CSE 30321 – Computer Architecture I – Fall 2010
Lab 05 –Introduction to Multi-core Processors and Parallel Programming
Assigned: November 16, 2010 **– Due:** December 2, 2010 **– Points:** 100

1. Introduction:
This lab will introduce you to how multi-core computer architectures can impact system-level performance, as well as conventional "software engineering." More specifically, we will work with a recursive mergesort algorithm – where a dataset will be sorted on both single and N-core machines. You will see that by understanding what the underlying (multi-core) architecture looks like, it is possible to write more efficient software. You will also see that performance evaluation techniques learned in previous lectures / used in previous assignments are just as relevant to multi-core processing too.

2. Background – Introduction to and Applications for Multi-Core Computing:
(Source: Intel white paper "From a Few Cores to Many: A Tera-scale Computing Research Overview)

(i) The Basics:
Intel processors with two and four cores are here now, and eight-core processors are right around the corner. In the coming years, the number of cores on a chip will continue to grow, launching an era of vastly more powerful computers.

At the highest-level, the motivation for this switch is because incremental improvements in performance and capabilities simply won't support things like:
- Real-time data mining across teraflops of data
- Artificial intelligence (AI) for smarter cars and appliances
- Virtual reality (VR) for modeling, visualization, physics simulation, and medical training
- Other applications that are still on the edge of being science fiction.

Also, data stores are becoming larger and more complex. In medicine, a full-body medical scan already contains terabytes of information. Even at home, people are generating large amounts of data, including hundreds of hours of video, thousands of documents, and tens of thousands of digital photos that need to be indexed and searched. Tera-scale[1] computing is the way to bring the massive compute capabilities of supercomputers to everyday devices, from servers, to desktops, to laptops.

For example, with a tera-scale computer, you could create studio quality, photo-realistic 3-D graphics in real time. Or you could manage personal media better by automatically analyzing, tagging, and sorting snapshots and home videos. Advanced algorithms could be used to improve the quality of movies captured on older, low-resolution video cameras. An advanced digital health application might assess a patient's health by interpreting huge volumes of data in a scan and aid in making decisions in real time.

The essential aspect of tera-scale technologies—and the heart of Intel's research—is being able to do such complex calculations in real-time, **primarily through the execution of multiple tasks in parallel**. That is the fundamental requirement for the complex and compelling applications we will see in the future.

(ii)
Question: *Why are we going to multi-core chips to find performance?*
Answer: *Because we have to.*

---
[1] The term itself—tera-scale—refers to the terabytes of data that must be handled by platforms capable of teraflops of computing performance. That's a thousand times more compute capability than is available in today's giga-scale devices.

In the last twenty years, Intel has delivered dramatic performance gains by increasing the frequency of its processors, from 5 MHz to more than 3 GHz, while at the same time, improving IPC (instructions per cycle[2]). Recently, power-thermal issues—such as dissipating heat from increasingly densely packed transistors—have begun to limit the rate at which processor frequency can also be increased. Although frequency increases have been a design staple for the last 20 years, the next 20 years will require a new approach. Basically, industry needs to develop improved microarchitectures at a faster rate, and in coordination with each new silicon manufacturing process, from 45 nm, to 32 nm, and beyond.

For this new approach we can take advantage of Moore's law. Transistor feature size is expected to continue to be reduced at a rate similar to that in the past. For example, a 0.7x reduction in linear dimensions enables a 2.0x increase in the transistor density. Thus, we should assume that with every process generation, we will be able to build chips with twice the number of transistors as on the previous process generation. New technologies, such as 3-D die stacking, may allow even greater increases in total transistor counts within a given footprint, beyond the increases made possible by improvements in lithography alone.

