Scalability and Data Placement on SGI Origin

Arun CHAUHAN
Barry SHERAW
Chen DING

April 28, 1997

Abstract

Cache-coherent non-uniform memory access (ccNUMA) architectures have attracted lots of academic and industry interests as a promising direction to large scale parallel computing. Data placement has been used as a major optimization method on such machines. This study examined the scalability and the effect of data placement on a state-of-the-art ccNUMA machine, SGI Origin, using 16 processors. Three applications from SPLASH-2 are used, FFT, Radix and Barnes-Hut. The results showed that FFT and Radix cannot scale to 16 processors with 70% efficiency even for the largest data sizes tested. Barnes-Hut doesn't scale for small data size but scales linearly for large input size. The results also showed that data placement does not make any difference on performance for all three applications. We attribute these results to the effect of the advanced uni-processor used on the Origin, R10K, the optimizing compiler, and the aggressive communication architecture.

Some of our results are quite different from the predictions of two recent simulation studies on directory-based ccNUMA machines ([HSH96] and [PRA97]), especially on FFT. These differences are partly due to the fact that the machine models used in previous simulation studies are different from the Origin machine in some important aspects. Our results also include data sizes that are larger than any of the previous simulation studies. To increase our confidence on the latency numbers and data placement tools, we also measured memory latencies using micro-benchmarks.

1 Introduction

In the last few years, there has been increasing interest in ccNUMA architectures, specifically in its potential for large scale parallel computing. As a result, many commercial machines based on the ccNUMA architecture have recently been introduced. These include machines such as the SGI Origin and HP-Convex Exemplar. Such machines utilize the ccNUMA architecture as an approach to scalable parallel architecture. One approach to ccNUMA architecture, directory based ccNUMA, is typified by the SGI Origin.

The basic ccNUMA architecture is composed of a collection of nodes connected via an interconnection network. Each node consists of a small number of high speed processors, a memory module, and a cache. Each node's memory is a portion of the machine's global shared memory.

The global shared memory is therefor distributed among the nodes. Cache coherence is maintained using a directory based update or invalidation scheme. Each node maintains a directory memory corresponding to its portion of the shared memory.

Cache misses are handled either locally or remotely. A cache miss is considered local if the cache is located at the same node as the memory module from which the physical address is allocated. In such a case, the memory at the node services the miss. A remote miss, however, implies that the cache and memory module where the data is allocated are in different nodes. Then the node which owns the cache line must service the request through the interconnection network.

Remote miss latency on ccNUMA machines is always greater than local miss latency due to the delay incurred by the interconnection network. It is therefore desirable that as many cache misses as possible be serviced by a node's local memory. Consequently, data placement influences the performance of many applications. Placing data in the memory of the node that access the data most frequently reduces the overall latency incurred by cache misses. Likewise, poor data placement can increase cache miss latency, increase communication, and lead to poor application performance.

1.1 Project Goals

The goal of this paper is to explore the scalability of several SPLASH-2 benchmark programs on a state of the art CC-NUMA machine. These experiments attempt to verify and compare performance characteristics of the selected SPLASH-2 applications with previous results. In addition, it investigates the performance benefits of data placement on applications running on the SGI Origin.

1.2 Previous Work

Several studies have been done on scaling applications on cache-coherent shared memory systems. Among the recent studies of scalability and effects of optimizations on parallel applications, is that done by Holts et al. [HSH96]. They report simulation results for a cross section of SPLASH 2 [WOT$^+$95] applications, in terms of scalability of applications as well as the effects of various optimizations. They found that most applications scaled well with relatively simple optimizations for locality and load balancing. In their paper, they presented results for four applications, viz., Barnes, Ocean, Radix, and FFT. They found good speedups on their simulator as well as the Stanford DASH machine for up to 1024 processors.

The first detailed simulation study was done by Pai et al [PRA97] using a much more detailed execution driven simulator that modeled a Release Consistent parallel machine based on R10000 like processors. They found that modern ILP processors improve CPU performance much more than memory performance, causing relatively poor parallel efficiencies in machines based on such processors. This is in contrast to the earlier study by Holt et al, that found good parallel efficiencies for a range of applications on their simulator.

A recent study, by Laudon and Lenoski [LL97], on SGI Origin reported consistently good speedups for NAS benchmarks. Among the SPLASH-2 applications, they found good speedups for Barnes and Ocean, but poor speedups beyond 16 and 32 processors for Lu, Radiosity, and Raytrace. They do not report results for other SPLASH-2 applications. At the time of writing the paper, they were still investigating the causes for this behavior. They conjecture that small data sets, owing to limited amount of memory per node, could be a reason for the fall in speedups beyond 32 processors.

An intersting study by Remzi et al [ACK+95] at UC Berkeley, involved what they called "microbenchmarks". These are small tight loops written to test a specific parameter of a parallel machine using the "grey box" approach. This inspired us to use similar small probes to test local and remote memory latencies on SGI Origin.

1.3 Motivation

One motivation behind this project is to validate the results of Holt et al, on a modern industrial machine. In the wake of their simulation model that models a previous generation processor, it is interesting to investigate whether the results obtained using such a simulation model can accurately predict the behavior of applications on a real modern ccNUMA machine that employs heavy weight ILP processors at every node. The detailed simulation study by Pai et al, indicates that there is reason to believe that an ILP based parallel machine, like SGI Origin, will perform differently.

Apart from the optimizations for locality and load balancing that these applications have built in them, compiler optimization can become important given that compiler technology has improved considerably. A real machine provides a good opportunity to study the effects of compiler optimizations on sequential and parallel versions of applications.

One optimization technique that was explored for all applications in the study by Holt et al, was the effect of data placement. A larger difference between remote and local memory latencies can enhance the benefits of data placement. In the simulation, the study assumes a remote to local latency ratio of 6:1. On SGI Origin, this ratio is no more than 2:1. Therefore, it is not clear at the outset the degree to which data placement can improve performance in parallel applications.

