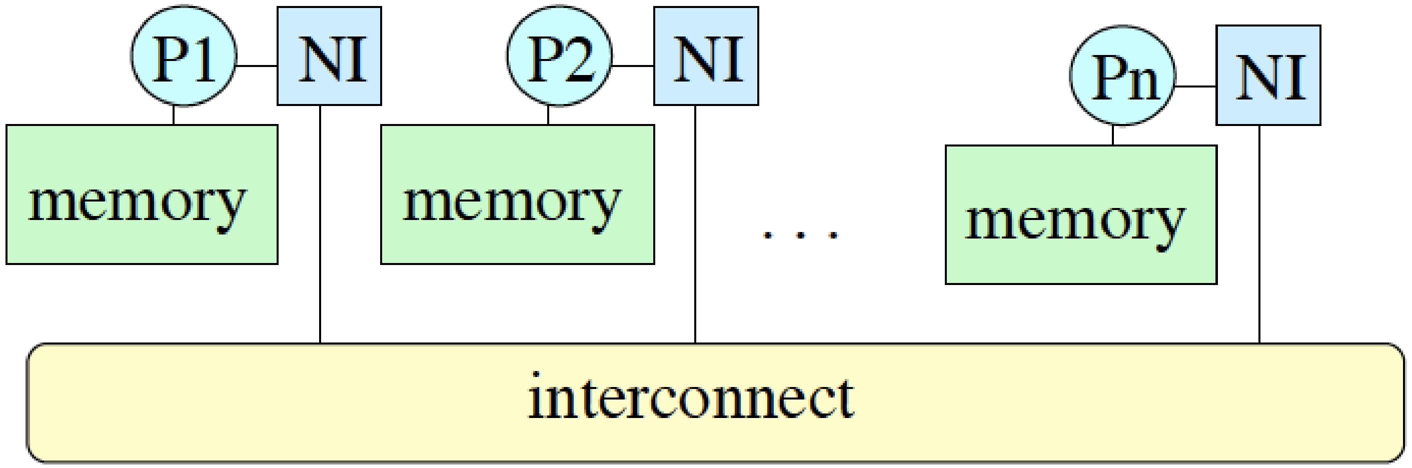
Programming Models: Message Passing



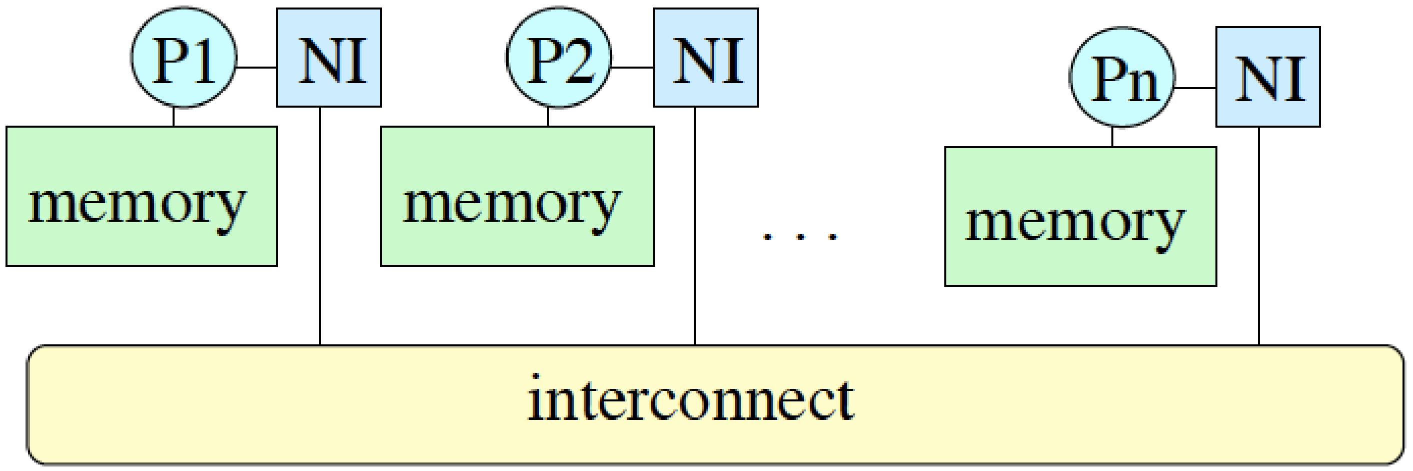
- **Programming model**
  - **Directly access only private address space (local memory), communicate via explicit messages**
    - **Send specifies data in a buffer to transmit to the receiving process**
    - **Recv specifies sending process and buffer to receive data**
  - **In the simplest form, the send/recv match achieves pair-wise synchronization**
- **Model is separated from basic hardware operations**
  - **Library or OS support for copying, buffer management, protection**
  - **Potential high overhead: large messages to amortize the cost**

Message Passing Architectures

- Complete processing node (computer) as building block, including I/O
  - Communication via explicit I/O operations
  - Processor/Memory/IO form a processing node that cannot directly access another processor's memory
- Each "node" has a network interface (NI) for communication and synchronization

