An Empirical Evaluation of the Convex SPP-1000 Hierarchical Shared Memory System

Thomas Sterling
Center of Excellence in Space Data
and Information Sciences
Code 930.5 NASA Goddard Space Flight Center
Greenbelt, MD 20771
tron@cesdis.gsfc.nasa.gov

Daniel Savarese
Department of Computer Science
University of Maryland
College Park, MD 20742
dfs@cs.umd.edu

Phillip Merkey
Center of Excellence in Space Data
and Information Sciences
Code 930.5 NASA Goddard Space Flight Center
Greenbelt, MD 20771
merk@cesdis.gsfc.nasa.gov

Kevin Olson
Institute for Computational Science
and Informatics
George Mason University
olson@jeans.gsfc.nasa.gov

Abstract

Cache coherency in a scalable parallel computer architecture requires mechanisms beyond the conventional common bus based snooping approaches which are limited to about 16 processors. The new Convex SPP-1000 achieves cache coherency across 128 processors through a two-level shared memory NUMA structure employing directory based and SCI protocol mechanisms. While hardware support for managing a common global name space minimizes overhead costs and simplifies programming, latency considerations for remote accesses may still dominate and can under unfavorable conditions constrain scalability. This paper provides the first published evaluation of the SPP-1000 hierarchical cache coherency mechanisms from the perspective of measured latency and its impact on basic global flow control mechanisms, scaling of a parallel science code, and sensitivity of cache miss rates to system scale. It is shown that global remote access latency is only a factor of seven greater than that of local cache miss penalty and that scaling of a challenging scientific application is not severely degraded by the hierarchical structure for achieving consistency across the system processor caches.

1 Introduction

The emergence of scalable shared memory parallel architecture with full cache coherence provides a new and important operating point for high performance computing. The hardware assisted guarantee of cache coherence across all processors of a distributed computer structure provides the opportunity for improved algorithms, more productive methods of parallel programming, and more efficient compilers than possible with fragmented address-space distributed systems. This capability may also simplify dynamic load balancing by runtime system software for adaptive resource management, and ease the development of distributed operating systems. The Convex SPP-1000 is the most recent entry in the commercial sector of scalable cache coherent architectures. Its structure is a two-level hierarchy with up to 16 clusters (hypernodes) of eight processors each. It employs a directory based cache management scheme at the lower (intra-hypernode) level and an SCI [11] protocol based global communication and cache control mechanism at the higher level. This paper provides an initial examination of the performance characteristics of the SPP-1000 hierarchical cache management.

In spite of its potential advantages, support for global shared memory and cache coherence on systems of wide diameter imposes a greater burden on the cache system itself for avoiding latency of data access than required of common bus based snooping mechanisms for small scale multiprocessors. This is due to the much larger possible miss penalty incurred from the longer remote access distances. While hardware mechanisms eliminate most of the overhead work attributed to managing remote access and data migration/distribution, the latency and possible consequences of shared resource contention still remain. This paper examines the Convex SPP-1000 to evaluate and quantify the costs incurred in performing critical actions across its two-level cache coherent structure. These results represent the first published findings of evaluation of the SPP-1000's hierarchical cache coherent architecture employing the SCI protocol and builds on earlier studies [15] of the performance properties of the first level of this new system. The findings presented in this paper are important to both parallel architecture and compiler designers in determining the feasibility of achieving effective parallel computation in a scalable framework for this class of architecture.

Cache coherence is a user transparent policy that maintains the appearance of sequential consistency across a physically distributed multiple access memory system. It ensures that read/write order to the memory address space in a physically distributed multiprocessor is equivalent to the access patterns of comparable uniprocessor execution. This includes correct manipulation of memory state for compound atomic operations such as test-and-set or fetch-and-add. While the semantic equivalence is achieved, time-domain behavior may be severely aggravated by the latencies intrinsic to the distributed nature of the large parallel systems and the additional work, even if hardware supported, required for realizing the complex local and global protocols for cache coherence. Other factors that may impact the overall performance include contention for shared resources such as memory banks and communication channels, side-effects of cache structural properties such as false sharing and conflict misses, and pollution of cache contents by other unrelated concurrent tasks.

