Using the Three Multiprocessor Servers from the 2009 Sun Microsystems Grant

Dale E. Parson, http://faculty.kutztown.edu/parson, Summer 2010

Version 2, January 2012

I. Introduction

Acknowledgements January 2012: This originally unplanned extension of the summer 2010 report, completed during winter break 2011-2012, has been made possible by a *Parallelism in the Classroom* Microgrant from **Intel Corporation**. The updates to the benchmarks reported in this document represent one component of the work completed thanks to the Intel grant. Other curricular components relating to the CSC 580 Special Topics course in Multiprocessor Programming from spring 2011, and the CSC 480 Special Topics course in Multiprocessor Programming from spring 2012, have benefited from the Intel grant as well. They are reported in separate documents, since they were not parts of the summer 2010 study. In addition, thanks go to the **NVIDIA Corporation** for the donation of a C2070 Tesla GPU (graphical processing unit) card to our department. I will be using that card housed within an Intel-based multicore PC purchased by Kutztown University to perform algorithm research reported outside the scope of this report.

All code and documentation reported herein will be contributed to Intel's Academic Communities' Educational Exchange under the Creative Commons 3 open-source copyright.

The major addition in this January 2012 revision of this report is the inclusion of measurements from an Intel-based multiprocessor PC in the benchmark reports. Subsections tagged with **January 2012** contain most of the added information.

Start of Original Report

This document is an outcome of a study that I conducted over the 2009-2010 academic year and the summer of 2010, with the summer work funded by a PASSHE Faculty Professional Development Grant, in order to determine ways to integrate the three multiprocessor servers that we received from Sun Microsystems at the end of summer 2009 into the computer science curriculum at Kutztown University.

The first activity planned for this document is to hold a **one-hour seminar for the Kutztown University Computer Science Faculty and appropriate members of the Kutztown IT staff** in order to summarize the results reported in this document. The hope is to make these machines useful across relevant portions of our curriculum. We will have this seminar early in the **fall 2010** semester after distributing this document.

The second activity is my planned offering of **CSC 580, Special Topics: Algorithms and Data Structures for Multiprocessing**, for graduate students in **spring 2011**. The Computer Science

Department has approved the syllabus for this course, attached as Appendix A. The current study supplies example software and starting points for lab exercises for this course.

There are also opportunities for using these machines for faculty and student research projects and for a new undergraduate course in parallel programming that is being planned. Professor Schaper is currently drafting a syllabus for that course.

The outline of the remainder of this document follows: II Three Multiprocessor Servers, III Textbooks, IV Code Examples and a Larger Planned Application, V External Benchmarks and VI References. Appendix A is the approved syllabus for the CSC580 special topics course mentioned above. Appendix B gives some compiler and command line utility documentation for using the multiprocessor machines.

File multip.final.zip holds the final version of this report as well as all of the code examples of Section IV. There are copies of this zip file under p:\Parson on the departmental P drive and in directory ~parson/multip on bill and the NFS-connected multiprocessor servers.

Acknowledgements: The Computer Science Department appreciates the generosity of Sun Microsystems in granting these server machines to us as the result of a proposal submitted by the author to Sun's "Change Your World" grant program in the summer of 2009. The author acknowledges the PASSHE Faculty Professional Development Committee for funding a stipend for the summer of 2010 during which most of the reported work was performed. Finally, Chris Walck of Kutztown's IT organization has been extremely helpful in getting and keeping these machines on the air, in installing software, in finding documentation, and generally in making this work possible.

II. The Four Multiprocessor Servers

Table I is a summary of the specifications of the three multiprocessor servers provided by Sun [26] and the fourth, Intel-based PC purchased by Kutztown University. All four servers are 64-bit architectures. This section examines the processing and memory architectures of these machines. Currently Harry, Hermione and Ron are available for computer science faculty and students within the Kutztown University network, and Dumbledore is available in the Old Main Computer Science lab room. Hermione came with Solaris 10 installed but has moved to Linux on a permanent basis. Harry and Hermione have one floating point unit per core. Ron has one floating point unit for the entire UltraSparc T1 multiprocessor. Dumbledore, officially named CSD-10 on the Kutztown University network, is running 64-bit Windows 7, and must be accessed from within the lab room. It houses the NVIDIA C2070 Tesla GPU card and accompanying CUDA software tools.

name	arch	cores	threads / core	total threads	clock speed	memory	cache	os
Harry	UltraSparc T2, T5120 server	8	8	64	1.2 Ghz	16 GB	16kb icache, 8kb dcache / core, 4 MB L2 cache (8 banks, 16 way)	Solaris
Hermione	AMD Opteron 885, x64, X4600 server	8	2	16	2.7 Ghz	32 GB	128kb / core, 1 mb L2 per core.	Linux
Ron	UltraSparc T1, T1000 server	8	4	32	1 Ghz	8 GB	16kb icache, 8kb dcache / core, 3 MB L2 cache (4 banks, 12 way)	Solaris
Dumbledore	Xeon[1] x5687 x 2	8	2	16	3.6 Ghz	16 GB	12 MB x 2	Windows 7

Table I: Overview of the specifications for the three server machines

[1] There are two x5687 processor packages with four dual-threaded cores in each, yielding 8 cores x 2 threads = 16 threads of execution. Each of the two processor packages has 12 Mbyte of L2 cache. All C++ benchmarks are run under Cygwin using g++ for compilation.

Figure 1 is a block diagram of the UltraSparc T2 processor architecture found in Harry [4]. An 8 x 9 crossbar switch fully interconnects each of the 8 cores to the 8 banks of L2 cache + the system interface unit for I/O. In addition to 8 hardware threading units and 2 integer execution units per core, there is also one floating point unit (FPU) and one stream processing unit for cryptography per core. As indicated by Table 1, the L2 cache is partitioned into 8 banks with 16-way access, shared by all cores across the crossbar of Figure 1.

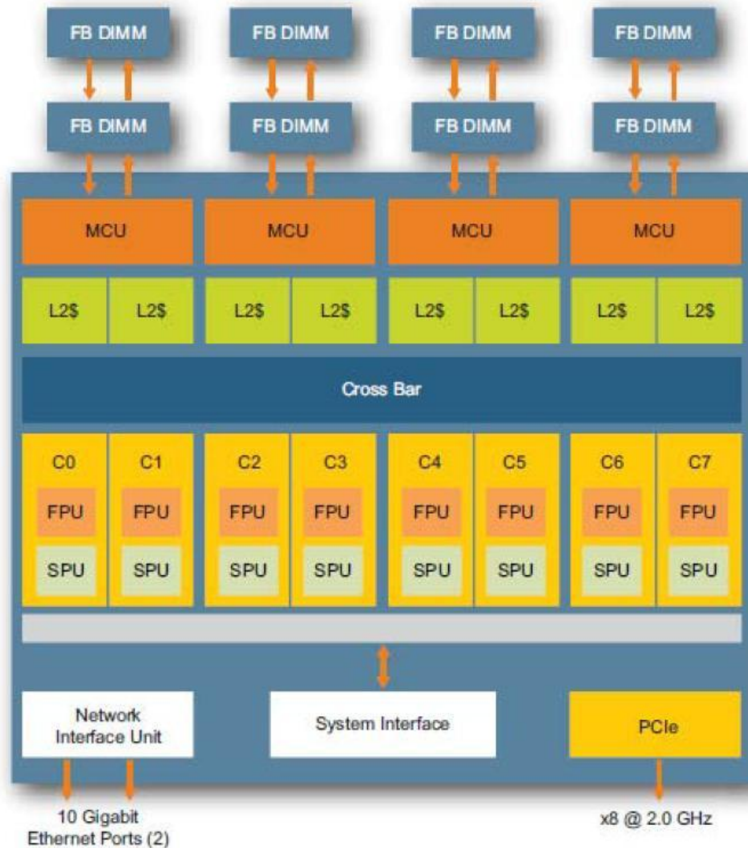


Figure 1: The UltraSparc T2 Processor Architecture in Harry

Each core also contains its own memory management unit (MMU) for mapping between virtual and physical memory address spaces. Each hardware thread can execute a distinct process in its own virtual memory space. Alternatively, multiple hardware threads up to all hardware threads in the server can execute within the virtual address space of a single process. This flexible memory address organization is true of all three servers of Table 1. The MMU in the T2 of Figure 1 supports page sizes of 8 Kb, 64 Kb, 4 Mb and 256 Mb.

Figure 2 gives an abstracted view of the execution timing for hardware threads executing in core 1; the source document duplicates this timing diagram identically for cores 2 through 8 to highlight the thread parallelism in the T2 processor [4]. The temporal intervals colored in blue indicate hardware threads that are actually performing computations. Thanks to the presence of

two integer execution units per core, two threads can be actively engaged in computation, while the others are waiting for staged memory accesses to complete. Memory latency is a primary bottleneck in processor systems including multiprocessors. Multithreaded hardware processors are designed to take advantage of this fact by providing work for a subset of the hardware threads, up to 2 per T2 core in Figure 2, while other threads block from necessity waiting for memory access to complete. Memory access, even to fast, intra-core L1 cache and inter-core L2 cache can be considered a form of I/O with respect to speed when compared with intra-core arithmetic/logic/control processing. Adding integer execution units would be fruitless if the memory access architecture cannot source and sink those execution units' data fast enough and cannot keep the hardware threads that drive the execution units filled with instructions.

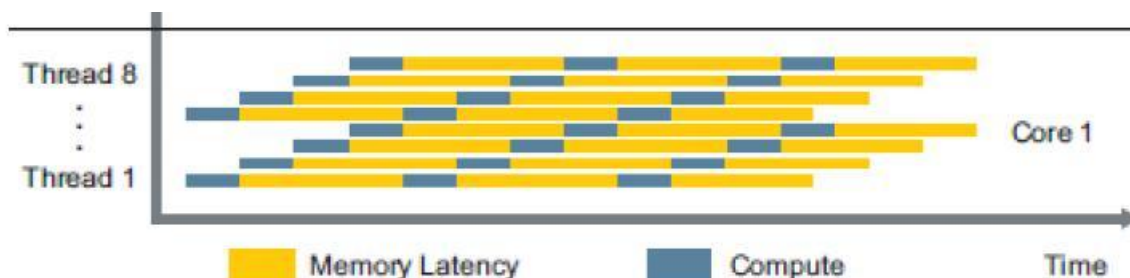


Figure 2: T2 thread parallelism amortizes memory latency across instruction streams

Each **hardware thread** in each of the three servers of Table 1 consists of that subset of processor resources required to control and maintain state for a single instruction stream, including data registers, control registers such as an instruction pointer, and indirection registers such as a stack pointer or frame pointer. A hardware thread does not include ALU and related execution logic. Execution logic is shared among hardware threads, e.g., the two integer and one floating point execution units per 8-threaded core of the UltraSparc T2 of Figure 1.

In addition to **thread parallelism** found in multiple instruction multiple data (MIMD) processors such as the T2, modern processors (in conjunction with optimizing compilers) support **instruction parallelism** in providing multiple operations per opcode (Very Large Instruction Word or "VLIW"), execution pipelines and operation reordering, as well as **data parallelism** such as vector operations in single instruction multiple data (SIMD) processors. The 64-bit register width in these server processors is a form of data parallelism. The stream processing unit (SPU) of Figure 1 is a SIMD processor used mostly for parallel computations applied to cryptography. SIMD processors require identical instruction sequences to operate on multiple data sets stored in arrays and matrices in order to achieve performance gains. Modern graphics processors used for general-purpose computation are primarily SIMD processors [15]. MIMD processors, in contrast, can diverge through different execution paths. Their execution bottlenecks occur in part when threads contend for access to shared memory and when they must synchronize their execution timing to control access to shared data. They operate most effectively when the hardware threads can perform compute-bound work independently of other hardware threads. The code examples of Section IV illustrate execution of MIMD algorithms.

