Performance Consistency on Multi-socket AMD Opteron Systems

TR-08-07

Allan Porterfield
Rob Fowler
Anirban Mandal
Min Yeol Lim

December 5, 2008

Performance Consistency on Multi-socket AMD Opteron Systems

Allan Porterfield Rob Fowler Anirban Mandal
Min Yeol Lim
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill NC
akp,rjf,anirban,mylim@renci.org

December 5, 2008

Abstract

Compute nodes with multiple sockets each of which has multiple cores are starting to dominate in the area of scientific computing clusters. Performance inconsistencies from one execution to the next makes any performance debugging or tuning difficult. The resulting performance inconsistencies are bigger for memory-bound applications but still noticeable for all but the most compute-intensive applications. Memory and thread placement across sockets has significant impact on performance of these systems. We test overall performance and performance consistency for a number of OpenMP and pthread benchmarks including Stream, pChase , the NAS Parallel Benchmarks and SPEC OMP. The tests are run on a variety of multi-socket quad-core AMD Opteron systems. We examine the benefits of explicitly pinning each thread to a different core before any data initialization, thus improving and reducing the variability of performance due to data-to-thread co-location. Execution time variability falls to less than 2% and for one memory-bound application peak performance increases over 40%. For applications running on hundreds or thousands of nodes, reducing variability will improve load balance and total application performance. Careful memory and thread placement is critical for the successful performance tuning of nodes on a modern scientific compute cluster.

1 Introduction

Today's high-performance computing systems, ranging from small blade centers to the largest supercomputers, tend to be clusters. The compute nodes in these systems are increasingly multi-core, multi-socket commodity servers and blades. Currently, on AMD-based systems, it is common for each of node to have 2 or 4 processor chips and each of the processor chips to have either 2 or 4 cores. Thus, each node is a coherent Non-Uniform Access Memory (NUMA) system with between four and 16 cores. The user can ignore the shared memory resource and program in a distributed memory model, like MPI [7, 8], or try to take advantage of the shared memory model and program in OpenMP [13, 14] or a PGAS language. The NUMA aspects of the system must be taken into account to maximize the performance of threads sharing a memory system. Although a model of the system hierarchy can guide *a priori* design decisions, this usually involves iterative tuning experiments.

If the performance of running a particular version of an application on a particular input is repeatable, *i.e.,* the time to completion as very little variance, then each iteration of the tuning cycle (change code, run, analyze) can be done with a single run of the program. If execution time varies from run-to-run by as little as one percent, then multiple runs will be required to build an adequate statistical model of the the effects of the code change. Even if the process is automated to reduce the labor and tedium involved, the process becomes a lot slower. If tests are being done on a parallel machine, it also becomes expensive. The complexity and cost of the tuning activity is therefore greatly reduced if execution times are repeatable from run-to-run and consistent between systems that are basically the same.

In particular, any variability in time between two runs should be should be caused by changes to the program, not by decisions made by system software. NUMA processors are especially sensitive to memory layout and by affinities between threads and data. Performance on a NUMA systems is impacted by not only distance to the remote memory (hence, latency), but often depends on the memory references being evenly spread between the memory resources. While past large systems interleaved data across all the memory banks in the system, modern microprocessor-based NUMA systems have multiple memory controllers that are generally configured to offer either separate physical memory spaces, or coarse interleaving at the virtual memory page granularity. Operating system and runtime memory and thread placement decisions can greatly effect overall performance, and any probabilistic variation in the outcome of these decisions will introduce a large variation in execution time. In turn, this variability will greatly increase the cost and complexity of performance tuning.

Performance variability can also adversely affect load balancing and limit parallel efficiency. In a statically partitioned application, it is reasonable to expect that equal amounts of work should result in equal execution times, but variability will invalidate this assumption. In adaptive applications with dynamic load balancing, non-repeatable execution times can confuse and defeat the load balancing protocol.

Unfortunately, the experiments reported in this paper indicate that default Linux memory and thread placement decisions on multi-socket systems can result in programs whose performance varies significantly between identical executions. The Linux thread scheduler does a good job of moving threads around to even utilization between cores, but the initial binding of threads to cores is often imbalanced and this can lead to inappropriate memory allocation and placement decisions. Specifically, a “first touch” allocation strategy places each page in the memory associated with the core from which the first page fault to the page is generated. If this allocation occurs before the thread migrates to its final “home”, those pages remain under the memory controller at the initial thread location. If there is an initial thread-scheduling hotspot, thread/process migration balances the CPU load, but the memory hotspot persists.

Due to the probabilistic nature of scheduling events, the degree of the problem varies from run-to-run. If one is able to pin threads/processes to specific cores before the data pages are touched, overall performance is improved and the variability is greatly reduced. This results in a sub-optimal placement.

The impact of pinning depends on program nature and structure. A completely cache-resident, compute-

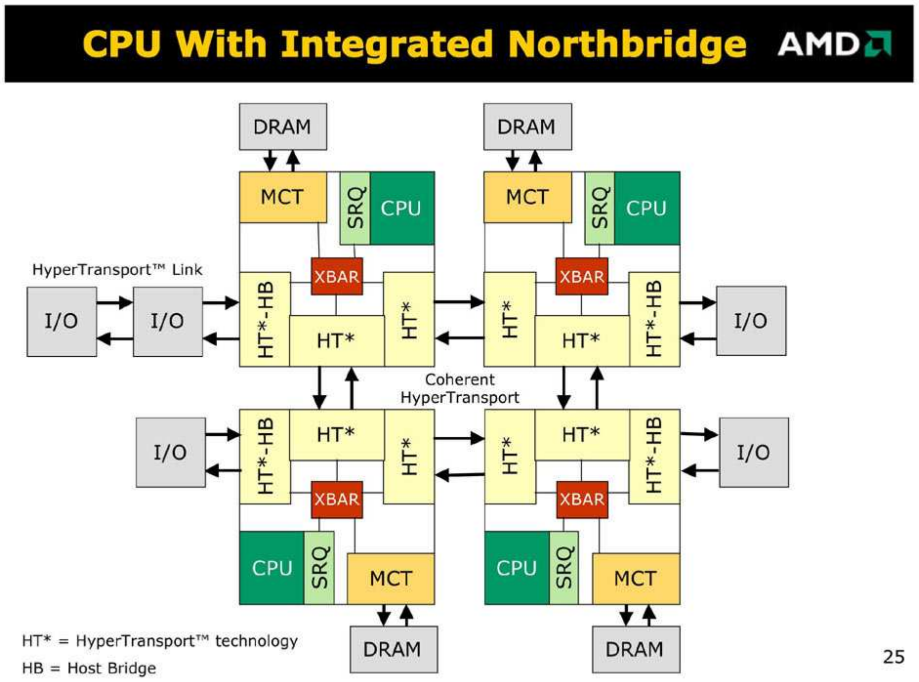


Figure 1: AMD Multi-Socket Architecture

bound job, will be minimally impacted by memory and thread placement. As long at the threads do not move, the data will be brought into cache and repeatedly used. The loading of the cache may take more time, but will be insignificant for a compute-bound job. Memory-bound jobs should be impacted much more by poor thread and memory placement. These jobs spend most of there execution moving data though the memory system and poor placement will definitely result in higher latencies. If the load is sufficiently high, the hotspot at the memory controller becomes a bottleneck at which memory requests are queued.

2 NUMA Multi-socket Multi-core Nodes

With the introduction of the coherent HyperTransport (HT) by AMD, it has been possible to construct multi-socket systems where each processor can access all of the memory attached to the system. Intel has introduced QuickPath with the Core i7, and the plans are to allow multi-socket systems using it in the future. AMD multi-socket systems are currently available and all tests in this paper are on those systems. The Operating System and runtime issues effecting memory and thread layout should apply equally to the Intel systems, although increases in the connection speed and increasing the number of available links should allow greatly improved performance.

