Evaluating Performance of the SGI Altix 4700 via Scientific Benchmark and Micro-Benchmarks

Mariam Salloum
University of California, Riverside
Computer Science Department
msalloum@cs.ucr.edu

ABSTRACT

I evaluated the performance of the SGI Altix 4700 by using several well-known benchmarks. In performing these experiments we hope to gain a better understanding of the capabilities and limitations of the system, and thus be able improve upon the design in future generations or develop tools that enhance the performance of the system.

INTRODUCTION

Micro-benchmarking is a popular technique used as a basis for application performance predication or to evaluate the performance of a particular system in comparison to other systems, or to be used as a predictor for real application performance. There are several benchmarks that have been developed to measure various aspects of the system. In the following sections I will (1) describe the target system, (2) list and describe the benchmarks used to evaluate the system, (3) provide results and analysis for each benchmark, (4) conclude and summarize the results.

TARGET SYSTEM

The system we are evaluating is the SGI Altix 4700. The following are the specs for this system[1]:

- cc-NUMA Distributed Shared Memory (DSM) with distributed cache directory,
- 64 1.5 GHz Itanium 2 processors,
- Deploys a modified version of the SUSE Linux 2.6.16 kernel,
- Fat-tree topology utilizing the NUMAlink 4 interconnect, which offers 3200 MB/s per link,
- gcc and Intel C compiler ,
- Utilizes a first-touch memory placement policy and highest priority first cpu scheduling.

BENCHMARKS

In the following section I will provide a brief description of the benchmarks used to evaluate the performance of the Altix. One thing to note is that some of the information provided by the benchmarks is similar; however, each benchmark is unique in how it obtains that particular information.

Stream

Sustainable Memory Bandwidth in High Performance Computers (STREAM) was developed by John McCalpin at the University of Virginia [2]. STREAM measures the sustainable memory bandwidth in MB/s. This information can also give us an idea about the memory latency. STREAM has become an industry standard, as a result, performance results for many other systems are available on the web for comparison.

Memperf

Memperf [3] is yet another benchmark developed by Kurmann and Stricker, which measures memory performance using access bandwidth by varying the strides and using different work loads. It measures the memory bandwidth in a 2D way. First it varies the block size information of the throughput in different memory system hierarchies (different cache levels). Secondly it varies the access pattern from contiguous block to different stride accesses [1].

CacheBench

CacheBench is a benchmark designed to evaluate the performance of the memory hierarchy of the system. The benchmark includes eight different cases: Cache Read, Cache Write, Cache Read/Modify/Write, Hand tuned Cache Read, Hand tuned Cache Write, Hand tuned Cache Read/Modify/Write, memset(), and memcpy(). Each case performs repeated access to data on varying vector lengths. Timings are taken for each vector length over a number of iterations and the bandwidth (megabytes per second) is calculated. The first three provides information about the performance of the compiler, while the second three are provided as points of comparison as they include hand tuned code. The benchmark provides the user with feedback about memory performance of the machine. For additional information regarding the CacheBench please refer to [4].

LMbench

LMBench is yet another benchmark that includes bandwidth and latency measurements. In terms of bandwidth, it measures cached file read, memory copy, memory read, memory write, pipe, and TCP. In terms of latency, it measures context switching, networking (connection establishment, pipe, TCP, UDP, and RPC hot potato), file system creates and deletes, process creation, signal handling, system call overhead, and memory latency. LMbench has become a popular benchmark, and as a result many measurements from different machines are posted online as a point of reference.

RESULTS

In this section I will present the results of each benchmark.

Stream

I compiled my application using the following options: gcc –fopenmp –D_OPENMP stream.c –o stream. I ran the Stream benchmark using array size of 18000000 of a 412 MB total memory. Each test was run 10 times, but the best result is reported below.

Table 1a: SGI Altix 4700 – 64 threads

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	662.5549	0.4348	0.4347	0.4350
Scale	601.9713	0.4786	0.4784	0.4788
Add	610.8624	0.7073	0.7072	0.7075
Triad	567.2530	0.7617	0.7616	0.7619

Table 1b: SGI Altix 4700 – 4 threads

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	668.2600	0.4311	0.4310	0.4313
Scale	602.8461	0.4778	0.4777	0.4781
Add	600.0067	0.7201	0.7200	0.7203
Triad	560.0540	0.7715	0.7714	0.7716

I reran this two cases by increased the array size to 180000000 which requires 4119.9MB . The results are shown in the following two tables.

Table 1c: SGI Altix 4700 – 64 threads

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	664.1074	4.3375	4.3366	4.3383
Scale	602.9207	4.7801	4.7767	4.7896
Add	617.5283	6.9969	6.9956	6.9979
Triad	566.6657	7.6418	7.6235	7.7631

Table 1d: SGI Altix 4700 – 4 threads

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	667.2964	4.3162	4.3159	4.3165
Scale	606.4606	4.7490	4.7489	4.7493
Add	618.6444	6.9833	6.9830	6.9836
Triad	566.8759	7.6211	7.6207	7.6214

This information will hold more value if we compare the results to another multiprocessors. Unfortunately, the STREAM website only lists results from systems with very large number of processors and as a result its difficult to compare the performance of our system to theirs. The bandwidth only changed slightly when we increased the array size and the bandwidth between spawning 64 vs. 4 threads showed only slight difference. I think the reason for this is that I did not test a very large array size. I think, if I were to test large array sizes versus small array the difference in bandwidth will be visible. Also, given the fact that other users are utilizing the machine at the time the accuracy of the results are influenced.

Memperf Micro-Benchmark

I ran the Memperf benchmark with varying workloads and stride sizes. The first figure plots the varying strides, workload, and the bandwidth achieved for load sum test case. The load sum test measures the memory load performance for all the block-sizes and access patterns. As can be seen by the figure, increasing the load and the stride size decreases the bandwidth.