An operating system's **software thread** houses register state similar to a hardware thread, but when a software thread is not currently executing, all of its register state is moved to memory, and that memory can be paged or swapped to disk by the operating system. An operating system's **process** is a collection of one or more software threads within a single, shared address space in virtual memory. The operating systems of all three servers of Table 1 are structured to minimize the costs of mapping a software thread that is ready to execute (i.e., not waiting for I/O, a timer, or another thread) to a hardware thread when one is available. These operating systems encourage the use of multithreaded processes, and as some of the examples of Section IV illustrate, at times it is effective to create more software threads than available hardware threads.

Figure 3 is a block diagram of the dual-core AMD Opteron processor architecture found in Hermione [23]. Unlike the cross-core L2 cache of the T2 of Figure 1, each Opteron core of Figure 3 has its own 1 Mb L2 cache, in addition to an integer and floating point execution unit. Hermione houses 8 of these dual core processors for a total of 16 hardware threads. Like Harry, each thread in Hermione can host the unique virtual address space of a process; alternatively, multiple hardware threads can execute within a single process' virtual address space.

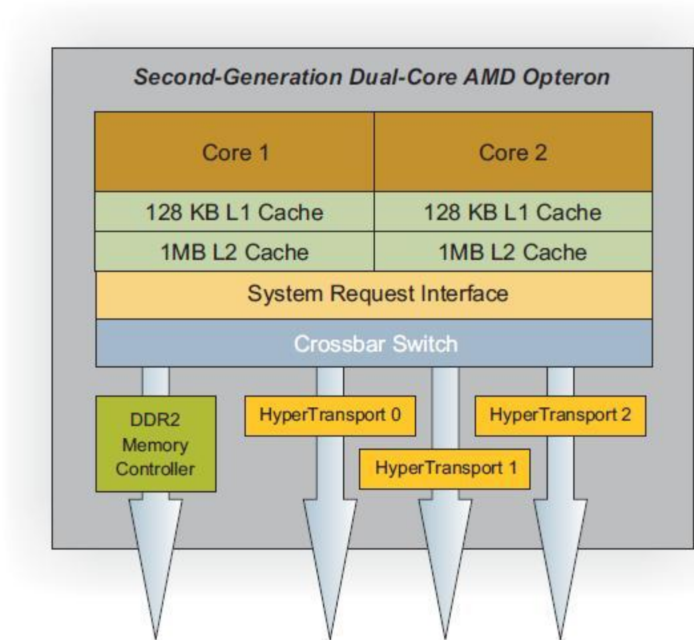


Figure 3: The Dual-core AMD Opteron Processor Architecture in Hermione

Rather than locate crossbar switches within the crossbar as shown in Figure 1 for Harry, Hermione's architecture locates a so-called crossbar switch within each dual-core package as shown in Figure 3. This arrangement leads to less than a full (core X memory bank) crossbar interconnection scheme. Figure 4 shows the multistage interconnection network topologies [9] available for the multicore Opteron processors used in Hermione [23]. With 8 dual-core units,

Hermione uses the upper left configuration of Figure 4. Each dual-core processor communicates with the L2 caches in other cores and with the two main memory ports at the bottom of the diagram via this network; there is only one main memory port for the configuration in the lower right. The 1 Ghz (8 Gbyte per second) "HyperTransport" ports of Figure 3 connect to the multiprocessor network of Figure 4, and the "DDR Memory Controller" is used for those processors that connect the network to main memory. Both the cached multistage interconnection network topology of Hermione and the cached crossbar topology of Harry are labeled Non-Uniform Memory Access (NUMA) access architectures in the specification documents, although this labeling may be incorrect for Harry. See further discussion for Ron below. Neither uses bus-based access to main memory that is typical of uniprocessors.

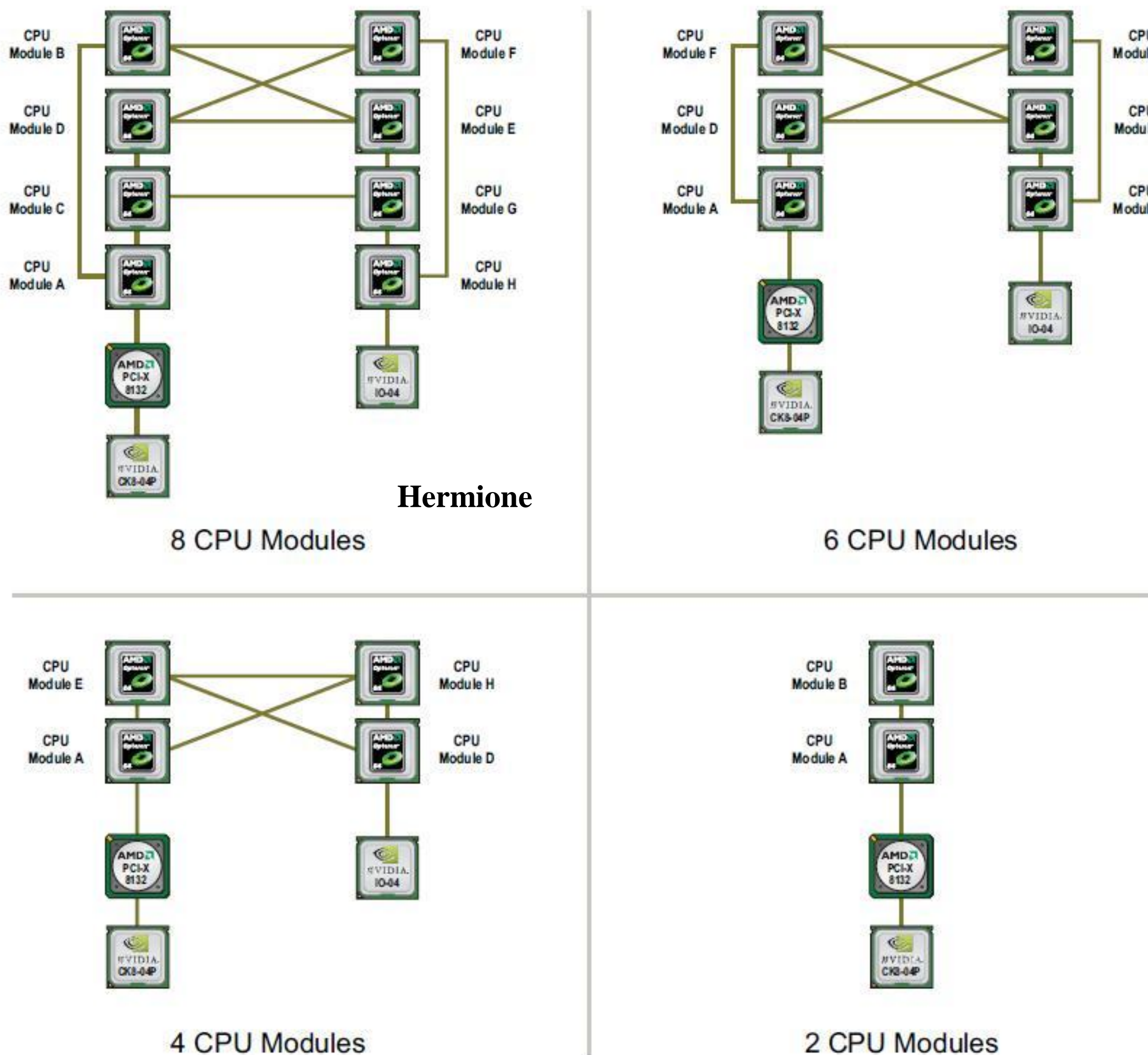


Figure 4: "Enhanced Twisted Ladder" X4600 Multistage Interconnection Topologies

Figure 5 is a block diagram of the UltraSparc T1 processor architecture found in Ron [25]. With only one integer execution unit per core, the concurrency diagram reduces to one half of Figure 2. Ron supports four threads per core, three of which would be waiting for memory response in Figure 2. Figure 5 shows that Ron has only one floating point unit for the entire multiprocessor, making it a poor choice for multithreaded scientific applications requiring floating point support. This deficiency was corrected in T2-based servers. The specification for the T1 labels the crossbar interconnection scheme to be a Uniform Memory Access (UMA) Architecture, in contrast with the NUMA labeling of the T2. It is not clear why the T2 would be considered NUMA. The UMA versus NUMA distinction does not take time variations caused by cache residency versus cache misses into account. Since each core in both Figure 1 and Figure 5 have full crossbar interconnection to shared L2 cache that is identical for each core, it appears that Harry and Ron both have a UMA architecture.

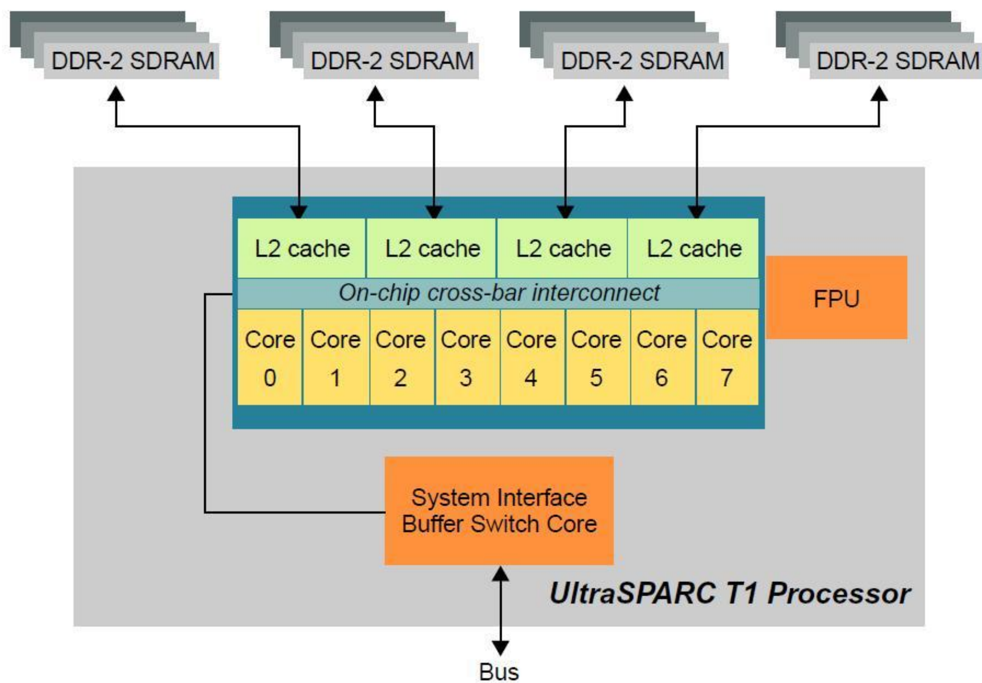


Figure 5: The UltraSparc T1 Processor Architecture in Ron

Like Harry and Hermione, each thread in Ron can host the unique virtual address space of a process; alternatively, multiple hardware threads can execute within a single process' virtual address space.

This completes the architectural overview of the three multiprocessors. Looking at the specifications and diagrams of Table 1 through Figure 5 do not give much insight into how multithreaded software must be written in order to take advantage of these processors. The code examples and benchmarks of sections IV and V help with that task.

III. Textbooks

This section is a very brief overview of the textbooks that I have examined for possible use in the spring 2011 offering of Special Topics in Multiprocessing (CSC 580). This section lists the books in decreasing order of usefulness and applicability to the course.

Java Concurrency in Practice by Goetz, et. al. [8] is probably the book that we will use as the main text in the course. The book's focus is on multithreaded object-oriented software architecture that makes effective use of the java.lang.concurrent library and related libraries. While I had originally intended to focus the course primarily on algorithms and data structures for multiprocessors, rather than taking the software architecture focus of this book, the book's quality both in terms of software design principles and readability make it the best book for the course. I will augment this textbook with materials on algorithms and data structures for multiprocessors from my own work and from other sources below. Much of the emphasis of the book is on effective use of existing Java library algorithms and data structures for multiprocessors, making it very useful for professional Java developers.

C++ Concurrency in Action, Practical Multithreading [27] is due out in January 2012 after several schedule slips. It addresses multithreading language and library support to be added to the impending C++0x standard [2, 3]. Since the book is not now available, review is impossible. I plan to obtain the book and the latest GCC/G++ implementation in support of these features as background material for the course. I hope to have one lab exercise that uses these features of C++.

