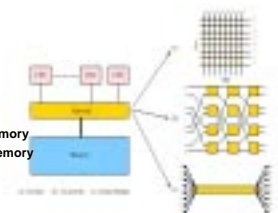


**CS 594 Applications of Parallel Computers**  
**Lecture 3:**  
**Introduction to Parallel Architectures**  
**and Programming Models**  
  
**Jack Dongarra**


**Different Architectures**

- SIMD
  - Vector Computers
- MIMD
  - Shared Memory
    - NEC SX Vector Computer
    - Cray J90/T90 Vector Computer
  - Distributed Shared Memory
    - SGI Origin Distributed Shared Memory
    - HP-Convex Distributed Shared Memory
    - SUN Enterprise Shared Memory
  - Distributed Memory
    - Cray T3E Distributed Memory
    - IBM SP Distributed Memory
    - Fujitsu VPP Vector Distributed Memory
    - Cluster of Processors (COWS, NOWS, Beowulf)




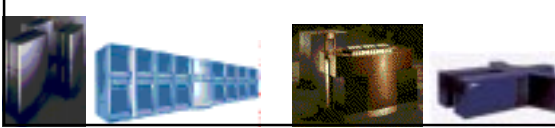
**SISD**

- RISC - Superscalar




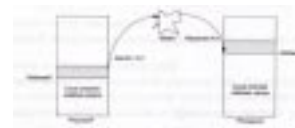

**Architectures**

- MIMD Shared memory
  - SGI Origin
  - HP-Convex Exempler
  - Cray T90/J90
  - NEC SX-4
  - Sun Enterprise

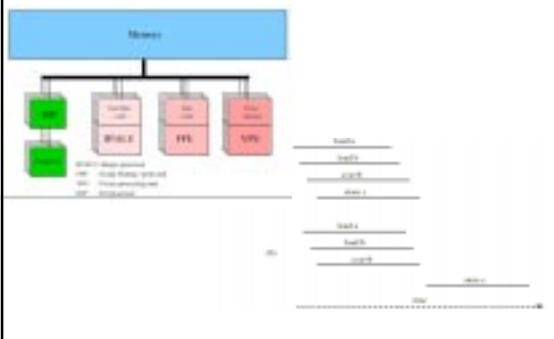



**Distributed**

- IBM SP
- CRAY/SGI T3E
- SGI Origin 2000
- HP/Convex Exempler
- Sun Enterprise
- (Intel)

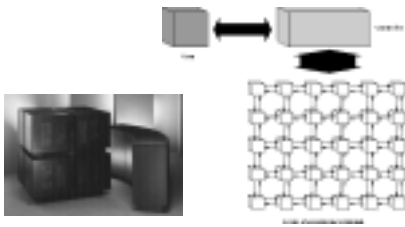




**Vector Architecture (SIMD)**



## SIMD

- (Thinking Machine CM-2)
- (MasPar)



## SGI/Cray Vector Computers

System parameters:

Model	Cray J90(se)	Cray T90
Clock cycle	10 ns (100 MHz)	2.2 ns (455 MHz)
Theor. peak performance	Per processor 200 Mflop/s	1.8 Gflop/s
No. of processors	4-32	1-32
Maximal	6.4 Gflop/s	58 Gflop/s
Main memory	<= 4 GB	<= 8 GB
Memory bandwidth		
Single proc. bandwidth	1.6 GB/s	24 GB/s

## Achieving TeraFlops

- In 1991, 1 Gflop/s
- 1000 fold increase
  - Architecture
    - exploiting parallelism
  - Processor, communication, memory
    - Moore's Law
  - Algorithm improvements
    - block-partitioned algorithms

## Future: Petaflops ( $10^{15}$ fl pt ops/s)

Today  $\approx \sqrt{10^{15}}$  flops for our workstations

- A Pflop for 1 second  $\approx$  a typical workstation computing for 1 year.
  - dynamic redistribution of workload
- From an algorithmic standpoint
  - concurrency
  - data locality
  - latency & sync
  - floating point accuracy
  - new language and constructs
  - role of numerical libraries
  - algorithm adaptation to hardware failure