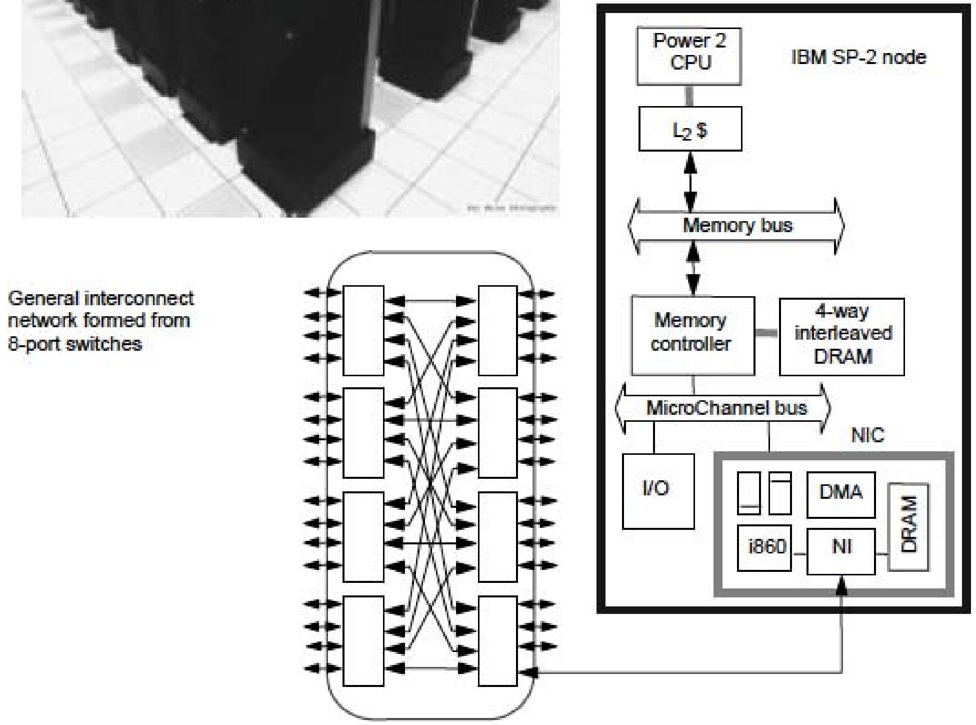


DSM vs. Message Passing



- **High-level block diagrams are similar**
- **Virtualization is possible, with some performance penalty, i.e.**
  - **MP programs can run on DSM architectures**
  - **DSM programs can run on MP architectures, with proper library or OS support for memory sharing**
- **Implications of DSM and MP on architectures**
  - **Fine-grained hardware supports for DSM**
    - **Communication integrated at I/O level for MP, needn't be into memory system**
    - **MP can be implemented as middleware (library)**
    - **MP has better scalability**
      - MP machines are easier to build than scalable address space machines

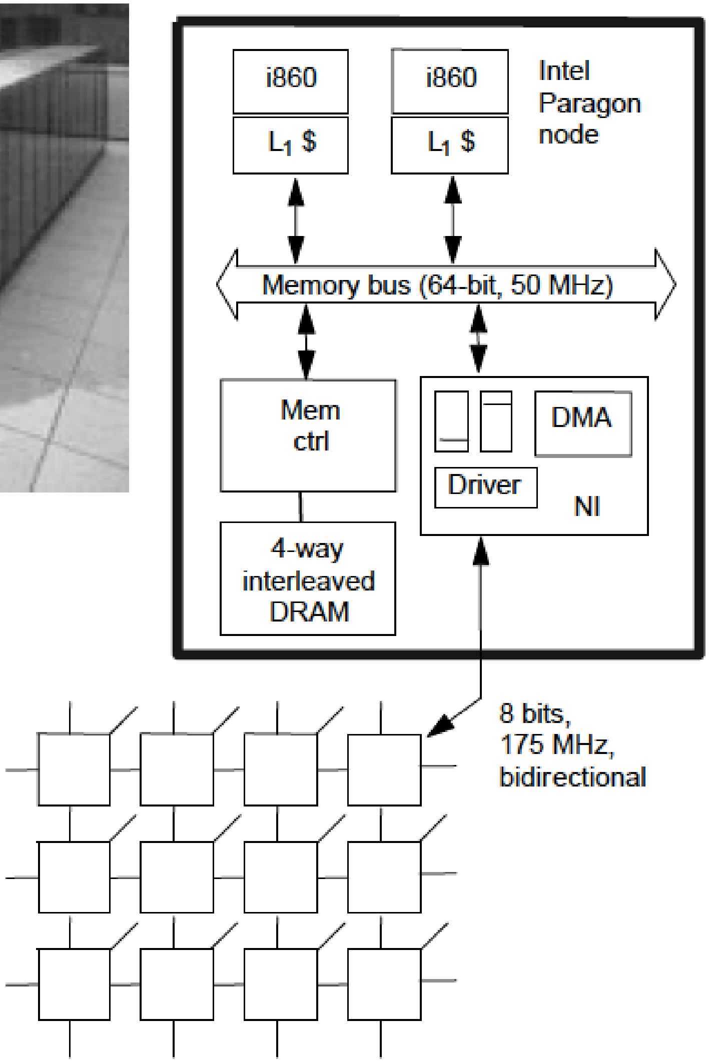
Example of MP: IBM SP-2



- Each node is a essentially complete RS6000 workstation;
- Network interface integrated in I/O bus (bw limited by I/O bus).

Example: Intel Paragon

Sandia's Intel Paragon XP/S-based Supercomputer



2D grid network with each processing node attached to a switch

Toward Architectural Convergence

- **Convergence in hardware organizations**
  - **Tighter NI integration for MP**
  - **Hardware SAS passes messages at lower level**
  - **Cluster of workstations/SMP become the most popular parallel architecture for parallel systems**
- **Programming models distinct, but organizations converging**
  - **Nodes connected by general network and communication assists**
  - **Implementations also converging, at least in high-end machines**

Programming Model: Data Parallel (1/2)

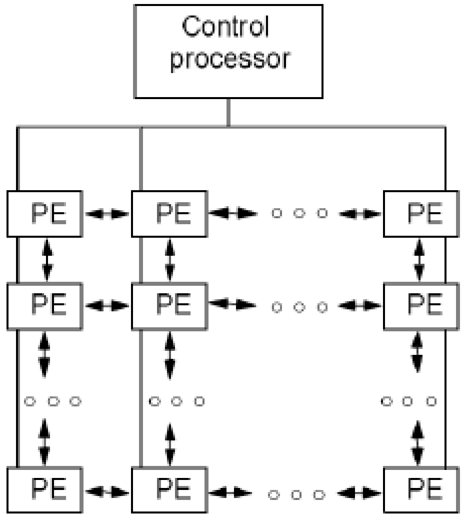
- Operations performed in parallel on each element of data structure
- Logically single thread of control (sequential program)
- Conceptually, a processor associated with each data element
- Coordination is implicit – statements executed synchronously
- Example:

```
                    float x[100];

for (i=0; i<100; i++)      ➔      x = x + 1;
  x[i] = x[i] + 1;
```

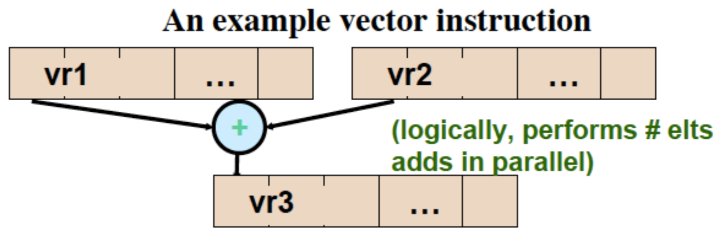
Programming Model: Data Parallel (2/2)

- **Architecture model**
  - A control processor issues instructions
  - Array of many simple cheap processors – processing element (PE) – each with little memory
  - A interconnect network that broadcasts data to PEs, communication among Pes, and cheap synchronizati



- **Motivation**
  - Give up flexibility (different instructions in different processors) to allow a much larger number of processors
  - Target at limited scope of applications
- **Applications**
  - Finite differences, linear algebra
  - Document searching, graphics, image processing

A Case of DP: Vector Machine


An example vector instruction
vr1
…
vr2
…
+
(logically, performs # elts adds in parallel)
vr3
…