The Art of Multiprocessor Programming [12] was one book that I had thought to use for the course. Its examples are in Java. Large sections of the book are devoted to building library algorithms and data structures for multithreaded applications. The book has three deficiencies for purposes of the course. First, it spends too many initial chapters considering how to build many flavors of locks and other low level synchronization primitives that are available in even the most basic libraries. Those sections are not practical for the course. Second, I have found several bugs in data structure code that I have reported back to the authors with acknowledgement of only one. Clearly, not all of the code in the book has been thoroughly tested. The bugs I found do not rely on hard-to-duplicate timing conditions. They are simple bugs. Third, there is no substantial treatment of the practical application of these algorithms and data structures to applications. That is the strong point of the Java book by Goetz, et. al. I will use this book to augment my own background in concurrent data structures and algorithms for the course.

Introduction to Parallel Computing, Second Edition [9] is a good all-around book for examining processor architectures, low-level software issues, high-level APIs, problem decomposition techniques, formal analysis and benchmarks for all varieties of multiprocessors. It is too broad for what I have in mind for the course, but it is a useful source of background material.

Programming Massively Parallel Processors [15] is a how-to guide for programming Graphics Processing Units (GPUs) for general purpose SIMD applications [5, 6]. I have only skimmed the

book. It looks useful for GPU SIMD programming. I hope to incorporate one GPU project into the course, and will use this book to build up my background.

Patterson and Hennessy's *Computer Organization and Design* (4th Edition) [21] is very good for background material on multiprocessor architecture. It is not appropriate as a textbook for this course.

Professional Multicore Programming, Design and Implementation for C++ Developers [13] is an in-the-trenches practical guide to developing multithreaded software for beginners. It includes a lot of detailed library documentation and some dated and very general processor documentation. It has some useful ideas on partitioning software systems, but overall much of the information in the book could be found on the Web. It is OK as a source of background material, but it is not a sufficient textbook.

Cellular Automata, A Discrete View of the World [22] is a good current overview on cellular automata. CA is a research area for these machines, but not for the course.

Programming Clojure [10] is a textbook for Clojure, a recent dialect of LISP with support for multiprocessing via wrappers to underlying Java Virtual Machine libraries. One of the example programs of Section IV (N queens) is coded in Clojure, Python and Jython in addition to C++ and Java as a test of interpreted languages. There have been in my opinion some questionable design choices in the architecture of Clojure, and I will probably not use it in the course other than as a framework for discussing the relationship of immutable data structures to multithreading.

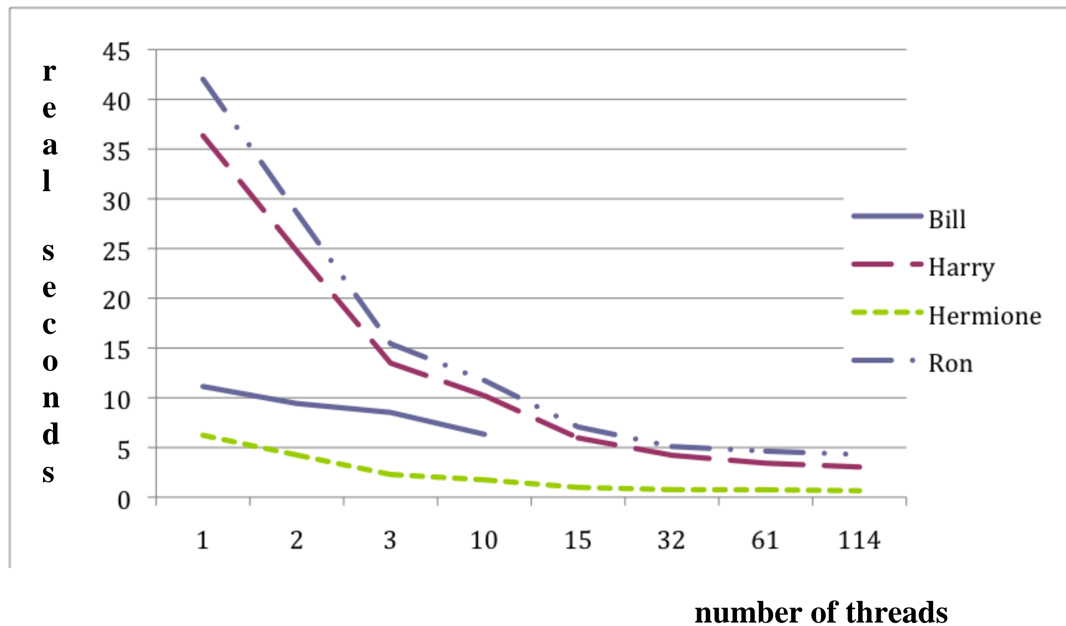
I skimmed the following books and found them inappropriate for the course, although potentially useful in preparing lectures. *Fundamentals of Parallel Processing* [14], like *Introduction to Parallel Computing*, appears to be a decent book for surveying a wide range of issues that include shared-memory and distributed multiprocessing, MIMD and SIMD approaches, libraries and algorithm development. It is too broad for the course. *Patterns for Parallel Programming* [18] takes an object-oriented pattern perspective. It is a little on the trite and verbose side, but a potentially useful source of ideas. *Principles of Concurrent and Distributed Programming* [1] is too broad and too formal for the course. *Principles of Parallel Programming* [16] is likewise too broad, too formal and too short.

IV. Code Examples and a Larger Planned Application

This section gives outlines of the algorithms and data structures used during my 2009-2010 investigation of the multiprocessor servers, along with graphs of their execution time results. Most algorithms were implemented in both C++ using POSIX threads and in Java using server JVMs. Algorithms include quicksort, mergesort, a closed address (chained) hash table store-and-lookup benchmark, and the N Queens puzzle. N Queens was also implemented in C-based Python, Jython and Clojure. I selected N Queens in order to measure a search algorithm with a large search space but with modest memory requirements. In general I was attempting to determine variations in processing bottlenecks, including synchronization, memory write, and

memory read bottlenecks, in comparing performance of the multiprocessors. This work will continue as part of the CSC 580 course. There is also discussion of a larger planned application of multiprocessing at the end of this section. We will complete the design and implementation of this algorithm in CSC 580.

Quicksort Version 1 is a concurrent version of quicksort in C++ using Pthreads that I implemented and measured in November 2009 through January 2010 on four machines, namely "Bill," a dual-core, time-shared Sparc machine used for academic computer science courses, along with multiprocessor servers "Harry" "Hermione" and "Ron" discussed in Section II. This is the only algorithm of this study to run on Ron. **Source code is available under ~parson/multip/multip/initial_qsort** on Bill or in multip.final.zip in P:\Parson. Command line arguments include number of integers to sort and a threshold on a subarray size. After partitioning the total array or a subarray, the quicksort implementation compares the size of that just-partitioned array to the threshold. If the array size is >= threshold elements, quicksort starts another thread to sort the left partition, then recursively sorts the right partition, and finally executes a join operation on the spawned thread, after which it returns. Once subarrays have been partitioned to sizes smaller than the threshold, quicksort proceeds recursively without spawning threads. Smaller threshold values lead to more software threads. There is a serial bottleneck at the beginning of the sort, because this version of quicksort does not spawn any threads until after partitioning. Concurrent techniques for partitioning have been proposed [9], although their added complexity and lack of thorough description kept me from using them.



number of threads

Graph 1: Multithreaded (threshold) quicksort run time as a function of thread count

Graph 1 shows real seconds in execution time (Y axis) for this quicksort of 10 million integers as a function of the number of software threads spawned (X axis). The irregular growth of the X

axis is due to the fact that unbalanced partitioning of subarrays causes thresholds to be exceeded, and thus threads to be spawned, at irregular rates. Later benchmarks eliminate this artifact.

When running a single thread, Ron is the slowest machine (42.0 seconds), followed by Harry (36.3 seconds), Bill (11.1 seconds) and then Hermione (6.2 seconds). The minimum time for each machine is 4.3 seconds for Ron, 3.0 seconds for Harry, 6.3 seconds for Bill, and 0.65 seconds for Hermione. Total CPU times (not shown) increase as threads are added, particularly on Harry and Ron, due to overhead entailed by software thread creation, synchronization, and thread contention for hardware resources. Bill's baseline speed for a single thread is significantly faster than its multiprocessing Sparc peers, and Hermione's single-threaded speed is the fastest.

Notice that Bill's speedup continues beyond two software threads even though Bill has only two hardware threads. Investigation of performance data reveals that imbalance in quicksort array partitioning accounts for speedup when there is an apparent excess of software threads. When a subarray is not divided equally between two software threads, one terminates well before the other. By invoking this quicksort with relatively small thresholds, and thus more software threads than the available hardware threads on Bill, a given hardware thread can execute another software thread on a small array partition after a previous software thread has completed its work on the small half of an unbalanced partition of a subarray. Using only two software threads on Bill would cause one hardware thread to run out of work well ahead of the other thread when sorting two unbalanced partitions of the total array. This implementation of quicksort uses the standard heuristic of using a pivot value that is the median of three values – the leftmost, the center, and the rightmost value – of the subarray being partitioned. This heuristic statistically improves the balance of partitions, but it is not perfect. Creating more software threads than available hardware threads helps eliminate performance penalties entailed by imbalanced partitioning. The effect is not as pronounced on Harry, Ron and Hermione because by the time that the number of software threads exceeds the available hardware threads, the partitions are small and statistically tend to be better balanced.

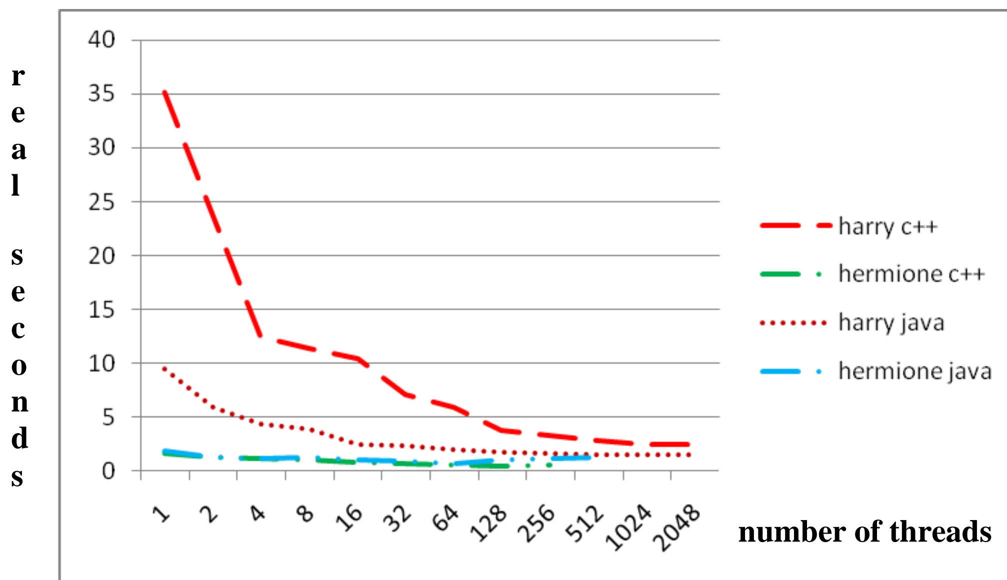
Harry and Ron eventually do exceed Bill's performance on this sort, but only when they apply roughly 30 hardware threads to the problem. Hermione is the clear winner in this test, both in terms of raw single-threaded speed and in terms of speedup through multiprocessing. In fact in a test run sorting one billion integers on Hermione while keeping all hardware threads busy completed its work in just under 60 seconds.