Figure 2a: Memperf (1 thread)

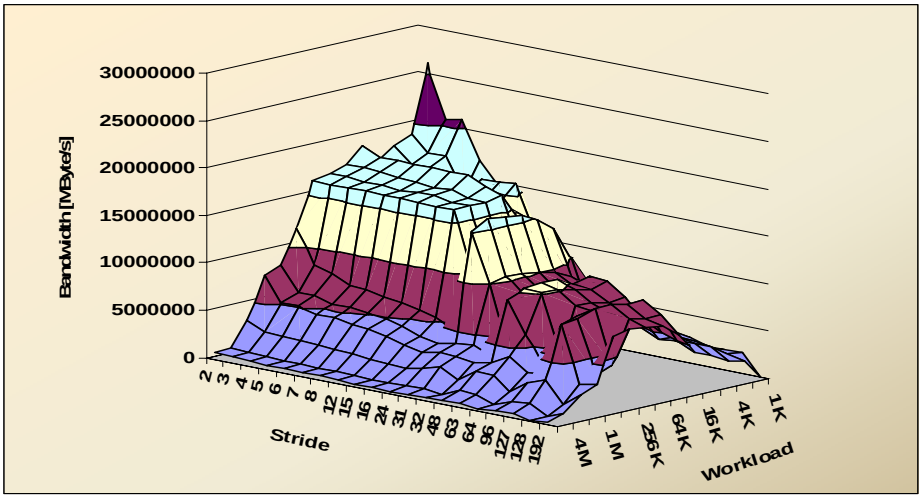


Figure 2b: Memperf (4 thread)

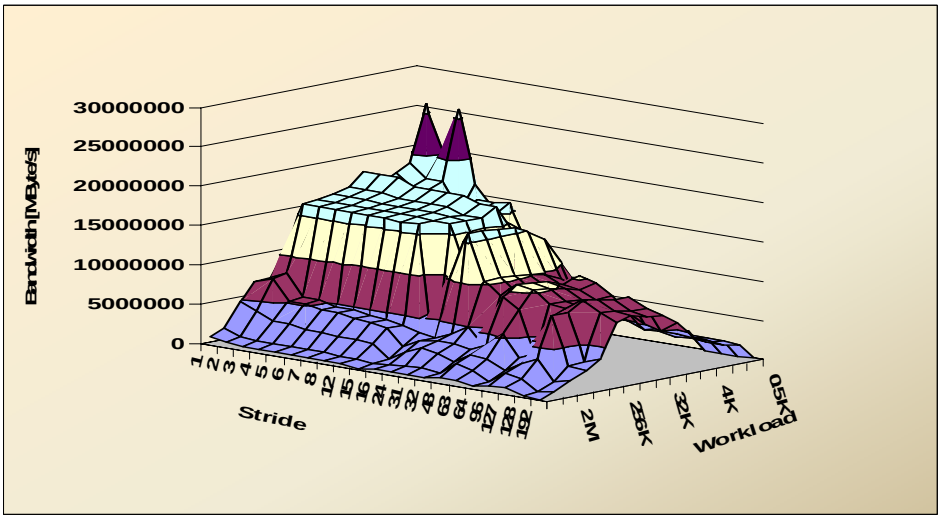
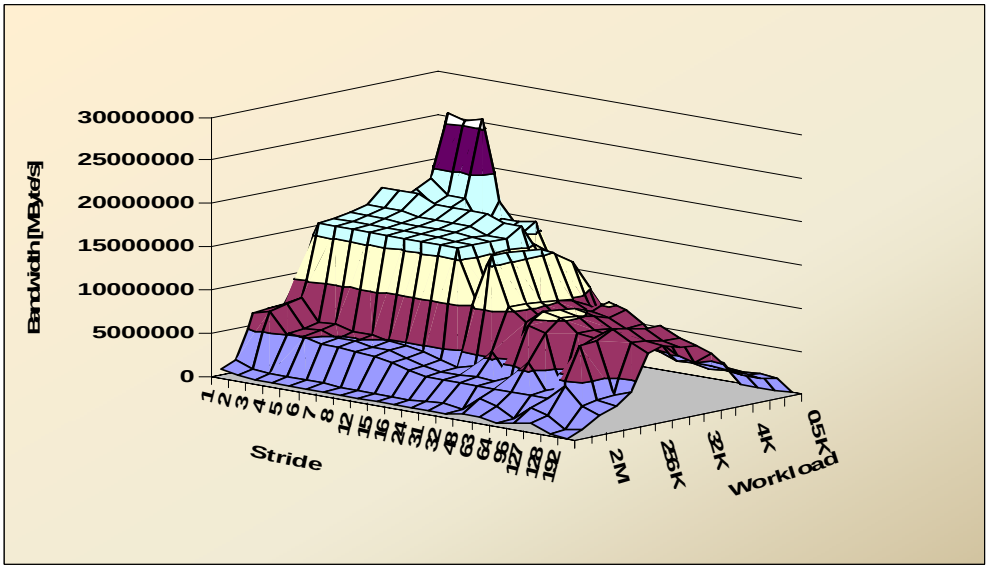


Figure 2c: Memperf (16 threads)