- Vector machine
  - Multiple functional units
  - All performing the same operation
  - Instructions may be of very high parallelism (e.g. 64-way) but hardware executes only a subset in parallel at a time
- Historically important, but overtaken by MPPs (massive parallel processors) in the 90s
- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors
    - SSE (Streaming SIMD Extensions), SSE2 (Intel: Pentium/IA64)
    - Altivec (IBM/Motorola/Apple: PowerPC)
    - VIS (Sun: Sparc)
- Enabling technique
  - Compiler does some of the difficult work of finding parallelism

Flynn's Taxonomy

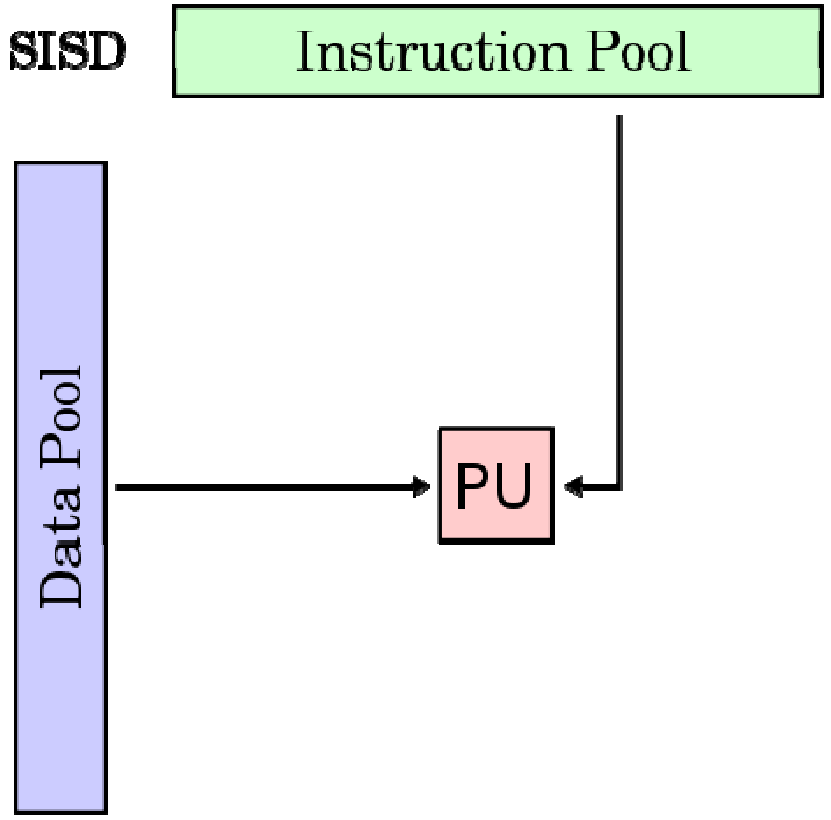
- **A classification of computer architectures based on the number of streams of instructions and data:**

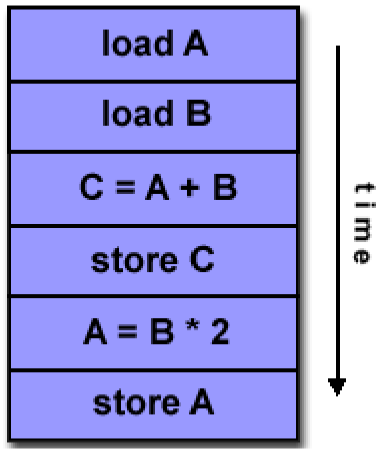
		Instruction stream	
		single	multiple
Data stream	single	SISD Single Instruction, Single Data	MISD Multiple Instructions, Single Data
	multiple	SIMD Single Instruction, Multiple Data	MIMD Multiple Instructions, Multiple Data