2 Experimental Environment

2.1 The ccNUMA Machine Used

The machine used is SGI's Origin 20000 (referred to as the Origin). It is a directory-based cache-coherent shared memory machine. Origin20000 architecture can scale to 128 processors using hypercube links and over 1000 processors using a ring connecting 128-processor units.

2.1.1 R10K Uniprocessor

The uni-processor used in Origin is MIPS R10000 (referred to as R10K). R10K is an advanced superscalar processor. It can fetch and decode four instructions per cycle and run them on five pipelined functional units: a non-blocking load-store unit, 2 integer ALUs, a pipelined floating-point adder, and a pipelined floating-point multiplier. It can graduate one floating-point addition and one floating-point multiply every cycle, so the peak rate is 390MFlops.

R10K uses register remapping, out-of-order and speculative execution in conjunction with non-blocking cache to achieve higher instruction throughput.

Each R10K has 32KB first-level(L1) instruction cache and 32KB first-level (L1) data cache. The second-level (L2) cache can be 1MB or 4MB. Both level caches are two-way set associative and non-blocking. Up to 4 outstanding misses from the combined two levels of cache are supported. The latency is 2 cycles for L1 hits and 8 to 10 cycles for L1 misses that hit in L2.

In order to hide the latency of cache misses, R10K dynamically executes instructions whose operands are available. Its instruction buffer can hold up to 32 active instructions at any given time. Register renaming is performed to allow out-of-order issuing. In order to do renaming, the processor has twice as many physical registers as logical registers (64 integer and 64 floating-point physical registers).

R10K can speculate 4 nested branches. According to the manufacturer's data, the branch prediction scheme used predicts correctly 87% of the times for SpecInt92 programs.

2.1.2 Origin Multi-processor Organization

Each node of Origin consists of a special communication device called the Hub, which connects to two R10K processors and a memory module. The Hub controls all the accesses of processors to memory. The cache-coherent protocol and all communications in and out the node are also managed by the Hub. Nodes are connected by linking all Hubs together in a hypercube architecture. The protocol used is a directory-based cache-coherent invalidation-based protocol. Directory information is maintained in the memory module. The Hub manages all memory accesses of the processors, both to local and to remote memory. Figure 1 illustrates the node structure.

The nodes are connected by a hypercube architecture. Routers are used to connect nodes and other routers. Figure 2 shows the organization of a 32-processor system.

Larger systems are linked by higher dimension hypercubes up to 128 processors. Due to the hypercube network architecture, the memory bandwidth and remote latency scales nicely with the number of processors. The following table gives the system bandwidth and memory latency.

		Peak Mem BW (GB/s)		Read Latency (ns)	
No. Nodes	No. Procs	Overall	Read	Max	Avg
1	2	0.68	0.59	313	313
2	4	1.37	1.19	497	405
4	8	2.73	2.38	601	528
8	16	5.47	4.75	703	641
16	32	10.94	9.50	805	710

The first column is the number of nodes in each configuration. The number of R10K processors