In Figure 1, we have a picture of the interconnect using HT from the ExtremeTech web site. In it, four sockets are connected with a HT ring. The path from a core to the DRAM connected to the same socket is obviously shorter and therefore lower latency than the path to any of the other DRAMs. There are multiple potential sources of bottlenecks in the system. A DRAM is connected to one memory controller (MCT) which is connected by single port to the on-socket crossbar (XBAR). Memory requests to the diagonal socket must cross two HT links and cross though a total three XBARs one way. Any of these may saturate and affect performance. Multi-socket AMD system effectively resemble Cache-Coherent NUMA (ccNUMA, normally just abbreviated NUMA).

Differences in the latency to access various memory modules can come from many source.

- Latency - distance (number of hops) to access particular memory
- Memory Controller - memory load on each banks memory controller
- Memory Concurrency - the maximum number of outstanding memory references allowed by a core or socket
- HT Traffic - load on each link in the HT connection

Each one of these can limit the memory performance observed by an application. Latency is a common reason for programming data placement on SGI systems[6], but the AMD NUMA nodes may see differing bandwidth/latency to memory because of any one (or multiple) of these.

Programming NUMA systems in the 90's [20] found that programing has shown that memory placement to reduce latency was critical. With incorrectly distributed memory, latencies are too great for the hardware to mask. Multi-sockets nodes have several serialization points which can not handle all of the traffic produced by several sockets. Good load balance across each of the components will be important to overall memory access bandwidth and latency. With the introduction of multi-socket nodes and systems, the number of programmers and applications exposed to NUMA problems will increase greatly.

3 STREAM Performance Inconsistency

Running the STREAM [11] benchmark on a multi-socket AMD Opteron system, significant performance variations were observed. To better understand the performance, a large number of copies of STREAM on 6 available AMD multi-socket systems focusing on the performance variability.

The STREAM performance reported here is not the maximum possible values for the AMD Opteron processors or even the systems tested. We examined out of the box reproducible results, rather than trying to optimize. We think this is closer to the original purpose of estimating the bandwidth available to a "typical" application. STREAM was compiled with gcc version 4.1.2 on the blades (LP, QB1, QB2) and gcc version 4.2.3 on the workstations (WS1, WS2, WS3). The compile options used were '-O3 -g'. Searching for faster compile options or compiler (PGI, PathScale) with more aggressive optimizations would produce higher numbers. Our interest is not the peak numbers but the variability between executions. Performance results from the AMD web site (Oct 08)[3], suggests that peak performance may be increased by 10-15% over what is presented here.

WS1 is a Dell PowerEdge T605 with 2 AMD 2.1GHz Operton processors and 2GB (4x512MB) main memory. It has Ubuntu Linux loaded and with a stock 2.6.25.9 kernel from kernels.org, with the perfmon2 hardware counter extensions. WS2 is the same system with twice the memory 4GB (8x512MB). WS3 is also a Dell PowerEdge T605 with slightly faster 2.2GHz processors with 16GB (8x2GB) of main memory running the same version of Linux.

In addition to the workstations, several blades were tested. LP is a Dell PowerEdge M600 dual-socket blade with 2 low-power 1.9GHz AMD Opteron processors. It has 16GB (8x2GB) main memory installed. It has CentOS loaded with a stock 2.6.26.2 kernel with the perfmon2 hardware counter extensions. QB1 is a Dell PowerEdge M905 quad-socket blade with 4 2.3GHz AMD Opteron processors. QB1 has 48GB (24x2G) main memory. It uses the same version of Linux as LP. QB2 is the same quad-socket system with only 32GB(16x2G) main memory installed.

STREAM was run 25 times with 1, 2, 3 or 4 threads on a single socket of each configuration[1]. Figure 2 shows the performance and variability. When only one socket was active the variation between results was small, with the difference between the maximum time and the minimum time of only 1.2%(WS1 - 2 threads) and a standard deviation of less than .32% (WS1 - 2 threads).

With 2 sockets active, the results show more variation between runs. The Linux command `set_affinity` was used to direct the threads to specific cores on individual sockets. It doesn't force one thread per core, but does prevent the threads running on any cores not specified. Figures 3 shows the minimum, average and maximum value, for the each configuration. The number of threads running is kept even across sockets. When more than one thread is running on each socket, the difference between the minimum and maximum observed value ranged between 33% and 95%. The mean value tends to be closer to the minimum than the maximum. Poor performance is likely to occur. For the blades, but not the workstations, only having one thread on each socket resulted in consistent results. Whether this is a function of the different motherboards, the different Linux kernels or something else is unknown at the present time.

For the quad-socket blades (QB1,QB2) having threads active on 3 or 4 sockets, figures 4, 5, are even less consistent. The difference between the minimum and maximum values are as high as 123% and is always above 24% (even when running one thread per socket). The maximum and average performance curves are not consistent. For QB1 using four sockets and 16 threads the average goes down while the maximum is rising. The minimum performance is consistent across both configurations and any thread count.

Tuning any program with this run-to-run performance behavior will be difficult. Either the performance problem needs to be resolved or a very large number of runs is going to be required to build valid statistics. On a cluster with hundreds of nodes, this performance will make good load balancing (nearly) impossible. One (or more) threads will always be running slow and all of the fast threads will wait for it at the next global barrier. Our first question was - What is happening?.

[1]The Linux kernels/BIOS used remap physical cores numbers to sockets during boot. Care was exercised to guarantee that threads ran on the appropriate socket.