- **Program model converges with SPMD (single program multiple data)**

SISD Architecture

Example: single-core computers

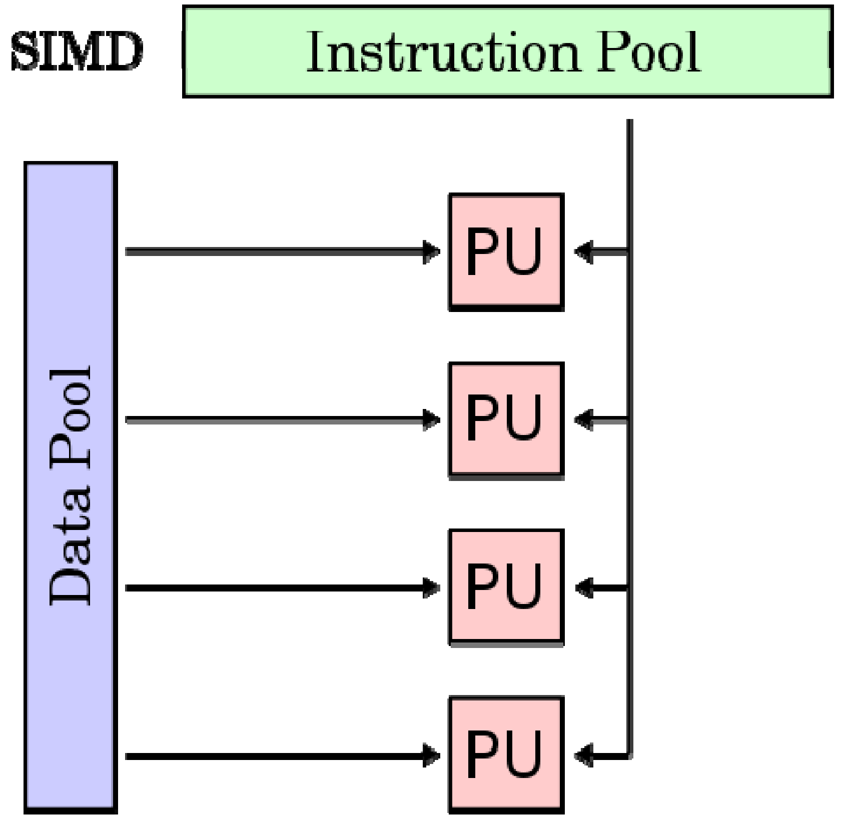


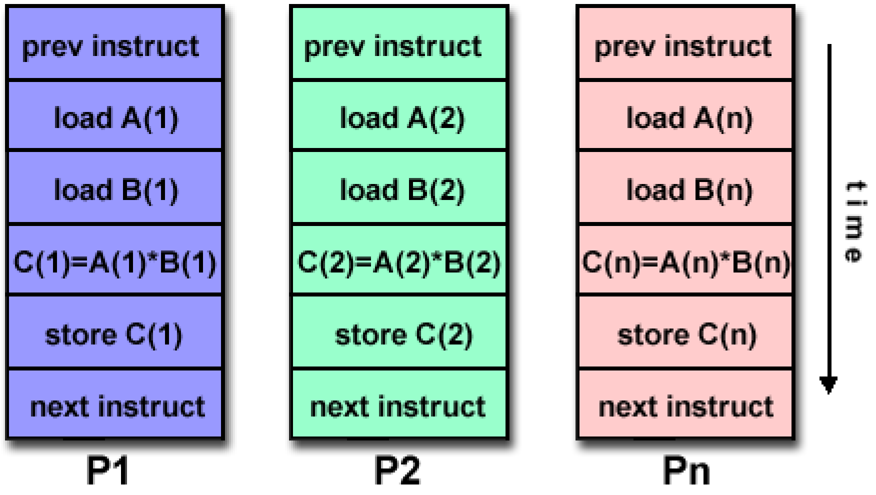


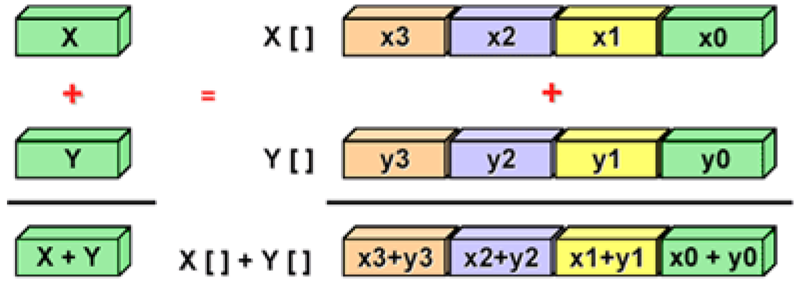
Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

