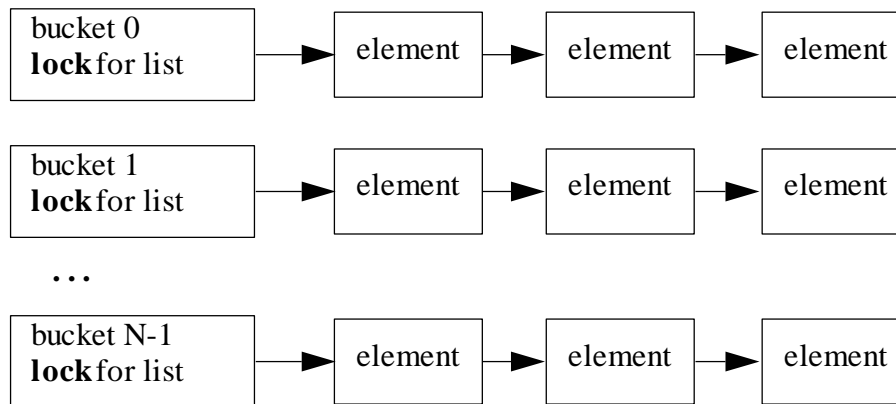


Outline of the Three Multiprocessor Servers from the 2009 Sun Microsystems Grant

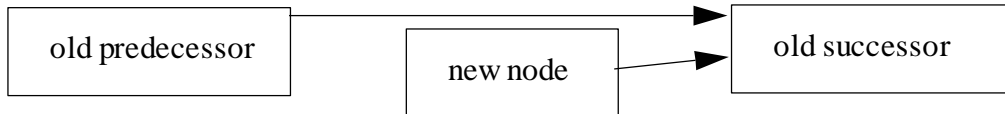
Dale E. Parson, <http://faculty.kutztown.edu/parson>, CSC 402, Fall 2010

A closed address (linked list) hash table can place a lock on every linked list, allowing only one thread at a time into each bucket's list. As long as concurrent accesses are scattered, the probability of colliding on a lock -- which entails waiting -- is low. A *striped* approach to locking allocates a fixed number **NL** of locks shared by one or more buckets, where **NL** does not grow with table growth. A bucket's lock is found by $\mathbf{NL} = \mathbf{bucketNumber} \% \mathbf{NL}$. This approach reduces memory overhead for locks with an increase in likelihood of waiting for a busy lock.



The reason that quicksort and mergesort as coded above do not need locks is because concurrent threads do not access shared data. Each partition of an array in quicksort or mergesort is separate from every other partition, allowing unsynchronized access. The only synchronization occurs when two threads *join* after they complete their work.

Some data sharing does not require the storage overhead and potential waiting overhead of locks. It is possible to avoid locking nodes in a linked list when they are being read, using locks to *serialize* writes to the nodes. Writer threads first act as readers until they find the node(s) that they must mutate. They then acquire a lock on any node to be mutated, re-check to ensure that no other concurrent writer thread has mutated the node in the meantime (e.g., by using a mutation counter in the node), and then perform the mutation. The mutation must be done in a series of steps that will not send readers to invalid nodes. Lockless mutation is easier in a language with garbage collection (Java or Jython), because a reader thread that accesses a deleted node will not crash. (It may access recently stale data -- temporal sequence becomes a partial ordering in concurrent processes.) In C++ if a mutator deletes a departing node, then a concurrent reader might crash when accessing that node. C++ requires reader locks or more complex solutions in such cases.



Insertion adjusts old predecessor's *next* link after setting the new node's link to avoid read locks.

Deletion adjusts the predecessor's *next* link around the node to be deleted, allowing the garbage collector to recover the node after all references (possibly by reader threads) to the node being deleted are gone. This approach requires link updates to be *atomic operations*, requires an atomic *isDeleted* boolean flag on a node, an requires complicated locking when 2 writers collide, e.g., a per-node *mutation counter*. Skiplists are preferable to balanced trees for concurrent threading, because lists support concurrency better. Tree balancing requires a global lock.

This document contains the illustrations of a comprehensive report. Contact the author if you would like the complete report.

Table I is a summary of the specifications of the three multiprocessor servers provided by Sun. All three servers are 64-bit architectures. Currently Harry and Hermione are available for computer science faculty and students within the Kutztown University network. Ron will become available again after an anticipated move of the servers to a site with increased electrical service. 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.