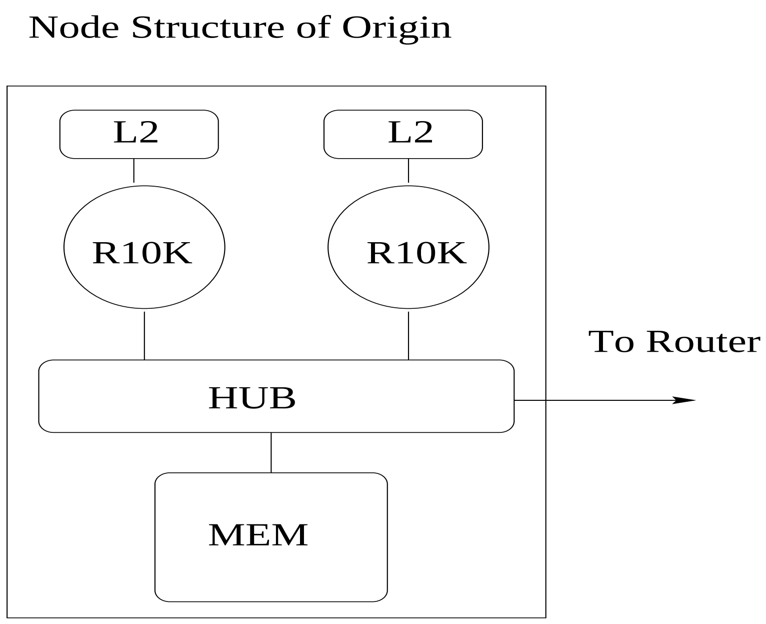

Figure 1: Structure of the Origin Node

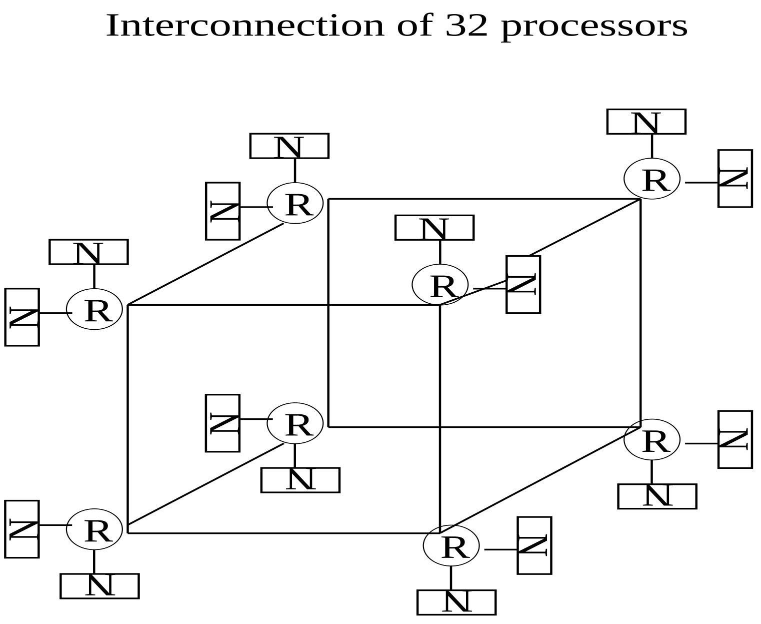

Figure 2: Organization of a 32 processor Origin

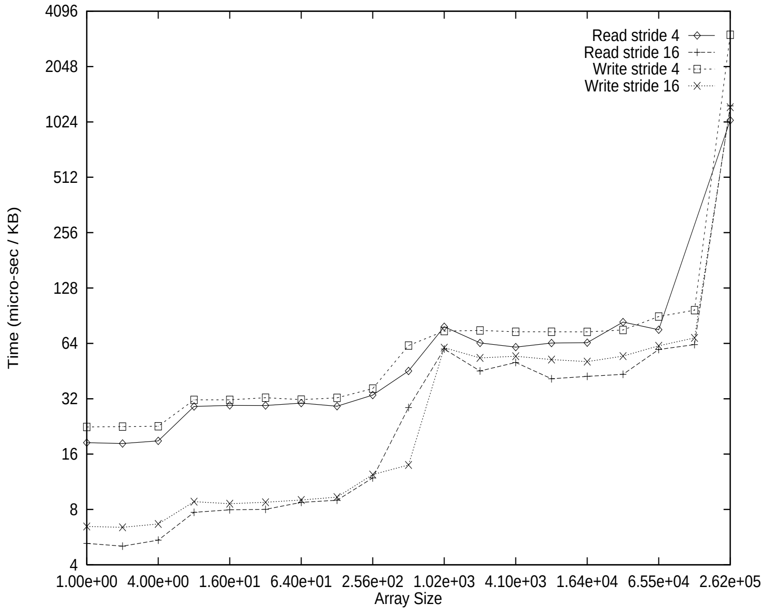

Figure 3: Memory Latencies

is two times the number of nodes (column two). The next two columns give the bandwidth and memory read latency for each of the configuration. The remote bandwidth scales linearly with the number of processors and the remote latency scales with the logarithm of the number of processors.

2.1.3 The Machine Used

The machine we used has 24 195MHz R10K processors. Each of the 12 memory modules has 320MB memory and the total memory is 3.84GB.

The 24 processors reside in 12 nodes, which are linked by 6 routers to form three quarters of a cube. Hence, the ratio between remote and local latency is less than 3 in the worse case and around 2 in an average case.

To verify these numbers we ran micro-benchmarks to measure relative latencies of different levels of memory. Figure 3 gives the relative read and write latencies for an array of long int's of varying size. The numbers are micro-seconds to traverse per KB of the array. We observe three distinct "jumps" in all the plots. The first, and smallest, jump is when array size grows beyond L1 cache. The second jump corresponds to going from L2 to memory. Finally, the third, and largest, jump occurs to the far right of the graph for very large input. This is likely to be due to increased thrashing of virtual memory system. The latency difference between L2 cache and memory is around 5, which agrees with the manufacturer's published figures.

2.2 Compilers and Tools

2.2.1 The Compiler Used

The compiler we used is MIPSpro 7.10. The major optimizations it performs are as follows. (A good text for all the following materials can be found in Ken's book [AKpt]).

- software pipelining
- inter-procedural analysis
- loop nested optimizations
  - loop interchange
  - out loop unrolling and blocking
  - loop distribution and loop fusion
  - prefetching

The flags we used are the recommended ones for best performance. All of the above optimizations are enabled. Although it is said that cache optimization may degrade performance of programs that have been manually optimized for cache performance, we found that the recommended optimization flags did give the best performance for the programs used.

2.2.2 Data Placement Tools

Origin provides powerful tools for data placement. A programmer can declare the number of threads to run and allocate a given number of memory modules. Different parts of the virtual address and different threads can then be explicitly associated with different memory modules and processors.

Once data and threads are placed, the operating system will always try to run a given thread on its allocated processor.

The data placement tools of Origin make it possible to study the effect of data placement optimizations. One can explicitly place data on one memory module or on multiple memory modules in either round-robin or blocked fashion.

3 Applications

This section describes the three applications used by our study – FFT, Radix, and Barnes Hut – out of the SPLASH-2 benchmark suite. Each description introduces the applications and mentions the expected behavior in terms of parallel efficiency observed by earlier simulation studies, and potential bottlenecks.

3.0.3 FFT

FFT is a fast Fourier transform program. It contains two transform phases, one forward and one backward, and two matrix transposes. The major data structures are square matrices of complex numbers. They are organized as row-major matrices. The size of each matrix must be an even number power of 2. Let the input size be 2^M, then the matrices are $2^{M/2} \times 2^{M/2}$.

The program is an explicit parallel program where each thread transforms and transposes a block of contiguous rows. The transformation phase is fully parallelizable. Data movement is local in this phase. The transpose phase, however, requires global communication in an all-to-all fashion, since each processor needs to read a contiguous column of one matrix and write to the rows (local) of another matrix. In the process, each processor needs to communicate with all other processors.

Two optimizations were proposed to improve its scalability on a ccNUMA machine ([HSH96]): data placement and staggered communication.

The data placement optimization puts the local rows of each processor into the processor's own private memory. Thus memory accesses of the transformation phase can be satisfied locally. There is a problem with the data placement. At the beginning of the transpose, all processors will contend with each other in communicating with the first processor. To solve this contention, the staggered communication optimization changes the access pattern of transpose so that at any given time, each processor is communicating with a different processor.

To avoid false sharing, each row of the matrix is padded in such a way that no two rows share the same cache line and no two partitions of rows share the same page.

The goals of experiment on FFT are:

- examine the scalability of FFT on Origin, and
- examine the effect of data placement optimizations.