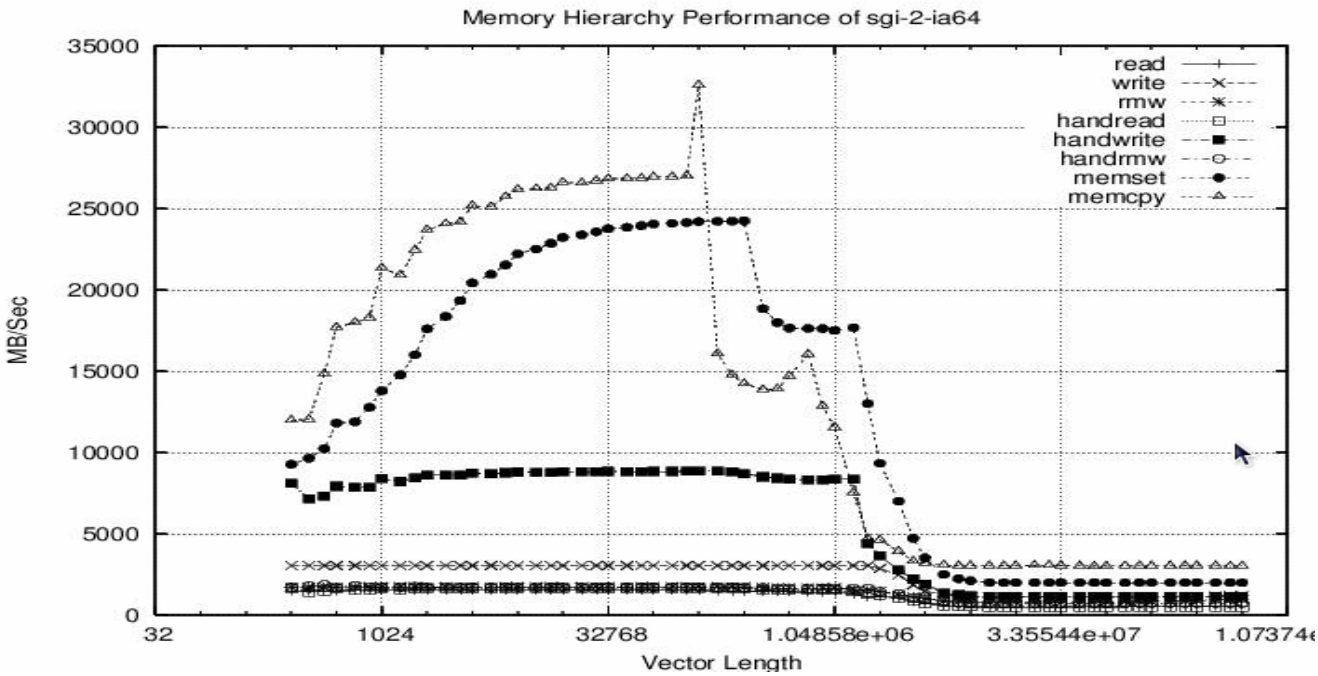


As can be seen by the above three figures, the bandwidth pattern is the same fro 1, 4 and 16 threads. The only visible difference is the peak achieved with small workload and stride. For one thread measurements, we see one clear peak. For four threads, we see two distinct peaks. For 16 threads we see a larger peak.

Copy bandwidth decreases for simultaneous access with 1, 4, and 16 processors. For small working sets that reside in caches the performance bandwidth remains the same for various threads, however, large working sets in memory generates interesting differences.

CacheBench

I ran CacheBench testing all cases, and varying the accessed vector length. The results are shown in the following figure.



LMbench

I ran the LMbench selecting to evaluate the OS and hardware. I selected to allow the scheduler to place jobs, instead of manually assigning each test thread to a processor. Also I set the benchmark to operate on only 4000MB (note the larger the rage of memory the more accurate the results, however, as I did not want to over load the system when others are utilizing it I opted for 4000MB). The following tables show the results for various tests. Unfortunately, its very hard to evaluate this type of data because we having nothing to compare it to. However, I have requested that the LMbench send me the results of other systems so that I will have an idea about the performance of the Altix (They acknowledged by request by have not sent the data yet).

Table 4a: Memory latencies (nanoseconds)

Host	**OS**	**Mhz**	**L1**	**L2**	**Main mem**	**Rand mem**
sgi-2	Linux 2.6.16	1595	1.2720	3.7620	151.4	181.2

The above table shows the latencies to L1, L2, and main memory. We are able to compare these results to that of the SGI Challenge. The SGI challenge has a L1 latency of 8 nanoseconds, L2 latency of 64 nanoseconds, and memory latency of 1189 nanoseconds. As we can see, the SGI Altix performs much better.

Table 4b: Basic integer operations (nanoseconds)

Host	OS	Intgr bit	Intgr add	Intgr mul	Intgr div	Intgr mod
sgi-2	Linux 2.6.16	0.6300	0.0100	3.2600	23.2	26.3

Table 4c: Basic float operations (nanoseconds)

Host	OS	Float add	Float mlt	Float div	float bogo
sgi-2	Linux 2.6.16	2.4800	2.5100	19.9	31.0

Table 4d: Basic double operations (nanoseconds)

Host	OS	Double add	Double mlt	Double div	double bogo
sgi-2	Linux 2.6.16	2.5100	2.5100	22.5	32.1

The previous three tables show the latencies for integer, float and double operations. The results look accurate. Its interesting to note the large difference between integer and float/double operations.

Table 4e: Local Communication latencies (microseconds)

Host	OS	Pipe	AF UNIX	UDP	RPC/UDP	TCP	RPC/TCP	TCP conn.
Sgi-2	Linux 2.6.16	17.2	8.87	27.3	32.2	31.4	101.0	78

Table 4f: Context Switching (microseconds)

Host	OS	2p/OK ctzsw	2p/16K ctzsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctzsw	16p/16K ctxsw	16p/64K ctzsw
Sgi-2	Linux 2.6.16	.501	1.160	1.9300	2.3100	4.3800	2.780	4.7800

The above table shows the latency for context switching. The benchmark performed test with 2, 8, and 16 processors and varying data size. Comparing these results with that of the SGI Challenge we see that Alix performs much better. The SGI Challenge results are 2p/0KB 63microseond, 2p/32KB 80microseond, and 8p/32KB 93microsecond. As we can see, even when the amount of data transferred was less the Challenge performed worse than the Atlix.