## A Petaflops Computer System

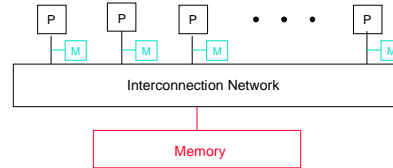
- 1 Pflop/s sustained computing
- Between 10,000 and 1,000,000 processors
- Between 10 TB and 1PB main memory
- Commensurate I/O bandwidth, mass store, etc.
- If built today, cost \$40 B and consume 1 TWatt.
- May be feasible and "affordable" by the year 2010

## Outline

- Parallel machines and programming models
- Steps in writing a parallel program
- Cost modeling and performance trade-offs

# Parallel Machines and Programming Models

## A generic parallel architecture



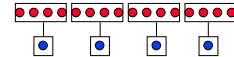
◦ Where does the memory go?

## Parallel Programming Models

- **Control**
  - how is **parallelism created**
  - what orderings exist between operations
  - how do different threads of control **synchronize**
- **Naming**
  - what data is **private vs. shared**
  - how logically shared data is **accessed or communicated**
- **Set of operations**
  - what are the basic operations
  - what operations are considered to be **atomic**
- **Cost**
  - how do we account for the cost of each of the above

## Trivial Example

- $\sum_{i=0}^{n-1} f(A[i])$
- **Parallel Decomposition:**
  - Each evaluation and each partial sum is a task
- **Assign n/p numbers to each of p procs**
  - each computes independent "private" results and partial sum
  - one (or all) collects the p partial sums and computes the global sum

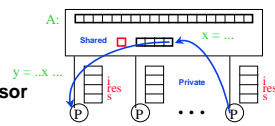


=> **Classes of Data**

- **Logically Shared**
  - the original n numbers, the global sum
- **Logically Private**
  - the individual function evaluations
  - what about the individual partial sums?

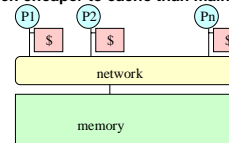
## Programming Model 1

- **Shared Address Space**
  - program consists of a collection of threads of control,
  - each with a set of private variables
    - e.g., local variables on the stack
  - collectively with a set of shared variables
    - e.g., static variables, shared common blocks, global heap
  - threads communicate implicitly by writing and reading shared variables
  - threads coordinate explicitly by synchronization operations on shared variables
    - writing and reading flags
    - locks, semaphores
- **Like concurrent programming on uniprocessor**



## Machine Model 1

- A shared memory machine
- Processors all connected to a large shared memory
- "Local" memory is not (usually) part of the hardware
  - Sun, DEC, Intel "SMPs" (Symmetric multiprocessors) in Millennium; SGI Origin
- **Cost: much cheaper to cache than main memory**



- **Machine model 1a: A Shared Address Space Machine**
  - replace caches by local memories (in abstract machine model)
  - this affects the cost model -- repeatedly accessed data should be copied
  - Cray T3E

### Shared Memory code for computing a sum

<p>Thread 1</p> <pre>[s = 0 initially] local_s1 = 0 for i = 0, n/2-1   local_s1 = local_s1 + f(A[i]) s = s + local_s1</pre>	<p>Thread 2</p> <pre>[s = 0 initially] local_s2 = 0 for i = n/2, n-1   local_s2 = local_s2 + f(A[i]) s = s + local_s2</pre>
---	---

What could go wrong?

### Pitfall and solution via synchronization

° Pitfall in computing a global sum  $s = \text{local\_s1} + \text{local\_s2}$

<p>Thread 1 (initially s=0)</p> <pre>load s [from mem to reg] s = s + local_s1 [local_s1, in reg] store s [from reg to mem]</pre>	<p>Thread 2 (initially s=0)</p> <pre>load s [from mem to reg; initially 0] s = s + local_s2 [local_s2, in reg] store s [from reg to mem]</pre>
---	--

Time

° Instructions from different threads can be interleaved arbitrarily  
 ° What can final result s stored in memory be?