Evaluation of the system capabilities and mechanisms involves revealing the attributes of both the system latency and cache mechanism overhead. In the previous work, the single level directory based cache coherence mechanism employed within a hypernode of eight processors was evaluated. This paper extends that work to expose the global system characteristics. Two basic mechanisms, synchronization and task scheduling, are studied. The first, using a non-cached barrier object, reveals the consequences of system latency without the cache consistency mechanisms. The measurements of task scheduling exposes the overhead costs to managing parallel resources and concurrent activities in a single common name-space system.

Of particular interest is the impact of the longer delays and SCI overhead on scalability across the entire system. This important factor is investigated through studies of a complete real-world application. This problem, taken from the Earth and space sciences community, is a tree-code algorithm for simulating N-body gravitational systems such as stellar clusters, galaxy formation, and cosmic scale structure evolution. This problem is peculiar to Earth and space science and imposes computational demands that are non-trivial to satisfy by parallel systems. The primary data structure, an Oct-Tree, is sparse, irregular, and time varying. The underlying physics requires significant amount of global access across the data structure and parallel system. Some runtime load balancing is required over the time frame of the simulation as the problem state distribution evolves. The scaling of this problem is studied with respect to scaling first with high locality (lower number of processors all in one hypernode) and second with uniform distribution of processors across hypernodes. It will be shown that the latter case shows some degradation over the highly local case but that the dominant scaling factor is the granularity of the work.

Finally, the cache miss rate is examined directly as it varies with problem size and system configuration scale. This is crucial to determining the overall effectiveness of cache consistency to system performance. If the cache miss rate were to increase for a fixed size workload because system size (number of processors) increases, then the value of this class of architecture would be in question as a means of enhancing scalability of distributed systems. It will be seen that cache miss rates do not vary significantly with system size although are heavily impacted by problem size. This study also measures the global latency and overhead times attributed to cache misses involving remote access.

A summary of the principal architectural characteristics of the Convex SPP-1000 is provided in section 2 of this paper with an emphasis on the system hierarchy and cache coherence mechanisms. Section 3 presents the results of experiments performed to study synchronization and thread scheduling costs. In this study, the basic latency and overhead factors are characterized. The scalability studies of the gravitational N-body application code are presented in Section 4 along with a brief explanation of the science problem addressed and a description of the parallel algorithm being applied. The results are provided to reveal the contributions of both problem size and task distribution. The sensitivity of cache misses to system and problem size is investigated in Section 5. Finally, the implications of the findings and the directions for future work are discussed in Section 6. It is noted that the researches represented here are of value because they provide an earliest quantitative examination at the full structure of this new architecture. But as a consequence, they suffer to a small degree due to the difficulties always present in the experimental context of any beta-test environment. Future work by this group in the near term will be directed to refining and expanding on the initial findings presented here.

2 Architecture

The objective of the Convex SPP-1000 architecture design is to leverage industry investment in high performance microprocessor technology, provide a highly scalable structure to satisfy a broad range of market needs, and support an easy to program parallel execution environment. The approach taken incorporates the HP PA-RISC [9] microprocessor in a hierarchical structure with hardware support for full global shared memory operation. Most notably, the architecture provides for full cache coherence, time and space sharing of system resources, and virtual memory for up to 128 processors in the current design. This combination of capabilities makes the SPP-1000 among the most advanced commercial parallel systems available and an interesting target for evaluation. This section briefly describes the SPP-1000 architecture to establish the context for the experimental results that follow.

2.1 Concepts

Key concepts embodied in the SPP-1000 architecture are:

- scalable hierarchical structure,
- global shared memory,
- hardware supported cache coherence,
- virtual memory across processor subcomplexes, and
- thread based and message based parallel execution.

The hierarchical organization of high performance microprocessors comprises three stages: functional units, clusters, and global interconnect. Functional units are the basic building blocks of the system incorporating dual processors and dual memory banks with interface logic. The cluster integrates up to four functional units and an I/O subunit coupled by a 5 port cross-bar switch. The global system stage interconnects up to sixteen clusters by four parallel rings using the Scalable Coherent Interface (SCI). The first generation SPP-1000 system may incorporate as many as 128 microprocessors.