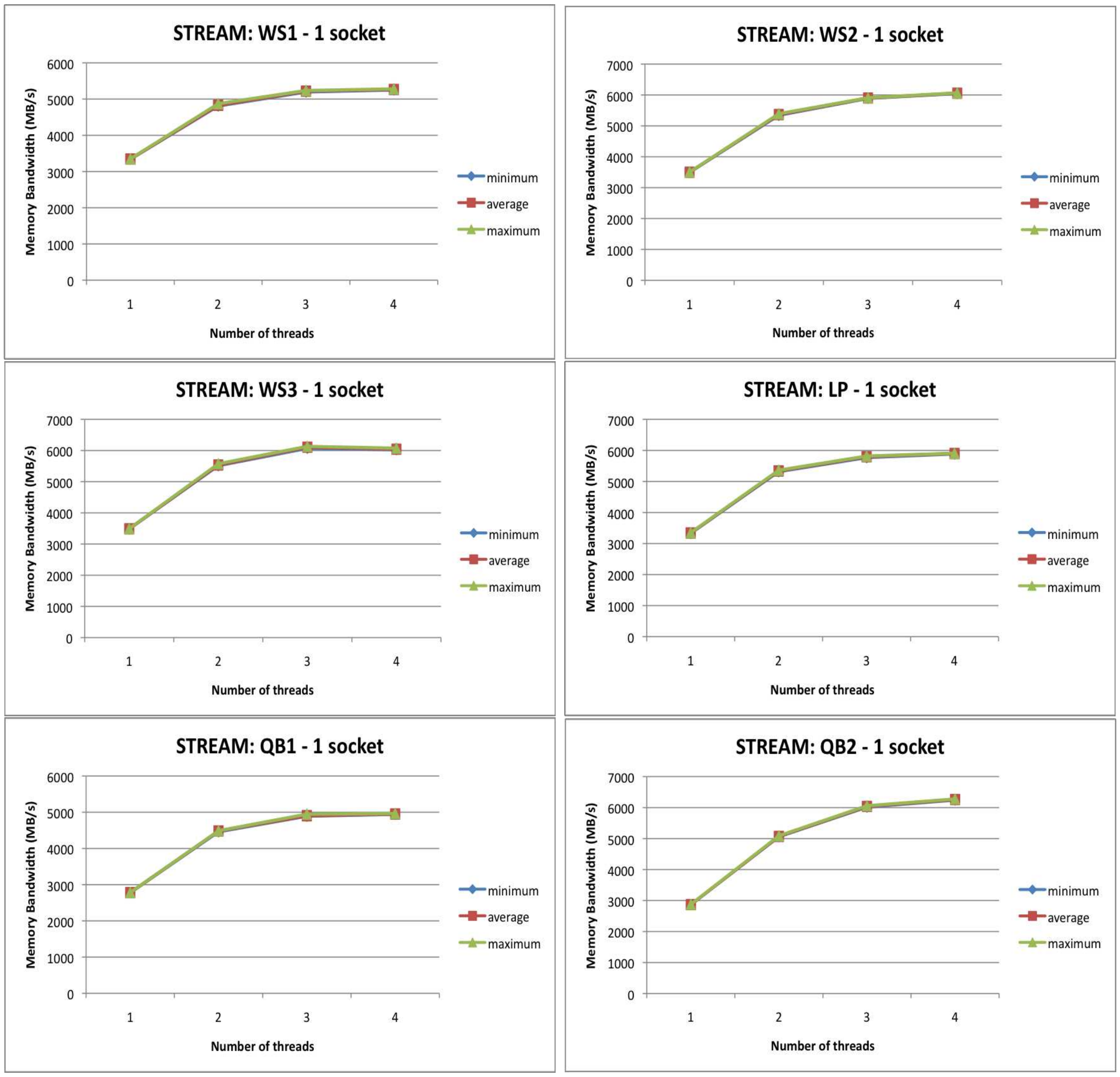


Figure 2: STREAM Performance on one socket

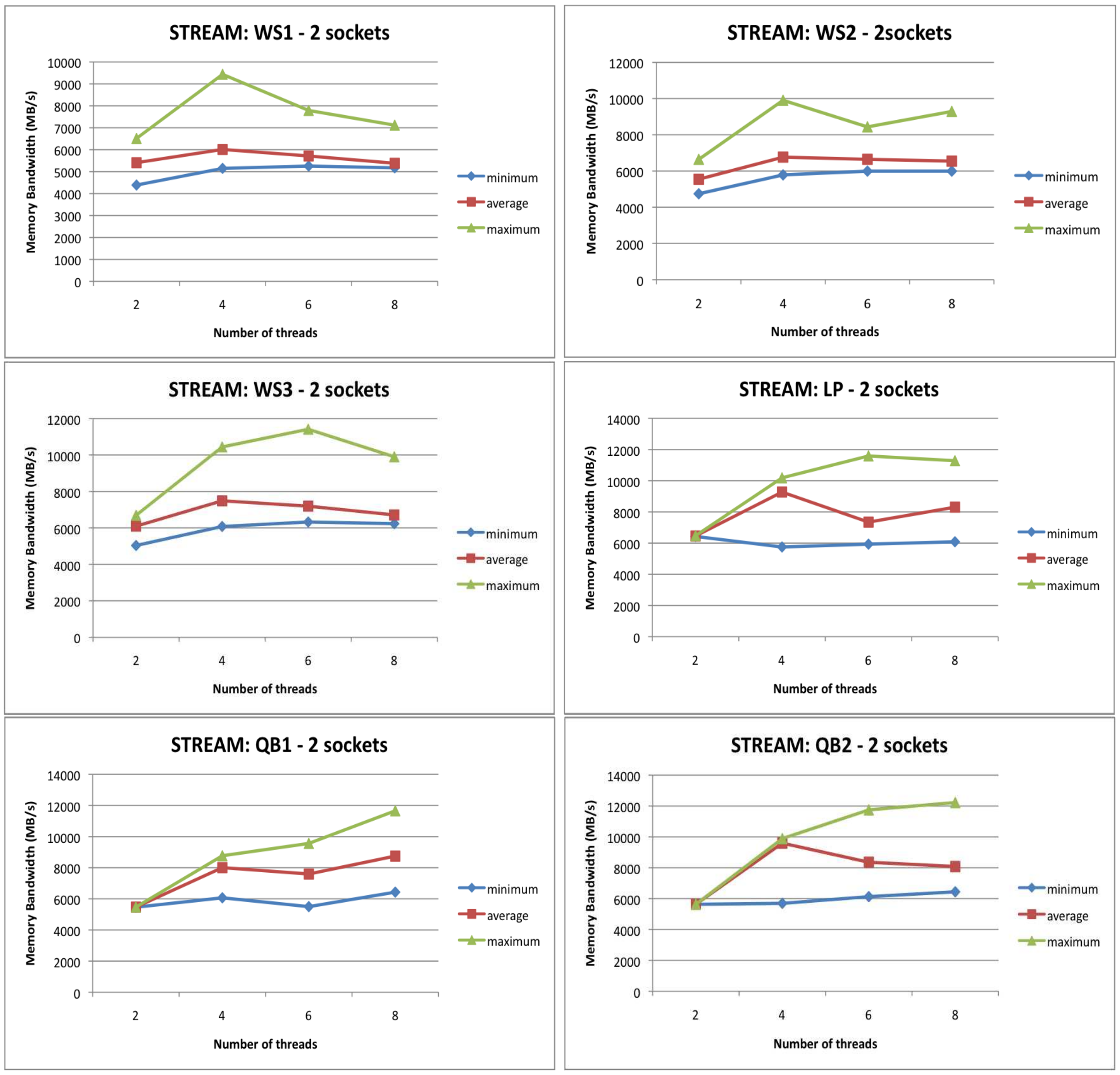
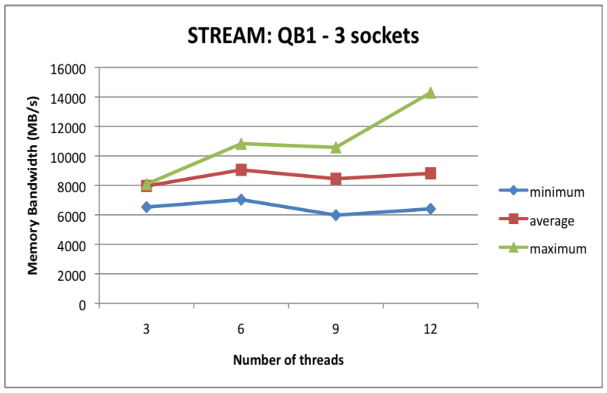


Figure 3: STREAM Performance on two sockets



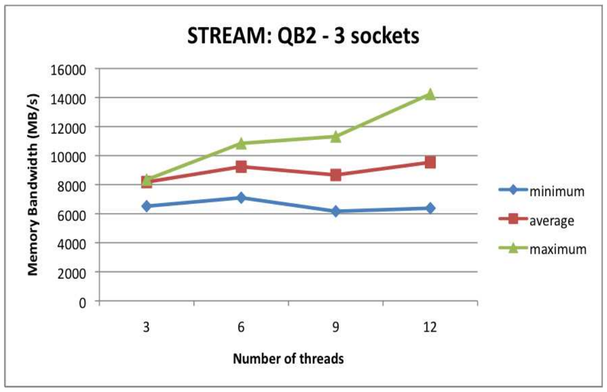
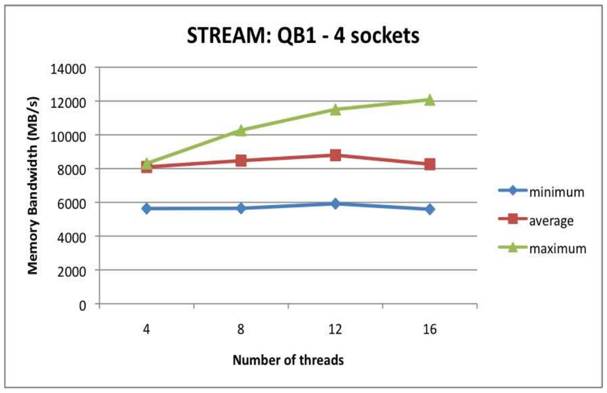


Figure 4: STREAM Performance on three sockets



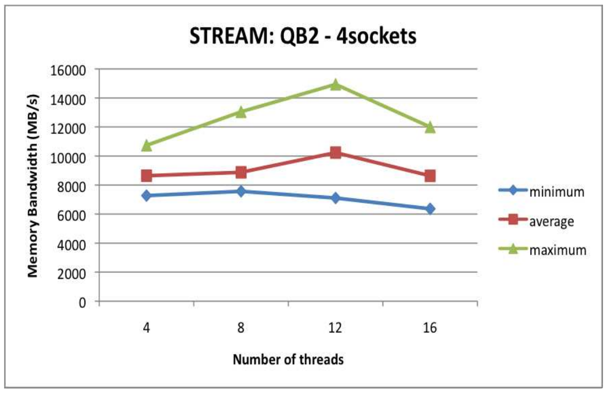


Figure 5: STREAM Performance on four sockets

When we executing STREAM 2 cores and 2 sockets under the `pfmon` hardware counter monitor (Figure 6), we see that the L3 cache misses for the two cores are within 0.2% of each other, but resulting memory accesses are heavily slanted towards one memory bank (2467 to 1 towards CPU0). Poor performance is resulting from poor memory allocation. Almost all of the memory actually being accessed is in one of the two memory banks. Accessing that memory is a severe bottleneck.