Table 5g: Local Communication bandwidth in MB/s

Host	OS	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Meme write
Sgi-2	Linux 2.6.16	1721	994	707	1498.0	318.5	654.2	351.6	615	655.4

Table 5h: File and Virtual Memory system latencies (microseconds)

Host	OS	0K file		10 K file		Mmap Latency	Prot Fault	Page fault
		Create	Delete	Create	Delete			
Sig-2	Linux 2.6.16	470.1	541.4	343.1	424.6	49.0	0.144	1.02810

Table 5g shows the local communication bandwidth, which includes inter-process communication and cache bandwidth. It evaluates the inter-process communication bandwidth, via evaluating several processes communicating through pipes or TCP sockets. It also evaluates cache bandwidth via read and mmap. Comparing these results with other systems we can see that SGI Altix performs much better. For example, other research posted the results for the SGI Challenge showing it has 17 MB/S for pipe and 31 MB/s for TCP bandwidth. In comparison, SGI Altix has 1721 MB/s for pipe and 707 MB/s for TCP bandwidth. Also, comparing the file and memory bandwidth for the two machines shows that SGI performs much better. For example, SGI Challenge has a Memory read bandwidth of 65 MB/s while the SGI Altix has a Memory read bandwidth 615 MB/S.

CONCLUSIONS

In this paper we ran several benchmarks to determine the performance of the SGI Altix system. The next step is to compare these results to other cc-NUMA systems and/or clusters to determine the level of performance we are receiving. This information can be used to better understand the limitations/capabilities of the Altix and thus design better applications. In addition, upon evaluating the performance we may be able to find ways to improve the design in the next generation. For example, the memperf and Stream benchmarks showed us that an automatic page migration/replication policy is desirable. Some the results obtained are hard to evaluate without having other results as a point of comparison. Thus, in the future I would like to compare some of these results with other systems.

REFERENCES

[1] SGI Alix Applications Development and Optimizations, Par No.: AAPPL-1.0-L2.4-S-SD-W, Release Date: August 1, 2003
[2] Stream www.cs.virginia.edu/stream
[3] Memory System Performance Characterization http://www.cs.inf.ethz.ch/cops/ECT/
[4] CacheBench http://icl.cs.utk.edu/projects/llcbench/**cachebench**.html
[5] LMbench www.bitmover.com/**lmbench**/

Memory Management Study on the SGI Altix

Mariam Salloum
University of California, Riverside
Computer Science Department
msalloum@cs.ucr.edu

ABSTRACT

The SGI Altix 4700 memory management policy is by default allocated on a ‘first-touch’ basis. Unlike the SGI IRIX machines, the Linux version of the SGI does not employ additional policies (such as page migration & replication) to improve data locality. In this study we would like to investigate whether the user can employ some tools to improve the data locality of a given

application versus settling for the default 'first-touch' policy. In this paper, we also look at research that have been done on this area and evaluate their results.

INTRODUCTION

The SGI Altix is a cc-NUMA system that employs distributed shared memory (DSM). Memory management plays an important role in achieving the best performance. Physical memory of the system is allocated on a "first-touch' basis, i.e. the thread that first referenced a memory location will maintain hold of that memory location throughout the application runtime. Although, 'first-touch' produces better results in comparison with round-robin, we would like to investigate whether there are other directives that can improve on data-locality without a high overhead cost.

The Linux OS on the SGI Altix, as mentioned above, employs "first-touch" policy which does a fairly good job of allocating memory close to where it will be referenced -- at least initially. However, even if we assume that the first-touch policy provides the best data locality, the Linux scheduler does not consider this factor and its load balancing policy may distort the initial assignment. The Linux scheduler takes no note of NUMA locality when migrating tasks between nodes.

If the user is familiar with the application memory accesses, he/she can employ the dplace, dmove, and cpuset commands to assign a give cpu/node to a memory slot. However, if the user does not have this specific information or the application exhibits a changing memory access pattern then this approach is not desirable. What we are seeking is an automatic method that does not rely on the user for any information. In the following section I will summarize of the papers published regarding this topic.

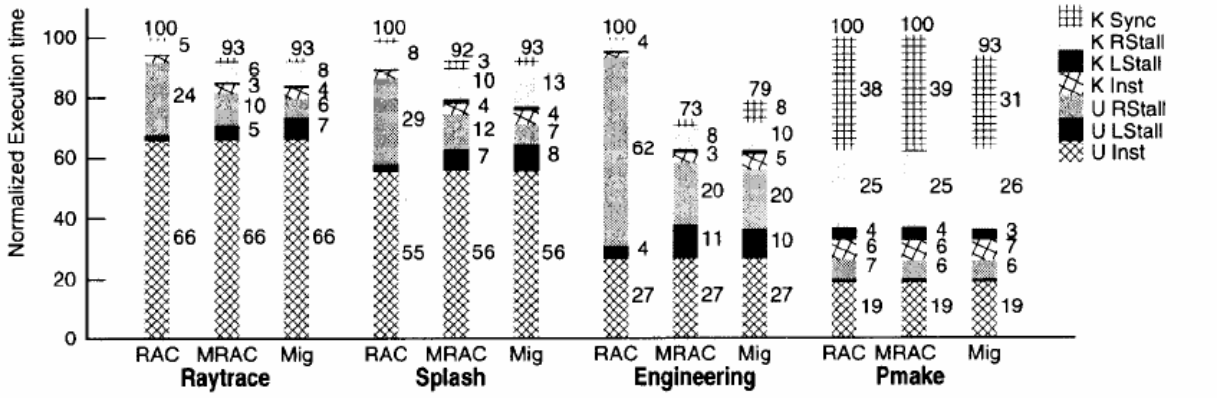
PREVIOUS WORK

One option to solve the locality problem is to develop an automatic page migration and replications policy embedded into the Linux kernel; this policy seeks to complement the first-touch memory placement policy not replace it. This idea was first proposed by Vergles et. al. at the Stanford FLASH group[3]. Their goal was to minimize the runtime of the user's application in a cc-NUMA system by converting remote misses to local misses through migration and replication of pages. They maintained a 'counter' for each page that tracked the cache miss rate. They also defined the 'threshold' for a page migration and replication (tolerance value that triggers page migration/replication). When a cache miss occurs, the controller looks at the 'page miss rate' and decides whether it's a good candidate for this policy. If the cache miss rate is low then there is no reason to apply these polices. Second, the controller looks at the type frequency and type of sharing. If the page is not shared by many then it's a good candidate for migration, otherwise, if the page is read shared by many then it's a good candidate for replicating. In addition, they take into consideration the cost of page migration and replication; if the cost of these two polices are high (above the defined threshold) then no policy is applied. Although this policy shows 30% performance improvement in simulations, this feature has not been added to the Linux OS because in real applications the overhead cost of page migration and replication is too high.