Part of the memory in each functional unit is designated as global shared memory and is potentially accessible by any

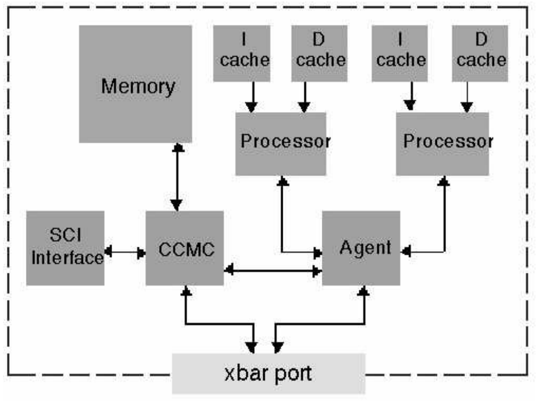

Figure 1: Convex SPP-1000 Functional Block

processor in the system. Therefore data movement between functional units in a cluster or between clusters is achieved by means of direct hardware support for load and store commands, independent of relative location of processor and the memory word being accessed. This distinguishes the memory system of the SPP-1000 from the distributed memory of the TMC CM-5 [14] or the Intel Paragon [10] with their fragmented address spaces. The CRI T3D [7] also has a global shared memory.

The primary means of ameliorating potential performance degradation due to access latency is through the use of caches which hold copies of data elements in fast buffer memory and rely on temporal and spatial locality for high hit rates. When a variable is shared by two or more processors at the same time, copies of the same variable in independent processors' respective caches can become corrupted such that different values for the same variable reside in the separate caches. To maintain consistency across caches, either special hardware is normally required (software techniques have been explored) or shared variables are expressly precluded from being stored in cache with the potential performance penalty this implies. The SPP-1000 employs a two-tier cache coherence scheme that is unique in the industry. It combines a directory based cache coherence mechanism within the cluster with the SCI protocol using an "interconnect cache" across clusters [5].

A true virtual memory system is supported. A parallel process with many concurrent sub-processes will have a single virtual address space. The virtual pages are mapped across the memory blocks of the functional units within a cluster to reduce the likelihood of bank conflicts and across clusters to distribute data objects more evenly among processors. Processors and physical memory pages may be organized in sub-complexes, a logical ensemble of processors from across the system that work together on a single problem. The virtual address space of a process running on the subcomplex is not limited to physical memory dedicated to that subcomplex. More than one parallel process may timeshare a given subcomplex.

The shared memory and cache coherence mechanisms enable a parallel thread model of execution to be directly supported along with the more common message passing model. The shared memory model facilitates implementation of a broad range of applications. Where memory accesses are global, dynamic, and non-uniform, parallel programs may run more efficiently than their message passing counterparts. Such primitive control flow operatives as barrier synchronization, fork-join task scheduling, and message passing are easily supported on a shared memory system. The rest of this section examines the physical elements of the SPP-1000 architecture in greater detail.

2.2 Cluster structure

A block diagram of the Functional Unit is shown in Figure 1 and presents the major elements of the SPP-1000 cluster. Each of two processors has its own separate 1 Mbyte data and instruction caches. The processor is the HP PA-RISC 7100 with a clock rate of 100 MHz. Two banks of memory are included, each of 64 Mbytes which may be increased to 256 Mbytes at a later time. The memory may be partitioned to serve as local storage for the cluster and global storage for subcomplexes spanning many clusters across the system. The Convex Coherent Memory Controller (CCMC) manages the interaction of the Functional Unit memory with the remaining system. Access requests can come from any processor within the cluster via the cross-bar switch interface or from other clusters via the SCI. Even access from processors within the same Functional Unit come through the cross-bar switch which provides arbitration. The CCMC also manages the directory based cache coherence protocol for the processor caches. The Processor Agent provides the interface between the processors and the rest of the system.

