COMP 422, Lecture 2: Parallel Computing Platforms and Memory System Performance
(Sections 2.2 & 2.3 of textbook)

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for today’s lecture

• Jack Dongarra (U. Tennessee) --- CS 594 slides from Spring 2008

• John Mellor-Crummey (Rice) --- COMP 422 slides from Spring 2007

• Kathy Yelick (UC Berkeley) --- CS 267 slides from Spring 2007
  —http://www.eecs.berkeley.edu/~yelick/cs267_sp07/lectures

• Slides accompanying course textbook
  —http://www-users.cs.umn.edu/~karypis/parbook/
Course Information

• Meeting time: TTh 10:50-12:05
• Meeting place: DH 1046

• Instructor: Vivek Sarkar
  — vsarkar@rice.edu, x5304, DH 3131
  — Office hours: By appointment

• TA: Raj Barik
  — rajbarik@rice.edu, x2738, DH 2070
  — Office hours: Tuesdays & Thursdays, 1pm - 2pm, and by appointment

• Web site: http://www.owlnet.rice.edu/~comp422
Homework #1 (due Jan 15, 2008)

• Apply for an account on the Ada cluster, if you don’t already have one
  — Go to https://rcsg.rice.edu/apply
  — Click on "Apply for a class user account"

• Send email to TA (rajbarik@rice.edu) with
  — Your userid on Ada
  — Your preference on whether to do assignments individually or in two-person teams (in which case you should also include your team partner’s name)
  — A ranking of C, Fortran, and Java as your language of choice for programming assignments
    – This is for planning purposes; we cannot guarantee that your top choice will suffice for all programming assignments
Consider three processor configurations, all of which consume the same power
- C1: 1 core executing at 2GHz
- C2: 8 cores executing at 1GHz
- C3: 64 cores executing at 500MHz each

Q1: Assuming 1 op/cycle, what is the ideal performance in ops/sec for each configuration?

Now consider a program P with N operations such that 50% of the ops have 8-way parallelism and 50% have 64-way parallelism

Q2: Ignoring memory/communication and other overheads, how much time will be needed to execute program P on each of C1, C2, and C3?
Section 2.3: Dichotomy of Parallel Computing Platforms

**Shared memory** - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

**Distributed memory** - each processor has its own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)
Shared-Memory: UMA vs. NUMA

Uniform memory access (UMA): Each processor has uniform access to memory. Also known as symmetric multiprocessors (Sun E10000)

Non-uniform memory access (NUMA): Time for memory access depends on location of data. Local access is faster than non-local access. Easier to scale than SMPs (SGI Origin)
Control Structure of Parallel Platforms

• Processor control structure alternatives
  — operate under the centralized control of a single control unit
  — work independently

• SIMD
  — Single Instruction stream
    – single control unit dispatches the same instruction to processors
  — Multiple Data streams
    – processors work on different data

• MIMD
  — Multiple Instruction stream
    – each processor has its own control control unit
    – each processor can execute different instructions
  — Multiple Data stream
    – processors work on different data items
A typical SIMD architecture (a) and a typical MIMD architecture (b).
SIMD Processors

• Examples include many early parallel computers
  — Illiac IV, MPP, DAP, CM-2, and MasPar MP-1

• SIMD control found today in vector units and co-processors
  — Examples of SIMD vector units: MMX, SSE, Altivec
  — Examples of SIMD co-processors: ClearSpeed array processor, nVidia G80 GPGPU

• SIMD relies on regular structure of computations
  — media processing
  — scientific kernels (e.g. linear algebra, FFT)

• Activity mask
  — per PE predicated execution: turn off operations on certain PEs
    – each PE tests own conditional and sets own activity mask
    – PE can conditionally perform operation predicated on mask value
Multi-Threaded Array Processing
- Hardware multi-threading
- Asynchronous, overlapped I/O
- Run-time extensible instruction set

Array of 96 Processor Elements (PEs)
- 64-bit and 32-bit floating point
- 210 MHz... key to low power
- 128 million transistors
- Low Power, Approx 10 Watts
Conditional Execution on SIMD Processors

conditional statement

initial values

execute “then” branch

execute “else” branch

if (B == 0)
  C = A;
else
  C = A/B;

(a)

Processor 0

A 5
B 0
C 0

Processor 1

A 4
B 2
C 0

Processor 2

A 1
B 1
C 0

Processor 3

A 0
B 0
C 0

Initial values

Step 1

Processor 0

A 5
B 0
C 5

Processor 1

A 4
B 2
C 0

Processor 2

A 1
B 1
C 0

Processor 3

A 0
B 0
C 0

Step 2

Processor 0

A 5
B 0
C 5

Processor 1

A 4
B 2
C 2

Processor 2

A 1
B 1
C 1

Processor 3

A 0
B 0
C 0
SSE/SSE2 as examples of SIMD vector units

• Scalar processing
  -- traditional mode
  -- one operation produces one result

• SIMD vector units
  -- with SSE / SSE2
  -- one operation produces multiple results

Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation
SSE / SSE2 SIMD on Intel

- **SSE2 data types:** anything that fits into 16 bytes, e.g.,
  - 4x floats
  - 2x doubles
  - 16x bytes

- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel

- **Challenges:**
  - Need to be contiguous in memory and aligned
  - Instructions provided to mask data and move data around from one part of register to another
Interconnect-Related Terms

- Both shared and distributed memory systems have:
  1. processors: now generally commodity RISC processors
  2. memory: now generally commodity DRAM
  3. network/interconnect: between the processors and memory (bus, crossbar, fat tree, torus, hypercube, etc.)

  — *Latency*: How long does it take to start sending a "message"? Measured in microseconds.