3.1 Radix Sort

Radix sort is an integer sorting kernel from the SPLASH-2 application suite. Radix performs an iterative sweep over the input keys for each radix digit. Each node forms a local histogram of its assigned keys and then merges this with the other nodes' local histograms to form a global histogram for all keys. Each node uses this global histogram to permute its keys for the next iteration.

Radix's major performance hindrance is its irregular communication structure. During the permutation phase, each processor iterates over its keys and writes them to a destination array in a scattered fashion. As the number of processors increase, the number of remote writes during this phase also increases. This leads to resource contention and hotspotting which increases the write latency and degrades performance. Another potential bottleneck of Radix is the merging of the local histograms. Some implementations utilize a linear communication structure to create the global histogram. This creates contention at the node that owns the global histogram.

The implementation of Radix used in these experiments benefits from several optimizations. To create the global histogram, this implementation utilizes a parallel prefix tree as opposed to the linear communication structure. The prefix tree affords more parallelism thus reducing contention and the time processors spend merging the local histograms. Another attempted optimization is data placement. This consists of distributing the portion of the key and individual rank data structures to the nodes that use them most. Similarly, the global rank data structure and prefix tree are distributed among the nodes such that the portion each node computes on is in the local node's memory. In this way, a node's keys are allocated in the nodes local memory, thus reducing remote communication.

3.1.1 Barnes Hut

Barnes Hut is an N-body simulation of a set of bodies evolving under the influence of gravitational force. To reduce the algorithmic complexity, it uses a hierarchical octree representation of three dimensional space and all bodies beyond a certain neighborhood are approximated by point mass.

The application executes in time steps and parallelism is exploited within the phases of each time step, across particles. Each time step involves an upward pass through the tree to find the center-of-mass of each cell, represented by a node of the octree, partition the bodies among processors, compute forces on all bodies, and advance the body positions. Almost the entire time is consumed in the force calculation phase. Only the tree building and center-of-mass computation phases require communication and synchronization.

Since the bodies evolve over time in an unpredictable way, the application is quite irregular. Communication patterns too are irregular. However, the application has been found to have a small working set that grows relatively slowly with the input data size. As a result, the working set fits almost entirely in the cache and locality enhancing optimizations are not expected to yield much benefit.

Indeed, the previously mentioned simulation study found that data placement and prefetching techniques were hardly useful for this application. The only major architectural bottleneck is latency. Most work needed to optimize the application is algorithmic in nature, like building trees with multiple particles per leaf, and dynamic partitioning of data. These form a part of the standard application that is included in SPLASH-2 suite.

Nevertheless, owing to its small working set behavior, Barnes Hut has been found to scale very well and deliver good speedups on relatively small input sizes.

4 Results

Figure 4 summarizes the speedups obtained on SGI Origin for the three applications. The figure shows speedups for the data sizes suggested in the study by Holt et al, and the maximum speedups obtained on Origin using maximum possible data sizes. For comparison, the figure also shows the speedup corresponding to 70% parallel efficiency for each application.