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

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

Figure 1 is a block diagram of the UltraSparc T2 processor architecture found in Harry. 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.

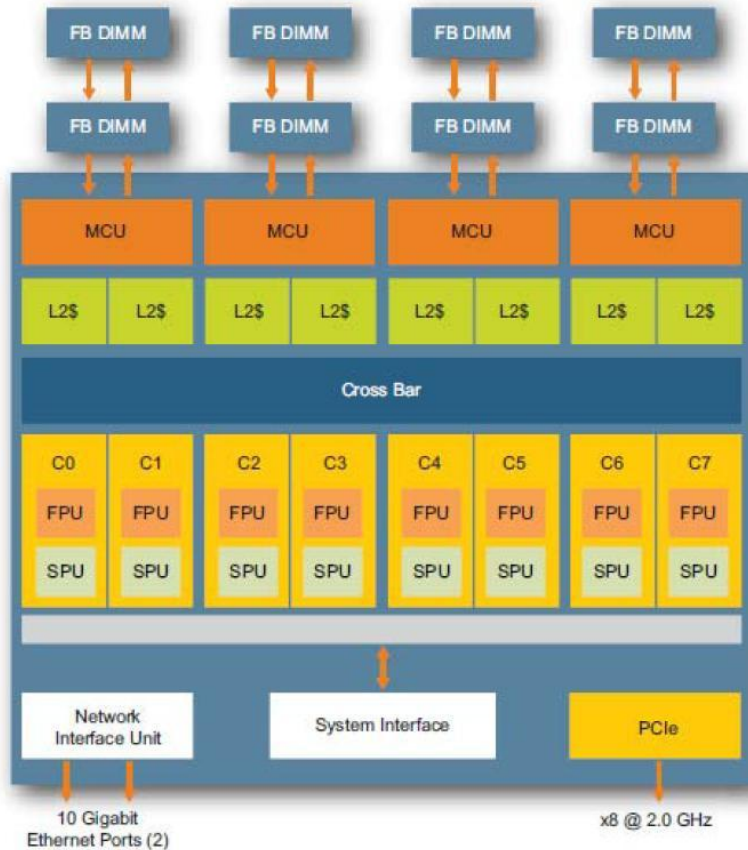


Figure 1: The UltraSparc T2 Processor Architecture in Harry

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. 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.

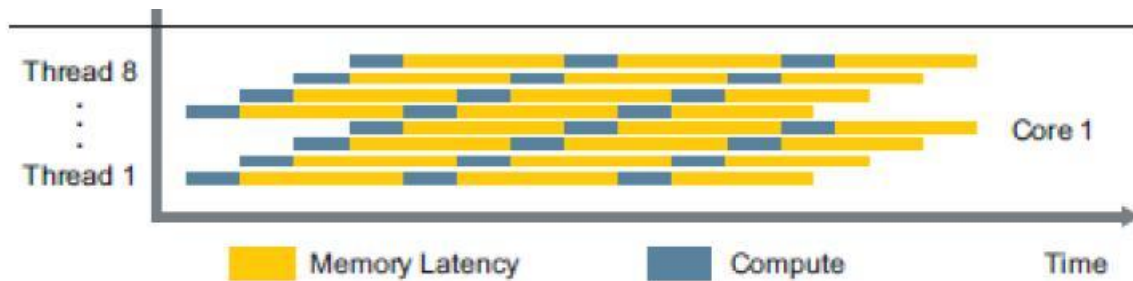


Figure 2: T2 thread parallelism amortizes memory latency across instruction streams

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.

