

**CS 267: Applications of Parallel Computers**

**Lecture 3:  
Introduction to Parallel Architectures  
and Programming Models**

Kathy Yelick

<http://www-inst.eecs.berkeley.edu/~cs267>

9/5/2001 CS267, Yelick

Recap of Last Lecture

- Memory systems on modern processors are complicated.
- The performance of a simple program can depend on the details of the micro-architecture.
- Simple performance models can aid in understanding
- Two ratios are key to efficiency
  - algorithmic:  $q = f/m = \# \text{ floating point ops} / \# \text{ slow memory ops}$
  - $t_m/t_f = \text{time for slow memory operation} / \text{time for floating point operation}$
- A common technique for improving cache performance (lowering  $q$ ) is called **blocking**
- Applied to matrix multiplication.

9/5/2001 CS267, Yelick

Outline

- Lecture 2 follow-up
  - Use of search in blocking matrix multiply
  - Strassen's matrix multiply algorithm
    - Bag of tricks for optimizing serial code
- Overview of parallel machines and programming models
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs
- Trends in real machines

9/5/2001 CS267, Yelick

Search Over Block Sizes

- Performance models are useful for high level algorithms
  - Helps in developing a blocked algorithm
  - Models have not proven very useful for block size selection
    - too complicated to be useful
      - See work by Sid Chatterjee for detailed model
    - too simple to be accurate
      - Multiple multidimensional arrays, virtual memory, etc.
- Some systems use search
  - Atlas – being incorporated into Matlab
  - BeBOP – <http://www.cs.berkeley.edu/~richie/bebop>

9/5/2001 CS267, Yelick

What the Search Space Looks Like

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-III, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

9/5/2001 CS267, Yelick

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has  $O(n^3)$  flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{2.81})$
- Consider a 2x2 matrix multiply
  - normally 8 multiplies, Strassen does it with 7 multiplies (but many more adds)

```

Let M = [m11 m12] = [a11 a12] * [b11 b12]
        [m21 m22]   [a21 a22]   [b21 b22]

Let p1 = (a12 - a22) * (b21 + b22)      p5 = a11 * (b12 - b22)
p2 = (a11 + a22) * (b11 + b22)        p6 = a22 * (b21 - b11)
p3 = (a11 - a21) * (b11 + b12)        p7 = (a21 + a22) * b11
p4 = (a11 + a12) * b22

Then m11 = p1 + p2 - p4 + p6
     m12 = p4 + p5
     m21 = p6 + p7
     m22 = p2 - p3 + p5 - p7
            
```

Extends to nxn by divide&conquer

9/5/2001 CS267, Yelick

### Strassen (continued)

$T(n)$  = Cost of multiplying  $n \times n$  matrices  
 $= 7 \cdot T(n/2) + 18 \cdot (n/2)^2$   
 $= O(n \log_2 7)$   
 $= O(n^{2.81})$

- Asymptotically faster
  - Several times faster for large  $n$  in practice
  - Cross-over depends on machine
  - Available in several libraries
- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error
  - Current world's record is  $O(n^{2.376...})$
- Why does Hong/Kung theorem not apply?

9/5/2001 CS267, Yelick

### Outline

- Lecture 2 follow-up
  - Use of search in blocking matrix multiply
  - Strassen's matrix multiply algorithm
  - Bag of tricks for optimizing serial code
- Overview of parallel machines and programming models
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs
- Trends in real machines

9/5/2001 CS267, Yelick

### Removing False Dependencies

Using local variables, reorder operations to remove false dependencies

```

a[i] = b[i] + c;
a[i+1] = b[i+1] * d;
    
```

false read-after-write hazard between a[i] and b[i+1]

↓

```

float f1 = b[i];
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 * d;
    
```

With some compilers, you can declare a and b unaliased.

- Done via "restrict pointers," compiler flag, or pragma

9/5/2001 CS267, Yelick

### Exploit Multiple Registers

Reduce demands on memory bandwidth by pre-loading into local variables

```

while( ... ) {
    *res++ = filter[0]*signal[0]
           + filter[1]*signal[1]
           + filter[2]*signal[2];
    signal++;
}
    
```

↓

```

float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( ... ) {
    *res++ = f0*signal[0]
           + f1*signal[1]
           + f2*signal[2];
    signal++;
}
    
```

also: register float f0 = ...;

Example is a convolution

9/5/2001 CS267, Yelick

### Minimize Pointer Updates

Replace pointer updates for strided memory addressing with constant array offsets

```

f0 = *x8; x8 += 4;
f1 = *x8; x8 += 4;
f2 = *x8; x8 += 4;
    
```

↓

```

f0 = x8[0];
f1 = x8[4];
f2 = x8[8];
x8 += 12;
    
```

Pointer vs. array expression costs may differ.