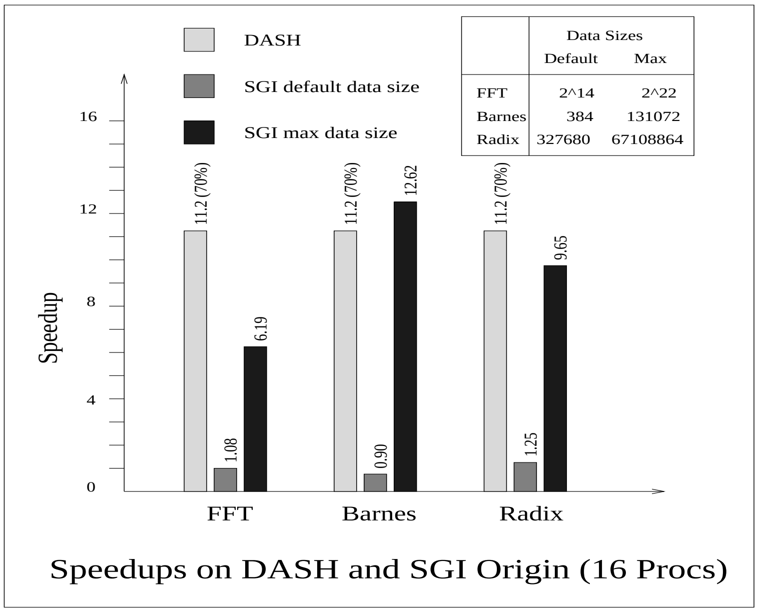

Figure 4: Comparison of Speedups with Default and Maximum Data Sizes

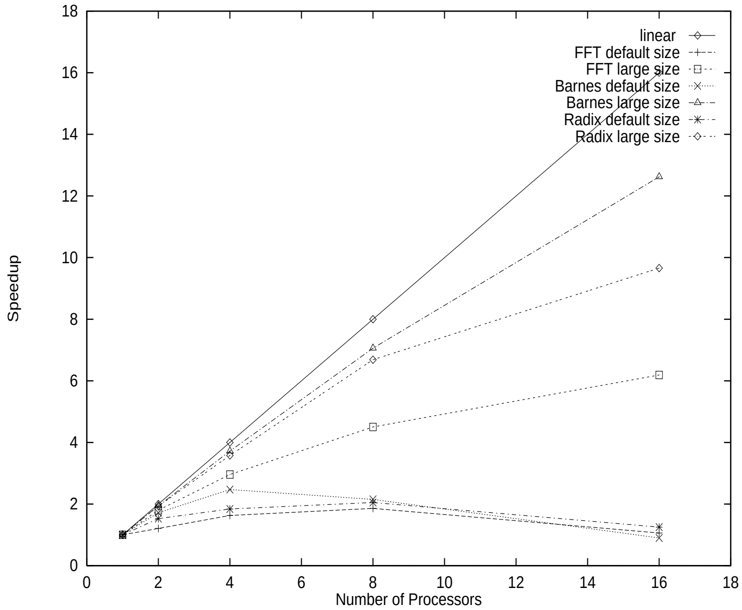

Figure 5: Speedups for Default and Maximum Data Sizes

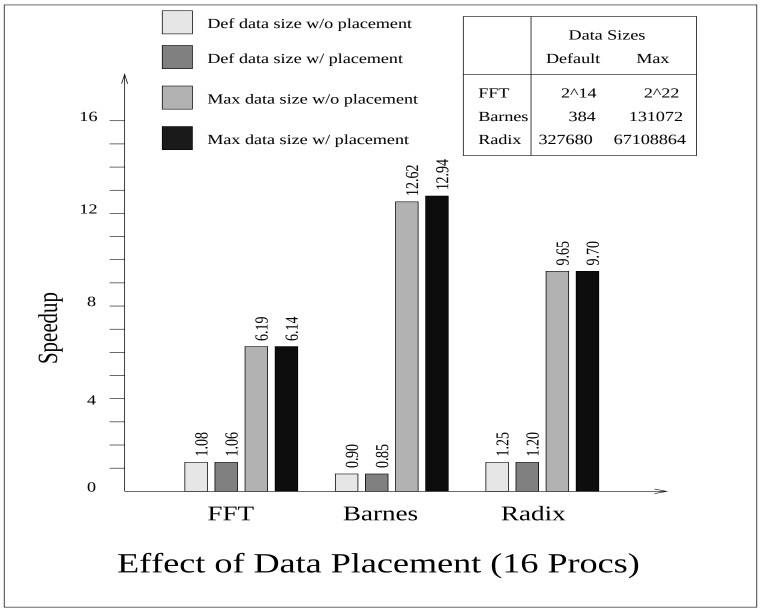

Figure 6: Effect of Data Placement on Speedups

The speedups obtained on 1 to 16 processors are summarized in figure 5. The figure shows the speedups for three applications for default as well as maximum data sizes.

Finally, figure 6 shows the effects of data placement on speedups for the three applications.

We found two things that are quite different from the previous belief on ccNUMA machines,

- the unexpected poor speedup for FFT and Radix, especially FFT, and
- data placement optimizations have no observable effect on performance up to 16 processors.

The Origin machine we used is different from the previous ccNUMA models in the following three aspects:

- the uni-processor used, R10K, is much more powerful than previous uni-processors,
- the optimizing compiler for the Origin is also more aggressive than previous compilers,
- the hypercube communication architecture is more effective in minimizing the latency and bandwidth difference of local and remote memory access,

In this section, we will show that these three differences are the reason for the unexpected results we found. First, the advanced uni-processor and compiler improve the sequential execution much more effectively than they do the parallel execution. Therefore, the parallel efficiency and the speedup are reduced. Second, the advanced uni-processor, along with the aggressive communication architecture also reduces the penalty of remote access. Thus, the effect of data placement is not significant for processor counts up to 16.

The next three subsections investigates the performance difference made by the advanced uni-processor, compiler, and the communication architecture of the Origin. We then compare these differences with the architectural assumptions used in two previous simulation studies to explain why some of our results are almost opposite to what was found before. The last three subsections give a detailed analysis on each of the applications we studied.

4.1 Impact of the Advanced Uni-processor

As explained in the previous section, the uni-processor used on the Origin machine, R10K, is an aggressive superscalar processor that is quite different from previous-generation micro-processors. R10K effectively improves the instruction throughput and reduces the memory access penalty for sequential executions. However, it does not always give similar improvement to parallel executions.

The dynamic and speculative execution of R10K can effectively hide penalties due to program dependences. With the increased capability of instruction throughput, the memory latency become an important, even dominant part of the execution time. To reduce the time waiting for memory, R10K exploits non-blocking caches to overlap memory latency with instruction execution and other memory accesses. With these techniques, R10K greatly improves its uni-processor performance.

Given the performance improvement of R10K over previous-generation processors, one might think that parallel programs using R10Ks can also benefit as well as sequential programs. However, a simulation study done by Pai et al showed that the advanced features generally have a negative effect on the speedup of a parallel machine. In another words, modern uni-processors give relatively poor improvement to parallel programs.

Pai et al compared parallel execution on multi-processors that use R10K-like uni-processors and multi-processors that use previous-generation uni-processors (single-issue, static scheduling, and with blocking reads). There are a number of differences between the machine and processor model they used and that of the Origin. We will present some of their results here and cover their study in more detail in a later section. They divide the major parallel execution time into the time spent on instruction execution (CPU time) and the time spent waiting for memory (memory time). They found that the system with advanced processors achieved significant speedup over CPU time (3.15 to 3.70) but the speedup over memory time is much lower than that of CPU time (0.74 to 2.61). In some cases, e.g. Radix (memory speedup of 0.74), the memory time was actually longer in parallel execution than in sequential execution.

Unfortunately, a parallel execution often spends a much greater portion of the execution time waiting for memory due to two reasons. First, the remote memory access takes longer than local memory access. Second, the coherence misses increase the number of memory accesses in parallel executions. Since memory time is a larger part in parallel executions than that in sequential executions, the low memory speedup of the advanced uni-processor causes an imbalanced effect. Indeed, the parallel execution benefits much less from the advanced processors than the sequential execution. This effect directly implies a lower parallel speedup on a multi-processor using advanced uni-processors than the parallel speedup using previous-generation processors.

4.2 Impact of the Optimizing Compiler

R10K, though having very powerful capacity, is limited in its optimizing scope since its instruction window is only 32 instructions long. Therefore, compiler optimizations are essential to help programs to maximaly exploit computing power, especially for scientific programs. We found that the optimizing compiler on the Origin can improve sequential execution by several times. However, the same optimizing compiler has a much limited effect on parallel executions for some applications. The imbalanced effect of the compiler further worsens the lowered speedup.