° **Race Condition**

° Possible solution: **Mutual Exclusion with Locks**

<p>Thread 1</p> <pre>lock load s s = s + local_s1 store s unlock</pre>	<p>Thread 2</p> <pre>lock load s s = s + local_s2 store s unlock</pre>
--	--

° Locks must be **atomic** (execute completely without interruption)

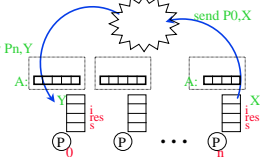
### Programming Model 2

° **Message Passing**

- program consists of a collection of **named** processes
  - thread of control plus local address space
  - local variables, static variables, common blocks, heap
- processes communicate by explicit data transfers
  - matching pair of send & receive by source and dest. proc.
- coordination is implicit in every communication event
- logically shared data is partitioned over local processes

° Like distributed programming

° Program with standard libraries: MPI, PVM



### Machine Model 2

° **A distributed memory machine**

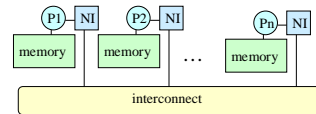
- Cray T3E (too!), IBM SP2, NOW, Millennium

° **Processors all connected to own memory (and caches)**

- cannot directly access another processor's memory

° **Each "node" has a network interface (NI)**

- all communication and synchronization done through this



### Computing $s = x(1)+x(2)$ on each processor

° First possible solution

<p>Processor 1</p> <pre>send xlocal, proc2 [xlocal = x(1)] receive xremote, proc2 s = xlocal + xremote</pre>	<p>Processor 2</p> <pre>receive xremote, proc1 send xlocal, proc1 [xlocal = x(2)] s = xlocal + xremote</pre>
--	--

° Second possible solution - what could go wrong?

<p>Processor 1</p> <pre>send xlocal, proc2 [xlocal = x(1)] receive xremote, proc2 s = xlocal + xremote</pre>	<p>Processor 2</p> <pre>send xlocal, proc1 [xlocal = x(2)] receive xremote, proc1 s = xlocal + xremote</pre>
--	--

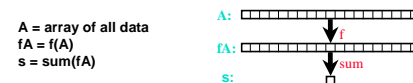
° What if send/receive act like the telephone system? The post office?

### Programming Model 3

° **Data Parallel**

- Single sequential thread of control consisting of **parallel operations**
- Parallel operations applied to all (or defined subset) of a data structure
- Communication is implicit in parallel operators and "shifted" data structures
- Elegant and easy to understand and reason about
- Not all problems fit this model

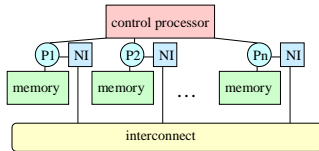
° Like marching in a regiment



° Think of Matlab

### Machine Model 3

- An SIMD (Single Instruction Multiple Data) machine
- A large number of small processors
- A single “control processor” issues each instruction
  - each processor executes the same instruction
  - some processors may be turned off on any instruction



- Machines not popular (CM2), but programming model is
  - implemented by mapping n-fold parallelism to p processors
  - mostly done in the compilers (HPF = High Performance Fortran)

### Machine Model 4

- Since small shared memory machines (SMPs) are the fastest commodity machine, why not build a larger machine by connecting many of them with a network?
- CLUMP = Cluster of SMPs
- Shared memory within one SMP, message passing outside
- Millennium, ASCI Red (Intel), ...
- Programming model?
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignore important part of memory hierarchy)
  - Expose two layers: shared memory and message passing (higher performance, but ugly to program)

### Programming Model 5

- Bulk synchronous
- Used within the message passing or shared memory models as a programming convention
- Phases separated by global barriers
  - **Compute phases**: all operate on local data (in distributed memory)
    - or **read** access to global data (in shared memory)
  - **Communication phases**: all participate in rearrangement or reduction of global data
- Generally all doing the “same thing” in a phase
  - all do f, but may all do different things within f
- Simplicity of data parallelism without restrictions

### Summary so far

- Historically, each parallel machine was unique, along with its programming model and programming language
- You had to throw away your software and start over with each new kind of machine - ugh
- Now we distinguish the programming model from the underlying machine, so we can write portably **correct** code, that runs on many machines
  - MPI now the most portable option, but can be tedious
- Writing portably **fast** code requires tuning for the architecture
  - Algorithm design challenge is to make this process easy
  - Example: picking a blocksize, not rewriting whole algorithm