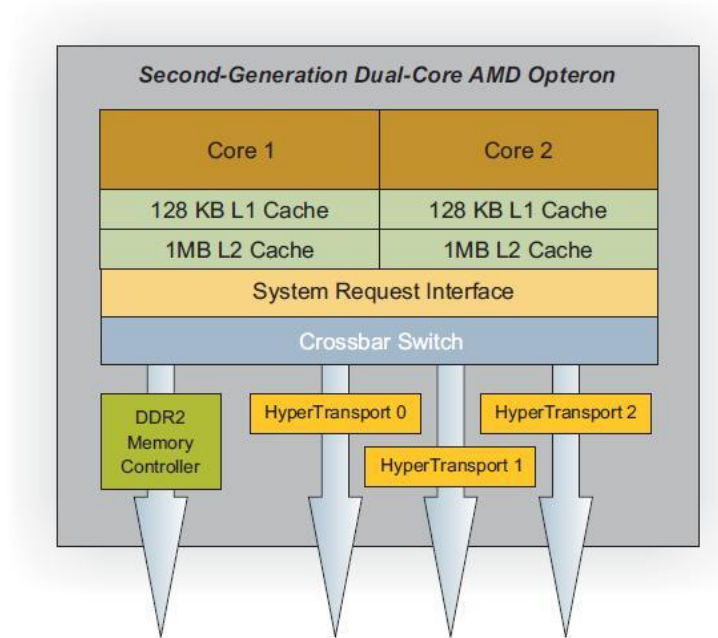


Figure 3: The Dual-core AMD Opteron Processor Architecture in Hermione

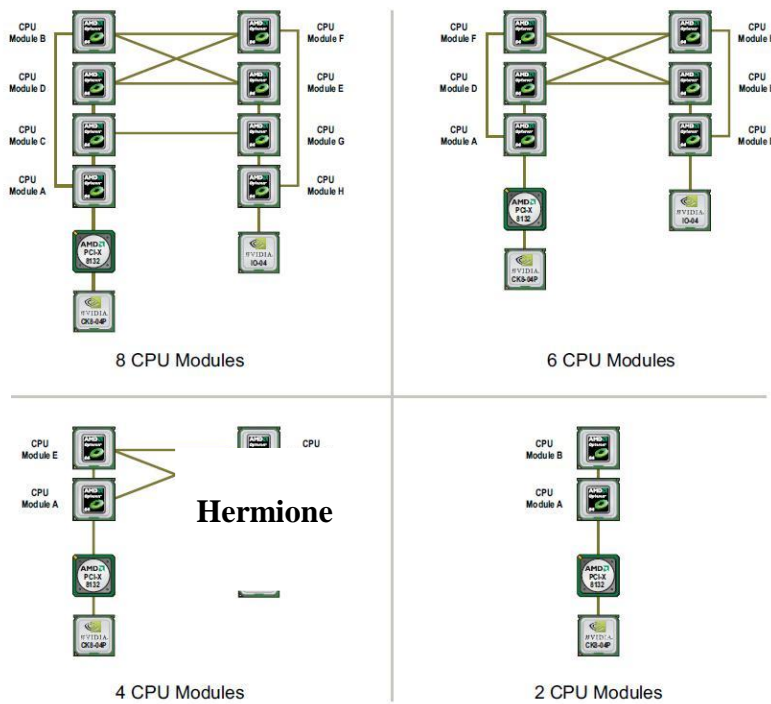