In the following description, we call the program compiled with the flags described in section 2.2.1 as the optimized version and the program compiled with the "-O1" flag as the non-optimized version. The optimized version is compiled with all the optimizations describe in (previous section), including software pipelining, loop restructuring and software prefetch, whereas the non-optimized version is compiled with only local optimizations.

The following table illustrates the performance difference between the optimized version and the non-optimized version for both sequential and parallel executions. Two input data sizes, the smallest and the largest, are shown for each application.

Application	Input Size	Speedup (1 procs)	Speedup (16 procs)
FFT	M = 12	3.8	1.2
FFT	M = 22	3.2	1.8
Radix	327K keys	2.2	1.2
Radix	67M keys	1.4	1.3
Barnes	384 particles	2.5	1.1
Barnes	128K part.	2.4	2.9

The third and fourth column give the sequential and the parallel (16 procs) execution speedup of the optimized over the non-optimized version respectively. The speedup figure in the third column is significantly greater than the figure in the fourth column (for FFT and Radix). The differences show that the compiler optimizations are more effective on the sequential programs than on parallel programs. Therefore, the effect of optimizing compiler reduces the parallel speedup.

Figure 7 shows the effect of the optimizing compiler in another view. It compares the scalability of the optimized version and the non-optimized version for all three applications. The scalability of non-optimized version is significantly better than that of the optimized version.

As we can see from the figure, although the optimizing compiler improves both sequential and parallel execution time, it does lower the scalability of applications due to its imbalanced effect on sequential execution and on parallel execution. Therefore, the effect of the optimizing compiler, along with the effect of the advanced processor R10K on the Origin counts for a dramatic reduction in the scalability of FFT and Radix.

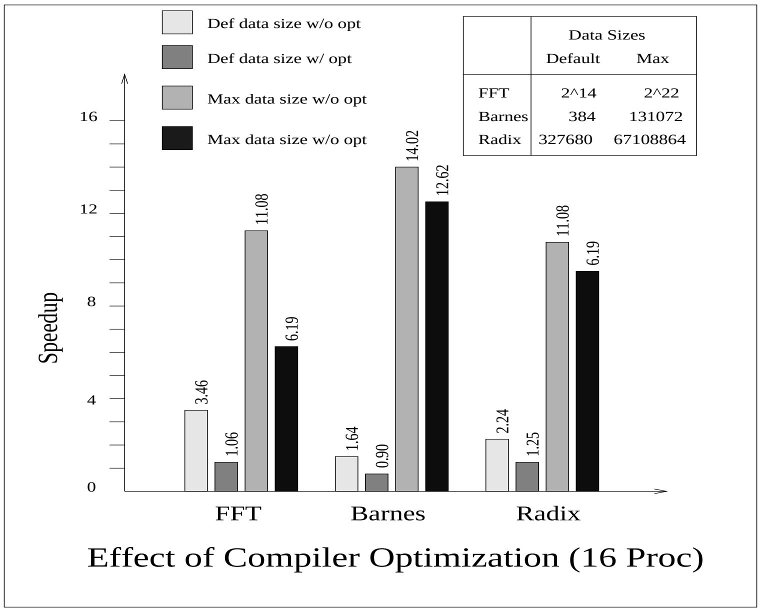

Figure 7: Effect of Compiler Optimization of Program Speedups

4.3 Effect of Data Placement on the Origin

The previous two sections investigate the causes for poor scalability of FFT and Radix. In this section, we will explore the reason why data placement optimizations do not perform as well on CC-NUMA machines as previously expected.

Data placement optimizations can benefit ccNUMA machines in many ways. First, as the remote latency is larger than the local latency, placing data locally can change remote access to local access and reduce memory latency. Second, when remote memory bandwidth is smaller than the local memory bandwidth, locally placed data can avoid adding contention to the network. Data placement can also be used to avoid contention to a given memory module in parallel execution.

The effect of the non-blocking cache used in R10K reduces the benefit of changing remote access to local access by data placement. A non-blocking cache makes it possible to overlap the memory loads (both remote and local) with instruction executions or other memory accesses. The load latency is no longer fully exposed to the execution time. If there are sufficient overlaps to hide remote latency, changing memory access from remote to local would not help execution time directly.

In addition to the non-blocking cache, the communication structure of the Origin is quite effective in reducing latency and bandwidth differences between remote and local memory access, therefore further reducing the need for data placement.

As we have described in section 1.4, the difference of remote and local access latency is not large for 16 processor configurations. The ratio between the average remote latency and the average local latency is close to 2. The table in section 2.1.2 also showed that the memory bandwidth increases linearly with the number of processors. Therefore, changing local access to remote access would not cause contention on the network. We believe these two features of the Origin reduce the need for data placement to the degree that data placement optimizations have no significant effect for the three applications up to 16 processors.

However, this result does not imply that data placement is not important for the Origin for larger number of processors. Next, we will describe one case where data placement did give a positive performance impact when we intentionally added heavier loads to the memory system.

To give heavier load to the memory system in the transpose phase of FFT, we disabled the cache blocking intentionally. With cache blocking, only one out of every 16 memory accesses is an L2 cache miss. Without cache blocking, the number of L2 cache misses (the number of memory accesses) should be increased. We ran FFT with the largest input data size, M = 22. The data placement version (with staggered communication) did achieve slightly better speedups over non-data placement version. However, we haven't investigated further the reason for this result.

4.4 Comparison with Previous Simulation Studies

Our work has a purpose similar to two previous simulation studies: the simulation study on the scalability of ccNUMA done by Holt et al, and the simulation study on the impact of advanced uni-processors on multi-processor performance done by Pai et al. The results we found on the Origin have some dramatic differences with both previous studies. In this section, we will compare our results with each of the simulation study. We will single out the differences between our machine model (the Origin) with their machine model that are responsible for the different results. For a complete description of the results and assumptions of the two simulation studies, we refer the reader to their original papers.