Experimentation with compiler options on Harry has yielded no improvement. The **pagesize** and **pagesize_heap** options recommended for programs with large in-core datasets [24] have no effect, and performance traces indicate that essentially no paging occurred during tests on the otherwise unoccupied machine. Various combinations of compiler **prefetch** options have no effect or detrimental effect. The machine-specific compiler options **-xtarget=ultraT2 -xarch=sparcvis2 -xcache=8/16/4:4096/64/16** on Harry have no measureable effect. Unlike the Opteron, the T1 and T2 processors do no instruction reordering or speculative execution, and every memory access causes an unconditional four-cycle stall on its thread. If the compiler does not reorder instructions to separate pairs of load-use instructions then the pipeline stalls more

than it does on the Opteron [17] (see Figure 2). With only one thread engaged per core, stall time is at a maximum, and for this program the partial solution is to fully engage all hardware threads.

An additional factor in the disparity between the Sparc and Opteron machines is the fact that the L2 cache location on Hermione leads to a tremendous advantage in memory-intensive applications. Hermione attaches 128Kb of L1 cache and 1 Mb of L2 cache directly to each of 16 single-threaded cores (Figure 3), while Harry and Ron attach only 8Kb of L1 cache directly to each multi-threaded core, forcing access to cross the crossbar to get to the respective 4 Mb and 3 Mb of L2 cache shared by all cores (Table 1 and Figures 1 and 5). With sufficiently small partitions on quicksort subarrays, a hardware thread on Hermione may be able to sort an entire subarray from within the L2 cache attached to that thread's core. Additional tests below suggest that the biggest problem for the architectures of Harry and Ron comes with manipulating large amounts of memory.

Quicksort Version 2 shown in Graph 2 is a modified version of the code of Graph 1 that takes an explicit number of software threads on the command line instead of a memory threshold size. **This code lives in subdirectories threadlim_qsort and jthreadlim_qsort**. Threads are spawned with one thread per quicksort array partition, top down, until the command line thread count has been reached, after which conventional recursive quicksort runs without spawning additional threads. Half of the remaining threads go to sorting the left partition and half to the right as long as threads remain to be allocated. Whereas the measurements in Graph 1 used the UNIX "time" command to measure entire process time, including the startup time to allocate and populate the 10,000,000 int array, Graph 2 and subsequent graphs use system calls to measure only that portion of the test in which multithreading applies. Times tend to be a bit smaller in Graph 2 when compared to Graph 1 for that reason.



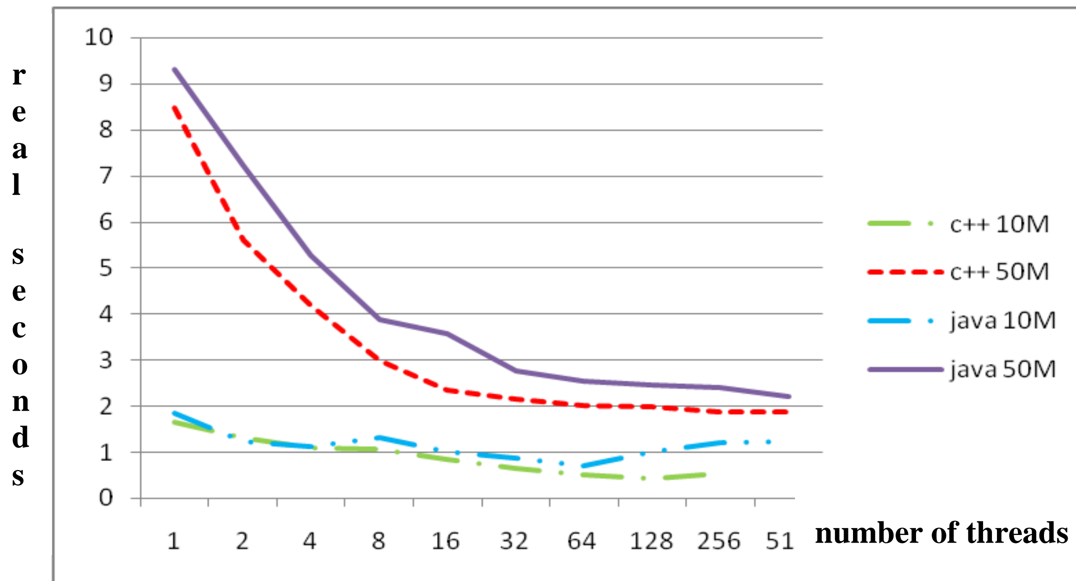
Graph 2: Multithreaded quicksort with explicit thread count on command line

Graph 2 shows the results of sorting 10 million integers on Harry and Hermione using both C++ with Pthreads and Java. Single-threaded Java on Harry runs roughly 3 times faster than compiled C++. C++ and Java on Hermione run at comparable speed, with Java becoming slightly slower after the productive thread count has been exceeded. The C++ and Java curves for Hermione in Graph 2 overlay at the bottom of the graph. Harry's curves in Graph 2 makes it apparent that the JVM combined with the just-in-time (JIT) compiler in server mode do a much better job than Sun's CC compiler of taking advantage of the T5120 architecture.

The cpustat utility on Harry supports reading counters for L1 and L2 data cache misses while the quicksort test is running (**sudo cpustat -c DC_miss,L2_dmiss_ld**). On an idle Harry running background operating system processes + cpustat, there is an average of 493 L1 data cache misses and 217 L2 data cache misses per hardware thread in 1 second. On Harry running **C++** quicksort using 1024 threads there is an average of 1,136,518 L1 data cache misses and 90,085 L2 data cache misses per hardware thread in 1 second, with the sort of 10 millions integers completing in 2.48 seconds. On Harry running **Java** quicksort using 1024 threads there is an average of 697,580 L1 data cache misses and 85,853 L2 data cache misses per hardware thread in 1 second, with the sort of 10 millions integers completing in 1.566 seconds. In going from C++ to Java there is 39% reduction in L1 misses, a 4.7% reduction in L2 misses, for a 37% reduction in real time. The Java JIT compiler appears to be beating the C++ optimizing compiler in ordering instructions to take advantage of the limited L1 cache. The source code for the inner portions of the quicksort algorithm are almost identical in C++ and Java, as is the storage consumed for an array of 10 million integers. Instruction reordering and other optimizations appear to account for the improvement in L1 cache performance with Java. According to a Sun expert (http://blogs.sun.com/d/entry/compiling_for_the_ultrasparc_t2), "To summarize, there is an UltraSPARC T2 specific compiler flag, but for most situations the best target to use would be -xtarget=generic which should give good performance over a wide range of processors." In fact –xtarget=native (the default for Harry) gives lightly better performance.

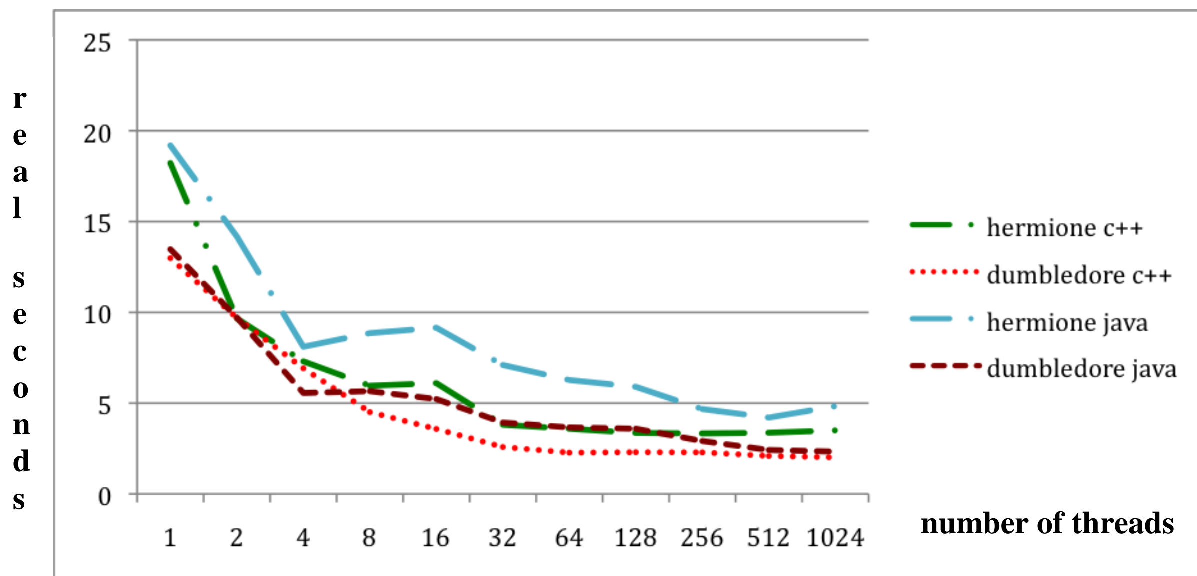
To summarize, Harry's JVM JIT compiler matches or exceeds CC optimized compilation in all tests in this study. Furthermore, Harry's T2 architecture requires heavily multithreaded applications in order to gain maximum effectiveness from threads used. Hermione's larger per-hardware-thread cache yields a fundamental advantage, and Opteron instruction reordering and speculative execution successfully augment optimizations in the C++ and Java JIT compilers.

Graph 3 repeats the C++ and Java quicksort tests of 10 million integers on Hermione, and adds C++ and Java sorts of 50 million integers on Hermione in order to scale measurements up into multiple seconds. The bottom two curves for 10 million integers in Graph 3 are identical to the bottom two curves in Graph 2. The 50 million curves give an indication that, unlike Harry, C++ is marginally faster than Java for this program on Hermione, but otherwise the multithreading benefit curves for the two languages on Hermione are on par.

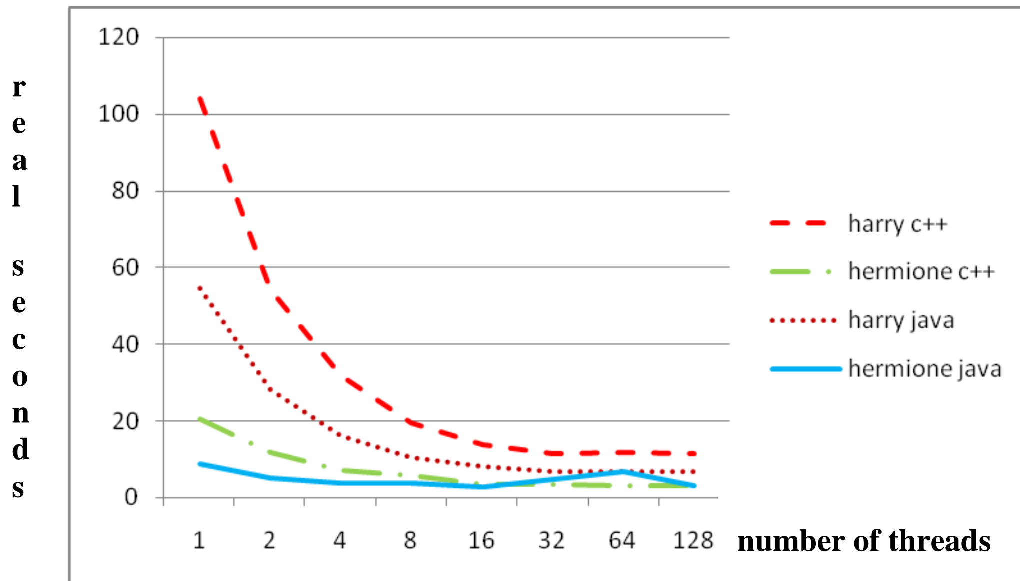


Graph 3: Quicksort of 10 and 50 million integers on <u>Hermione</u> using C++ and Java

January 2012: Graph 3B shows the results of running the above quicksort benchmark on arrays of 100 million integers on Hermione (AMD) and Dumbledore (Intel) using C++ and Java. Dumbledore's clock speed is 1.33x faster than Hermione's. Both have 16 hardware threads. Hermione's C++ performance is on par with Dumbledore's Java performance beyond 2 threads. It appears that the primary source of benefit for the Intel architecture is clock speed. There does not appear to be a significant difference in memory access performance.