Figure 4: “Enhanced Twisted Ladder” X4600 Multistage Interconnection Topologies

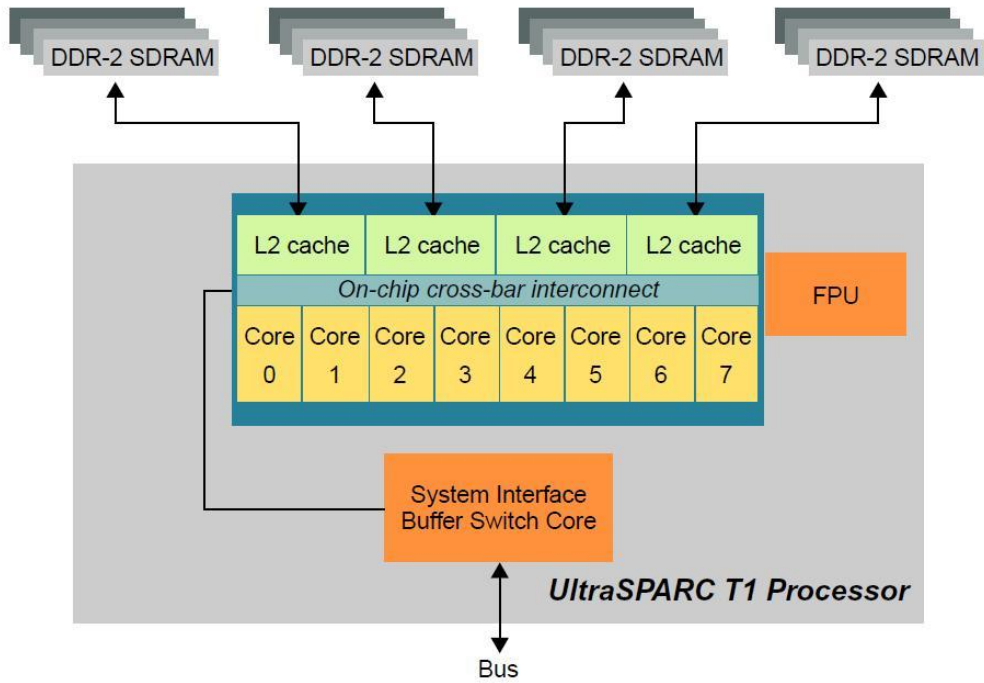
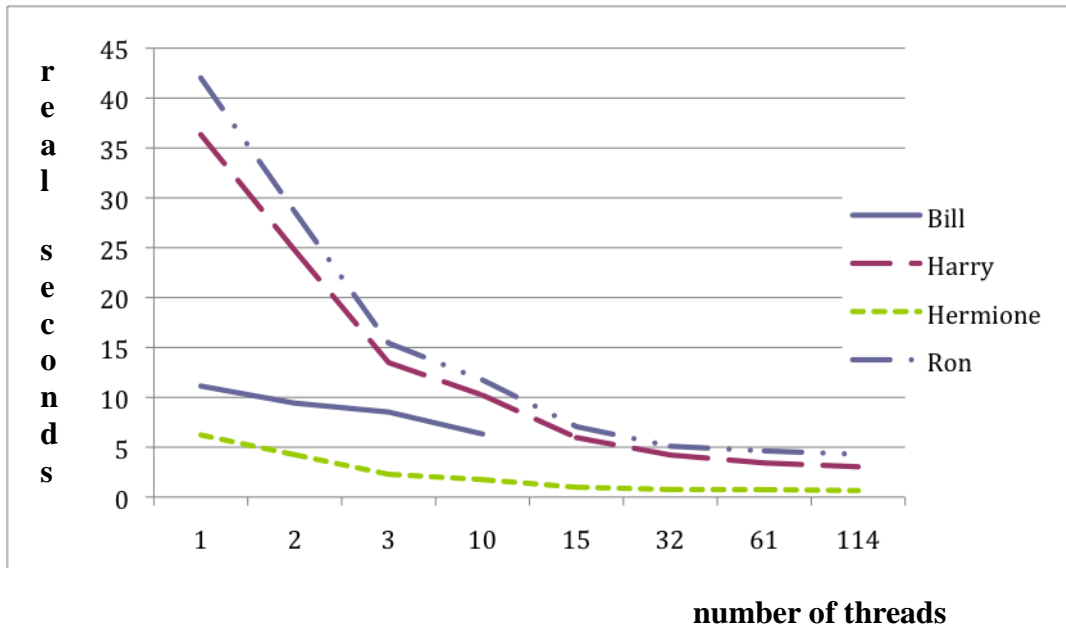
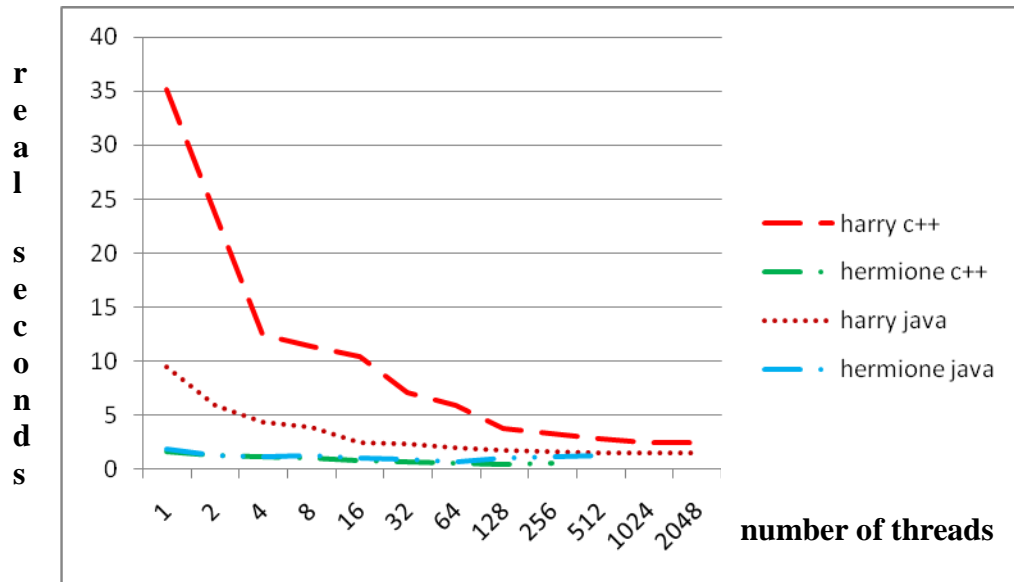