SIMD Architecture

Example: vector processors, GPUs



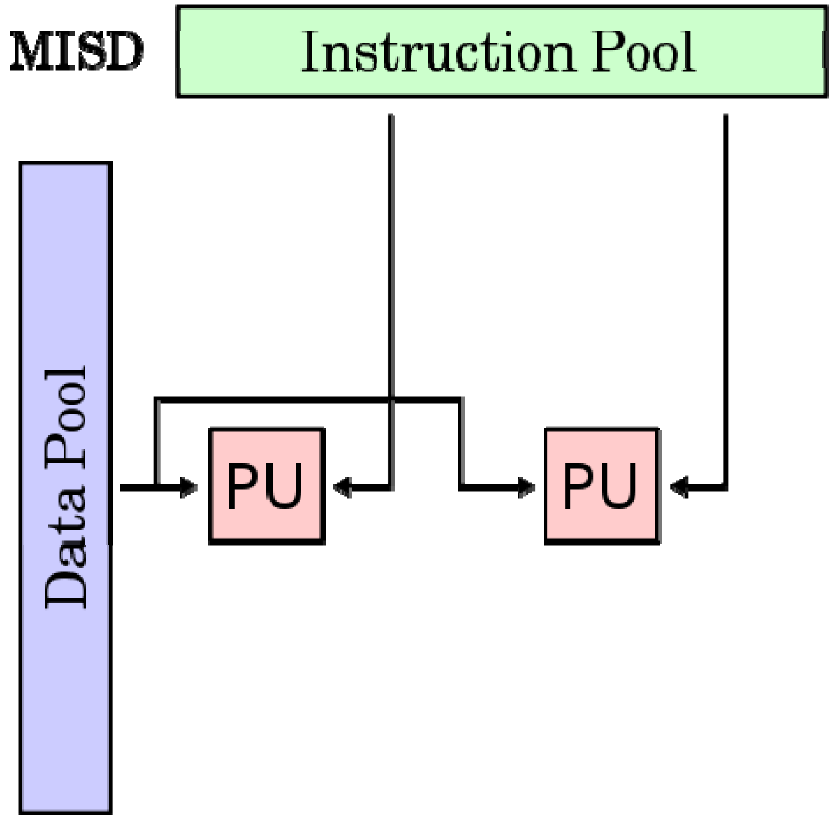


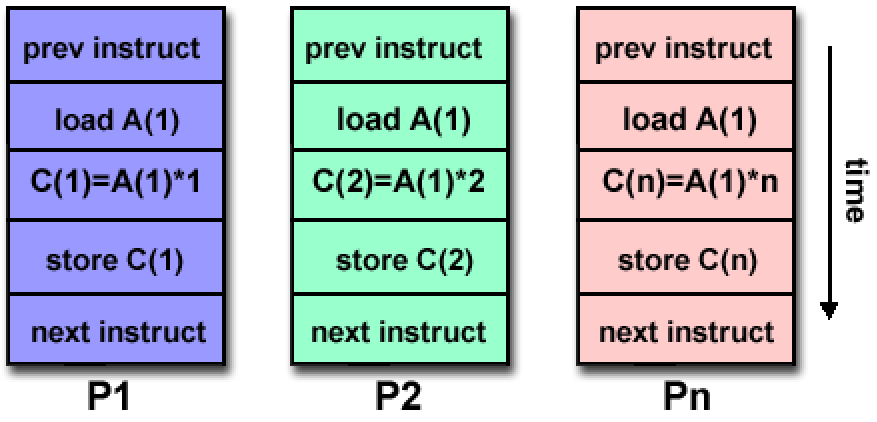


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

MISD Architecture

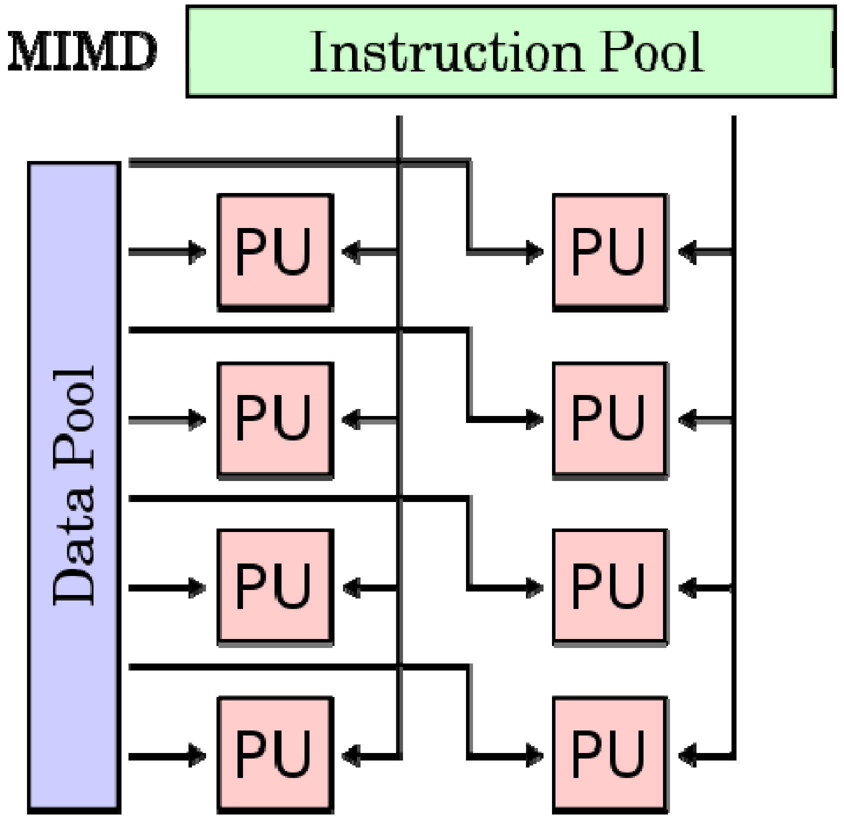
* no well-known systems fit



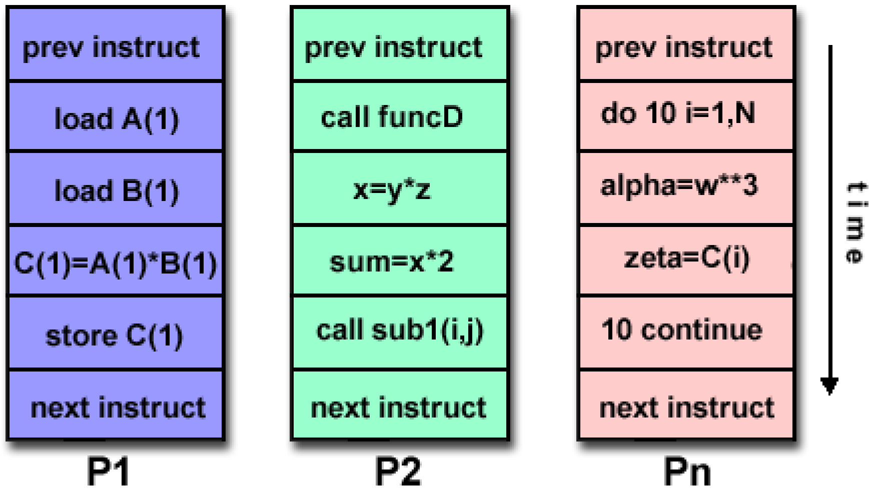


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

MIMD Architecture



Example: modern parallel systems



Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

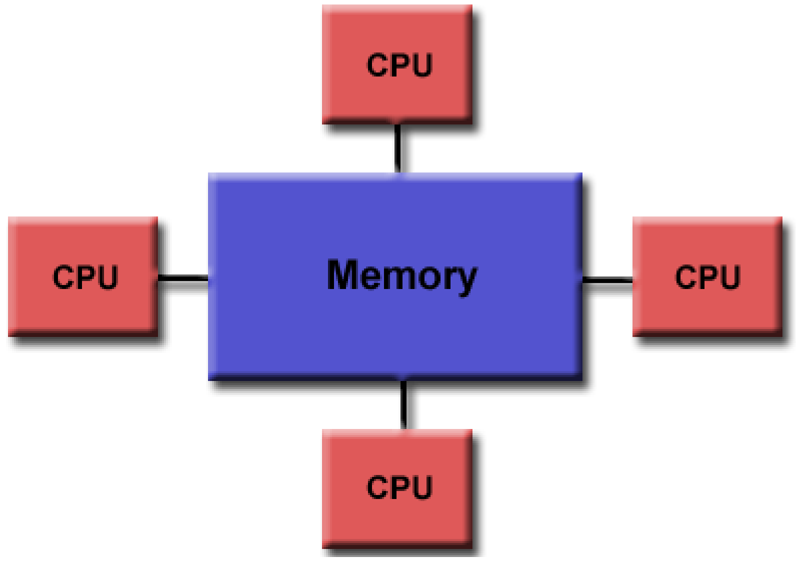
A Further Breakdown of MIMD

- **Shared-memory architecture**
  - **Symmetric multiprocessors (SMP), or uniform memory access (UMA)**
  - **Nonuniform memory access (NUMA) systems**
    - **Cache-coherent NUMA (ccNUMA)**
- **Distributed-memory architecture**
  - **Massively parallel processors (MPP)**
    - **Tightly coupled, specialized processors and network infrastructure**
  - **Clusters**
    - **Off-the-shelf computers connected by an off-the-shelf network**
- **Hybrid distributed-shared memory architecture**
- **Grids**
  - **Distributed, heterogeneous resources connected by LANs and/or WANs**

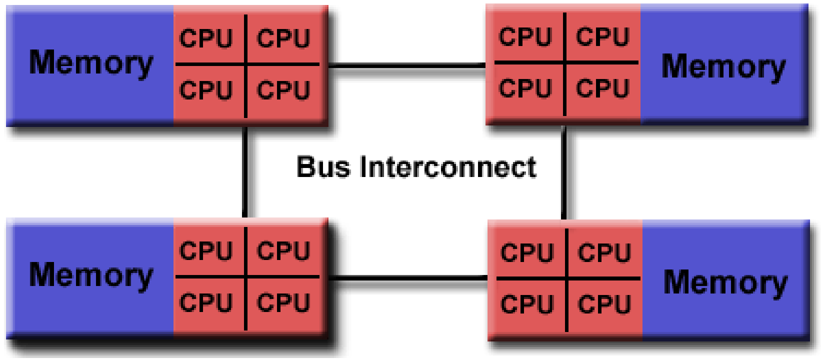
From T.G. Mattson, et al., "Patterns for Parallel Programming," Addison-Wesley Professional, 2004.

Shared-Memory Architecture

Symmetric Multiprocessors (SMP)

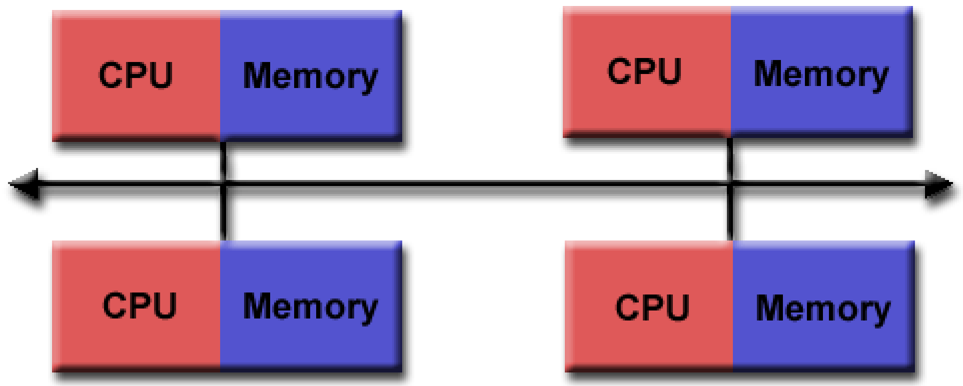


Nonuniform Memory Access (NUMA)