How does this poor allocation happen? During other testing, the NAS Parallel Benchmark OMP suite member SP (CLASS=B) was instrumented and run on the same configuration. The graphs in Figure 7 were created by at every OpenMP control point recording the time, the current thread number and the current CPU number. Figure 7(a) is from the fastest execution and Figure 7(b) from the slowest. The threads do not move for the bulk of the execution (only one swap in either after the first couple of seconds). The first 2 seconds of both graphs are blown up (right-hand sides), and much movement is evident. In the fast case, 4 cores are active at time 0 and 6 by 0.1 second and all 8 at about 0.4 seconds. The threads are fixed on threads in less than 1/2 second. In the slow case, only 3 cores are active at time 0, 5 at about 0.3 second, 7 at 0.5 second, and the last does not start until about 1.3 seconds into execution. The fast case has threads move 5 times, the slow version has threads move 11 times. It is clear that threads remain in one place after

```
CPU0                      61597365709 CPU_CLK_UNHALTED
CPU0                       1252786946 L3_CACHE_MISSES:ALL
CPU0                       3318429026 DRAM_ACCESSES_PAGE:ALL
CPU1                      71580890154 CPU_CLK_UNHALTED
CPU1                       1254950288 L3_CACHE_MISSES:ALL
CPU1                          1344951 DRAM_ACCESSES_PAGE:ALL
```

Figure 6: `pfmon` output for 2 cores of STREAM executing on 2 sockets

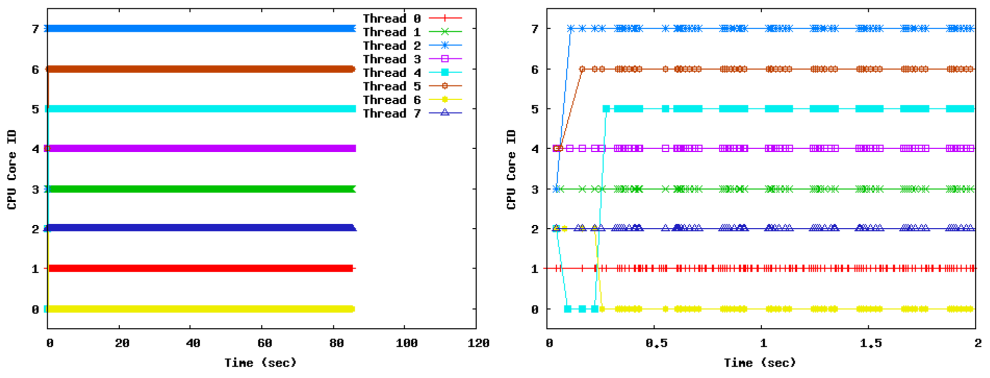
an early scramble. When Linux creates the threads, it must decide on which core to place them. The first thread goes on the most idle thread in the previous scheduling quanta. When later threads show up before the next scheduling quanta, they see the same core as the most idle and also schedule themselves there. By not taking into the load already scheduled, some cores are overloaded. Placement of jobs within a workflow has a similar problem [10]. Linux quickly realizes some core are overloaded and spreads the threads out to one per core after which the threads are not moved again. The problem is that during this initial phase, the program is doing data initialization. The Linux memory allocator uses a first-touch policy to determine memory placement. Since many the threads start on one core, most of the memory is allocated close to that core and the huge memory access mismatch observed by hardware counters potentially occurs.

4 Thread Pinning

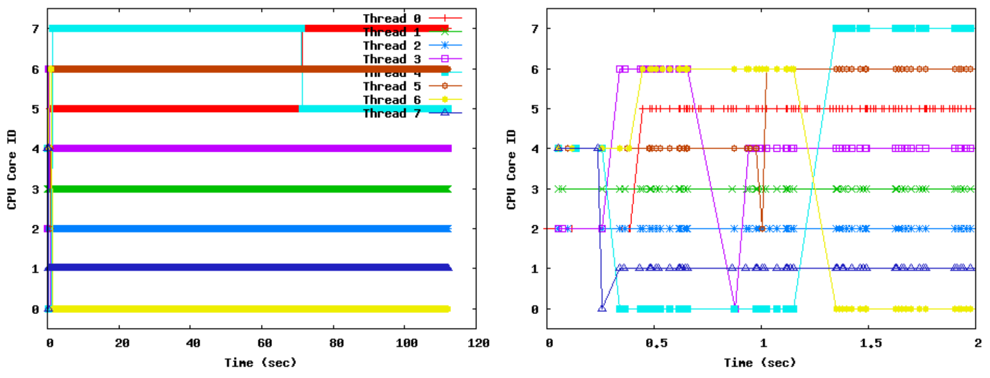
Multi-socket, multi-core performance is dependent on good memory performance. Good memory performance is dependent on good memory placement. A first-touch policy for memory placement is common. As a consequence, many applications are careful to have threads allocate and initialize their own memory. The initial thread migration for computational load balancing defeats this. Making sure that Linux places allocated memory correctly is important to any performance tuning effort.

Since, as previously noted, threads are initially allocated in an unbalanced way, part of the process of pairing a core and a particular thread will be to evenly distribute the threads over the allowed cores. Linux (versions 2.6+) have introduced several functions that can be used to facilitate this process. `sched_setaffinity()` allows a program to specify which processor ids it can execute. `sched_getaffinity()` allows a program to see what cores it is allowed to execute. We wrote a function that reads the currently allowed affinity and assigns the n^{th} thread to the n^{th} available core. For system with only one node, the OpenMP thread number or the pthread number can be used as the thread number. For multi-node systems some means generate thread numbers starting at 0 to the number of threads in memory sharing domain will be required. That function, `pin_thread()`, was placed in a library where it can be added to any program.

The linux command `taskset` does not make the allowable affinity visible to the user using `sched_getaffinity()`. A program running on a number of core limited by `taskset` will see the entire system as available. At NC State, Tyler Bletsch has produced a second version, `cpuaffinity`, that does set the system affinity in a user visible manner and is used in all of the tests reported here.



(a) SP.B - execution time 85 second



(b) SP.B - execution time 111 second

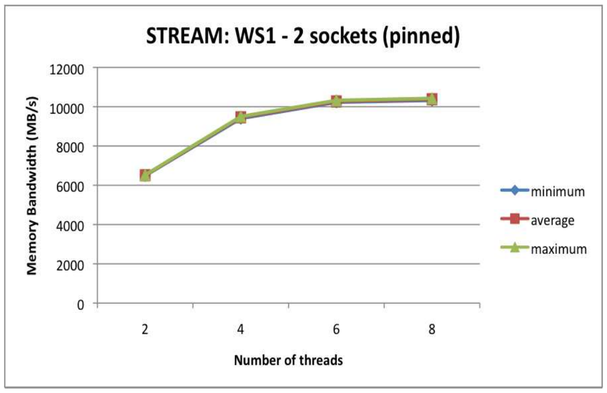
Figure 7: NPB OMP SP.B thread id/core id mapping

4.1 Pinned STREAM Performance

We modified version 5.8 of the STREAM benchmark was to include a new OpenMP parallel section, which calls `pin_thread`, after the number of threads is printed but before any initialization. This pins each OpenMP thread to a different core before any data is touched.