The study of Holt et al found that the minimal data size for achieving 11.2 speedup on a 16-processor is surprisingly small for FFT, Barnes and Radix, and data placement is key to the performance of all applications. Our results for the scalability of FFT and Radix and the effect of data placement optimizations are almost the opposite. The reason their results are not accurate for the Origin machine lies on the assumptions they made. The uni-processor they modeled is single-issue, static scheduling, and with blocking cache. As we described previously, this assumption could not take into account the effect of modern uni-processor and today's optimizing compiler. Therefore, their estimation of the scalability of FFT and Radix was quite over-optimistic.

The simulated multi-processor in their model has a much higher remote access penalty than the Origin. The network topology they used is a two-dimensional mesh whereas the network of Origin is hypercube. As a result, they assumed a longer remote access latency. For 64 processors, the ratio of remote latency over local latency is about 2.9. The remote memory bandwidth is also smaller than the local bandwidth in their model. As described in the previous section, the blocking cache of their uni-processor model and the less aggressive communication architecture they used can lead to their observation that data placement optimizations have significant effect on 16 processors.

The study done by Pai et al is the first one that performed an extensive experiment and analysis on the impact of modern micro-processors on the performance of ccNUMA multi-processor. The uni-processor they modeled, in fact, is very similar to the processor used on Origin. However, our result of FFT is quite different from theirs. They observed a 5.9 speedup of FFT on problem size of M equal to 16; our result showed a speedup of 3.4. There are a number of reasons that may cause this difference. The memory consistency model they used is release consistency and Origin uses sequential sequency. Sequential consistency is more sensitive to memory waiting time than release consistency. It can be expected that with sequential consistency, the disparity between the speedups of memory and CPU time is even larger than the release consistency case. Therefore, the effect of the modern uni-processor causes more reduction of the parallel speedup on Origin than on the model used in their study. The other important reason is the difference in the compiler [1]. The compiler they used is a version of the SPARC V9 gcc compiler with flags of "-O2 -funrollloop". This compiler is unlikely to have the advanced optimizing abilities of the MIPS Pro compiler on Origin. Some important optimizations, software pipelining for example, are highly sensitive to the machine architecture that is targeted. Any such optimizations not tuned for R10K specifically is likely to be much less effective compared to a compiler tuned for Origin. Therefore, their study may not have fully considered the effect of the optimizing compilers.

4.5 Detailed Study of Individual Applications

4.5.1 FFT

For FFT, timing was also collected for the transpose phase alone. By computing the parallel speedup for the transpose phase and for the rest part of the execution, we found that the transpose phase scales much poorer than the rest of FFT. For data size of M equal to 12, the transpose phase is slowed down for all processor counts greater than 1, ranging from 71% for 2 procs to 42% for 16 procs. The speedup of the remaining part is always greater than 1 and is about 1.77 for 16 procs. A similar difference of scalability is also shown for the largest data size (M=22). The speedup of transpose phase for 16 procs is 2.4 whereas the speedup for the rest part is 12.2.

The causes of the poor speedup of the transpose phase are still under investigation. We haven't characterized the effects of ccNUMA architecture on this kind of computation.

4.6 Radix Analysis

Timing was also collected for the ranking and sorting subsections of Radix. The results indicate two points. First, the sort time for Radix is typically much larger than the rank time. In general, the sort time is at least 4 times the rank time. Second, the sort phase of Radix scales much better than the rank time. For a data size of 4194304 keys and 16 processors, the sort phase demonstrates a parallel efficiency of 56% (speedup of 8.90) while the rank phase has an efficiency of only 37% (speedup of 5.98). The performance gap decreases as the data size increases, but the sort phase still outperforms the rank phase. For 67108864 keys and 16 processors, the sort efficiency is about 63% and the rank efficiency about 61%.

[1] They did perform many optimizations at the source level, such as cache blocking and prefetching. However, our experience showed that the optimizing effect of the MIPS Pro compiler is orthogonal to many of these optimizations.

In addition, the sorting portion of Radix dominates the overall performance of Radix. Because the sort phase is much longer, its affects on the total performance overshadow those of the rank phase. Hence, the speedup of Radix as a whole is influenced mostly by the performance of the sort phase.

The poor speedup of Radix can again be attributed to the aggressive nature of the processor and the irregular communication requirements of the application itself. For small data sizes, the number of keys each processor computes on is not large enough to overcome the communication bottlenecks and the high uniprocessor performance of SGI system. As a result, the speedups obtained are generally poor. As the number of input keys increases, however, each processor obtains more keys to compute on. For very large data sizes, the processors have enough keys that the communication penalties are somewhat overcome by the increased computation. Nevertheless, performance improvement increases very slowly with data size and may not be practical for greater than 16 processors.

4.6.1 Barnes Hut

Barnes Hut behaved as predicted by the simulation study. It scaled very well, and data placement hardly affected the performance.

Particles	Number of Processors							
	2		4		8		16	
	w/ P	w/o P	w/ P	w/o P	w/ P	w/o P	w/ P	w/o P
384	1.72	1.72	2.58	2.47	2.65	2.15	0.85	0.90
16384	1.94	1.93	3.66	3.63	6.51	6.46	8.49	8.52
131072	1.94	1.94	3.76	3.72	7.22	7.06	12.94	12.62
1048576	1.87	1.83	3.60	3.38	6.78	6.20	12.52	12.31

In the above table "w/ P" indicates with placement and "w/o P" indicates without placement strategy. The above results are the best of several experiments staggered over time, in order to minimize the effect of varying load conditions.

We notice very good speedups except for the first set which is for 384 particles – the data size that the simulation found adequate for 70% efficiency on 16 processors – and the second set which is the default input size. Clearly, these sizes are too small for the SGI Origin due to its high performance single processor. Beyond 128K particles, the speedups remain almost unchanged.

Also, as predicted by the earlier study, data placement has practically no effect on parallel performance of Barnes.

Numbers for just the force calculation phase, which is the major computation phase, show near perfect speedup. Notice that this part shows very good speedups even for small data sizes, thus validating the hypothesis that CPU performance scales well while the memory performance (latencies) are the bottleneck.