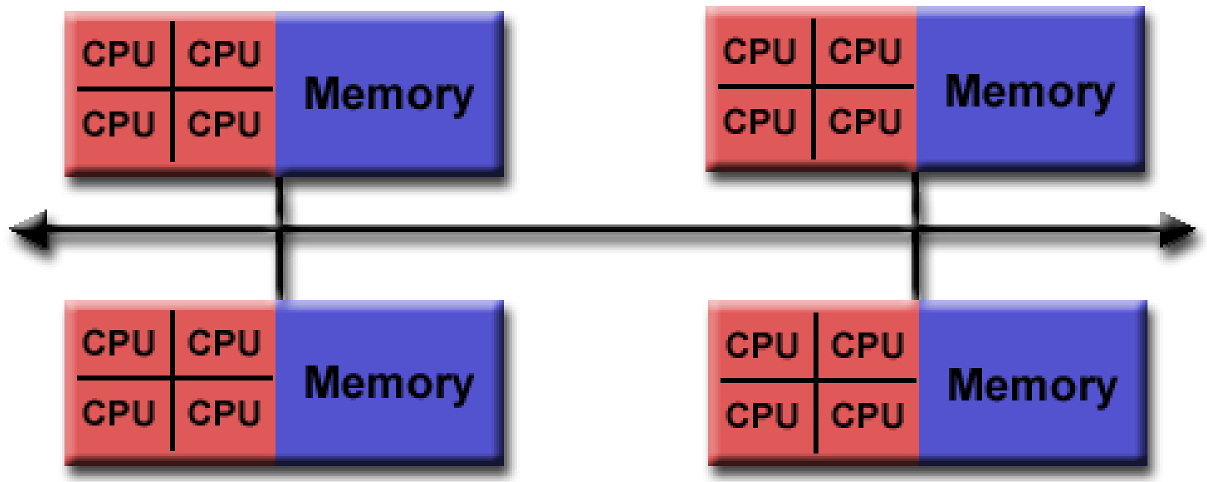
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Distributed-Memory Architecture

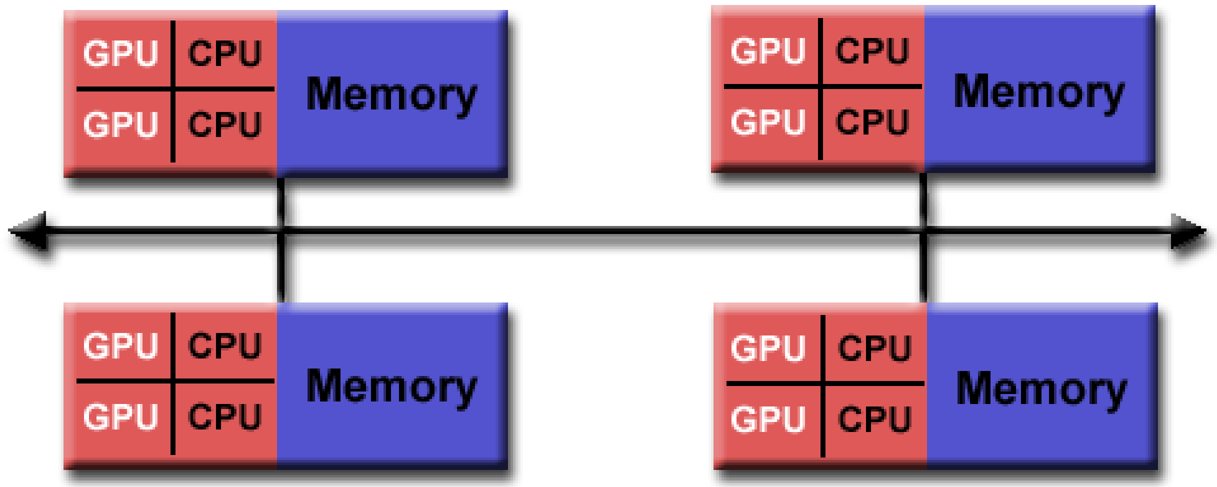


Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Hybrid Architecture



Hybrid & Homogeneous

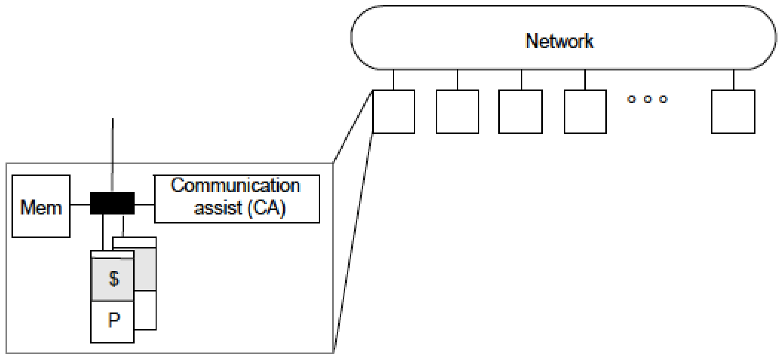


Hybrid & Heterogeneous

Source: Blaise Barney (LLNL), Introduction to Parallel Computing

Convergence: Generic Parallel Architecture

- A generic modern multiprocessor



- Node: processor(s), memory, plus communication assist
  - Network interface and communication controller
- Scalable network
- => Convergence allows lots of innovation, now within the same framework
  - integration of assist within node, what operation, how efficiently…