The FLASH team presented another paper titled "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors[5]." In this paper, they compared the performance of Remote-Access-Cache (RAC), OS based page migration / replication, and MIGRA a kernel-based migration / replication that handles coarse-grain locality decisions. The RAC policy utilizes part of local memory to perform remote-caching. This policy

is said to perform 14% better than standard CC-NUA base protocol, however, the performance is heavily dependent on the size of the RAC and many take away memory from the OS.

The OS based page migration/replication (the policy presented in the previous mentioned paper) is said to perform 30% better however, this policy is strict in terms of which types of sharing can occur. Thus, they proposed a scheme called MIIGRAC which includes kernel-based migration and replication and RAC protocol. The idea behind MIGRAC is as follows: the RAC can capture fine-grain sharing and short-term locality (thus temporarily solving the cache miss problem) and Mig/Rep can move or replicate a page that has a high miss rate to local memory thus alleviating RAC from having to continue tracking these pages.



The above figure was copied out of the paper. It shows a comparison of the three methods, RAC, MRAC, and Mig/Rep for four applications. As can be seen by the figure, MRAC performs better on most applications.

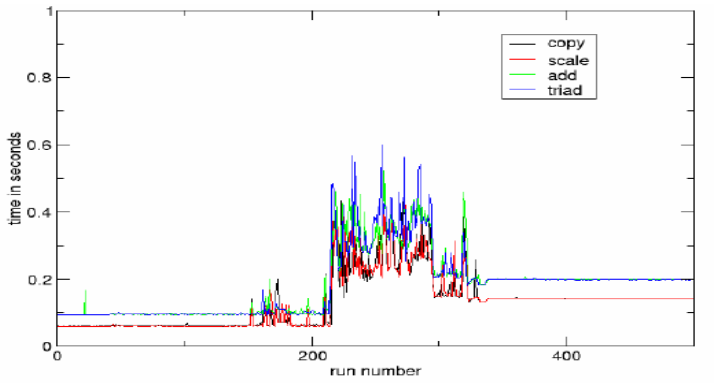
Workload	RAC hit rate(%)		Pages Replicated	
	RAC	MRAC	Mig/Rep	MRAC
Raytrace	48.0	55.5	2661	1083
Splash	40.1	50.4	2982	1460
Engr	34.6	46.7	3494	2083
Pmake	25.5	24.8	664	160

A more subtle advantage of MRAC can be seen by the above table. The MRAC policy is able to reduce the number of pages replicated compared with Mig/Rep and increase the RAC hit rate. Replicating fewer pages implies that there is more free memory for other uses and having a higher hit rate translate to better performance.

However, none of the above policies have been implemented in Linux OS. Lee Schermerhorn at HP/Open Source & Linux Organization wrote a paper titled “Automatic Page Migration for Linux” and has developed patches to the Linux OS that implements automatic page migration (note: these patches do not support page replication as yet) [6]. The team utilized the STREAM benchmark to show that without manual intervention optimal data placement has low probability and that the probability decreases with increasing node count. Also, even with near optimal ‘initial’ placement, transient workloads can distort locality. This problem occurs when the kernel is performing inter-node load balancing; the kernel is unaware of cc-NUMA data locality issues and thus it does not know where to best schedule threads as to optimize data locality. They

proposed a policy called automatic lazy page migration which has the same idea as Vergles policy however they try to cut-down overhead.

I ran the STREAM benchmark several times initially binding the threads to a given cpu to provide optimal performance. But, as seen by the graph below, although the high initial performance showed that data placement was optimal, running the benchmark several times shows a decrease in perforce. This decrease in performance is attributed the Linux scheduler which tries to employ load balancing between cpus. Thus, this confirms that another method is required to improve data locality throughout the application execution time.



This project is in its infancy and has yet not developed fully but shows great promise in the future. The team is currently working on perfecting the migration policies and are also working on adding the page replication. Their work is very much based on the paper by V…., however, at every step they try to minimize the overhead cost of maintaining the counter information.

A paper by Corbalan, Martorell, and Labarta titled “Evaluation of the Memory Page Migration Influence in the System Performance: The Case of the SGI O2000” proposes yet another policy to minimize data locality[4]. In this paper they evaluate the SGI Origin 2000 IRIX memory management policy. They proposed to enhance the IRIX page migration and replication policy by coupling it with the process scheduler. The basic intuition behind this policy is that we should utilize the cache miss rate data by passing it to the process scheduling, that way, the scheduler can make a more informed decision on how to assign a given thread to a cpu. In this paper, they evaluated this policy using the IRIX native scheduling policy, equiparition, and Performance Driver – Processor Allocation (PDPA). Equiparition is a dynamic space-sharing policy which tries to allocate a equal amount of applications among the available processors. PDPA is a dynamic space-sharing policy which tries to allocate the maximum number of processors to the running applications that reach a given target efficiency. The PDPA decides the processor allocation based on the application request and the application performance (measured at run-time). Re-allocations are done a job arrival, job completion, and when jobs inform about their performance. And finally, they also evaluated the IRIX native scheduler that uses time-sharing approach. They used four different benchmarks/applications including swim, hydro2d, apsi from specfp95, and bt from NAS parallel Benchmark suite. The results show that the memory page migration policy in IRIX does indeed improve the application performance, but aht the benefit or the level of performance depends on the application and on the scheduling policy. The performance was shown to improve by 10% with the IRIX schedule, 50% with the Equiparition, and 13% with the PDPA.