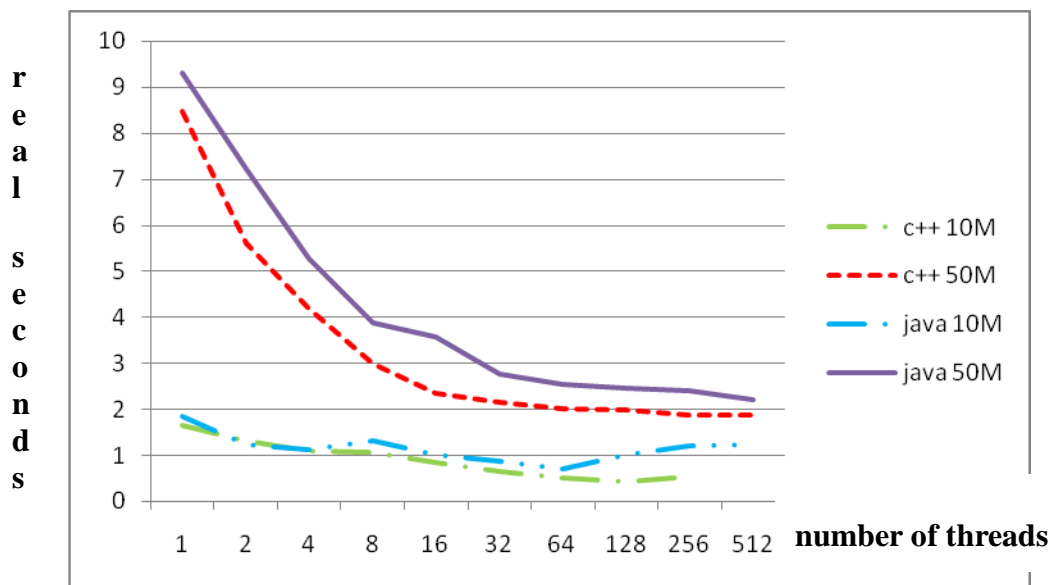
Figure 5: The UltraSparc T1 Processor Architecture in Ron



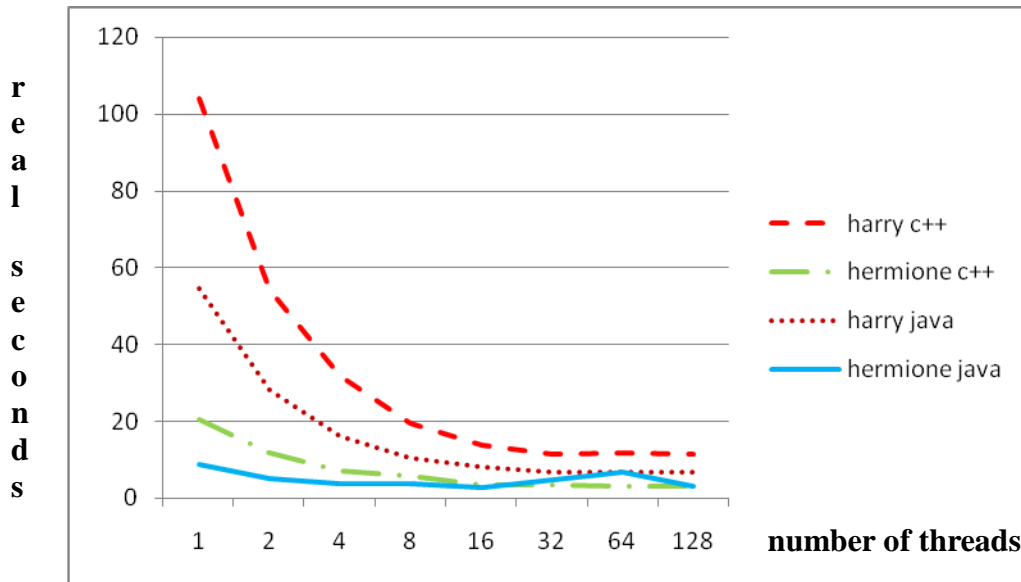
Graph 1: Multithreaded (threshold) quicksort run time as a function of thread count