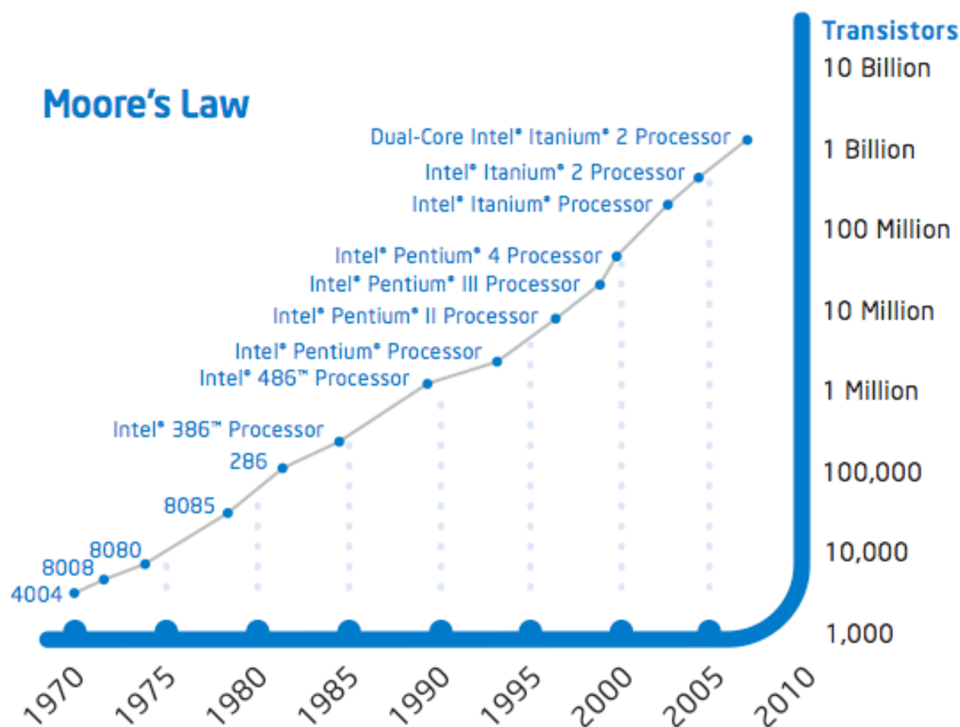

Figure 1. Scaling transistors. The number of transistors is expected to continue to double about every two years, in accordance with Moore's Law. Over time, the number of additional transistors will allow designers to increase the number of cores per chip.

Since Moore's law is expected to continue to deliver more transistors every process generation, and since platform power and energy budgets will be increasingly limited, the trend is to deliver increased performance through parallel computing. Essentially, to achieve the desired improvements in performance without a corresponding increase in energy bills, we must increase the efficiency and

[2] Recall that this is simply the inverse of CPI.

number of cores on a chip, rather than increase clock frequency.

With so many transistors available, we have already began to design chips with multiple processor cores (also called CMP or chip-level multiprocessing). Instead of focusing solely on performing individual tasks faster, we will execute many more tasks in parallel at the same time. We will also distribute those tasks across a grouping of cores that work in a coordinated fashion.

Many simple cores can be built within the same area as a small number of large complex cores. In addition, power consumption can be optimized by using multiple types of cores tuned to match the needs of different usage models. Also, cores that are not busy can be powered down to reduce power consumption during idle times. These advanced power-saving techniques are enabled by multiple cores working in a coordinated fashion.

(iii) Impact on Software:

Intel has identified several key attributes that will be required of future tera-scale platforms:

- Programmability. Without optimized software, tera-scale platforms will not live up to their potential. Platforms must effectively address the needs of new and existing programming models. This also includes software development/debug tools, as well as new performance benchmarks consistent with highly parallel execution.

- Adaptability. The platform must be able to change configuration to match varied usage and workloads, as well as adapt to changes in the hardware environment, such as from power and thermal factors.

- Reliability. The platform must preserve current levels of reliability or increase its reliability despite the increased complexity inherent in these platforms.

- Trust. The platform must provide a trustworthy environment, despite its flexibility and the complexity of its design.

- Scalability. The platform must deliver performance that increases in proportion to the number of cores, with hardware and software that also effectively scales.

More will be said about programmability issues in future assignments. However, if you are interested in learning more about how multi-core chips might impact software *now*, please see Sec. 2.3 of the Intel white paper linked on the course website.