## Steps in Writing Parallel Programs

### Creating a Parallel Program

- Pieces of the job
  - Identify work that can be done in parallel
  - Partition work and perhaps data among processes=threads
  - Manage the data access, communication, synchronization

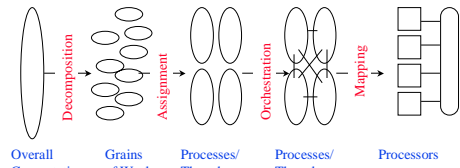
- Goal: maximize Speedup due to parallelism

$$\text{Speedup}_{\text{prob}}(P \text{ procs}) = \frac{\text{Time to solve prob with "best" sequential solution}}{\text{Time to solve prob in parallel on } P \text{ processors}}$$
$$\text{Efficiency}(P) = \frac{\text{Speedup}(P)}{P}$$

$\leq P$  (Brent's Theorem)  
 $\leq 1$

- Key question is when you can solve each piece
  - **statically**, if information is known in advance
  - **dynamically**, otherwise

### Steps in the Process



- **Task:** arbitrarily defined piece of work that forms the basic unit of concurrency
- **Process/Thread:** abstract entity that performs tasks
  - tasks are assigned to threads via an assignment mechanism
  - threads must coordinate to accomplish their collective tasks
- **Processor:** physical entity that executes a thread

### Decomposition

- **Break the overall computation into grains of work (tasks)**
  - identify concurrency and decide at what level to exploit it
  - concurrency may be statically identifiable or may vary dynamically
  - it may depend only on problem size, or it may depend on the particular input data
- **Goal:** enough tasks to keep the target range of processors busy, but not too many
  - establishes upper limit on number of useful processors (i.e., scaling)

### Assignment

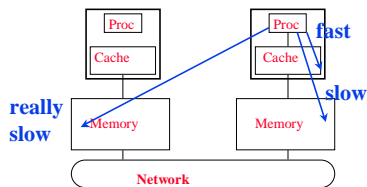
- **Determine mechanism to divide work among threads**
  - functional partitioning
    - assign logically distinct aspects of work to different threads
      - eg pipelining
  - structural mechanisms
    - assign iterations of "parallel loop" according to simple rule
      - eg proc j gets iterates  $j*n/p$  through  $(j+1)*n/p-1$
    - throw tasks in a bowl (task queue) and let threads feed
  - data/domain decomposition
    - data describing the problem has a natural decomposition
    - break up the data and assign work associated with regions
      - eg parts of physical system being simulated
- **Goal**
  - Balance the workload to keep everyone busy (all the time)
  - Allow efficient orchestration

### Orchestration

- **Provide a means of**
  - naming and accessing shared data,
  - communication and coordination among threads of control
- **Goals:**
  - correctness of parallel solution
    - respect the inherent dependencies within the algorithm
  - avoid serialization
  - reduce cost of communication, synchronization, and management
  - preserve locality of data reference

### Mapping

- **Binding processes to physical processors**
- **Time to reach processor across network does not depend on which processor (roughly)**
  - lots of old literature on "network topology", no longer so important
- **Basic issue is how many remote accesses**



### Example

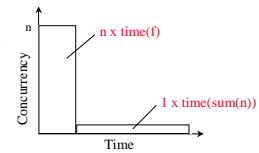
- $s = f(A[1]) + \dots + f(A[n])$
- **Decomposition**
  - computing each  $f(A[j])$
  - n-fold parallelism, where n may be  $\gg p$
  - computing sum s
- **Assignment**
  - thread k sums  $s_k = f(A[k*n/p]) + \dots + f(A[(k+1)*n/p-1])$
  - thread 1 sums  $s = s_1 + \dots + s_p$  (for simplicity of this example)
  - thread 1 communicates s to other threads
- **Orchestration**
  - starting up threads
  - communicating, synchronizing with thread 1
- **Mapping**
  - processor j runs thread j