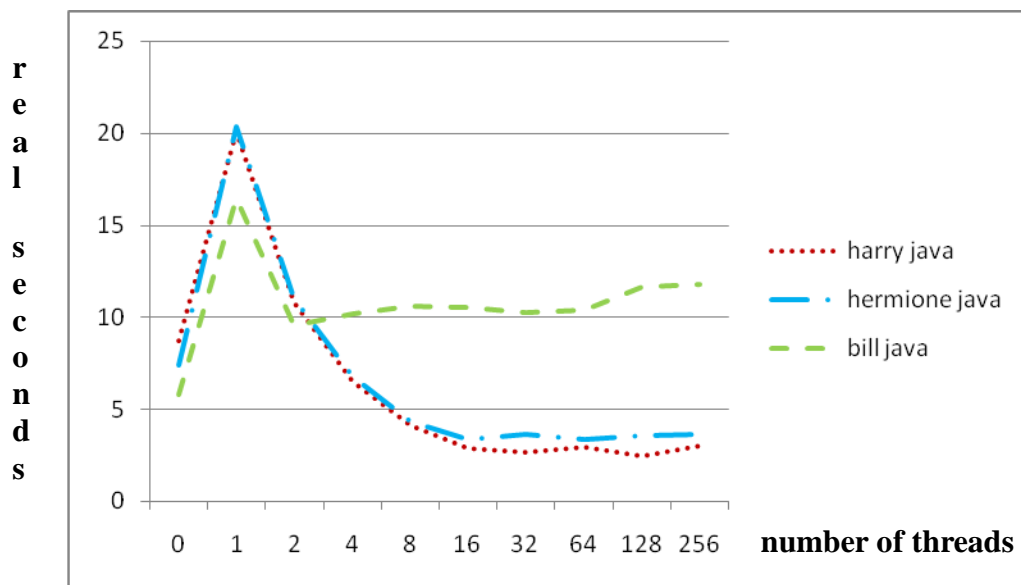
Graph 2: Multithreaded quicksort with explicit thread count on command line



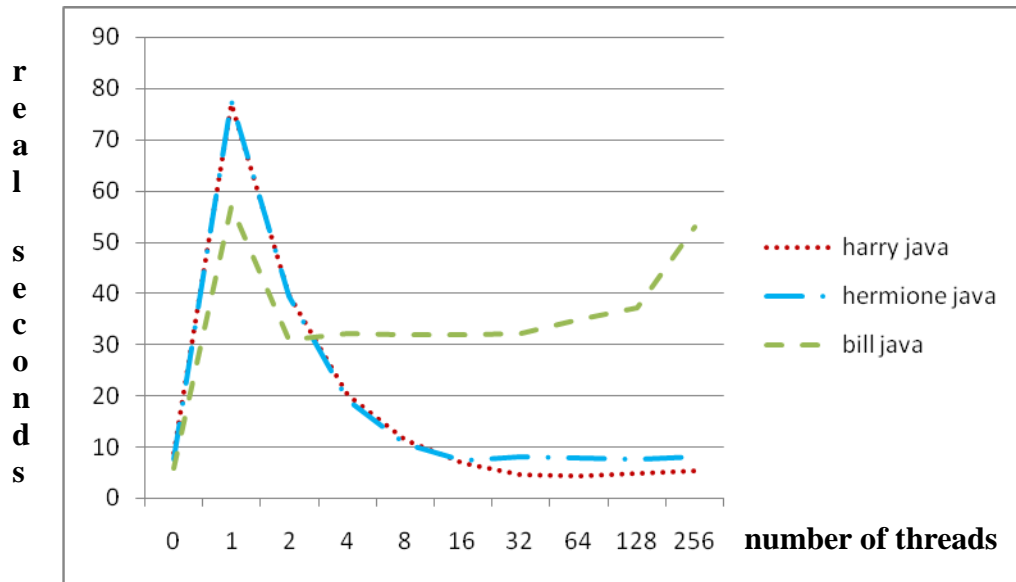
Graph 3: Quicksort of 10 and 50 million integers on Hermione using C++ and Java