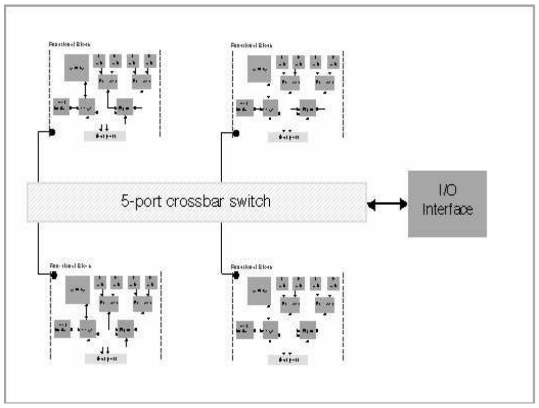

Figure 2: Convex SPP-1000 Cluster

The SPP-1000 cluster is depicted in Figure 2 showing the principal subsystems making up the cluster and their relationship to each other. Four Functional Units provide the processing and memory. The cluster processors access the cluster memory by means of the 5 port cross-bar switch. The remaining fifth port is used to access the I/O Interface which connects the cluster to external devices. The I/O Interface supports Ethernet, FDDI, SCSI, and other interface standards. A SPP-1000 cluster is self sufficient and can stand alone as a full and operational computing system.

2.3 Global Organization

Scalability for the SPP-1000 is achieved through a scheme that combines data consistency with global communications. In a manner reminiscent of the KSR-1 [12], the clusters are interconnected using rings. However, unlike its predecessor, the SPP-1000 connects all 16 clusters using four parallel rings. All 16 clusters are connected to all four rings. This is achieved through the SCI interfaces to the Functional Units within each cluster. Every Functional Unit of a cluster is connected to one of the four global ring interconnects. The effect is that every memory bank is connected to the processors within its cluster by the 5 port cross-bar switch and to the other clusters by the specific ring to which the memory block's host Functional Unit is connected. Processor request to global memory goes first to the local cross-bar switch, the switch connects it to the Functional Unit in the same cluster that is in turn associated with the global ring interconnect to which the remote Functional Unit holding the target data is also connected. The request is then forwarded along the SCI ring interconnect to the remote cluster in which the destination memory resides. This structure is shown in Figure 3 which presents the global interconnect for the SPP-1000 system. The second level of cache coherence is provided by part of the memory in each Functional Unit reserved as interconnect cache and is managed by the CCMC. Data which is accessed from a remote cluster is copied into the interconnect cache and from there sent to the cache of the requesting processor.

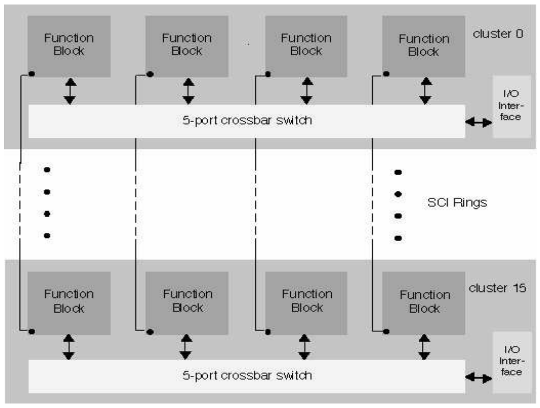

Figure 3: Convex SPP-1000 System Organization

3 Global Control Mechanisms

The scalabilty of parallel programs can be greatly affected by the cost of the overhead for managing concurrency. Programs requiring fine grain parallelism demand efficient implementations of parallel control mechanisms, while those with coarser requirements are not significantly impacted by more costly implementations. The scaling and temporal costs of basic control constructs on a single hypernode were presented in [15]. This section presents the results of some of the same kinds of measurements on a two hypernode system with global shared memory. The experiments expose the limits on the degree of granularity of parallelism achievable on the system, and provide some insight as to its scalability.

A set of synthetic codes were written to measure the temporal cost of specific parallel control constructs. These measurements were conducted across both hypernodes, first scaling with high locality (first 8 threads are on one hypernode) and next with uniform distribution (each hypernode has an equal number of threads running on it). The accuracy of the measurements were limited by the resolution of the timing mechanisms available and the intrusion resulting from their use. The multitasking nature of resource scheduling also proved to be a source of error, prompting the execution of many experimental runs to expose the true system behavior. Depending on the measurement of interest, either averages of the combined measurements or the minimum values observed were used.

