CSE 30321 – Computer Architecture I – Fall 2009
Lab 06 –Introduction to Multi-core Processors and Parallel Programming
Assigned: November 3, 2009 **– Due:** November 17, 2009

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.

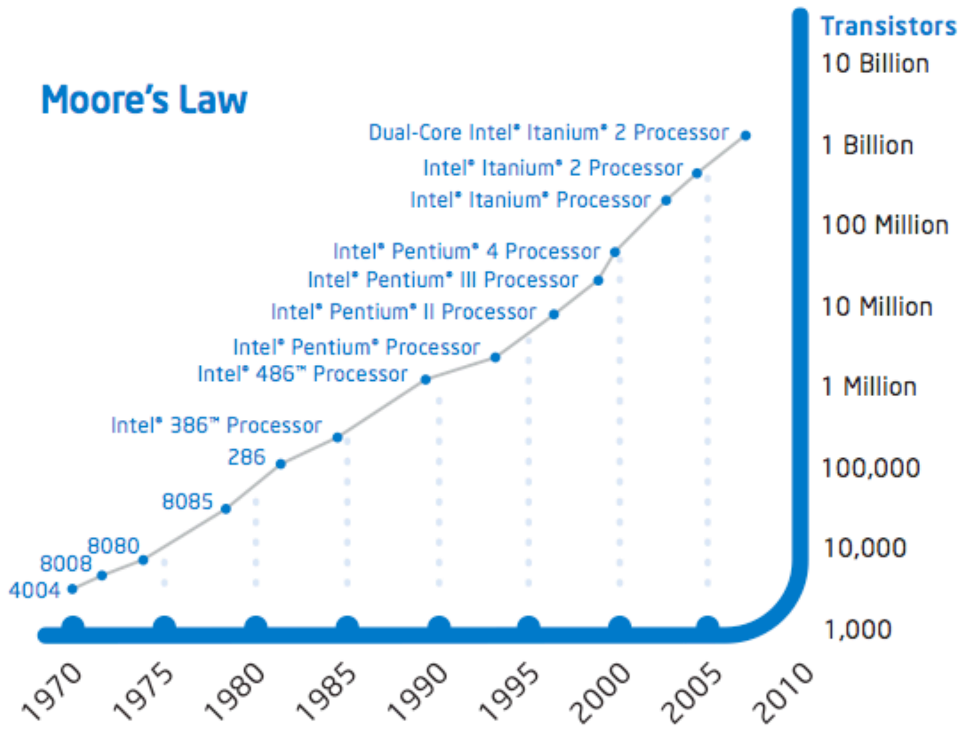

Figure 1. Scaling transistors. The number of transistors is expected to continue to double about every two years, in accordance with Moore's Law. Over time, the number of additional transistors will allow designers to increase the number of cores per chip.

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. <u>Performance</u> <u>in</u> <u>Multi-Core</u> <u>Chips</u>:

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, we list the number of CCs each instruction type requires on each design, as well as the percentage of each instruction type:

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.

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

4. Mergesort:

The material covered thus far in CSE 30321 – e.g. on datapath design – is just as relevant in the multi-core regime too. Code written in a high-level language still must be translated to assembly instructions, and there can be multiple (identical) datapaths on the same chip. In the Section 5, we'll work with a parallel version of mergesort – that will still be based on a "core" mergesort function like that described below.

The code below performs a `mergesort` of an array that has a power-of-two size. We assume that we have an external routine `merge()` that merges two sorted arrays into one sorted array of twice the size; this is for simplicity (a real `mergesort` needs to handle boundary cases in the merge loop that would be too complicated here). Translate the code into MIPS assembly, with stack handling appropriate for the MIPS calling convention.

```
void mergesort(int *array, int size)
{
      if (size == 1) return;                /* recursive base-case */

      mergesort(&array[0], size/2);
      mergesort(&array[size/2], size/2);

      /* assume that merge() can do an in-place operation,
         writing the merged list back into array[] */

      /* note that you do not have to write the merge() function! */
      /* you might just have a line that says "jal merge" */
      merge(&array[0], &array[size/2], &array[0]);
}
```

5. Parallel Mergesort:

The challenge of multicore is to make use of the multiple processors: it is not enough to simply split a program into N equal chunks for N processors. The separate subproblems 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 the sorting problem with a modified mergesort algorithm. Essentially, this parallel mergesort will split the input into equally-sized chunks for each core, and then after each core sorts its partial list, these lists are combined (merged) into the final answer.

Roughly, parallel mergesort looks like this:

```
Parallel_Mergesort(S)
{
      $S_1$, $S_2$, ..., $S_n$   = Partition(S, number_of_cores)

      for i = 1 to n:
                 Dispatch_Core(i, $S_i$)

      Wait_for_Cores_to_Finish()

      Sorted_$S_1$, Sorted_$S_2$, ..., Sorted_$S_n$ = Receive_From_Cores()

      return merge(Sorted_$S_1$, Sorted_$S_2$, ..., Sorted_$S_n$)
}

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

      $S_1$, $S_2$ = Split(sublist)

      $S_1$ = Core_Mergesort($S_1$)
      $S_2$ = Core_Mergesort($S_2$)

      return Merge($S_1$, $S_2$)
}
```

(Note that Core_Mergesort is essentially the same function that you wrote in Section 4 – and this 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?

6. What to turn in:

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

- An answer (with quantitative justification) to the question in Section 3.
- The MIPS code for mergesort
- An answer (with quantitative justification) to the question in Section 5.
- A group evaluation sheet (see link on course website).

Notes:

- To preserve anonymity (if needed), group evaluation sheets should be (a) turned in individually, and (b) folded in half with your name on the outside of the form. These forms will not be returned.
- Also, please submit your MIPS code and a PDF of your lab report in the dropbox of one of your group members.