- Some compilers do a better job at analyzing one than the other

9/5/2001 CS267, Yelick

### Loop Unrolling

Expose instruction-level parallelism

```

float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
    signal += 3;
    s0 = signal[0];
    res[0] = f0*s1 + f1*s2 + f2*s0;

    s1 = signal[1];
    res[1] = f0*s2 + f1*s0 + f2*s1;

    s2 = signal[2];
    res[2] = f0*s0 + f1*s1 + f2*s2;

    res += 3;
} while( ... );
    
```

9/5/2001 CS267, Yelick

### Expose Independent Operations

- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

```

f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
    
```

9/5/2001 CS267, Yelick

### Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality

Original matrix  
(numbers are addresses)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Reorganized into  
2x2 blocks

0	2	8	10
1	3	9	11
4	5	12	13
6	7	14	15

9/5/2001 CS267, Yelick

### Outline

- Lecture 2 follow-up
  - Use of search in blocking matrix multiply
  - Strassen's matrix multiply algorithm
  - Bag of tricks for optimizing serial code
- ➔ Overview of parallel machines and programming models
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs
- Trends in real machines

9/5/2001 CS267, Yelick

### A generic parallel architecture

◦ Where is the memory physically located?

9/5/2001 CS267, Yelick

### Parallel Programming Models

- Control
  - How is parallelism **created**?
  - What **orderings** exist between operations?
  - How do different threads of control **synchronize**?
- Data
  - What data is **private** vs. **shared**?
  - How is logically shared data accessed or **communicated**?
- Operations
  - What are the **atomic** operations?
- Cost
  - How do we account for the **cost** of each of the above?

9/5/2001 CS267, Yelick

### Simple Example

Consider a sum of an array function:  $\sum_{i=0}^{n-1} f(A[i])$

- 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.

Two Classes of Data:

- Logically Shared
  - The original  $n$  numbers, the global sum.
- Logically Private
  - The individual function evaluations.
  - What about the individual partial sums?

9/5/2001 CS267, Yelick

### Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Many languages allow threads to be created dynamically, i.e., mid-execution.
- Each thread has 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 using synchronization operations on shared variables

Address:  $y = \dots x \dots$   $x = \dots$

9/5/2001 CS267, Yelick

### Machine Model 1a: Shared Memory

- Processors all connected to a large shared memory.
  - Typically called Symmetric Multiprocessors (SMPs)
  - Sun, DEC, Intel, IBM SMPs (nodes of Millennium, SP)
- "Local" memory is not (usually) part of the hardware.
- Cost: much cheaper to access data in cache than in main memory.
- Difficulty scaling to large numbers of processors
  - <10 processors typical

9/5/2001 CS267, Yelick

### Machine Model 1b: Distributed Shared Memory

- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI Origin is canonical example (+ research machines)
  - Scales to 100s
  - Limitation is cache consistency protocols – need to keep cached copies of the same address consistent

9/5/2001 CS267, Yelick

### Shared Memory Code for Computing a Sum

```

static int s = 0;

Thread 1
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
s = s + local_s1

Thread 2
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
s = s + local_s2
    
```

What is the problem?

- A **race condition** or **data race** occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized)

9/5/2001 CS267, Yelick

### Pitfalls and Solution via Synchronization

° Pitfall in computing a global sum  $s = s + local\_si$ :

<p>Thread 1 (initially s=0)</p> <p>load s [from memto reg]</p> <p><math>s = s + local\_s1</math> [local_s1, in reg]</p> <p>store s [from reg to mem]</p>	<p>Thread 2 (initially s=0)</p> <p>load s [from memto reg; initially 0]</p> <p><math>s = s + local\_s2</math> [local_s2, in reg]</p> <p>store s [from reg to mem]</p>
--	---

Time

- Instructions from different threads can be interleaved arbitrarily.
- One of the additions may be lost
- Possible solution: **mutual exclusion** with **locks**

<p>Thread 1</p> <p>lock</p> <p>load s</p> <p><math>s = s + local\_s1</math></p> <p>store s</p> <p>unlock</p>	<p>Thread 2</p> <p>lock</p> <p>load s</p> <p><math>s = s + local\_s2</math></p> <p>store s</p> <p>unlock</p>
--	--

9/5/2001 CS267, Yelick

### Programming Model 2: Message Passing

- Program consists of a collection of **named** processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit **send/receive** pairs
  - Coordination is implicit in every communication event.
  - MPI is the most common example

9/5/2001 CS267, Yelick

### Machine Model 2: Distributed Memory

- Cray T3E, IBM SP, Millennium.
- Each processor is connected to its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a network interface (NI) for all communication and synchronization.