3.1 Fork-Join Mechanism

For a program to achieve parallelism, it must be able to divide its work and distribute it among multiple processors. This is done by spawning execution threads that share the same virtual memory space. The fork-join mechanism performs all actions required to generate a set of parallel threads, synchronize their termination, and create the follow-on actions. The utility of parallelizing a portion of a program is diminished as the time required to spawn and join the threads approaches the same order as the execution time of the part of the program under consideration.

To evaluate the efficiency of the fork-join primitive on the SPP-1000, a set of experiments were conducted across a range of 1 to 16 threads being executed on 1 to 16 processors. This was done by distributing the threads with high locality and also by distributing them uniformly across the two hypernodes. The threads were spawned using the vendor's Compiler Parallel Support (CPS) library function `cps_ppcall()`. The time of the actual thread length and the intrusive cost of timing were subtracted from the total time to yield a good estimate of the overhead time to execute the fork-join primitive.

Figure 4 shows the fork-join time in microseconds as a function of the number of threads spawned. The graph shows two plots that highlight the increased cost of a fork-join across two hypernodes. The high locality plot demonstrates the cost of the fork-join where the first 8 threads are spawned on the same hypernode and subsequent threads are spawned on the remaining hypernode. The uniform distribution plot shows the cost of the fork-join where an equal number of threads are spawned on each hypernode (except in the 1 thread case).

The principal observations to be garnered from Figure 4 are:

- The fork-join time is proportional to the number of threads spawned with high locality across a single hypernode. Moving from 2 to 8 processors each additional pair of threads costs approximately 10 microseconds.
- The fork-join time is roughly proportional to the number of threads spawned with uniform distribution between hypernodes. Moving from 2 to 16 processors each additional pair of threads costs approximately 20 microseconds.
- A significant overhead, on the order of 50 microseconds, is incurred once threads start to be spawned on two hypernodes.

As will be seen in the next subsection, the cost of a simple barrier is significantly cheaper than the fork-join. This can be attributed to the additional functionality required for the fork-join to create and maintain intra-thread contexts while retaining the shared context of the parent thread.