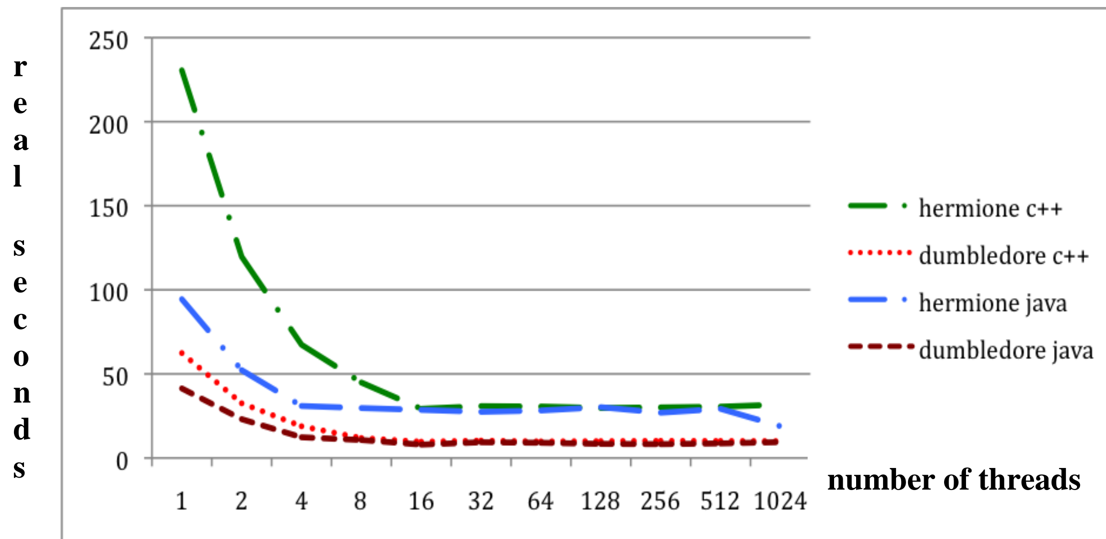
Graph 3B: Quicksort of 100 million ints on <u>Hermione & Dumbledore</u> using C++ and Java



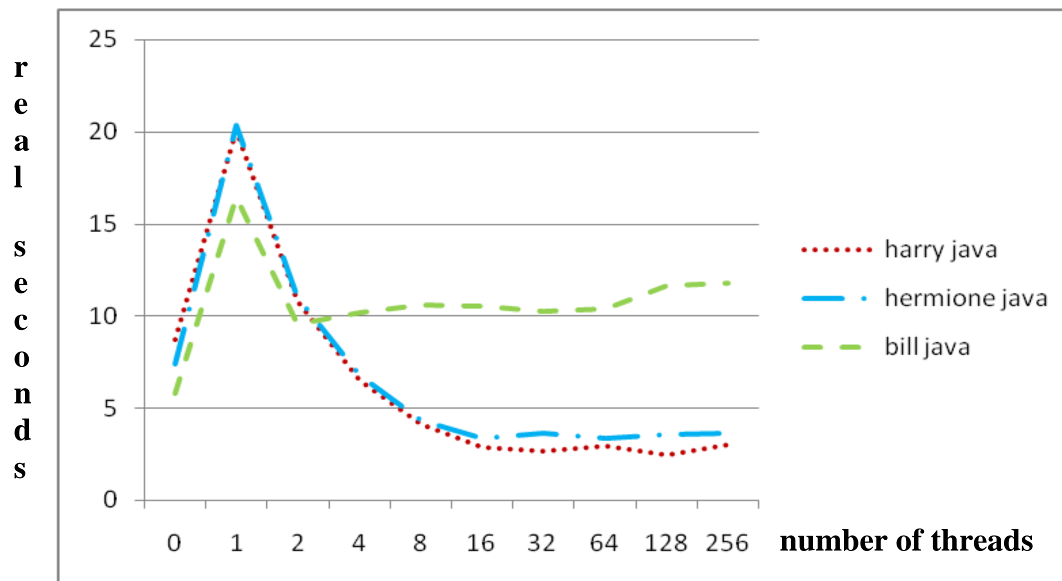
Graph 4: Multithreaded mergesort with explicit thread pool size on command line

Graph 4 shows a set of curves comparable to Graph 2 for mergesort for 10 million integers on Harry and Hermione using C++ and Java. **This code lives in subdirectories threadlim_pool_mrgsort and jthreadlim_pool_mrgsort**. This implementation of mergesort creates a pool of threads >= 1 in number and then initially divides the array to be sorted into that many partitions. Each thread in the pool applies serial mergesort to its partition, resulting in a "run size" (sorted subsequence size) equal to the total array size divided by the number of threads. In subsequent passes over the data only half of the threads employed in the previous pass are used to merge adjacent runs. This cycle repeats until the run size meets the array size. Thus, like quicksort, the algorithm takes full advantage of all allocated threads only at the bottom of the partitioning tree. In quicksort this division of labor occurs top down, while in mergesort it occurs bottom up. The overall time for mergesort is greater than quicksort because of copy overhead in mergesort, but otherwise the curves of Graphs 2 and 4 are similar. Unlike quicksort, mergesort guarantees equal division of its partitions, so all threads remain occupied in the initial sorting phase.

January 2012: Graph 4B shows the results of running the above mergesort benchmark on arrays of 100 million integers on Hermione (AMD) and Dumbledore (Intel) using C++ and Java. Mergesort entails much more copy overhead than quicksort, hence the overall growth in execution time on the Y axis. As in Graph 4, Java surpasses C++ on both machines for this benchmark. Dumbledore does somewhat better than expected from the clock speed differential alone, indicating a contribution of memory architecture and cache size for a large data set with much copy overhead.



Graph 4B: Multithreaded mergesort on 100 million integers

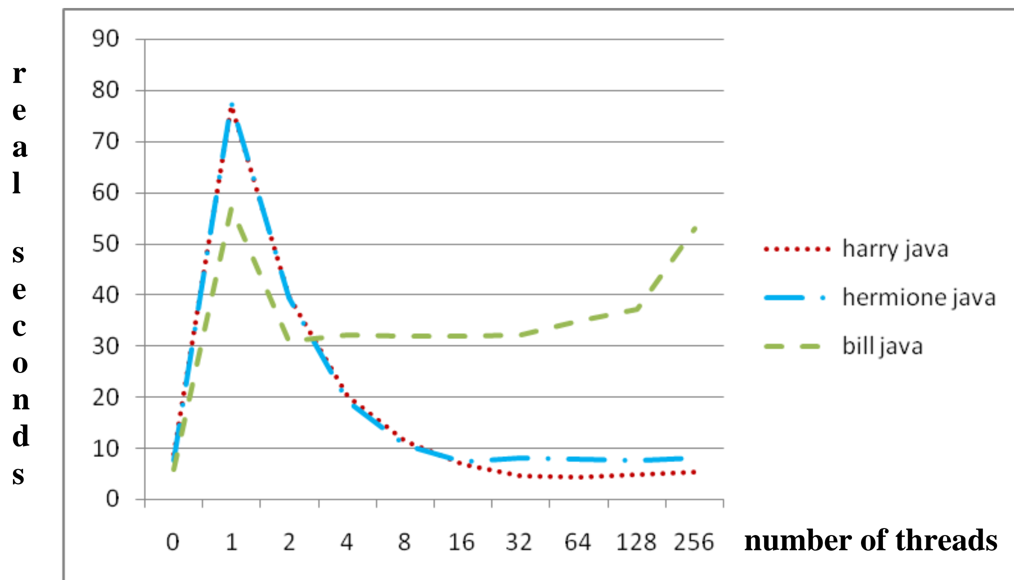


Graph 5: Hash set 1:10 insert:lookup ratio using Herlihy/Shavit striped linked hash table

Graph 5 shows concurrent insertion of 1 million integers from a pseudo-random sequence followed by concurrent lookup of 10 million integers using a concurrent hash table provide by the Herlihy / Shavit textbook [12] with a bug fix by Parson. **Code is in subdirectory jhash_closed**. The zero-thread measurement corresponds to using java.util.HashSet with no threads spawned, while the one-thread measurement corresponds to using the custom hash set with one thread spawned to use it. The custom hash set uses a striped approach to locking, where the linked lists for a group of buckets within a stripe share a common lock. This approach is a compromise between locking the entire set, which minimizes storage consumption by locks while also minimizing concurrent access, and providing a distinct lock for each list at each

bucket, which maximizes concurrency and storage cost. There were approximately four buckets per lock stripe in this test run. The 0 thread, 1 thread and minimal multithreading times in seconds were (8.7, 20.0 and 2.4 at 128 threads for Harry), (7.4, 20.4 and 3.4 at 64 threads for Hermione), and (5.8, 16.5 and 10.3 at 32 threads for Bill). Harry recouped the added costs of multithreading at 4 threads (6.5 seconds) as did Hermione at 4 threads (6.8 seconds).

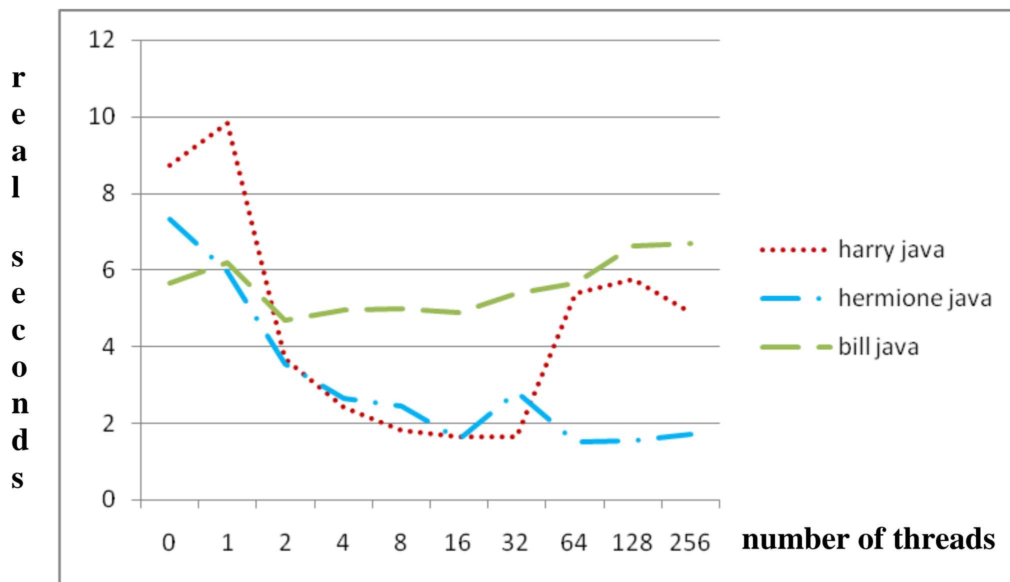
Given the fact that there is no inter-thread contention for access to the hash table in the 1 thread case, the spike is caused by switching from java.util.HashSet to the custom striped hash table data structure, with the increase being a combination of locking / unlocking costs for uncontended locks and a less effective data structure. In order to get a feel for the effectiveness of the data structure, a similar test run shown in Graph 6 replaces the custom striped hash table with the library set implementation in java.util.concurrent.ConcurrentSkipListSet. **Code is in subdirectory jhash_closed/skiplistset**. That set class guarantees average log(n) insertion and lookup time rather than constant time for an ideal closed address hash table. The 0 thread, 1 thread and minimal multithreading times in seconds for Graph 6 are (8.7, 77.0 and 4.3 at 64 threads for Harry), (7.6, 77.4 and 7.5 at 128 threads for Hermione), and (5.9, 57.0 and 32.0 at 8 threads for Bill). Based on the extreme similarity of the curves in Graphs 5 and 6 despite the use of decidedly different data structures, it appears that most of the 1-thread performance hit comes from locking and unlocking overhead in the absence of thread contention. The cost of the thread-safe set outweighs any multithreading benefit for Bill in both test cases, and Hermione barely manages to recoup the cost when using ConcurrentSkipListSet. On the test of Graph 5 Harry and Hermione recouped the cost at 4 threads. An additional test run using the library's CopyOnWriteArraySet performed so badly that I aborted its run. Documentation for that set warns against using it for large or mutable sets. This test actually performs all mutations (insertions) into a ConcurrentSkipListSet object and then constructs the CopyOnWriteArraySet from that object, avoiding mutation of the CopyOnWriteArraySet object; performance was poor.