FUTURE WORK

All the paper presented raised interesting ideas. In the future, I would like to test the Linux 'patches' presented by Lee Schermerhorn and compare the performance of various benchmarks with migration policy enabled and with the policy disabled. The MRAC policy presented by the FLASH group also looks very promising especially for the SGI Altix because the machine has sufficient memory to maintain RAC information. Implementing MRAC policy would be a interesting and promising project. Lastly, I'm interested in the work presented by Corbalan, Martorell, and Labart in their paper. I think their idea is correct in that we need a migration/ replication controller in addition to a good scheduling policy that takes data locality into consideration. For future research I would like to see if I can apply the modified Linux kernel presented by Schermerhorn with the idea presented by Corbalan et. al. One thing to note is that the IRIX scheduler differs from the Linux scheduling policy and we would need to conduct further work to evaluate the level of performance achieved.

CONCLUSIONS

In this paper we reviewed the various methods that have been proposed to improve data-locality on cc-NUMA machines. The work performed by Lee Schermerhorn looks promising. However, the kernel patches proposed are still not complete and require the refinement of the lazy page migration policy to better identify the thresholds that trigger page migration. Also, Schemerhorn should also consider including page replication policies. The MRAC policy presented by the FLASH group looks very promising and it would be interesting to see this policy implemented on the SGI machine. The proposal put forth by Corbalan et. al. looks very promising and I think this is the path that SGI should consider in the future.

REFERENCES

[1] SGI Alix Applications Development and Optimizations, Par No.: AAPPL-1.0-L2.4-S-SD-W, Release Date: August 1, 2003
[2] Mattew Dobson, Patricia Gaughen, Michael Hohnbaum, Erich Focht, "Linux Support for NUMA Hardware," Proceedings of the Ottawa Linux Sumposium, Ottawa, Ontario, Canada July 2003
[3] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," ACM, 1996.
[4] J. Corbalan, X. Marorell, J. Labarta, "Evaluation of the Memory Page Migration Influence in the System Performance: The Case of the SGI O2000. ICS 2003.
[5] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, J. Hennessy, "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors," IEEE 1998.
[6] L. Schermerhorn, "Automatic Page Migration for Linux," HP/Open Source & Linux Organization

CS213 Parallel Processing Homework 2

Mariam Salloum

1. OBJECTIVE

The objective of this assignment is to compare the execution time of a well-known and tested benchmark on the SGI machine, varying the number of threads created, and the size of the simulated data. We would like to answer some of the following questions: Does the performance increase as we increase the number of threads? How does the number of threads relate to the actual number of processors available? At what point does the creation of multiple threads hinder performance?

2. ENVIRONMENT

We will perform the simulation on the SGI machine, which has the following specs:

SGI Altrix 4700
Dual Core Intel Itanium 2 Series 9000
1600MHz, 24M L3

3. BENCHMARK

The benchmark chosen to perform the simulation is the SPLASH2 application Ocean.

4. PERFORMACE METRICS

To measure the performance of the SGI machine, I will look at the total execution time vs. the number of threads created, # instructions for each cpu, and the # cycles for each cpu. All the SPLASH2 application produce output that specifies the total number of seconds taken by the program, however, I noted that this value was rounded. Thus, I decided to use the time command to get the exact execution time of the application. For each simulation, I will vary the grid size and the number of processors/threads used and plot the execution time of the application, # of instructions, and # of cycles. I will also look at the cache misses and finally compare the gcc and icc results.

5. RESULTS

I ran the simulation on the SGI machine for the Ocean application. I ran each test case twice and averaged the execution time received; each time I varied the simulated data size and the number of threads created. To make sure that the threads were bind to a processor, I used the 'taskset' command with the 0xFFFFFFFF option to specify that the application should distribute threads to all processors.

Results – Ocean

Table 1a: This table shows the Ocean application simulation on the sgi machine, varying the number of processors and the grid size and using the gcc compiler

	Gird Size					
Ocean	n=258	n=514	n=1026	n=2050	n=4098	n=8194

	1	0.701	3.404	14.685	59.449	240.712	1656.576
	2	0.354	1.558	7.245	29.42	123.514	926.321
	4	0.298	0.831	3.414	15.317	64.011	279.885
	8	0.383	0.987	2.476	8.167	34.642	152..093
	16	0.478	1.053	2.438	6.591	19.712	77.054
	32	0.641	1.226	2.927	7.132	15.204	53.913
	64	1.1.57	1.705	3.998	9.057	22.946	97.091

As shown by Table 1, performance improves as we increase the number of threads created (processors allocated), however, for each simulation there is an optimal point and then the performance will converge. However, there seems to be a correlation between the grid size and performance receiving by increasing the number of threads. For grid size n=258, the optimal solution was achieved by 4 threads; for grid size n=1026 the optimal solution achieved was by 8 threads, for n=2050 the optimal solution achieved was by 16 threads, and for n=4098 the optimal solution achieved was by 32 threads. Thus as we increase the size of the simulation data creating threads and thus applying parallelize improves the overall execution time.

The following two graphs shows the performance for grid size n = 2050.