The effect of this simple pinning of threads to cores before initialization is noticeable. Figure 8 shows the same minimum, maximum, and average values over 25 runs on both sockets of WS1 as in section 3. On the right-hand side is the average performance for both pinned and unpinned. The average performance for 1 thread is 8% better for the pinned version. With 4 cores are active the pinned version is 92% faster.

Performance variability is also effectively eliminated on the other systems tested. Figures 9 and 10 shows the average case improvement on each of the systems. For systems running a large number of threads, 9+, every pinned thread test reported noticeably higher bandwidth than any test of the original version. The more cores and sockets available, the larger the chance of bad decision, the more opportunity for improvement. Equal banks sizes potentially helps performance significantly (10% for 3 socket case with 9



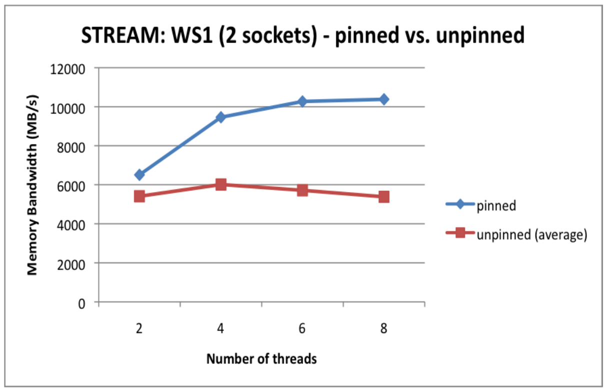


Figure 8: Pinned and UnPinned STREAM Performance on two sockets of WS1

concurrent references). For in depth study of AMD multi-socket memory performance see [19].

Pinning the OpenMP threads to specific cores before data initialization improves average bandwidth and greatly reduces the run-to-run variability in the results for the STREAM memory benchmark.

The foremost component of the performance variability is not the NUMA latency to reach a remote memory but is caused by unbalanced memory allocation. Which causes some portions of the memory system to be overloaded and serialize their transactions. Although the difference in latency between memories is relatively small, the effective bandwidth of each portion of the memory system is not designed for use by more than one socket. NUMA-like memory scheduling decisions need to occur to reduce memory imbalances.

5 Other Benchmarks

With pinnings success at reducing performance variability and actually increasing STREAM peak performance, we looked a number of other pthread and OpenMP benchmarks. These included pChase , a pthreads memory benchmark, the NAS NPB, OpenMP high performance computing benchmark and the SPEC OpenMP 2000 benchmark.

5.1 pChase

pChase is a memory performance benchmarks suite which provide both the latency and the bandwidth of different access patterns, for cache and main memory [15, 9] and has been used on a number of IBM systems [16, 18, 17] to accurately model memory performance. pChase among other values, provides the the number of memory references which were required to support any obtained performance level. An