3. Performance in Multi-Core Chips:

In this question, you're going to leverage techniques that you've learned so far in class to quantitatively see how a multi-core computer architecture might improve overall performance (i.e. decrease execution time). We'll keep the discussion pretty simple for now…

Given the above context, assume that we want to compare 2 designs – each with its own execution model:

- Design 1 is a single-core machine with a 4 GHz clock rate.
- Design 2 is a dual-core machine with a clock rate that is 20% slower.

Assume that we are interested in how long it will take to execute all of the instructions associated with 2 processes on each design.

You know the following:

- Process 1 requires 2.5 million MIPS instructions
- Process 2 requires 6 million MIPS instructions
- In the tables below, I've listed the number of CCs each instruction "class" requires. Note that the number of CCs per class differs from design-to-design. The percentage of each instruction class per process is also listed.

Instruction Type	% (Process 1)	% (Process 2)
ALU	45%	65%
Store	12%	5%
Load	22%	15%
Branch/Jump	21%	15%

Instruction Type	CCs on Design 1	CCs on Design 2
ALU	4	4
Store	4	5
Load	5	6
Branch/Jump	3	3

(Note difference in shaded boxes)

On Design 1, Process 1 will be executed first, there will be a context switch (where we update the register file with the data for Process 2, etc. that will take 100,000 CCs), and then Process 2 will run until completion. On Design 2, each process can be mapped to a different core so there is no context switch overhead.

What performance improvement do we get by executing the instructions for these two processes on the dual core machine?

Solution:

CPU Time – Design 1, Process 1:

= 2.5M Instructions **x** [(0.45)(4) + (0.12)(4) + (0.22)(5) + (0.21)(3)] CCs / Inst **x** 0.25×10^{-9}s / CC

= 0.002506 s

Overhead:

= 100,000 CCs **x** 0.25×10^{-9}s / CC

= 0.000025 s

CPU Time – Design 1, Process 2:

= 6M Instructions **x** [(0.65)(4) + (0.05)(4) + (0.15)(5) + (0.15)(3)] CCs / Inst **x** 0.25×10^{-9}s / CC

= 0.006 s

Total: ~0.0085 s

CPU Time – Design 2, Process 1:

= 2.5M Instructions **x** [(0.45)(4) + (0.12)(5) + (0.22)(6) + (0.21)(3)] CCs / Inst **x** 0.313×10^{-9}s / CC

= 0.0034 s

CPU Time – Design 2, Process 2:

= 6M Instructions **x** [(0.65)(4) + (0.05)(5) + (0.15)(6) + (0.15)(3)] CCs / Inst **x** 0.313×10^{-9}s / CC

= 0.007875 s

Total: ~0.007875 s (b/c the processes run in parallel)

Therefore, 0.0085 / 0.00785 ~ 1.08 (therefore Design 2 is about 8% faster)

4. A Real Life Example:

The focus of my research is on computational devices beyond the transistor – which may ultimately be limited by physics, cost, and manufacturing-related issues. My research efforts have primarily targeted computational systems where magnetic elements with nanometer feature sizes are used to both process and store binary information.

Why is this good? In conventional electronics, most computation is charge-based. A power supply maintains state, and thousands of electrons (*each* dissipating ~1.66×10^{-19} J of energy) are needed to perform a single function (when you consider a chip with a billions transistors on it, this adds up quickly!). Alternatively, nanomagnet logic (NML) devices can process information in a cellular-automata like architecture, could dissipate less than 1.66×10^{-19} J per switching event for a *gate* operation, will retain state without power, and are intrinsically radiation hard.

Thus, looking up to applications, NML has the potential to mitigate increasing chip-level power densities that are currently exacerbated by device scaling, could help to improve battery life in mobile information processing systems, and may operate in environments where transistor-based logic and memory cannot.