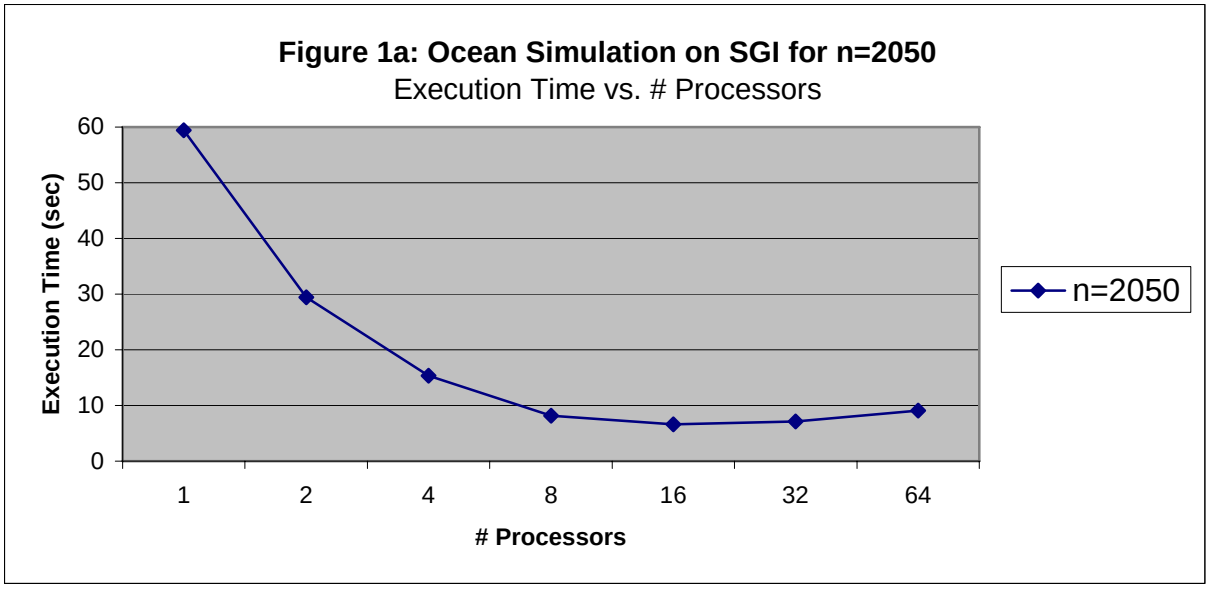


Table 1b: # instruction vs # threads for n=2050

		Ocean- # of simulated particles
		2050
# of processors	**1**	62726343296

2	31365465169
4	15691356926
8	7851191381
16	3932172795
32	1968428500
64	990438120

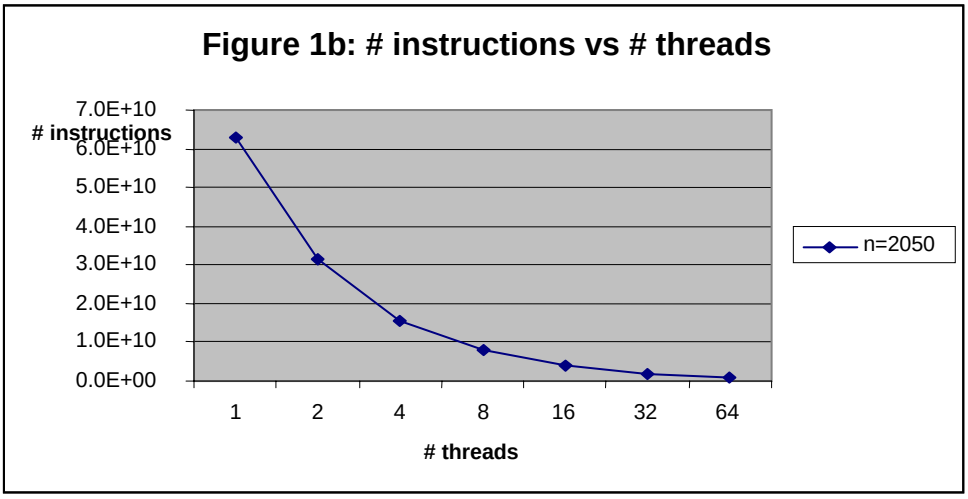


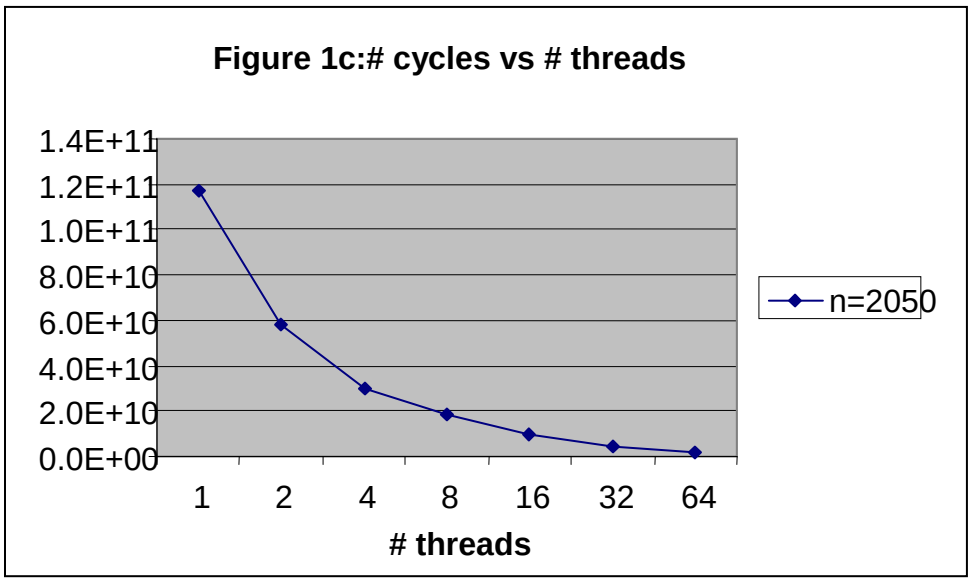
Table 1c: # cycles vs # threads for n=2050

		Ocean - # of simulated particles
		2050
# of processors	**1**	116950196762
	2	57860292901
	4	29958624771
	8	18406620724
	16	9813847799
	32	4232873095
	64	2161704279