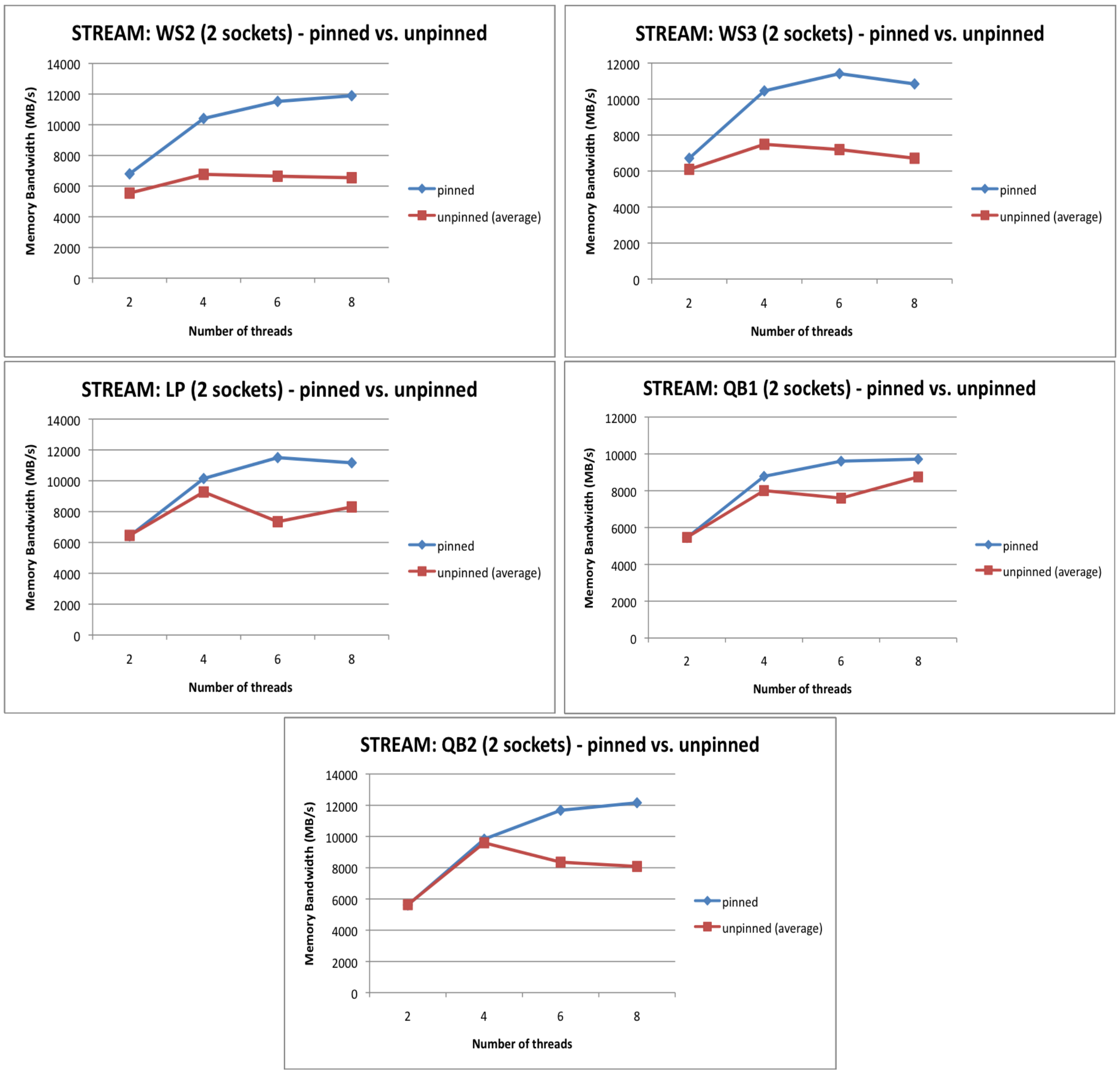


Figure 9: UnPinned and Pinned STREAM Performance on two sockets

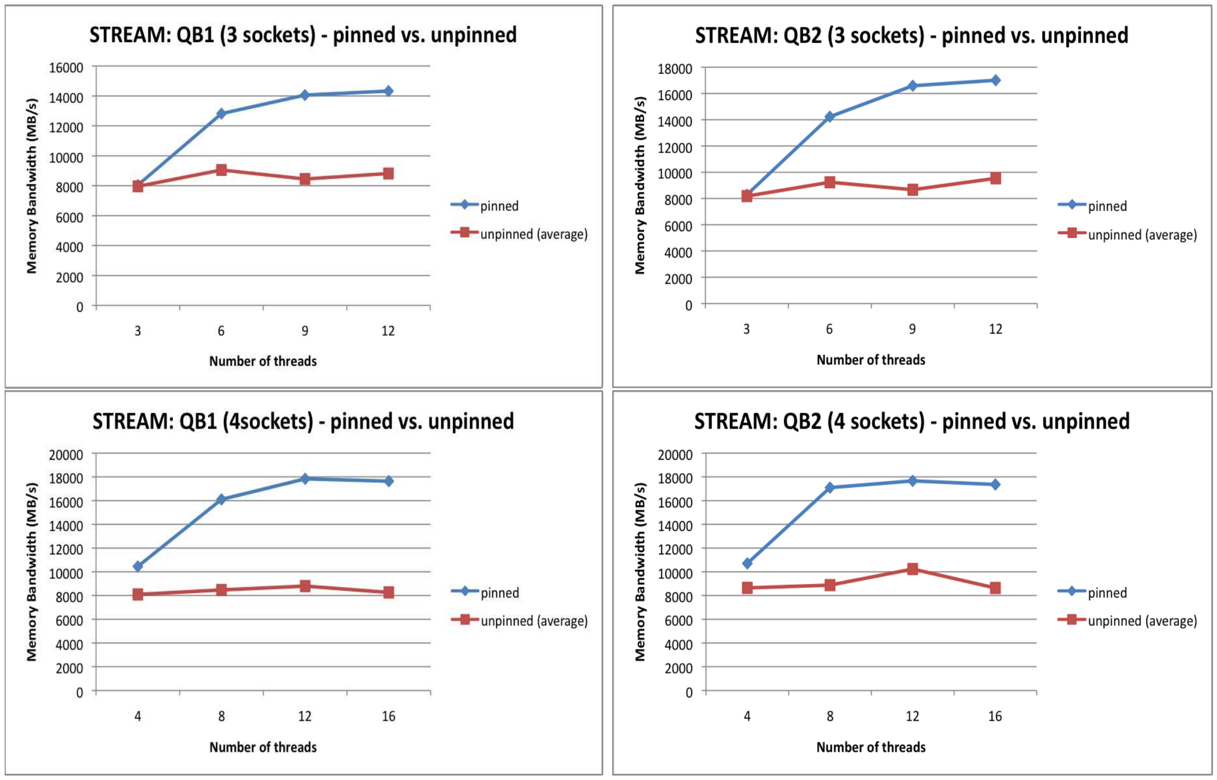


Figure 10: UnPinned and Pinned STREAM Performance on three and four sockets

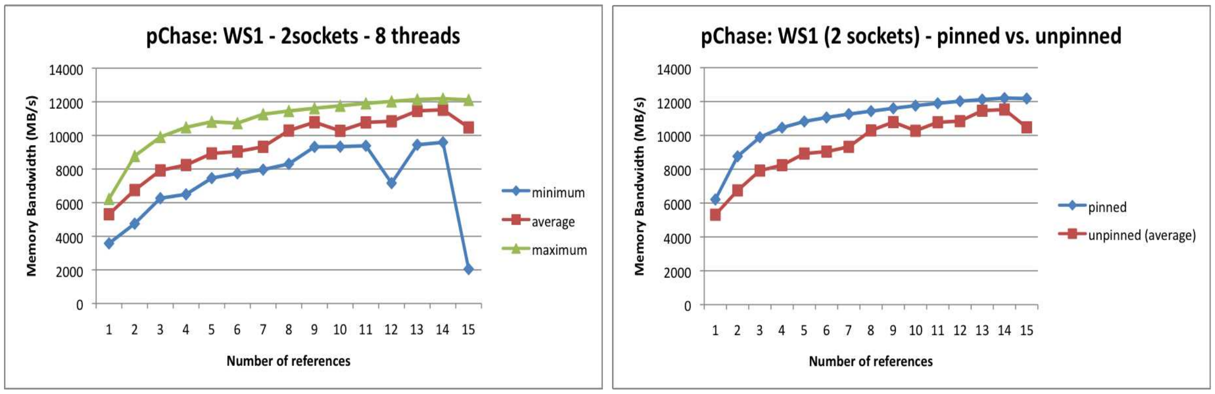


Figure 11: UnPinned and Pinned pChase Performance on two sockets

experiment using pChase consists a series of executions varying one parameter. Performance consistency between executions is critical to understanding the results.

As another memory benchmark, it is not surprising that the results, Figure 11 comparable to those observed by STREAM, but slightly better. The figure shows the improvement in average performance with 8 memory references active. Other performance graphs show this to be approximately the effective to maximum performance on AMD Opterons [19]. Pinning pthreads seems to be just as important as pinning OpenMP threads for memory bound programs. Memory bound programs must arrange their memory accesses evenly across the available hardware resources to take advantage of all of the bandwidth available.

5.2 NAS NPB Suite

Thread and data placement was expected to have a significant impact on memory benchmarks. What is the impact for a computationally intensive application?

To better understand that question, we studied several other OpenMP suites benchmarks. The NAS Parallel Benchmark (NPB) is a series of programs used to benchmark various supercomputers for the 17+ years [5, 1]. By examining the effect of thread pinning on computational benchmarks, it's usefulness for real programs can be predicted.

We did 25 runs of each NAS Parallel Benchmark (version 3.2.1) on QB2. Most of the tests used C class as there size, but FT and MG were only run B class because the compiler would not compile the larger (C class) version (data sections greater than 2GB). The pinning code was manually added to each test, normally as the first statement of the program. Figure 13 shows the results on QB2 running 16 threads on all 16 cores.

The variability between runs for several of the programs is very good. EP runs are effectively identical, and IS and FT have less than 10% between the minimum and maximum run time. Most CG and LU runs are close to the minimum time but both have cases where the execution was 20% slower than normal. On a