Graph 6: Set 1:10 insert:lookup ratio using library ConcurrentSkipListSet

The marked differences in absolute time values – ConcurrentSkipListSet from the standard concurrent library was 2 to 4 times slower than the striped hashed set for this program – highlights the need to benchmark and choose algorithms and data structures carefully when they lie in the critical path of execution with respect to time.

With that thought in mind, I decided to run one more variation of this test, using ConcurrentHashMap from java.util.concurrent to implement the set object. The results appear in Graph 7, and happily, the Java library class is extremely effective in this benchmark in comparison to Graphs 5 and 6. **Code is in directory jhash_closed/concurrenthashmap**. The 0 thread, 1 thread and minimal multithreading times in seconds were (8.7, 9.8 and 1.6 at 32 threads for Harry), (7.3, 6.0 and 1.5 at 64 threads for Hermione), and (5.7, 6.2 and 4.7 at 2 threads for Bill). Harry recouped the added costs of multithreading at 2 threads (3.7 seconds) as did Bill at 2 threads (4.7 seconds). Hermione showed no penalty for going from HashSet to ConcurrentHashMap. Unlike java.util.Hashtable, which locks the entire table during access, ConcurrentHashMap's documentation states, and "even though all operations are thread-safe, retrieval operations do *not* entail locking." Lock avoidance usually takes the form of having threads spin on atomic variables, polling those variables when entering critical sections. While poll spinning for heavily contended critical sections can be too expensive for use on uniprocessors, on multiprocessors using data structures with very short critical section times, poll spinning can typically take the form of a few trips around a loop on an entering thread, in contrast to the system call overhead of locking and unlocking. Graph 7 shows that this approach can be very effective.[2]

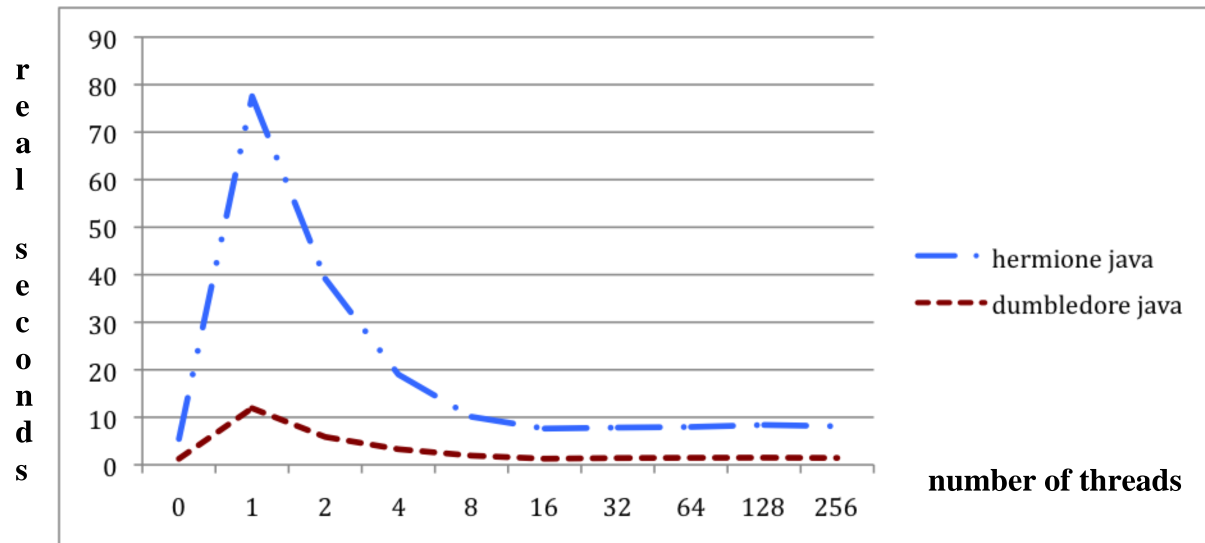


Graph 7: Hash set 1:10 insert:lookup ratio using library ConcurrentHashMap

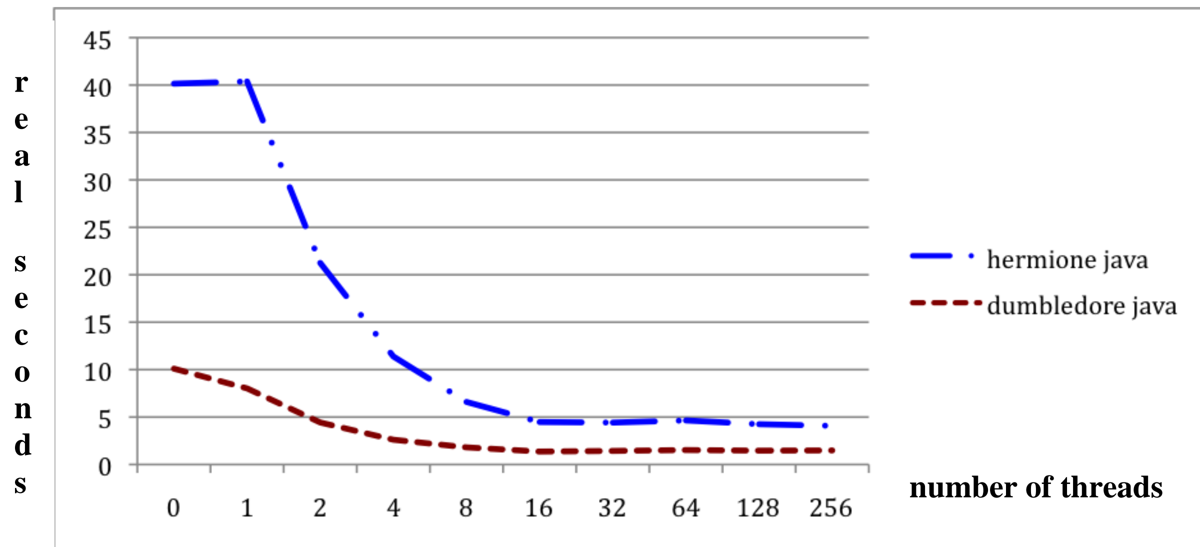
[2] A subsequent reading of Chapter 11 of the Goetz textbook [8] reveals that ConcurrentHashMap uses striped locks for write operations and some read operations, presumably the read operations that conflict with writes. The majority of read operations entail checking uncontended atomic variables.

Contention for memory access during poll spinning is the most likely culprit for the sharp rise in Harry's run time after exceeding 32 threads. Measurements by the *mpstat* utility in comparing the 16-thread and 128-thread tests of Graph 7 show a marked increase in "spins on mutexes (lock not acquired on first try)" in going from 16 to 128 threads. Since ConcurrentHashmap does not use mutexes (locks), presumably this increased lock contention occurs elsewhere in the JVM. Mpstat also reveals a decrease in per-hardware-thread idle time from 76% in the busiest 24 hardware threads for the 16-thread test (the JVM uses some additional threads beyond 16) during a 1.66 second run to 70% for all 64 hardware threads for the 128-thread test during a 5.74 second test run. More hardware threads become busier for a longer period of time accomplishing the same amount of application work. The UNIX "time" command shows that most of the increase takes place in user CPU time, not in system CPU time. The biggest danger with poll-spinning on an atomic variable in order to gain access to a critical section is that active hardware threads pollute the memory access hardware with polling activity that is fruitless for most polls, thereby stalling other threads. Goetz, et. al. sum up the trade-off between locks and polling of atomic variables by stating, "The performance reversal between locks and atomics at differing levels of contention illustrates the strengths and weaknesses of each. With low to moderate contention, atomics offer better scalability; with high contention, locks offer better contention avoidance." [8, p. 328] Locks are better in high contention cases because they stop threads from contending and over-utilizing memory access hardware. The high-thread ends of the curves for Harry in Graphs 5 through 7 illustrate this trade-off. Hermione with only 16 hardware threads never reaches this trade-off.

January 2012: Graphs 6B and 7B below give results corresponding to Graphs 6 and 7 for 1 million insertions and 100 million lookups (up from 10 million) on Hermione (AMD) and Dumbledore (Intel). Again, ConcurrentSkipList never exceeds the performance of single-threaded use of HashSet in the threads = 0 case at the left end of Graph 6B.

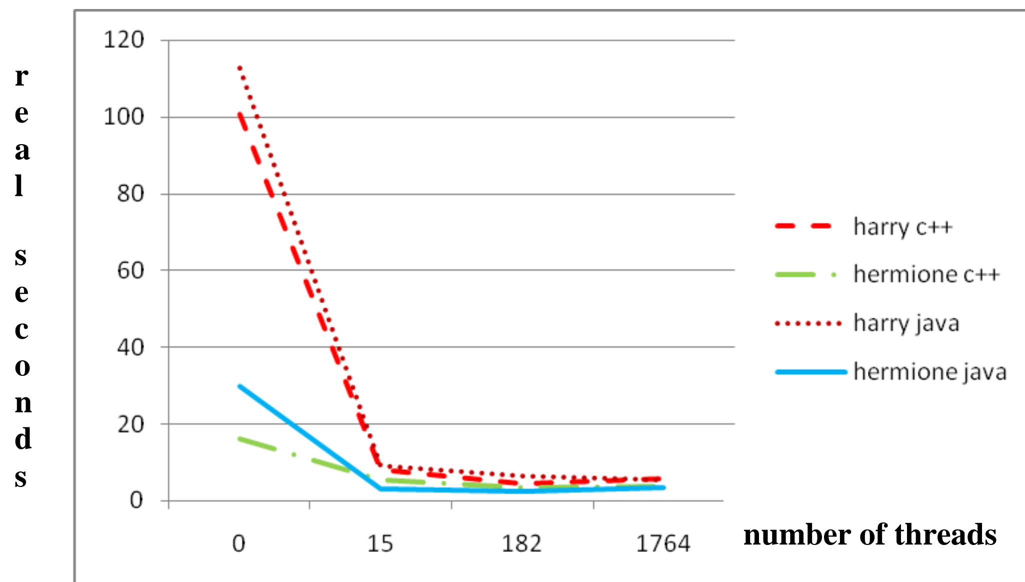


Graph 6B: Set 1:100 insert:lookup ratio on Hermione (AMD) & Dumbledore (Intel) using library ConcurrentSkipListSet



Graph 7B: Hash set 1:100 insert:lookup ratio using library ConcurrentHashMap on Hermione (AMD) and Dumbledore (Intel)

ConcurrentHashMap again fares much better. Multithreading has a more dramatic effect on the AMD machine. Performance on this benchmark is fairly good without multithreading on the Intel machine.



Graph 8: 15 X 15 N Queens on Harry and Hermione in C++ and Java

The final benchmark in this series, the N Queens problem, represents an effort to measure performance for a large search space problem that does not use a large amount of memory. **Code is in and beneath directory threadlim_nqueens**. The N Queens puzzle looks for every possible configuration of an N x N game board that places N chess queens in positions where they do not

confront any other queen horizontally, vertically or diagonally. The implementations measured use a board state object that houses one small vector of integers and three small vectors of byte-size booleans to record occupation by a queen at some position of a given column, row, forward-slash and backward-slash diagonal. The full N x N matrix was not stored, in the interest of minimizing memory consumption. In all implementations except a Clojure implementation [10], mutation with backtracking restoration of previous state occurred within single-threaded subsets of the problem space exploration in order to minimize copy overhead and memory consumption. Clojure works mostly with immutable data structures. It attempts to minimize the cost of constructing new structures from parts of existing data structures.