  — *Bandwidth*: What data rate can be sustained once the message is started? Measured in Mbytes/sec.

  — *Topology*: the manner in which the nodes are connected
Bandwidth vs. Latency in a Pipeline

Dave Patterson’s Laundry example: 4 people doing laundry
wash (30 min) + dry (40 min) + fold (20 min) = 90 min

- In this example:
  - Sequential execution takes $4 \times 90\text{min} = 6\text{ hours}$
  - Pipelined execution takes $30 + 4 \times 40 + 20 = 3.5\text{ hours}$

- Bandwidth = loads/hour
- $BW = 4/6 \text{ l/h w/o pipelining}$
- $BW = 4/3.5 \text{ l/h w pipelining}$
- $BW \leq 1.5 \text{ l/h w pipelining, more total loads}$
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipe stages
Example of Memory System Performance Limitations (Section 2.2)

• Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches).
  — Assume that the processor is capable of executing one floating-point instructions per cycle, and therefore has a peak performance rating of 1 GFLOPS.

• On the above architecture, consider the problem of adding two vectors
  — Each floating point operation requires two data accesses
  — It follows that the peak speed of this computation is limited to one floating point operation every 200 ns, or a speed of 5 MFLOPS, a very small fraction of the peak processor rating!
Consider the architecture from the previous example. In this case, we add a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions $32 \times 32$. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C.
Impact of Caches: Example (continued)

• The following observations can be made about the problem:
  — 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. For our problem, this corresponds to 64K operations, which can be performed in 64K cycles (or 64 µs)
  — The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200 + 64 µs.
  — This corresponds to a peak computation rate = \( (64K \text{ flop}) / (264 \mu s) \) = or 248 MFLOPS.
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).
Impact of Memory Bandwidth: Example

• Consider the same setup as before, except in this case, the block size is 4 words instead of 1 word. We repeat the vector-add computation in this scenario:
  —Assuming that the vectors are laid out linearly in memory, four additions can be performed in 200 cycles.
  —This is because a single memory access fetches four consecutive words in the vector.
  —This corresponds to a FLOP every 50 ns, for a peak speed of 20 MFLOPS.
Experimental Study of Memory (Membench)

- Microbenchmark for memory system performance

- for array A of length L from 4KB to 8MB by 2x
  for stride s from 4 Bytes (1 word) to L/2 by 2x
  time the following loop
  (repeat many times and average)
  for i from 0 to L by s
  load A[i] from memory (4 Bytes)
Membench: What to Expect

- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most ¼ miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
  - Picture assumes only one level of cache
  - Values have gotten more difficult to measure on modern procs
Memory Hierarchy on a Sun Ultra-2i

Sun Ultra-2i, 333 MHz

L1: 16 KB, 2 cycles (6 ns)

L2: 2 MB, 12 cycles (36 ns)

L1: 16 KB

L2: 64 byte line

Mem: 396 ns (132 cycles)

8 K pages, 32 TLB entries

See [www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps](http://www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps) for details
Memory Hierarchy on a Pentium III

Katmai processor on Millennium, 550 MHz

Array size

- 4KB
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1MB
- 2MB
- 4MB
- 8MB
- 16MB
- 32MB
- 64MB

L1: 32 byte line?

L2: 512 KB
60 ns

L1: 64K
5 ns, 4-way?
Memory System Performance: Summary

• The series of examples presented in this section illustrate the following concepts:
  — 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.
Prefetching and Multithreading Approaches for Hiding Memory Latency

• Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of two possible ways:
  — we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
  — we 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*
A thread is a single stream of control in the flow of a program. We illustrate threads with a simple example:

\[
\text{for } (i = 0; i < n; i++) \\
\quad c[i] = \text{dot_product(} \text{get_row(a, i), b)};
\]

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:

\[
\text{for } (i = 0; i < n; i++) \\
\quad c[i] = \text{create_thread(} \text{dot_product, get_row(a, i), b)};
\]
• In the code, the first instance of this function accesses a pair of vector elements and waits for them.

• In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.

• After $l$ units of time, where $l$ is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.

• In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.
Multithreading for Latency Hiding (contd)

• The execution schedule in the previous example is predicated upon two assumptions: 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.

• Machines such as the HEP, Tera, and Sun T2000 (Niagara-2) rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

• Sun T2000, 64-bit SPARC v9 processor @1200MHz
  — Organization: 8 cores, 4 strands per core, 8KB Data cache and 16KB Instruction cache per core, L2 cache: unified 12-way 3MB, RAM: 32GB
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!
**Stanza Triad**

- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- **Stanza (L)** is the length of a unit stride run

```
while i < arraylength
    for each L element stanza
        A[i] = scalar * X[i] + Y[i]
    skip k elements
```

Source: Kamil et al, MSP05
Stanza Triad Results

- This graph (x-axis) starts at a cache line size (>=16 Bytes)
- If cache locality was the only thing that mattered, we would expect
  —Flat lines equal to measured memory peak bandwidth (STREAM) as on Pentium3
- Prefetching gets the next cache line (pipelining) while using the current one
  —This does not “kick in” immediately, so performance depends on L
Tradeoffs in Multithreading and Prefetching

- Multithreading and prefetching are critically impacted by the memory bandwidth. Consider the following example:
  - Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB.
  - If the computation makes one data request in every cycle of 1 ns, you may notice that the first scenario requires 400MB/s of memory bandwidth and the second, 3GB/s.
Tradeoffs in 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 of Today’s Lecture

- Section 2.3: Dichotomy of Parallel Computing Platforms
- Section 2.2: Limitations of Memory System Performance

Reading List for Next Lecture
- Sections 2.4, 2.5