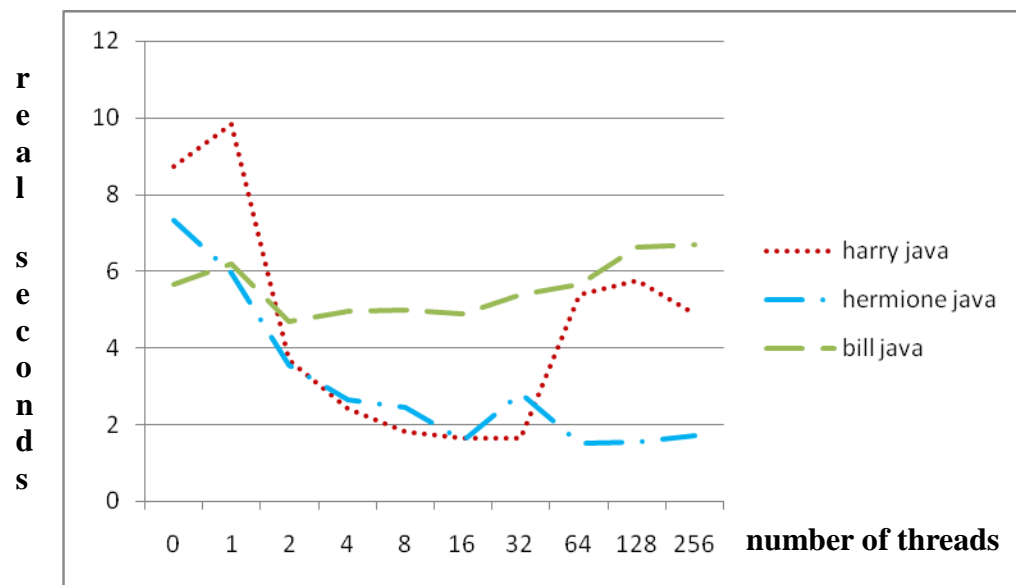
Graph 4: Multithreaded mergesort with explicit thread pool size on command line



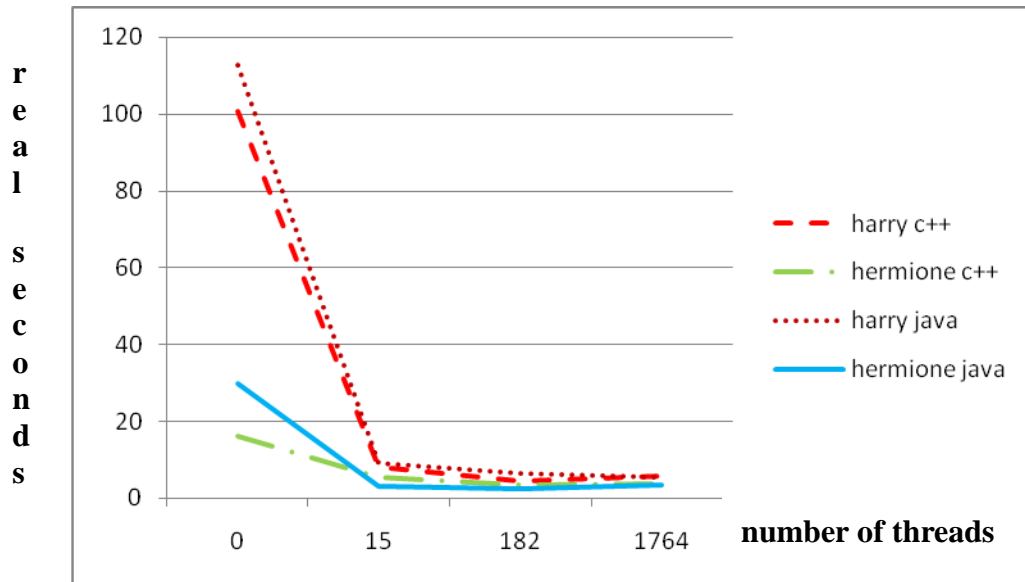
Graph 5: Hash set 1:10 insert:lookup ratio using Herlihy/Shavit striped linked hash table



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



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



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

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

OpenMPBench is an open source multiprocessor benchmarking project. 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