The basic algorithm consists of iterating over a column and finding each safe location to place a queen. With a queen in place in the current column, the single-threaded version uses recursion and backtracking to search the next column, while the multithreaded version starts a thread in the next column (given the current placement of the queen in the current column), and then proceeds to find the next safe spot for a queen in the current column, again spawning a thread for the next column with this next safe queen placement in the current column. Once it reaches the end of its column, the multithreaded version joins all threads that it has spawned. This spawn-join behavior can happen in 0, 1, or more columns of the board search space, as controlled by a command line parameter. Graph 8 gives the set of curves. The number of threads correspond to a column depth for multithreading specified on the command line. A depth of 0 spawns no threads. A depth of 1 spawns 15 threads, one for each row in the first column. A depth of 2 has these 15 threads spawn a total of 182 threads to search deeper; the 15 spawn fewer than 15 x 15 = 225 because some positions in the second column are invalid for a given queen placement in the first column. A depth of 3 spawns 1764 threads below the 182. Once all sub-threads are running, the thread that spawned them waits to join them. Any thread that finds a solution increments an atomic counter before returning. Once beyond the thread-spawning depth limit, each thread invokes the single-threaded, recursive, backtracking algorithm.

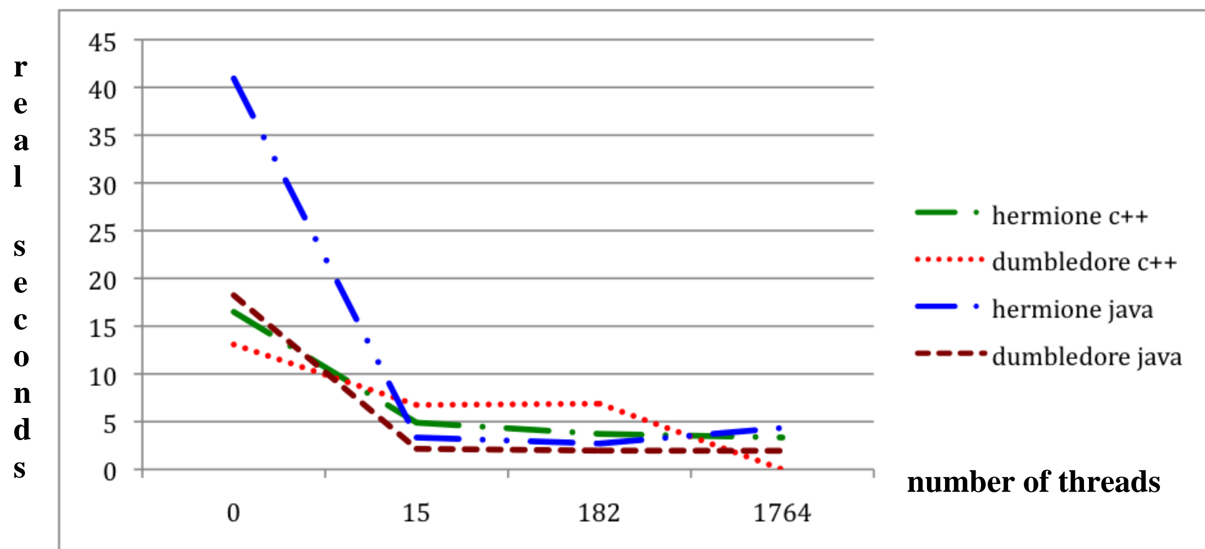
number threads	**0**	**15**	**182**	**1764**
harry c++	100.54	8.07	4.37	5.87
harry java	112.602	8.936	6.253	5.391
harry python	19028.483	1517.602	745.118	737.132
harry jython	24001.803	9654.617	12164.258	15132.37
harry clojure	5883.59	475.08	215.809	223.569
hermione c++	16.12	5.71	3.59	3.82
hermione java	29.867	2.916	2.316	3.446
hermione python	2825.602	211.302	179.349	178.108
hermione jython	5807.911	11283.732	7065.99	9025.306
hermione clojure	1534.754	413.526	397.673	410.039

Table 2: 15 X 15 N Queens on Harry and Hermione in compiled and interpreted languages

Table 2 gives the measurements for the above 15 X 15 N Queens test runs from Graph 8, and it adds test runs for interpreted languages Python (C-based implementation of Python compiler and VM), Jython (Python atop a JVM), and Clojure (a dialect of LISP with support for multithreading, compiling to the JVM). I have used a table because a graph compresses the faster measurements too much to make them readable.

The Python implementation uses multiple UNIX processes to achieve hardware multithreading via Python's *multiprocessing* library module. Python allows only one thread at a time to execute within the VM, necessitating use of multiple processes to achieve multiprocessing. It is a demonstration of the fact that processes running in distinct address spaces can take advantage of hardware threads on these processors.

Clojure is the performance winner among interpreted languages on Harry; Python wins on Hermione. With two or more orders of magnitude performance hit over the C++ and Java, interpreted languages have a long way to go in using multiprocessors effectively. Both Jython and Clojure generate code for the JVM, and one would hope that they could leverage some of the performance potential demonstrated by the Java test runs. An effort to improve multiprocessing for the functional language Haskell [11] may be more successful because Haskell does not support dynamic typing with its run time overhead, opting instead for compile-time type inference. All of the interpreted languages of Table 2 use dynamic typing.



Graph 8B: 15 X 15 N Queens on Hermione and Dumbledore in C++ and Java

January 2012: Graph 8B replaces Harry (Sparc) of Graph 8 with Dumbledore (Intel). The graph does not show that g++ running under Cygwin on Windows 7 crashed when attempting the 1764-thread case. The process could not create that many threads. An attempt to duplicate the multiple-process Python test run suffered a similar fate on N Queens. Python crashed even on the 15-process case. There are reasons to suspect that the Python multiprocessing library may be faulty on Windows 7.

A larger planned example will be a complete multithreaded application or at least a substantial multithreaded component of a framework. Originally I had considered attempting to adapt GNU Chess [7] for multithreading, but the code architecture is not readily amenable to multithreading because of global variables and hard-to-find state mutation. The current plan is to design and implement a multithreaded intelligent Scrabble opponent as part of the spring 2011 CSC 580 course, using a Java game-to-music framework constructed in earlier Java Programming classes [20].

Anticipated uses in other courses include using Harry and Hermione as they are currently configured for courses in parallel programming, Web server development, server architecture, theory of parallel algorithms, and UNIX programming that includes Linux on Hermione. Harry and Hermione are also available for research and independent study projects. When Ron is on line again we can use it for offloading student compute-bound projects from Bill and for student exercises in UNIX administration that entail taking the machine off line and reconfiguring it. Ron would also be useful for student study in virtualization techniques.

V. External Benchmarks

OpenMPBench is an open source multiprocessor benchmarking project [19]. It includes a growing set of multiprocessor benchmarks relevant to Automation and Industry Control, Networking, Office Applications and Security. It is source code limited to testing 8 concurrent threads, although the source should be amenable to easy extension to additional threads. It is all written in ANSI C using Pthreads.

My plan is to have students extend and test these benchmarks on our machines in spring 2011 CSC 580. I have run the string matching benchmarks in order to check out the ease of compiling and using these benchmarks. The results for applying a multithreaded version of the Pratt-Boyer-Moore string search algorithm against a 32 Mbyte test data set appear in Table 3.

harry c	hermione c	threads
290.4	49.2	1
142.4	25.5	2
71.2	18.1	4
35.7	17.6	8

Table 3: Pratt-Boyer-Moore string search in ANSI C [19]

Harry shows performance doubling with thread doubling within the range of 8 threads, while Hermione levels off in this test run. The results suggest that adding threads could be beneficial, particularly on Harry. Extension of these and the other benchmarks on this site should be a useful exercise for CSC 580.

VI. References

1. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Second Edition, Addison-Wesley, 2006.

2. Free Software Foundation, C++0x Support in GCC, http://gcc.gnu.org/projects/cxx0x.html, August 2010.

3. Free Software Foundation, C200x Support in GCC, http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.200x, August 2010.

4. Fujitsu, Sparc Enterprise T5120, T5220, T5140 and T5240 Server Architecture, http://www.fujitsu.com/downloads/SPARCE/whitepapers/T5x20-T5x40-wp-e-200907.pdf, July 2009.

5. *Genetic Programming On General Purpose Graphics Processing Units* (GP GP GPU) home page, http://www.gpgpgpu.com/, 2010.

6. *General-Purpose Computation on Graphics Hardware* home page, http://gpgpu.org/, 2010.

7. GNU Chess, http://www.gnu.org/software/chess/, August 2010.

8. Goetz, Peierls, Bloch, Bowbeer, Holmes and Lea, *Java Concurrency in Practice*, Pearson, 2006.

9. Grama, Gupta, Karypis and Kumar, *Introduction to Parallel Computing*, Second Edition, Pearson, 2003.

10. Halloway, *Programming Clojure*, Pragmatic Bookshelf, 2009.

11. Haskell and Multi-core Systems, http://haskell.org/opensparc/ and http://www.haskell.org/~duncan/opensparc/announce.email, 2008.

12. Herlihy and Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

13. Hughes and Hughes, *Professional Multicore Programming, Design and Implementation for C++ Developers*, Wrox, Wiley, 2008.

14. Jordan and Alaghband, *Fundamentals of Parallel Processing*, Prentice Hall, 2003.

15. Kirk and Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

16. Lin and Snyder, *Principles of Parallel Programming*, Addison-Wesley, 2009.

17. Lippmeier, Ben, personal communication, http://www.cse.unsw.edu.au/~benl/, August 8, 2010.

18. Mattson, Sanders and Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005.

19. OpenMPBench: An Open-Source Benchmark for Multiprocessor Based Embedded Systems, http://code.google.com/p/multiprocessor-benchmark/, August 2010.

20. Parson, Dale, "Apprenticeship in Undergraduate Java Programming," Proceedings of the 25th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators, West Chester University, West Chester, PA, April 9-10, 2010, http://www.pacise.org/, http://faculty.kutztown.edu/parson/pubs/PACISE2010DaleParsonFinal.pdf.

21. Patterson and Hennessy, *Computer Organization and Design, The Hardware / Software Interface*, Fourth Edition, Elsevier, 2009.

22. Schiff, *Cellular Automata, A Discrete View of the World*, Wiley, 2008.

23. Sun Microsystems, Sun Fire™ 4600 M2 Server Architecture, http://www.sun.com/servers/x64/x4600/arch-wp.pdf, June 2008.

24. Sun Microsystems, *Sun Studio 12: C++ User's Guide*, http://dlc.sun.com/pdf/819-5267/819-5267.pdf, 2007.

25. Sun Microsystems, Sun Fire™ T1000 and T2000 Server Architecture, http://www.sun.com/servers/x64/x4600/arch-wp.pdf, December, 2005.

26. Sun Sales Quotation, Change Your World Grants Program, Kutztown University Computer Science file P:\grants\accepted\sun_servers_parson_2009_granted_july.zip, July 23, 2009.

27. Williams, Anthony, *C++ Concurrency in Action, Practical Multithreading*, Manning Publications, 2011.

Appendix A: Department-approved syllabus for CSC580 Special Topics, Spring 2011

Kutztown University
Kutztown, Pennsylvania

Computer Science Department
College of Liberal Arts and Sciences

I. Course Description: CSC 580: Algorithms and Data Structures for Multiprocessing

This special topics course explores the concepts and practices of creating software that makes effective use of modern multiple-processor computers. Emphasis is on partitioning program code and data for efficient execution on multiple processors that share machine resources such as memory. Lab exercises include construction, execution and benchmarking of multithreaded programs on several multicore, multithreaded computers.

3 s.h. 3 c.h. Prerequisites: Fluency in a programming language that supports multithreaded programming such as C++ or Java; unconditional acceptance to the graduate program.

II. Course Rationale

Physical limits in circuit design for speed and size are forcing computer designers to interconnect multiple processors that share resources such as memory, disk, and network interfaces. The goal is to allow multiple processors to divide problems into smaller sub-problems that the processors can solve concurrently. Software architectures must partition code and data properly to achieve maximum effectiveness in using multiple instruction streams that share resources. Students must learn how to partition software for multiprocessing in order to work successfully with such computers.

III. Course Objectives

Upon satisfactory completion of this course the student will be able to:

A. Determine limiting factors in multiple processor hardware architectures by reading manufacturer specifications and user performance guides.

B. Partition program algorithms and data structures to make effective use of multiple processor computing systems.

C. Apply the multithreading software library constructs of modern programming languages to achieve concurrent execution and synchronization of multiple computer instruction streams.