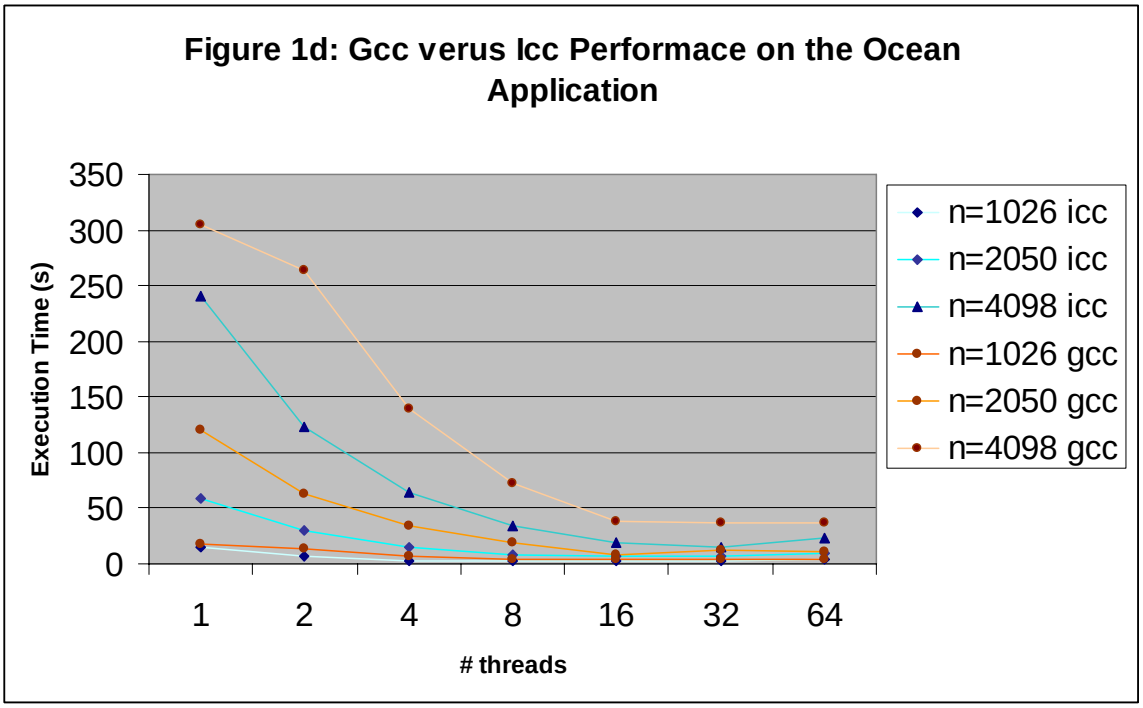
The following table shows the cache miss/access for n=1026.

Data	p=64	p=16	p=4	p=1
L1 data accesses are	288703	319368	502366	938747
L1 data misses are	80190	69026	82537	73951
L1 Insn accesses are	4209234	12349834	45889607	179263497
L1 Insn misses are	76057	55558	49934	18539
L2 data accesses are	2512665	9072628	35426940	140733375
L2 data misses are	58802	134930	1112225	4745652
L2 Insn accesses are	85524	63747	56337	21389
L2 Insn misses are	1062	1442	1441	1476

The results seem to be excepted. As we increase the number of processes the data access increase and the data misses decrease. It also shows that the L1 Insn access decrease with increasing number of processors, while the L2 Insn. access decrease with decreasing number of processes.

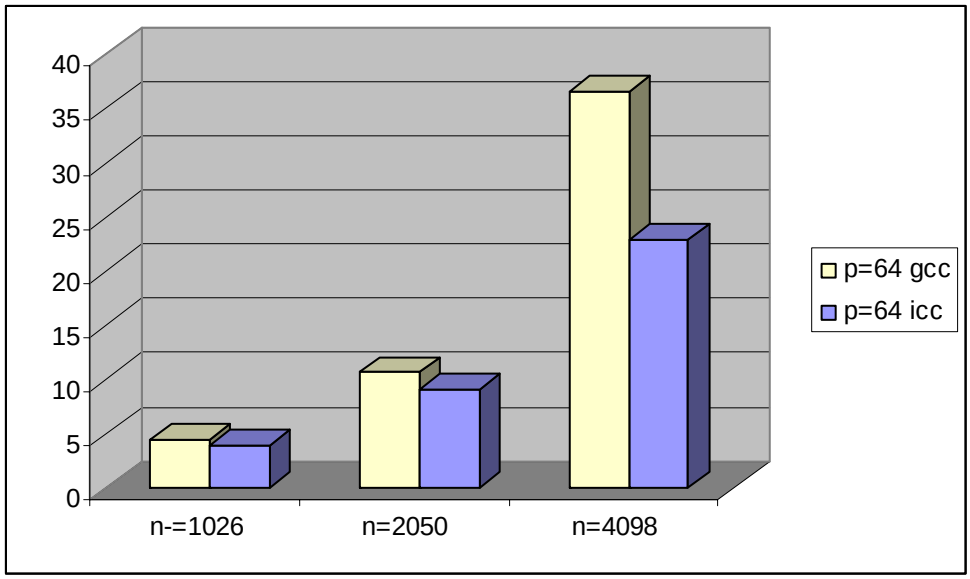


I redid the results using the intel icc compiler and compared the execution time of the application using gcc versus using icc.



The lines in orange show the execution for varying threads using the gcc compiler. The blue lines shows the same data for the icc compiler. As can be seen, the icc compiler generates better results.

Here is another graph that shows the performance of the ocean application with spawning 64 threads.



6. CONCLUSION

The simulation showed that increasing the number of threads or processors used does not always yield the optimal solution, however, for each test case there is an optimal number of threads which yields the optimal solution. We were also able to observe that when increasing the size of the simulation data, utilizing multiple threads yielded better results however when the size of the simulation data was small utilizing fewer threads yielded better results. This is expected because for a large data set we can create more parallelism and the delay caused by the thread dependency is minimal, but when the simulation data is small the delay caused by multi-processor communication is more obvious. It seems since we are not using CC and OpenMp the application is not correctly portioning the simulated data among the processors memory modules, but rather storing it in only one memory. Thus evey processor must access that memory and that could be a cause for bad simulations at times. We also looked at the # of instructions and # of cycles as we increased the # of threads. All the simulations showed that as we increase the # of threads spawned the # of instructions/cycles decrease for each cpu. Also, we compared the performance of gcc versus icc, and as expected icc performed much better than gcc.