To design and study magnetic logic systems, both my graduate students and I do lots and lots of physical-level simulations. In particular, we use a simulation suite called "OOMMF" – or the "Objected Oriented MicroMagnetic Framework". OOMMF simulations give us an idea of how an ensemble of magnets (which forms a magnetic logic gate or circuit) might perform when "clocked" after being subjected to new inputs. These simulations are critical for experimental design (there is excellent correlation between simulation results and experimental results), as well as for projecting how well NML might ultimately perform and scale.

The current version of OOMMF used by our group is multi-threaded, and the user can specify the number of threads that a given simulation will use – i.e. 8 cores could be used to simulate 1 magnetic ensemble, or 8 cores could be used to simulate 8 different magnet ensembles simultaneously. As you might expect, presumably the single threaded simulations would have longer execution times.

While the correlation between simulation and experimental results is strong, one drawback to this simulation-based approach to design is that the simulations themselves can be quite time consuming (i.e. 2-3 days are sometimes needed to see how just a 10-15 magnet ensemble will respond to just 1 input). Moreover, because of the rather complex magnetization dynamics, any changes to the material system being studied, magnet geometries, etc. necessitate an entirely new simulation.

At present, there are 7 graduate students in my research group. Four students (+ two faculty members) do micromagnetic simulations. Thus, it is not uncommon for the group to have between 300-500 simulations running simultaneously.

Not surprisingly, we need significant computational resources to manage this simulation overhead. We currently use machines managed by Notre Dame's Center for Research Computing (CRC). While access to these machines is good, there are times when our jobs will wait for 24-48 hours in a queue (as other research groups also have access to these resources).

There are times when we would like to have instantaneous access to computing resources (e.g. to meet a paper or proposal deadline, support experimental work, etc. As such, as part of a recent proposal to the Department of Defense to investigate magnetic logic systems, we included a line item in our budget that would allow for the purchase of compute nodes that are managed by the CRC, but where our research group would receive priority scheduling on said nodes[3].

[3] i.e. if we are not running jobs, other users' jobs can run on these nodes; however, if we submit jobs, our jobs will be scheduled and the jobs running on our nodes will be returned to the queue.

Presently, there are two compute server configurations that are available for purchase:

1. A server with 2, AMD 6134 chips (8 cores/chip @ 2.3GHz) and 24GB DDR3 RAM at $1,746.37
2. A server with 2, Intel E5620 chips (4 cores/chip @ 2.4GHz) and 24GB DDR3 RAM at $1,848.83

Given only a cursory glance, the choice of server would be somewhat obvious:

- The server with an AMD chipset offers 2X more cores that the server with an Intel chipset
- The server with an AMD chipset has a marginally lower clock rate than the server with an Intel chipset
- The server with an AMD chipset has the same amount and type of memory as the server with an Intel chipset

However, after consulting with CRC personnel, we learned that code used by certain research groups can perform significantly better on the Intel serves (even given the reduced number of cores). To determine which compute server best meets our needs, the CRC recommended running a set of representative simulations on different servers – where different compilers were also used to compile the simulation engine.

The results of the test simulations are summarized in the table below. Note that in all cases, jobs were submitted to maximize CPU usage. Thus, for the 4-core chip Intel server (with two chips), 8 single core jobs were submitted simultaneously. Similarly, for the 8-core chip AMD server (with two chips), 16 single core jobs were submitted simultaneously. (This was done to ensure that there was not an imbalance regarding main memory usage.)

	Server Type	Intel - 8 Total Cores		AMD - 16 Total Cores	
	# Cores used per job	1	4	1	4
g++ Compiler	Average time per job (Hours:Minutes:Seconds)	3:45:53	1:21:55	4:41:00	2:06:51
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:31:45	10:55:20	4:41:00	8:27:25
Intel Compiler (icpc)	Average time per job (Hours:Minutes:Seconds)	3:47:00	1:18:30	4:34:38	1:51:30
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:34:00	10:28:00	4:34:38	7:26:00
PGI Compiler (pgCC)	Average time per job (Hours:Minutes:Seconds)	6:44:37	1:44:00	5:40:45	2:31:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	13:29:15	13:52:00	5:40:45	10:05:00
open64 Compiler (openCC)	Average time per job (Hours:Minutes:Seconds)	3:50:23	1:22:00	5:10:37	2:01:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:40:45	10:56:00	5:10:37	8:05:00

Given our typical usage patterns, a $25,000 budget, and the statistics summarized in the above table answer the questions below. You should justify your answer quantitatively:

- What servers (chipset + number of servers) would you recommend buying and why?
- Does the choice of compiler have a significant impact on your results?