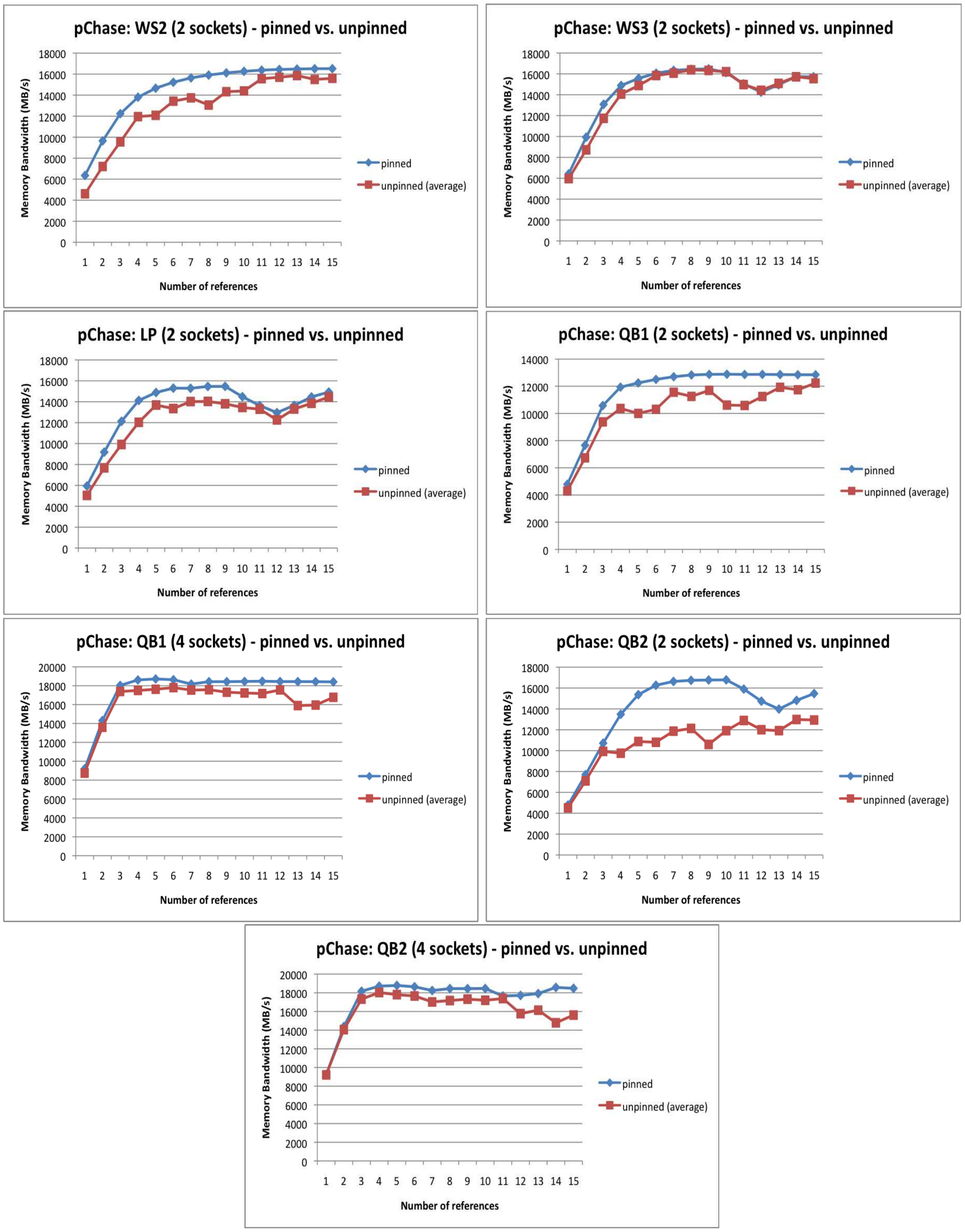


Figure 12: UnPinned and Pinned pCHASE Performance on multiple sockets

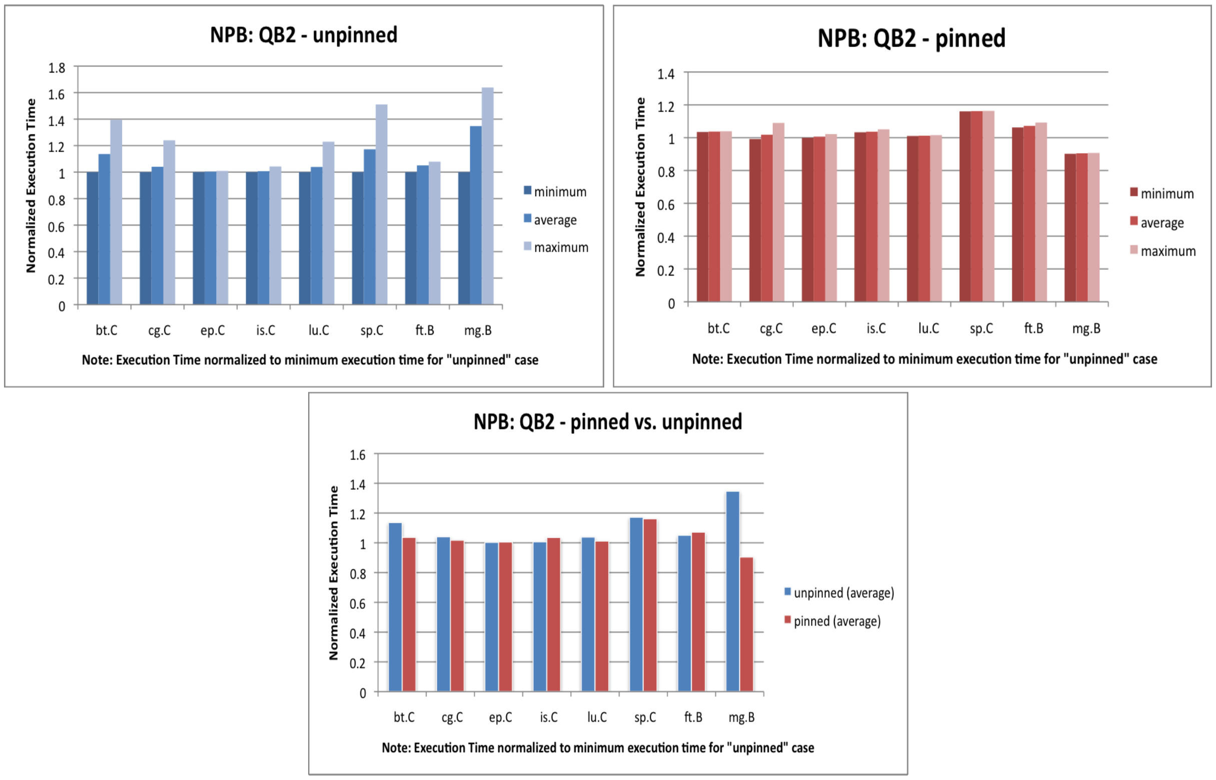


Figure 13: QB2 NAS Parallel benchmarks

cluster with a lot of nodes this could cause very poor scaling as every node waits for the slow node. BT, SP and MG performance was much more varied. The fastest run of each was between 40 and 60% faster than the slowest and the variety of execution times was greater producing an average time someplace between to two extremes.

One of the NPB benchmarks, EP, has very little communication between threads or even beyond the data that can reside within each core's local cache. Since very little memory traffic, local or remote, occurs, EP performance consistency is very good before pinning and all that pinning does is slightly increase the overhead of the OpenMP program. On average EP is fractionally slower by 0.2%, but the performance variation actually increased. The standard deviation of the performance rose from 0.2 to 0.5%.

IS and FT are other problematic tests. IS's performance decreased by 2.8% and FT's decreased by 2%. In both cases, the original version had good run-to-run consistency but the pinned version slightly better (standard deviation IS unpinned 1.2% pinned 0.4% FT unpinned 1.7% pinned 0.7%). BT, CG, LU, SP all see reductions in average execution time, ranging from 0.9% to 9.6%. Performance consistency was also improved (standard deviation unpinned 1.2% to 11.1% pinned 0.06% to 1.99%).

The performance of the final test, MG, improves 49% on average. The slowest pinned execution was 10% faster than the fastest unpinned execution. Unpinned MG has a standard deviation of about 10%, and pinned that falls to 0.16%. MG touches a large fraction of its memory before Linux balances the initial thread placement. By moving threads to their own processor before memory initialization and not allowing them to move after initialization, much better memory load balance occurs. Unpinned one memory bank handled 40% to 90% of the total memory accesses. After pinning the four banks all see about a quarter of the accesses. The memory balance allows more consistent performance and higher performance since one bank is not not bottlenecking the entire system.

Pinning computationally intense benchmarks improves consistency of results between runs. From an absolute performance point-of-view, there were 3 trivial losses (increase in overhead), 3 trivial gains, 1 significant increase ($\tilde{1}0\%$) and one spectacular improvement ($\tilde{5}0\%$). From a single node performance perspective, pinning seems worthwhile but not overly significant. But when performance tuning or running on large clusters, the benefits from not having to make multiple runs or have nodes idle waiting on other slow cores to finish will be important. The near complete consistency between runs of the NPB, shows that correct thread and memory placement is likely to be important even for computationally intensive programs.

5.3 SPEC OMP2001

SPEC OMP 2001 is a benchmark suite that measures performance of applications using the OpenMP standard. The applications are weighted towards computation although not as much as the NPB suite. SPEC OMP 2001 comes in two sizes M (shorter runs) and L (originally several hours to execute). We ran the M series of tests which consists of 11 programs, 2 of which did not compile on our system (MGRID and GALGEL). Figure 14 shows the minimum, maximum and average execution time for 10 runs of each program on QB2 for both unpinned and pinned versions. The pinning of threads was accomplished by adding an OpenMP parallel section directly after main which calls `pin_thread()`. (Note: the large difference in scale between the two graphs.)

Two programs, APSI and ART, have very little performance variation when run without threads being pinned to cores. The other programs show variation ranging from 3% for AMMP to 44% for EQUAKE. Once the threads are pinned only SWIM (7%) and APPLU (10%) show visible variation. Most of the applications have had any performance variation significantly reduced.

On average, only AMMP (-2%) slowed because of the pinning. To add the pinning code to the first OpenMP region, one OpenMP pragma was split into two and the pinning code added between them. If that program runs without the pinning code, however, the execution time is greater than with the pinning code. The performance of AMMP is significantly improved though optimizations that the single pragma allow this implementation.

The minimum times for pinned and unpinned runs where close (pinned from 5% to 4% slower). The elimination of variability increased average performance in 8 of the 9 tests, in some cases significantly (EQUAKE 22% and SWIM 12%). The reduced execution times could be the result of better load balancing between barriers or reduced memory latency from using nearby memory. In either case, thread pinning to cores allows the application to better understand and use the hardware resources.