Outline

- Parallel programming models and architectures
- Memory limitations

Limitations of Memory System Performance

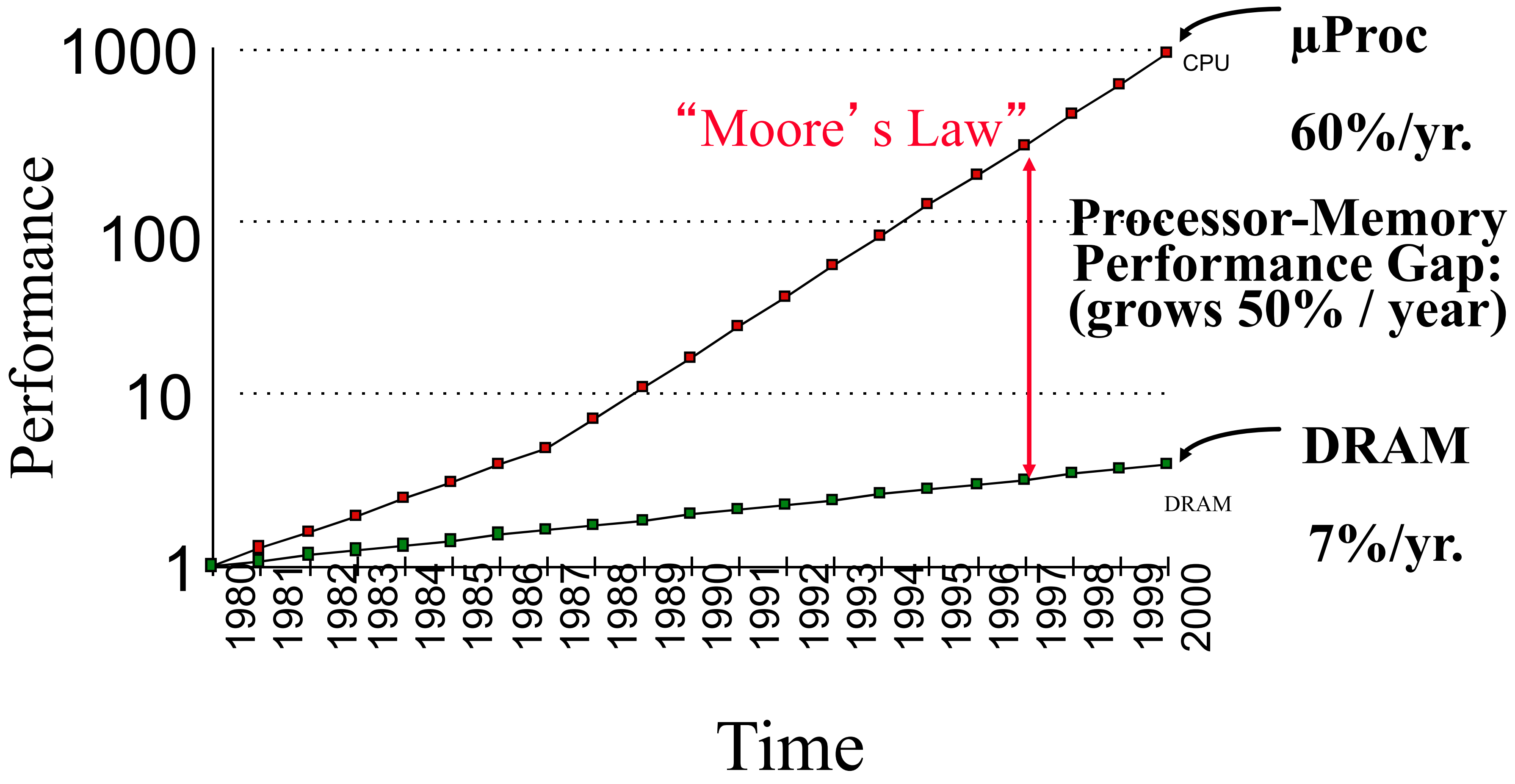
- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Difference between Bandwidth and Latency

- **An example of a fire-hose**
  - **If the water comes out of the hose two seconds after the hydrant is turned on, latency = two seconds.**
  - **Once the water starts flowing, if the hydrant delivers water at the rate of 20 liters/second, bandwidth = 5 liters/second.**
- **Serve different purposes**
  - **If you want immediate response from the hydrant, it is important to reduce latency.**
  - **If you want to fight big fires, you want high bandwidth.**

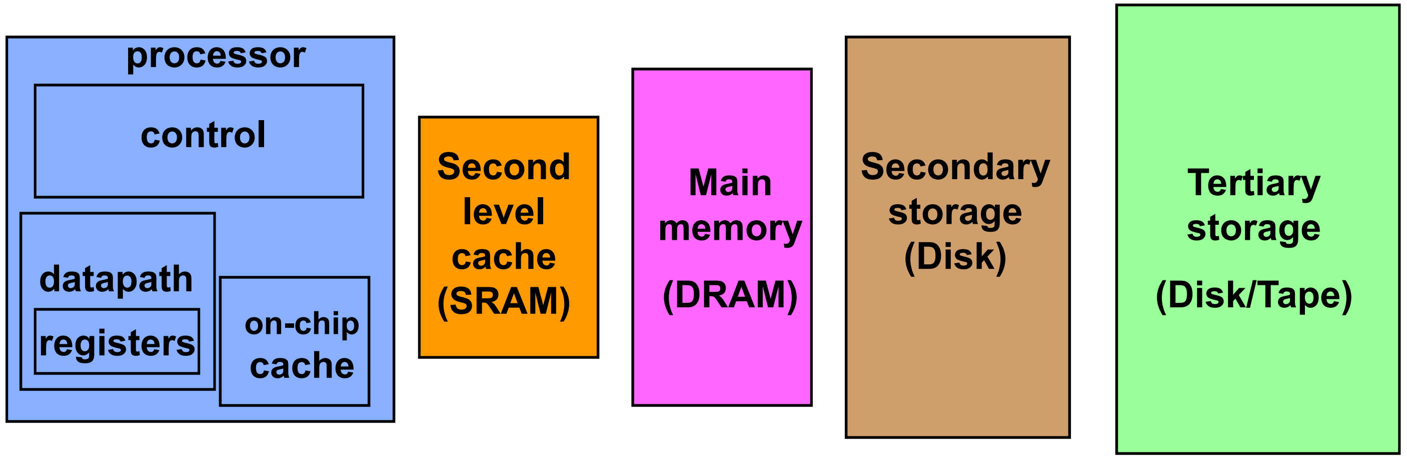
Processor-DRAM Gap (Latency)

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory



Memory Hierarchy

- Most programs have a high degree of locality in their accesses
  - spatial locality: accessing things nearby previous accesses
  - temporal locality: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

Memory Latency: An Example

- A processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches).
- Assume the processor has 2 multiply-add units; capable of executing 4 instructions in each clock cycle
  - The peak processor rating is 4 GFLOPS.
  - Every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Memory Latency: An Example (Cont'd)

- **On the previous architecture, consider the problem of computing a dot-product of two vectors.**
  - **Performs one multiply-add on each pair of vector elements**
    - **Each floating point operation requires one data fetch.**
  - **Limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS,**
    - **A very small fraction of the peak processor rating!**

Improving Effective Memory Latency Using Caches

- **Caches are small and fast memory elements between the processor and DRAM.**
  - **Acts as a low-latency high-bandwidth storage.**
  - **If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.**
- **The fraction of data references satisfied by the cache is called the cache *hit ratio*.**
- **Cache hit ratio achieved by a code on a memory system often determines its performance.**

Impact of Caches: Example

- **Introduce a cache of size 32 KB with a latency of 1 ns or one cycle in previous example.**
- **Assume we multiply two matrices A and B of dimensions 32 $\times$ 32.**
  - **Both A and B (1K each), as well as the result matrix C can be in cache**
- **Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.**
- **Multiplying two *n* $\times$ *n* matrices takes *2n*^{3} operations.**
  - **=> 64K operations, which can be performed in 16K cycles (or 16 µs) at 4 instructions per cycle.**
- **The total time for the computation = 200 + 16 µs.**
  - **A peak computation rate of 64K/216 or 303 MFLOPS**
  - **Much better!**

Impact of Caches

- Repeated references to the same data item correspond to temporal locality.
- In our example, we had *$O(n^2)$* data accesses and *$O(n^3)$* computation. This asymptotic difference makes the above example particularly desirable for caches.