Particles	Number of Processors							
	2		4		8		16	
	w/ P	w/o P	w/ P	w/o P	w/ P	w/o P	w/ P	w/o P
384	1.92	1.94	3.62	3.71	6.94	7.19	14.26	14.20
16384	1.99	1.99	3.95	3.94	7.78	7.80	15.35	15.33
131072	1.99	1.99	3.95	3.92	7.85	7.77	15.57	15.45
1048576	1.99	1.73	3.90	3.28	7.78	6.50	15.31	13.21

Comparison of program performance for optimized and non-optimized versions of the code for 384 and 131072 (128K) input particles is summarized in the table below. It shows the computation ratio for unoptimized and optimized versions of the program on different number of processors for the two input sizes.

Particles	Number of Processors				
	1	2	4	8	16
384	2.50	2.25	1.74	1.32	0.72
131072	2.36	2.32	2.33	2.25	2.17

Optimization results in a more than two fold improvement in performance for sequential program. But the advantage diminishes with increasing number of processors. This is due to the memory component of the program, as mentioned elsewhere in this report, that does not benefit from compiler optimization. Also, for a larger data size the gains of optimization reduce more gradually with increased parallelism, thereby maintaining a higher parallel efficiency – this is corroborated by the observed behavior.

5 Conclusion

This study examined the scalability and the effect of data placement on a state-of-art directory-based ccNUMA machine, SGI Origin. We were able to run large test cases and perform data placement explicitly for up to 16 processors.

We tested three applications from SPLASH2: FFT, Radix and Barnes-Hut. The result of two input data sizes were studied in detail. The first data size is the one that achieved 70% efficiency (11.2 speedup on 16 processors) in a recent simulation study by the FLASH group. The second data size is the largest one that fits in the memory of the machine.

Here is a summary of scalability results on 16 processors. We found that the speedups of all three applications are less than 1.3 for the first data size, corresponding to a parallel efficiency of about 8%. For larger data sizes, Barnes-Hut achieves consistently good speedups of more than 12 for data sizes of beyond 128K particles. However, both FFT and Radix cannot scale to 70% efficiency even for the largest data size. The maximum speedup is 6.2 for FFT and 9.7 for Radix. We found that FFT does not scale to 70% efficiency even for 8 processors.

For all input data sizes of all three applications, we tested the performance with and without data placement optimizations. We found that data placement does not make any observable differences for all cases.

These results are quite surprising given the recent simulation study by Holt et al[reference]. They found that FFT, Radix and Barnes-Hut can achieve 70% efficiency for 16 processors for fairly small input data sizes. They also found that data placement optimization is vital to achieve scalable performance.

We attribute the reason for the unexpectedly poor scalability to the effect of the advanced uni-processor and the optimizing compiler. The uni-processor, R10K, is a four-way issue, dynamically scheduled, with non-blocking cache and speculative execution. As observed by [refer to Pai], these features generally lower the parallel efficiency. However, our result for FFT still scales much worse than the result of Pai et al. One important reason for this is the difference of the compiler. The optimizing compiler on the Origin is designed to maximaly utilize the advanced architecture. For FFT and Radix, we found that the optimizing compiler improved the performance of sequential executions much more than it does parallel execution. For FFT, the compiler optimizations improved sequential executions by more than a factor of 3 but less than a factor of 2 for parallel executions using 16 processors.

The reasons for the negligible effect of data placement on the Origin are similar to those for scalability. The latency-hiding techniques of the R10K processor can tolerate more remote latency penalties. The aggressive communication architecture of the Origin achieves a remote memory latency close to local memory latency and the same remote memory bandwidth as local bandwidth for the processor configurations we used. These reasons together avoid the remote access penalty and memory contention due to poor data placement for the application we studied and the processor counts for which we tested.

6 Future Work

To get a more complete idea on the scalability of a ccNUMA machine like the Origin, we need to test more applications using more processors. In particular, we need to examine data parallel benchmarks as well as other applications from SPLASH-2.

Data placement should be important given the hardware limit of the communication architecture. We need to examine data placement using larger number of processors to see at which point data placement becomes important for ccNUMA machines.

We also need to identify the performance problems of FFT and Radix more specifically.

We found that current compiler optimizations provide different benefits for sequential and parallel programs. However, the exact reasons and the effects of individual compiler optimizations is not very clearly understood.

The SGI Origin is a directory-based ccNUMA architecture. A very different ccNUMA architecture, represented by HP-Convex Exemplar machine, is also commercially available. Studying its scalability and effects of data placement would also be helpful in understanding the inherent characteristics of ccNUMA architectures in general.

7 Acknowledgement

We wish to thank Prof Kennedy, Dr Mellor-Crummey, and Dr Dennis Moreau for providing us access to the Origin. Thanks to Mr Eaton who provided prompt and complete help for every question we asked about the machine. Dr Vikram Adve gave us very valuable help on the whole process of this project. Hazim's encouragements and support were helpful in dealing with implementation issues. Finally, special thanks to Dr Sarita Adve, for her classes and her inspirations.

References

[ACK+95] Remzi H Arpaci, David R Culler, Arvind Krishnamurthy, Steve G Steinberg, and Katherine Yelick. Empirical evaluation of the Cray-T3D: A compiler perspective. In *Proceedings of the 22nd annual ISCA*, June 1995.

[AKpt] R Allen and K Kennedy. *Advanced Compilation for Vector and Parallel Computers*. Manuscript.

[HSH96] Chris Holt, Jaswinder Pal Singh, and John Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *Proceedings of the 23rd Annual ISCA*, 1996.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th annual ISCA*, 1997.

[PRA97] Vijay S Pai, Parthasarathy Ranganathan, and Sarita Adve. The impact of instruction-level parallelism on mutliprocessor performance and simulation methodology. In *Proceedings of HPCA-3*, February 1997.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual ISCA*, June 1995.