9/5/2001 CS267, Yelick

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

- First possible solution:

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

- Second possible solution -- what could go wrong?

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

- What if send/receive acts like the telephone system? The post office?

9/5/2001 CS267, Yelick

### Programming Model 2b: Global Addr Space

- Program consists of a collection of **named** processes.
  - Usually fixed at program startup time
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes
  - Remote data stays remote on distributed memory machines
- Processes communicate by writes to shared variables
  - Explicit synchronization needed to coordinate
  - UPC, Titanium, Split-C are some examples
- Global Address Space programming is an intermediate point between message passing and shared memory
- Most common on the Cray t3e, which had some hardware support for remote reads/writes

9/5/2001 CS267, Yelick

### Programming Model 3: Data Parallel

- Single thread of control consisting of **parallel operations**.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously
- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines

9/5/2001 CS267, Yelick

### Machine Model 3: SIMD System

- A large number of (usually) small processors.
  - A single "control processor" issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.
- Machines are not popular (CM2), but programming model is.

- Implemented by mapping  $m$ -fold parallelism to  $p$  processors.
- Mostly done in the compilers (e.g., HPF).

9/5/2001 CS267, Yelick

### Machine Model 4: Clusters of SMPs

- SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network
- Common names:
  - CLUMP = Cluster of SMPs
  - Hierarchical machines, constellations
- Most modern machines look like this:
  - Millennium, IBM SPs, (not the t3e)...
- What is an appropriate programming model #4 ???
  - Treat machine as "flat", always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.

9/5/2001 CS267, Yelick

**Outline**

- Lecture 2 follow-up
  - Use of search in blocking matrix multiply
  - Strassen's matrix multiply algorithm
  - Bag of tricks for optimizing serial code
- Overview of parallel machines and programming models
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs

➡ Trends in real machines

9/5/2001 CS267, Yelick

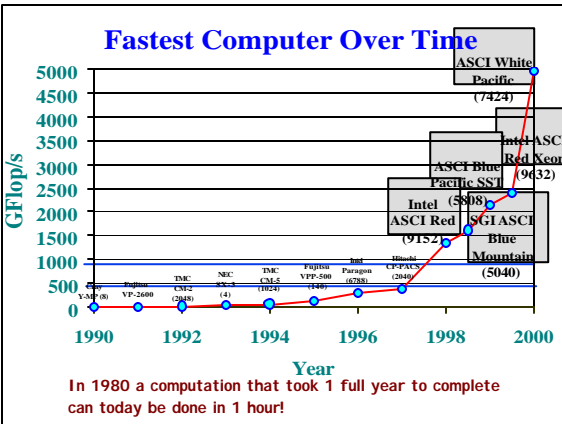
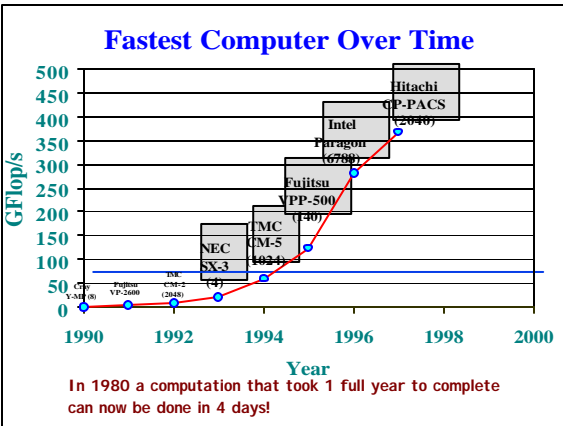
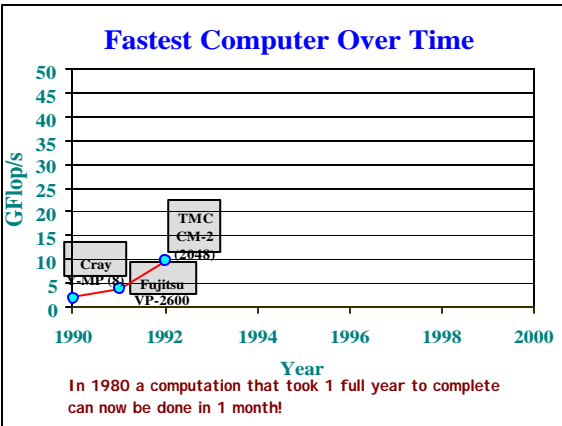
**Top 500 Supercomputers**