- Data reuse is critical for cache performance.

Impact of Memory Bandwidth

- **Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.**
- **Memory bandwidth can be improved by increasing the size of memory blocks.**
  - **The underlying system takes *l* time units (where *l* is the latency of the system) to deliver *b* units of data (where *b* is the block size).**
- **Bandwidth has improved more than latency**
  - **23% per year vs 7% per year**

Impact of Memory Bandwidth: Example

- **In previous example, if block size is 4 words instead of 1 word.**
- **For the dot-product computation:**
  - **Assuming that the vectors are laid out linearly in memory, 8 FLOPs (4 multiply-adds) can be performed in 200 cycles.**
    - **A single memory access fetches 4 consecutive words in a vector.**
    - **2 accesses can fetch 4 elements of each of the vectors.**
  - **One FLOP every 25 ns, for a peak speed of 40 MFLOPS**
- **Bandwidth helps to hide (some) latency**

Impact of Memory Bandwidth

- **Note that increasing block size does not change latency of the system.**
- **Physically, it can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.**
- **In practice, such wide buses are expensive to construct.**
- **In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.**

Impact of Memory Bandwidth

- **In order to take advantage of full bandwidth**
  - **The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions**
  - **Spatial locality of reference**
- **If spatial locality is poor**
  - **Computations often need to be reordered to enhance spatial locality of reference.**

Impact of Memory Bandwidth: Example

Consider the following code fragment:

```
for (i = 0; i < 1000; i++)
     column_sum[i] = 0.0;
     for (j = 0; j < 1000; j++)
          column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector `column_sum`.

Potential Problem

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- What if the matrix is stored in row major (often the case)
  - The strided access results in very poor performance.

How to Fix the Problem?

We can fix the above code as follows:

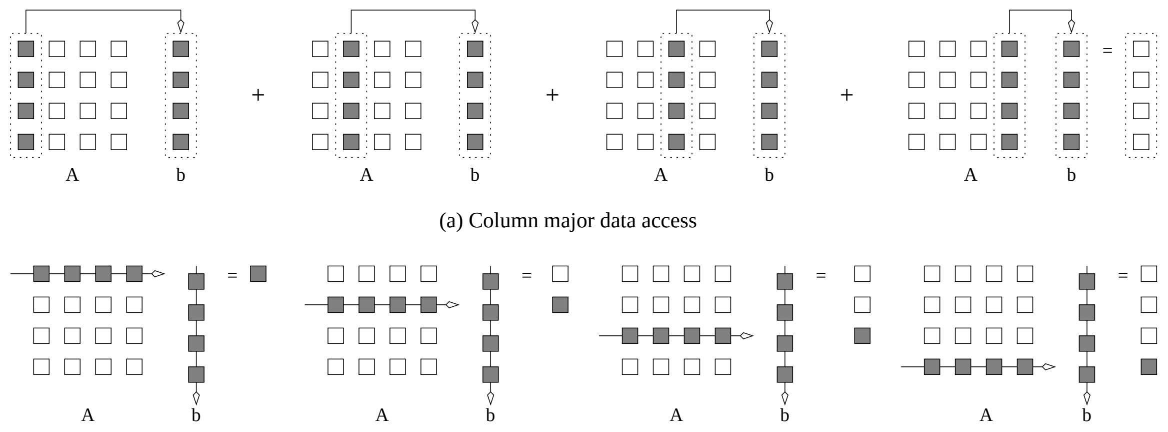
```
for (i = 0; i < 1000; i++)
     column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
     for (i = 0; i < 1000; i++)
          column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Memory Bandwidth Utilization: Another Example

- **Matrix-vector multiplication**

A x = b



Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Other Approaches for Hiding Memory Latency

- **Example: When browsing the web on a very slow network connection, one may:**
  - **anticipate which pages we are going to browse ahead of time and issue requests for them in advance;**
  - **open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or**
- **The first approach is called *prefetching*, the second *multithreading*.**

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)
     c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product,get_row(a, i), b);
```

Multithreading for Latency Hiding

- The previous example assumes:
  - The memory system is capable of servicing multiple outstanding requests, and
  - The processor is capable of switching threads at every cycle.
- It also requires the program to have an explicit specification of concurrency in the form of threads.
- GPUs and multi-threaded CPUs can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Tradeoffs of Multithreading and Prefetching

- **Multithreading and prefetching are critically impacted by the memory bandwidth.**
- **Assume**
  - **A processor with 1 GHz clock, 32K cache with line size of 16 bytes, single cycle access to the cache, and 100 ns latency to DRAM.**
  - **The computation has a cache hit ratio at**
    - **1 KB of 25%; 32 KB of 90%.**
  - **one data request in every 10 cycles**
- **1-thread and 32-threads cases**
  - **A 1-thread execution in which the entire cache is available**
    - **Required bandwidth = (1-90%) * 16B / (10 * 1ns) = 160 MB/s**
  - **A 32-thread execution where each thread has a cache residency of 1 KB**
    - **Required bandwidth = (1-25%) * 16B / (10 * 1ns) = 1.2GB/s**

Tradeoffs of Multithreading and Prefetching

- **Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.**
- **Multithreaded systems become bandwidth bound instead of latency bound.**
- **Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.**
- **Multithreading and prefetching also require significantly more hardware resources in the form of storage.**

Summary

- **Convergence in parallel architecture**
  - **Initially: close coupling of programming model and architecture**
    - **shared address space, message passing. data parallel**
  - **Now: separation and identification of dominant models/architectures**
    - **Programming models:**
      - shared address space
      - message passing
      - data parallel
    - **Architectures:**
      - small-scale shared memory
      - large-scale distributed memory
      - large-scale SMP cluster

Summary

- **Memory performance is critical:**
  - **Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.**
  - **The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.**
  - **Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.**
- **Latency and bandwidth are two related but different concepts**
  - **Memory reuse helps both latency and bandwidth**
  - **Prefetching and multi-threading can help to deal with latency assuming sufficient bandwidth**