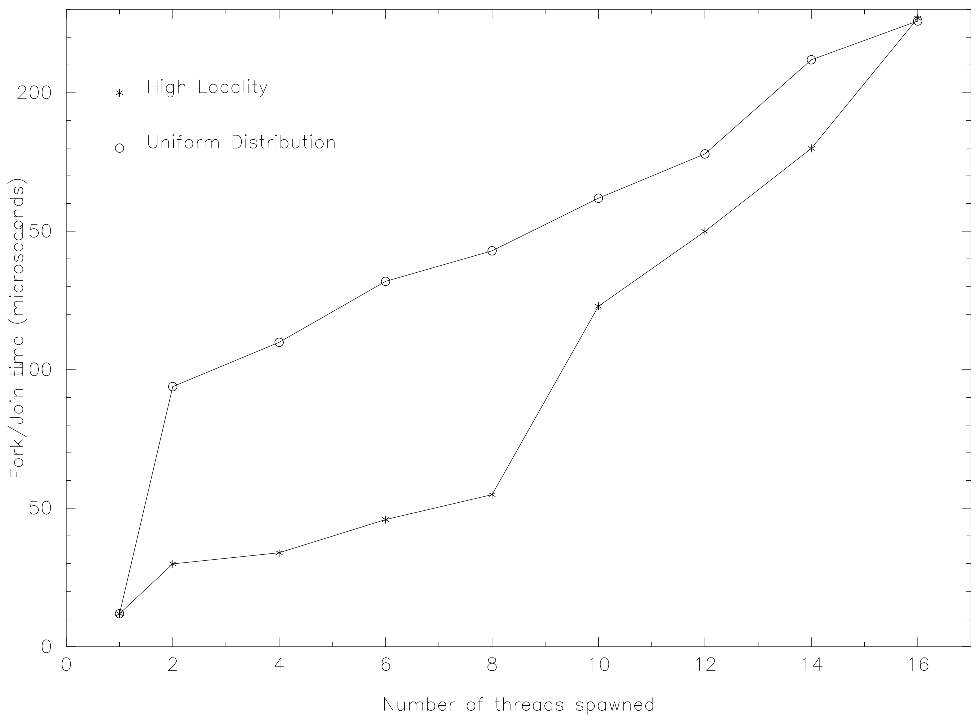

Figure 4: Cost of Fork-Join

3.2 Barrier Synchronization

Among the most basic of parallel control constructs is the barrier, a method for synchronizing the continuation of multiple activities. It is closely related to the semaphore, used to achieve exclusive access to shared mutable data objects and for realizing critical sections in data parallel code. The temporal cost of performing a barrier determines the minimum granularity of threads that can be effectively used to spread out the work among processors in order to achieve a performance gain through parallelism.

The temporal cost of the barrier synchronization primitive on the SPP-1000 was measured across a range of 1 to 16 threads using from 1 to 16 processors. As with the fork-join experiment, measurements were taken where the threads were distributed with high locality and also where they were distributed uniformly across the two hypernodes. The barrier synchronization was performed using the vendor's CPS library function `cps_barrier()`.

Figure 5 reports two metrics for both the high locality and uniform distribution cases:

Last In - First Out: the minimum time measured from when the last thread enters the barrier to when the first thread afterward continues.

Last In - Last Out: the minimum time measured from when the last thread enters the barrier to when the last thread continues.

The results from our earlier study of only one hypernode of the SPP-1000 [15] are also shown. Both the previous and current study used the same experimental method. A time-stamp was taken before each thread entered the barrier and after each thread exited the barrier. From this data an approximation of the barrier costs could be derived. All timing data have been corrected for the overhead involved in performing the measurements.

Figure 5 shows that the minimum time for a barrier (last in - first out) involving more than one thread is approximately 3.5 microseconds on a single hypernode, incurring an additional cost of 1 microsecond once threads on a second hypernode become involved. The release time of the barrier, the total time to continue all suspended threads, possesses a more complex behavior. In the high locality case on just one hypernode, the barrier appears to cost roughly 2 microseconds per thread beyond the second thread involved. Once threads on a second hypernode become involved, there is an additional penalty, as evidenced by both the high locality and uniform distribution cases.

This behavior may be caused by the implementation of the barrier primitive, which has each thread decrement an uncached counting semaphore [3] and then enter a while loop, waiting for a shared variable to be set to a particular value. The last thread to enter the barrier sets the shared variable to the expected value, thus releasing the other threads from their spin waiting. Because this shared variable is cached by all of the threads, coherency mechanisms are invoked when the final thread alters its value. This incurs a variable temporal cost depending on the status of the system reference tree. Figure 5 would seem to reflect the increased cost of maintaining coherency and updating the reference tree as a greater number of processors become