A Solution:

	Server Type	Intel - 8 Total Cores		AMD - 16 Total Cores	
	# Cores used per job	1	4	1	4
g++ Compiler	Average time per job (Hours:Minutes:Seconds)	3:45:53	1:21:55	4:41:00	2:06:51
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:31:45	10:55:20	4:41:00	8:27:25
	%CPU usage	100%	69%	100%	55%
	(Average Intel)/(Average AMD)	80%	65%		
Intel Compiler (icpc)	Average time per job (Hours:Minutes:Seconds)	3:47:00	1:18:30	4:34:38	1:51:30
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:34:00	10:28:00	4:34:38	7:26:00
	%CPU usage	100%	72%	100%	62%
	(Average Intel)/(Average AMD)	83%	70%		
PGI Compiler (pgCC)	Average time per job (Hours:Minutes:Seconds)	6:44:37	1:44:00	5:40:45	2:31:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	13:29:15	13:52:00	5:40:45	10:05:00
	%CPU usage	100%	97%	100%	56%
	(Average Intel)/(Average AMD)	119%	69%		
open64 Compiler (openCC)	Average time per job (Hours:Minutes:Seconds)	3:50:23	1:22:00	5:10:37	2:01:15
	Time to complete 16 identical jobs on each server (Hours:Minutes:Seconds)	7:40:45	10:56:00	5:10:37	8:05:00
	%CPU usage	100%	70%	100%	64%
	(Average Intel)/(Average AMD)	74%	68%		

(Average Intel)/(Average AMD) needs to be < 50% before Intel machines become more cost effective per core

Throughput is significantly better when running single core jobs

Compiler did not have great influence over results

When running a Single Simulation

- When running 1 simulation and using the same number of cores, the Intel machine takes roughly 70% of the time it takes the AMD machine.
- When running 1 simulation, running with 4 cores instead of 1 core allows the job to finish 2.7 times faster.

So, when trying to maximize our speed on *single* simulations, the Intel machine will provide better performance.

When running Multiple Simulations

However, if we want to run a very large number of jobs, and expect to completely fill all of our servers, the performance gain shifts to AMD.

When running 16 single core simulations, the AMD machine takes 60% of the time the Intel machine requires.

When running 16 4 core simulations, the AMD machine takes 77% of the time the Intel machine requires.

When comparing 16 single core jobs to 16 4 core jobs, the single core jobs complete the task in roughly 60% of the time it takes the 4 core jobs. (Note, this is assuming that we have more jobs than can be fit on all of our machines)

Summary

Because we are often looking at running massive numbers of simulations, I think the AMD machines are probably better suited to our tasks. If we think that we will have some ultra-long jobs that we only want to run once, we could look into getting an Intel machine. The compiler did not make a major difference with the Intel compiler slightly outperforming the g++ compiler.

5. **Parallel Mergesort:**

A challenge of multi-core computing is figuring out how to best make use of the multiple processors cores. It is not enough to simply split a program into N equal chunks for N processors. The separate sub-problems solved by each CPU core usually must communicate with each other to coordinate their work and produce the final solution. Though the product of the number of cores, and the computational power per core, may be much greater for a multicore than a single core of the same size, this inter-core communication is often very costly and can affect the design of the algorithms involved.

We will look at a very simple application of a multicore microprocessor, to solve a sorting problem with a modified mergesort algorithm. Essentially, this parallel mergesort will (1) split the input into equally-sized chunks for each core, and then (2) after each core sorts its partial list, these lists will be combined (merged) into the final answer.