## Cost Modeling and Performance Tradeoffs

### Identifying enough Concurrency

- **Parallelism profile**
  - area is total work done

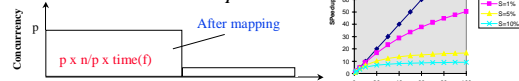
Simple Decomposition:  
f ( A[i] ) is the parallel task  
sum is sequential



- **Amdahl's law**

- let s be the fraction of total work done sequentially

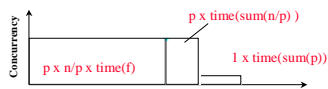
$$Speedup(P) \leq \frac{1}{1-s} \leq \frac{1}{s + \frac{1-s}{P}}$$



### Algorithmic Trade-offs

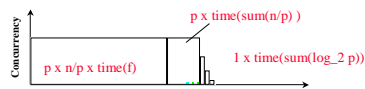
- **Parallelize partial sum of the f's**

- what fraction of the computation is "sequential"



- what does this do for communication? locality?
- what if you sum what you "own"

- **Parallelize the final summation (tree sum)**



- Generalize Amdahl's law for arbitrary "ideal" parallelism profile

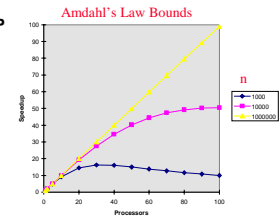
### Problem Size is Critical

- Suppose Total work = n + P

- Serial work: P

- Parallel work: n

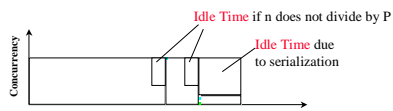
- s = serial fraction = P / (n+P)



In general seek to exploit a fraction of the peak parallelism in the problem.

### Load Balance

- Insufficient Concurrency will appear as **load imbalance**



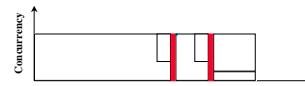
- Use of coarser grain tends to increase load imbalance.
- Poor assignment of tasks can cause load imbalance.
- Synchronization waits are instantaneous load imbalance

$$Speedup(P) \leq \frac{Work(1)}{\max_p(Work(p) + idle)}$$

### Extra Work

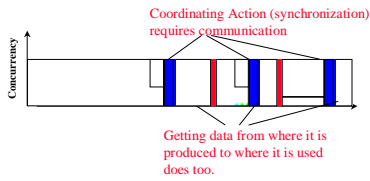
- There is always some amount of extra work to manage parallelism

- e.g., to decide who is to do what



$$Speedup(P) \leq \frac{Work(1)}{\max_p(Work(p) + idle + extra)}$$

### Communication and Synchronization

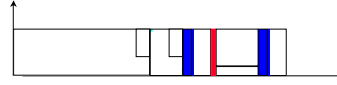


$$Speedup(P) \leq \frac{Work(1)}{\max(Work(P) + idle + extra + comm)}$$

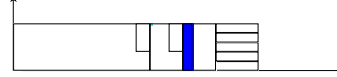
- There are many ways to reduce communication costs.

### Reducing Communication Costs

- Coordinating placement of work and data to eliminate unnecessary communication



- Replicating data
- Redundant work



- Performing required communication efficiently
  - e.g., transfer size, contention, machine specific optimizations

### The Tension

$$Speedup(P) \leq \frac{Work(1)}{\max(Work(P) + idle + comm + extraWork)}$$

Minimizing one tends to increase the others

- Fine grain decomposition and flexible assignment tends to minimize load imbalance at the cost of increased communication

- In many problems communication goes like the surface-to-volume ratio
- Larger grain => larger transfers, fewer synchronization events



- Simple static assignment reduces extra work, but may yield load imbalance

### The Good News

- The basic work component in the parallel program may be more efficient than in the sequential case
  - Only a small fraction of the problem fits in cache
  - Need to chop problem up into pieces and concentrate on them to get good cache performance.
  - Similar to the parallel case
  - Indeed, the best sequential program may emulate the parallel one.
- Communication can be hidden behind computation
  - May lead to better algorithms for memory hierarchies
- Parallel algorithms may lead to better serial ones
  - parallel search may explore space more effectively