D. Measure improvements or deterioration in processing throughput or latency as a result of applying multithreading by using software profiling utilities.

E. Identify sources of improvements or deterioration, enhancing the former and minimizing the latter in iterative program development.

F. Demonstrate the structure and application of computing mechanisms of shared memory, message passing, distributed computation and synchronization through diagrams and programs.

G. Explain multiprocessing hardware and software mechanisms such as intra-core versus inter-core shared resources, run-time system mapping of threads to hardware processors, and application program interaction with these underlying mechanisms.

IV. Course Assessment

The course assessment will be a subset of tests, projects, papers, presentations, quizzes, homework, team assignments and final exam.

V. Course Outline

A. Multiple processor hardware architectures
1. Single instruction stream, multiple data stream processors
2. Multiple instruction stream, multiple data stream processors
3. Multiple cores, contexts, and hierarchies of shared resources
4. Shared memory and message passing hardware mechanisms
5. Cache coherence, multiple-port caches, memory hierarchies

B. Multiple processor software mechanisms
1. POSIX threads, processes, Java threads, java.util.concurrent
2. Fork, join, shared memory, critical sections, non-recursive and recursive mutexes, semaphores, monitors, condition variables, message queues, events, thread-local store, futures and continuations
3. Program pragmas, OpenMP™, pinning threads to processors
4. Run-time mapping of software constructs to hardware processors

C. Software architecture and partitioning
1. Porting classic sort and search algorithms for multithreading
2. Porting classic data structures for multithreading
3. Minimizing shared resource contention and synchronization
4. Maximizing multithread locality for large-memory problems
5. Parallel exploration of low-memory, large search spaces
6. Parallelization and pipelining in algorithm refactoring
7. Latency versus throughput in leveraging multiprocessing
8. Functional programming, immutable data, Erlang, Clojure
9. Profiling and iterative tuning of multithreaded software
10. Finding, running and interpreting multiprocessor benchmarks

D. Applications and frameworks
   1. Multithreaded application frameworks
   2. Multithreaded web servers
   3. Server virtualization
   4. Compute bound and I/O bound classes of services

E. Embedded and special purpose multiple processor computing
   1. Network processors and stateless versus stateful flows
   2. Digital signal processors and multithreaded media channels
   3. SIMD applications of graphics processing units
   4. Grid computing
   5. Cellular automata, genetic algorithms, and other local-reference abstractions of computation

VI. Instructional Resources

Advanced Micro Devices, *AMD Processors*, http://www.amd.com/us/products/Pages/processors.aspx, 2010.

Butenhof, David, *Programming with POSIX Threads*, Addison-Wesley, 1997.

Clojure programming language home page, http://clojure.org/, 2010.

Comer, D., *Network Systems Design Using Network Processors, Agere Version*, Prentice Hall, 2005.

Erlang programming language home page, http://erlang.org/, 2010.

Genetic Programming On General Purpose Graphics Processing Units (GP GP GPU) home page, http://www.gpgpgpu.com/, 2010.

General-Purpose Computation on Graphics Hardware home page, http://gpgpu.org/, 2010.

Giladi, Ran, *Network Processors: Architecture, Programming, and Implementation*, Morgan Kaufmann, 2008.

Goetz, Peielrs, Bloch, Bowbeer, Holmes, Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.

Grid Computing Information Center home page, http://www.gridcomputing.com/, 2010.

Herlihy, Maurice and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

Halloway, Stuart, *Programming Clojure*, Pragmatic Bookshelf, 2009.

Hughes, Cameron and Tracey Hughes, *Professional Multicore Programming: Design and Implementation for C++ Developers*, Wrox, 2008.

Lewis, Bill and Daniel Berg, *Multithreaded Programming With PThreads*, Prentice Hall PTR, 1997.

Mattson, Timothy, *Patterns for Parallel Programming*, Addison-Wesley, 2004.

Miller, Russ Miller and Laurence Boxer, *Algorithms Sequential & Parallel: A Unified Approach*, Charles River Media, 2005.

Patterson, David and John Hennessey, *Computer Organization and Design: The Hardware/Software Interface*, Fourth Edition, Morgan Kaufmann, 2008.

Robbins, Kay and Steve Robbins, *UNIX Systems Programming*, Prentice Hall PTR, 2003.

Scarpino, Matthew, *Programming the Cell Processor: for Games, Graphics and Computation*, Prentice Hall PTR, 2008.

Sun Microsystems, *Sun Chip Multithreading Servers*, http://www.sun.com/servers/coolthreads/overview/index.jsp, 2010.

Toffoli, Tommaso and Norman Margolus, *Cellular Automata Machines*, MIT Press, 1987.

VanderHart, Luke, *Practical Clojure*, Apress, 2010.

Appendix B: Compiler and other command line utilities for multiprocessing

On Harry the following utilities are useful:

My account uses the following environment adjustments:

```
export PATH=$PATH:/export/home/faculty/parson/temporary/harry/bin:/usr/sfw/bin
export JAVA_HOME=/usr
export TERM=xterm
alias vi=/bin/vi
```

/usr/bin/CC -fast -xO4 -m64 -mt *.cxx –o

compiles C++ code. Experimentation with various –xtarget prefetch, pagesize and related options brought no improvement on Harry's test runs.

"mpstat 1" updates a display of processor status every second.

"sudo cpustat -c DC_miss,L2_dmiss_ld" measures L1 and L2 data cache misses.

Chris Walck found the following graphical utilities useful:

To analyze data cache misses, create an experiment using the following.

$ collect -h DC_miss ./thread_qsort 50000000 8 12345

Then with the resulting .er file, open it with analyzer.

$ analyzer test.1.er

Total and per-function DC_miss events are shown in a GUI.

On Hermione the following utilities are useful:

My account uses the following environment adjustments:

```
export PATH=$PATH:/export/home/faculty/parson/temporary/hermione/bin
export JAVA_HOME=/usr
```

"mpstat -P ALL 1" updates a display of processor status every second.

Appendix C: Department-approved syllabus for CSC480 Special Topics, Spring 2012

This appendix was added January 2012.

Kutztown University
Kutztown, Pennsylvania

Computer Science Department
College of Liberal Arts and Sciences

I. Course Description: CSC 480: Multiprocessing and Concurrent Software Design

This special topics course explores the concepts and practices of creating software that makes effective use of modern multiple-processor computers. Emphasis is on partitioning program code and data for safe and efficient execution on multiple processors that share machine resources such as memory. Lab exercises include construction, execution and benchmarking of multithreaded programs on several multicore, multithreaded computers.

3 s.h. 3 c.h. Prerequisites: CSC 237 and CSC 243 or permission of the instructor.

II. Course Rationale

Physical limits in circuit design for speed and size are forcing computer designers to interconnect multiple processors that share resources such as memory, disk, and network interfaces. The goal is to allow multiple processors to divide problems into smaller sub-problems that the processors can solve concurrently. Software architectures must partition code and data properly to achieve maximum effectiveness in using multiple instruction streams that share resources. Students must learn how to partition software for multiprocessing in order to work successfully with such computers.

III. Course Objectives

Upon satisfactory completion of this course the student will be able to:

H. Determine limiting factors in multiple processor hardware architectures.

I. Partition program algorithms and data structures to make effective use of multiple processor computing systems.

J. Apply the multithreading software library constructs of modern programming languages to achieve concurrent execution and synchronization of multiple computer instruction streams.

K. Measure improvements or deterioration in processing throughput or latency as a result of applying multithreading by using software profiling utilities.

L. Identify sources of improvements or deterioration, enhancing the former and minimizing the latter in iterative program development.

M. Demonstrate the structure and application of shared memory, message passing, distributed computation and synchronization through diagrams and programs.

N. Explain the mapping of software mechanisms onto multiprocessing hardware.

IV. Course Assessment

The course assessment will be a subset of tests, projects, papers, presentations, quizzes, homework, team assignments and final exam.

V. Course Outline

A. Multiple processor hardware architectures

6. Single instruction stream, multiple data stream processors
7. Multiple instruction stream, multiple data stream processors
8. Multiple cores, contexts, and hierarchies of shared resources
9. Shared memory and message passing hardware mechanisms
10. Cache coherence, multiple-port caches, memory hierarchies

B. Multiple processor software mechanisms

5. POSIX threads, processes, Java threads
6. Fork, join, shared memory, critical sections, non-recursive and recursive mutexes, semaphores, monitors, condition variables, message queues, events, thread-local store, futures and continuations
7. Program pragmas, OpenMP™, pinning threads to processors
8. Run-time mapping of software constructs to hardware processors

C. Java and C++ concurrency language and library constructs

1. Thread safety, object publication, object sharing and object composition
2. Thread pools, scheduling and priorities, task cancellation, shutdown
3. Multithreaded interaction with single threaded GUI event threads
4. Liveness, race conditions, performance, testing and benchmarks
5. java.util.concurrent library classes and their application
6. C++0x concurrent library classes and their application

D. Software architecture and partitioning

11. Porting classic sort and search algorithms for multithreading
12. Porting classic data structures for multithreading
13. Minimizing shared resource contention and synchronization
14. Maximizing multithread locality for large-memory problems
15. Parallel exploration of low-memory, large search spaces
16. Parallelization and pipelining in algorithm refactoring

17. Latency versus throughput in leveraging multiprocessing
18. Functional programming, immutable data, Erlang, Clojure
19. Profiling and iterative tuning of multithreaded software
20. Finding, running and interpreting multiprocessor benchmarks

E. Applications and frameworks
5. Multithreaded application frameworks
6. Multithreaded web servers
7. Server virtualization
8. Compute bound and I/O bound classes of services

VI. Instructional Resources

Advanced Micro Devices, *AMD Processors*, http://www.amd.com/us/products/Pages/processors.aspx, 2010.

Butenhof, David, *Programming with POSIX Threads*, Addison-Wesley, 1997.

Clojure programming language home page, http://clojure.org/, 2010.

Comer, D., *Network Systems Design Using Network Processors, Agere Version*, Prentice Hall, 2005.

Erlang programming language home page, http://erlang.org/, 2010.

Free Software Foundation, C++0x Support in GCC, http://gcc.gnu.org/projects/cxx0x.html, August 2010.

Genetic Programming On General Purpose Graphics Processing Units (GP GPU) home page, http://www.gpgpgpu.com/, 2010.

General-Purpose Computation on Graphics Hardware home page, http://gpgpu.org/, 2010.

Giladi, Ran, *Network Processors: Architecture, Programming, and Implementation*, Morgan Kaufmann, 2008.

Goetz, Peielrs, Bloch, Bowbeer, Holmes, Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.

Grid Computing Information Center home page, http://www.gridcomputing.com/, 2010.

Herlihy, Maurice and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

Halloway, Stuart, *Programming Clojure*, Pragmatic Bookshelf, 2009.

Hughes, Cameron and Tracey Hughes, *Professional Multicore Programming: Design and Implementation for C++ Developers*, Wrox, 2008.

Lewis, Bill and Daniel Berg, *Multithreaded Programming With PThreads*, Prentice Hall PTR, 1997.

Mattson, Timothy, *Patterns for Parallel Programming*, Addison-Wesley, 2004.

Miller, Russ Miller and Laurence Boxer, *Algorithms Sequential & Parallel: A Unified Approach*, Charles River Media, 2005.

Patterson and Hennessy, *Computer Organization and Design, The Hardware / Software Interface*, Fourth Edition, Elsevier, 2009.

Robbins, Kay and Steve Robbins, *UNIX Systems Programming*, Prentice Hall PTR, 2003.

Scarpino, Matthew, *Programming the Cell Processor: for Games, Graphics and Computation*, Prentice Hall PTR, 2008.

Sun Microsystems, *Sun Chip Multithreading Servers*, http://www.sun.com/servers/coolthreads/overview/index.jsp, 2010.

Toffoli, Tommaso and Norman Margolus, *Cellular Automata Machines*, MIT Press, 1987.

VanderHart, Luke, *Practical Clojure*, Apress, 2010.

Williams, Anthony, *C++ Concurrency in Action, Practical Multithreading*, Manning Publications, 2011.