- Listing of the 500 most powerful computers in the world
  - Yardstick: Rmax from LINPACK MPP benchmark  
 $Ax=b$ , dense problem

- Dense LU Factorization (dominated by matrix multiply)

- Updated twice a year SC'xy in the States in November
  - Meeting in Mannheim, Germany in June
  - All data (and slides) available from [www.top500.org](http://www.top500.org)
  - Also measures N-1/2 (size required to get 1/2 speed)

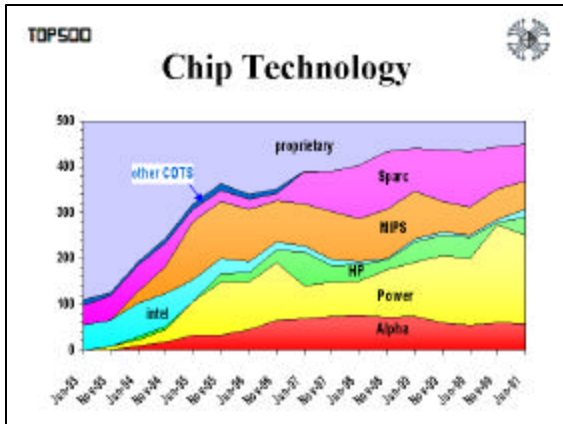
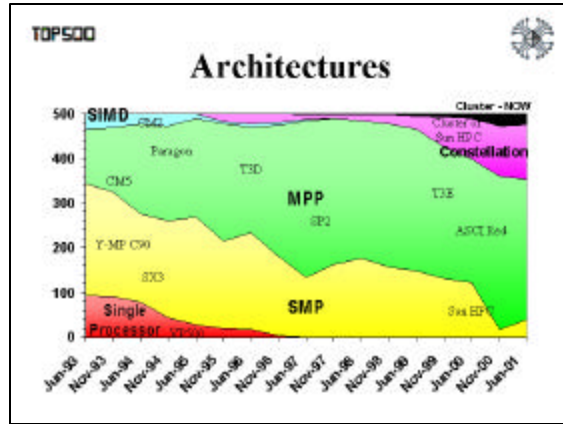
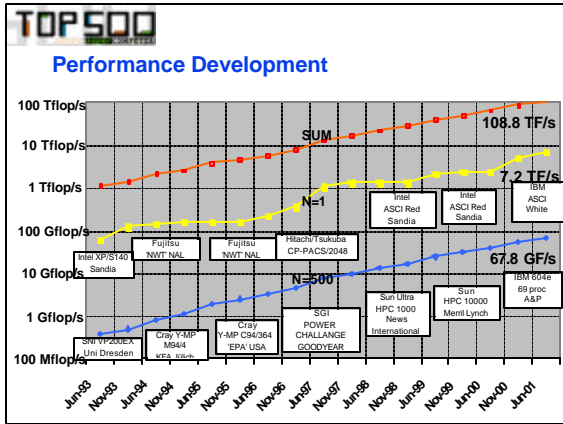
9/5/2001 CS267, Yelick



**Top 10 of the Fastest Computers in the World**

Rank	Computer	GFlop/s	Year
1	ASCI White, SP Power3	1104	2001
2	SP Power3 375 MHz 16	1179	2001
3	ASCI Blue, Pacific SST	9632	1999
4	ASCI Blue, Pacific SST	9152	1998
5	Intel (S408)	9632	1999
6	SG/ASCI Red	7424	2000
7	Intel (S408)	7424	2000
8	Intel (S408)	7424	2000
9	Intel (S408)	7424	2000
10	IBM SP Power3 375 MHz	1104	2001

9/5/2001 CS267, Yelick



**Summary**

- Historically, each parallel machine was unique, along with its programming model and programming language.
- It was necessary to throw away software and start over with each new kind of machine.
- Now we distinguish the programming model from the underlying machine, so we can write portably **correct** codes that run 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.

9/5/2001 CS267, Yelick