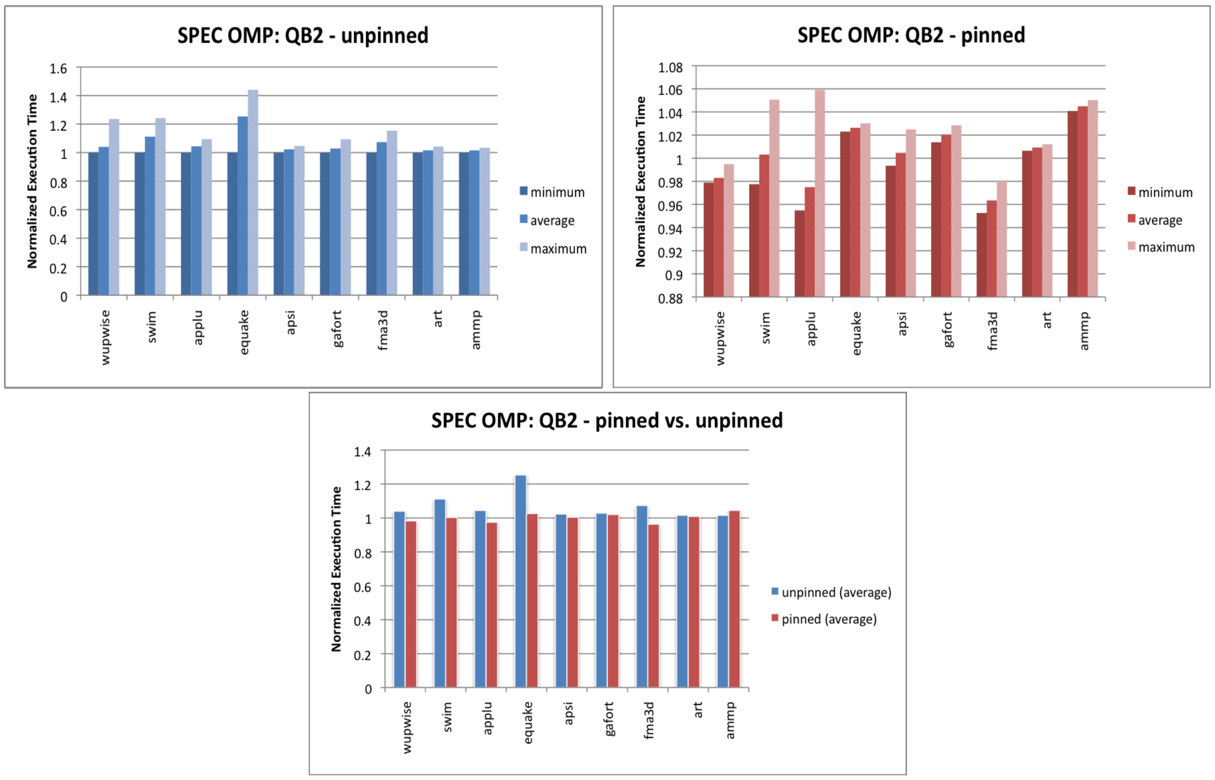


Figure 14: QB2 SPEC OMPM 2001 benchmarks

6 Related Work

With the introduction of multi-socket multi-core systems, previous work on NUMA programming for SMP systems will apply to desktop systems. SGI has built several generations of NUMA architectures has a book, Topics in IRIX Programming [6], several issues dealt with in Chapter 1 (Mapping, Locking and Unlocking, and Using Nonuniform will apply to NUMA x86 systems. The actual memory page placement functions are specific to SGI and may not be applicable on Linux x86 systems.

Efforts like NSORT [12] have examined methods to algorithmically increase the performance of NUMA systems by maximizing the work done where the data is located, by either moving work or moving data to available resources. We have seen that in addition to memory locality, the total number of outstanding requests is limited and performance will suffer if too many references are outstanding.

Anthony, Janes, and Rendell [4] examined thread and memory placement for a single-core AMD quad-socket system and a Sun 12 processor UltraSPARCIII Cu system. They observe the latency of the remote memory to be up to 53% expensive on the AMD and 20% more expensive on the Sun. With one thread per memory bank, they do not see the DIMM bandwidth and coherence protocol bottlenecks, that we observe with quad-core systems.

Alam *et al.* [2]showed the effective bandwidth falling per core on multi-socket systems, but beyond noting performance significantly worse than expected did not look into the causes or comment on any inconsistencies.

7 Conclusions

By adding a function call at the beginning of execution that pins a thread to a specific core, we have greatly decreased the run-to-run variation and some cases dramatically increased performance. Memory bound programs are most likely have significant increases, but some compute intensive programs have significant increases. For the small number of benchmarks tested no program's performance was significantly reduced although several showed minor performance degradation.

Understanding what effects a change makes in the execution time of an application, is critical to performance tuning. Modern multi-socket nodes are NUMA-like systems and can have significant on-node bottlenecks getting to memory, so that correct data placement is critical. Linux initial thread allocation is often not optimal, although cpu usage is balanced within a small number of seconds. Memory pages touched during this initial phase are misplaced, resulting in poor (and worse inconsistent run-to-run) performance.

For large clusters, the variation between runs will cause significant idle time across the system. At every global barrier, the entire system will wait on any slow node. If memory and thread layout, causes one node to be 10 or 20% slow, every other node will be idle for that time. Making sure that no slow nodes exist is important to most load balancing problems on large clusters.

References

[1] National Aeronautics and Space Administration. NAS parallel benchmarks changes. http://www.nas.nasa.gov/Resources/Software/npb_changes.htm.

[2] Sadaf R. Alam, Richard F. Barrett, Jeffery A. Kuehn, Philip C. Roth, and Jeffrey S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *IISWC*, pages 225–236. IEEE, 2006.

[3] AMD. Memory bandwidth (STREAM) four-socket servers. http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_8800̃124987,00.html.

[4] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *High Performance Computing - HiPC 2006, 13th International Conference*, Lecture Notes in Computer Science, pages 338–352. Springer, December 2006.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[6] David Cortesi, Arthur Evans, Wendy Ferguson, and Jed Hartman. Topics in IRIX programming, 1996.

[7] Message Passing Forum. MPI: a message passing interface, 1993.

[8] Message Passing Forum. MPI-2: extensions to the message-passing interface, 1996.

[9] R. L. Knapp, R. L. Pase, and K. L. Karavanic. ARUM: application resource usage monitor. In *9th Linux Clusters Institute International Conference on High-Performance Clustered Computing*, April 2008.

[10] Howard M. Lander, Robert J. Fowler, Lavanya Ramakrishnan, and Stevern R. Thorpe. Stateful grid resource selection for related asynchronous tasks. Technical Report TR-08-02, RENCI, North Carolina, April 2008.

[11] John McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December, 1995.

[12] Chris Nyberg, Charles Koester, and Jim Gray. Nsort: A parallel sorting program for NUMA and SMP machines. Technical report, Ordinal Technology Corp and Microsoft, August 2000.

[13] OpenMP Architecture Review Board. OpenMP application program interface version 2.5, May 2005.

[14] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[15] Douglas Pase. The pChase benchmark page. http://pchase.org/.

[16] Douglas M. Pase. Linpack HPL performance on IBM eServer 326 and xSeries 336 servers. Technical report, IBM, July 2005.

[17] Douglas M. Pase and Matthew A. Eckl. Performance of the AMD Opteron LS21 for IBM Bladecenter. Technical report, IBM, August 2006.

[18] Douglas M. Pase and Matthew A. Eckl. Performance of the IBM System x3755. Technical report, IBM, August 2006.

[19] Allan Porterfield, Rob Fowler, Anirban Mandel, and Min Yeol Lim. Empirical evaluation of multi-socket, multi-core memory concurrency. Technical Report RENCI Technical Report TR-09-01, Renaissance Computing Institute, 2009.

[20] X. Zhang and X. Qin. Performance prediction and evaluation of parallel processing on a numa multiprocessor. *IEEE Transactions on Software Engineering*, 17(10):1059–1068, 1991.