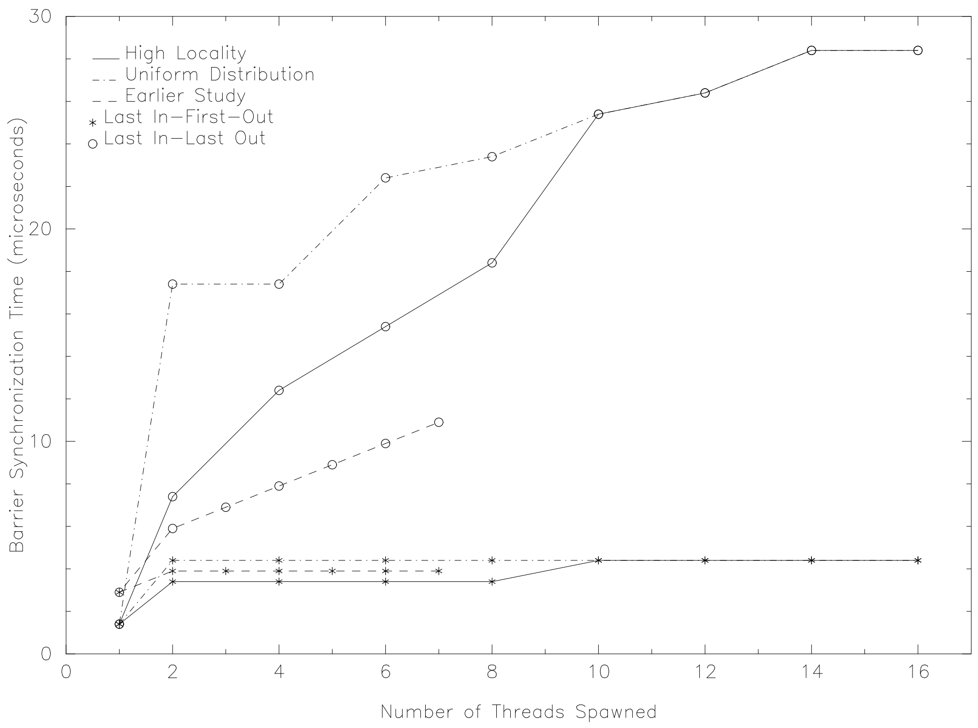

Figure 5: Cost of Barrier Synchronization

involved. The behavior of the uniformly distributed case is accounted for by the parallel updates of internal system data structures of the two hypernodes.

It is of particular note that the barrier synchronization time has increased under the multi-cluster global shared memory operating system. This is a reflection of the additional system management necessary to provide parallel support across multiple hypernodes. The difference between the high locality and uniform distribution graphs demonstrates another aspect of this overhead. The difference in the last in-last out costs of these cases from 2 to 8 threads ranges from 5 to 10 microseconds. Even though their behavior is not linear, if one were to approximate it as such, the average cost per additional thread for the high locality case starting with 2 threads is about 1.5 microseconds and 1.3 microseconds for the uniform case. The single hypernode study [15] showed an average additional cost of 1 microsecond per thread.

4 Application: The Gravitational N-body Problem

The solution of the gravitational N-body problem in Astrophysics is of general interest for a large number of problems ranging from the breakup of comet Shoemaker/Levy 9 to galaxy dynamics to the large scale structure of the universe. This problem is defined by the following relation where the gravitational force on particle i in a system of N gravitationally interacting particles is given by,

$$\vec{F_i} = \sum_{j=1}^{N} \frac{G m_i m_j \vec{r_{ij}}}{(r_{ij}^2 + \epsilon^2)^{3/2}} \tag{1}$$

where G is the universal gravitational constant, m_i and m_j are the masses the particles i and j, $\vec{r_{ij}}$ is the vector separating them, and ϵ is a smoothing length which can be nonzero and serves to eliminate diverging values in $\vec{F_i}$ when $\vec{r_{ij}}$ is small. This parameter also serves to define a resolution limit to the problem. This equation also shows that the problem scales as N^2 and modeling systems with particle numbers larger than several thousand is infeasible.

Tree codes are a collection of algorithms which approximate the solution to equation 1 [2, 8, 13]. In these algorithms the particles are sorted into a spatial hierarchy which forms a tree data structure. Each node in the tree then represents a grouping of particles and data which represents average quantities of these particles (e.g. total mass, center of mass, and high order moments of the mass distribution) are computed and stored at the nodes of the tree. The forces are then computed by having each particle search the tree and pruning subtrees from the search when the average data stored at that node can be used to compute a force on the searching particle below a user supplied accuracy limit. For a fixed level of accuracy this algorithm scales as $Nlog(N)$ although $O(N)$ algorithms are also possible. Since the tree search for any one particle is not known *a priori* and the tree is unstructured, frequent use is made of indirect addressing. Further, the tree data is updated during a simulation as the