Roughly, parallel mergesort looks like this:

```
Parallel_Mergesort(S)
{
      S1, S2, ..., Sn   = Partition(S, number_of_cores)

      for i = 1 to n:
                 Dispatch_Core(i, Si)

      Wait_for_Cores_to_Finish()

      Sorted_S1, Sorted_S2, ..., Sorted_Sn = Receive_From_Cores()

      return merge(Sorted_S1, Sorted_S2, ..., Sorted_Sn)
}

Core_Mergesort(sublist)
{
      if (length(sublist == 1)) : return sublist

      S1, S2 = Split(sublist)

      S1 = Core_Mergesort(S1)
      S2 = Core_Mergesort(S2)

      return Merge(S1, S2)
}
```

(Note that Core_Mergesort is essentially the same function that you wrote in Lab 02 – and that code would be part of a multi-core implementation.)

For simplicity, presume the multi-core machines can do everything in parallel until the final merge step. All of the CPUs have the exact same instruction set and have the same rate of execution for each core, and the clock rates for all machines/cores are equal. However, the time to transfer the sorted sublists back to the master routine is given in clock cycles per list element – and does depend on the number of cores.

We'll consider 3 CPU designs:

- CPU 1:
  - Single core.
- CPU 2:
  - 2 cores.
  - Time to message between CPUs: 70 CC / element
- CPU 3:
  - 4 cores.
  - Time to message between CPUs: 120 CC / element

Additionally (so you don't have to extract data from your code), you can simply assume that on each core:

- the Split(L) function takes 2 CC per element in L
- the Merge(L1, L2) function takes 15 CC per element in L1 or L2.

Finally, recall that the depth of recursion for a mergesort will be logarithmic (base 2) – specifically, for a power-of-two input size 2^N, there will be N+1 levels of recursion in `Core_Mergesort` (since the recursion bottoms out at N = 0, not 1.

Find the fastest CPU for each of the following input sizes: 64, 256, 2048 elements. Explain why you believe a given CPU is best for a each of the above data sizes. What do these results suggest if you're writing / compiling code that can be parallelized?

Solution:
In general, running time of Core_Mergesort is $(\log_2(N) + 1) * N *(15+2)$.

Single core (no overhead for message-passing):

- Input 64:
  - Core_Mergesort takes: 7 x 17 x 64 = **7616 CC**
- Input 256:
  - Core_Mergesort takes: 9 x 17 x 256 = 39168 CC
- Input 2048:
  - Core_Mergesort takes: 12 * 17 x 2048 = 417792 CC

Dual Core:

- Input 64:
  - Core_Mergesort (N = 32) takes: 3264 CC
  - Message-passing takes 70 CCs x 64: + 4480 CC
  - Total: 7744 CC
- Input 256:
  - Core_Mergesort (N = 128) takes: 17408 CC
  - Message-passing takes 70 CCs x 256: + 17920 CC
  - Total: **35328 CC**
- Input 2048:
  - Core_Mergesort (N = 1024) takes: 191488 CC
  - Message-passing takes 70 CC x 2048: + 143360 CC
  - Total: 334848 CC

Quad Core:

- Input 64:
  - Core_Mergesort (N = 16) takes: 1360 CC
  - MP takes 120 CC x 64: + 7680 CC
  - Total: 9040 CC
- Input 256:
  - Core_Mergesort (N = 64) takes: 7616 CC
  - MP takes 120 CC x 256: + 30720 CC
  - Total: 38336 CC
- Input 2048:
  - Core_Mergesort (N = 512) takes: 87040 CC
  - MP takes 120 CC x 2048: + 245760 CC
  - Total: **332800 CC**

Summary: Single-core wins at N = 64, dual-core wins at N = 256, quad-core wins (barely) at N = 2048. Students should note that, in general, with larger data sets the parallelism pays off more significantly because the linear-time cost to communicate between cores can be amortized over the O((N/k) log (N/k)) time of the `mergesort` on each core.

6. What to turn in:

A **typed** lab report that contains the following:

- An answer (with quantitative justification) to the question in Section 3-5.