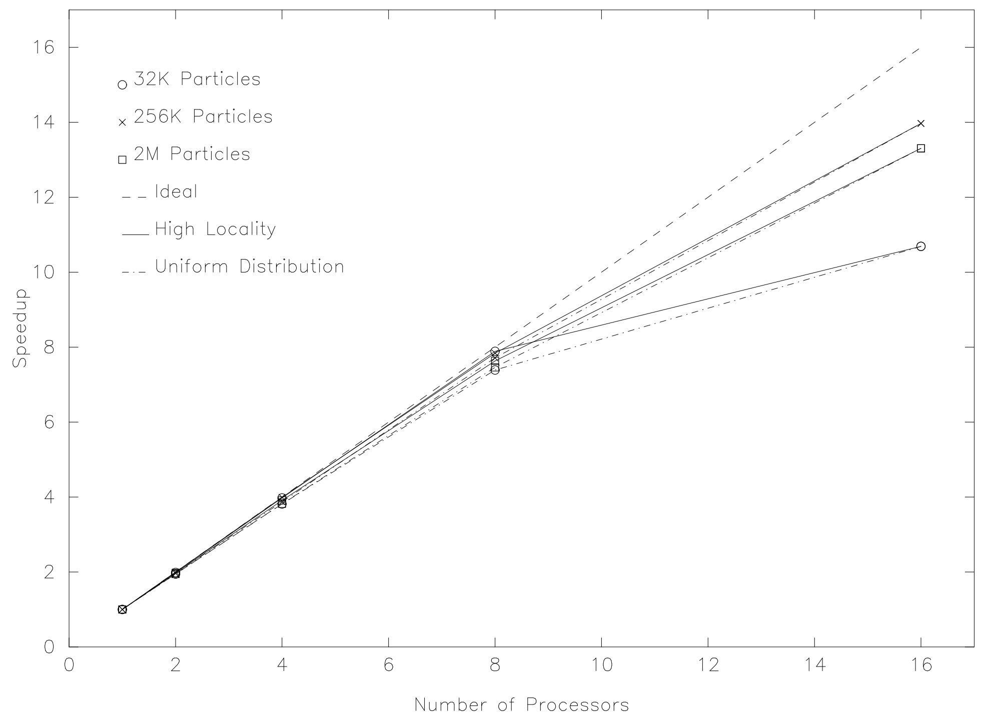

Figure 6: N-Body Performance Scaling

particles move through space. Therefore, this algorithm is not only of interest for its scientific application, but is also of computational interest due to its unstructured and dynamic nature.

4.1 Experimental Results

We have ported a FORTRAN 90 version of a tree code to the Convex SPP which was initially developed for the Maspar MP-2 and implemented using the algorithm described in Olson and Dorband [13]. The changes to the original code were straightforward and the compiler directives and the shared memory programming model facilitated a very simple minded approach to be taken. The main alterations to the MasPar code were to distribute all the particle calculations evenly among the processors and make all intermediate variables in the force calculation thread-private. Each processor then calculates the forces of its subset of particles in a serial manner. All indirect accesses are made by each thread of execution into the tree data stored in global shared memory. Further, these indirect addresses are made in the innermost loop of the tree search algorithm, thus relying on the ability to utilize rapid, fine grained memory accesses allowed by the shared memory programming model. This scheme also allows for more efficient use of the data cache on subsequent floating point operations.

The program was run on three problem sizes (32K, 256K and 2M particles), applying from 1 to 16 processors in two configurations of the processors. The first configuration ran 1,2,4 and 8 processors on a single hypernode and the second ran 2,4,8 and 16 processors across two hypernodes. Figure 6 shows the parallel speedup for each of the cases measured relative to the single processor rate of 27.5 Mflop/s. We see that the performance degradation incurred across multiple hypernodes is small; it is between 2 and 7 percent. It is also clear that the performance at 16 processors is affected by the problem size. The task granularity changes linearly with the problem size as do the overall memory requirements. However, the balance between local and global memory accesses varies non-linearly; it is determined by the proportion of information at each level of the tree and by the proportion of the depths searched by the algorithm. To determine the effect of multiple hypernodes on the scaling of this application, tests should be run on a system with more than two hypernodes. From this initial data it is not possible to predict how speedup will change as additional hypernodes are added. Finally, the 16 processor 384 Mflop/s result compares favorably to a highly vectorized, public domain tree code [8] which achieves 120 Mflop/s on one head of a C90.

5 Cache Consistency Behavior

The Convex SPP-1000 cache coherency mechanism operates at two levels: the directory based mechanism within a given hypernode and the SCI method that maintains consistency across hypernodes. The Tree code used in the scaling studies of the previous section is employed again to expose cache miss dependencies upon system and problem size scaling. Hardware instrumentation has been incorporated as part of the SPP-1000 design to measure the number of cache misses encountered and the duration of the misses. The allocation

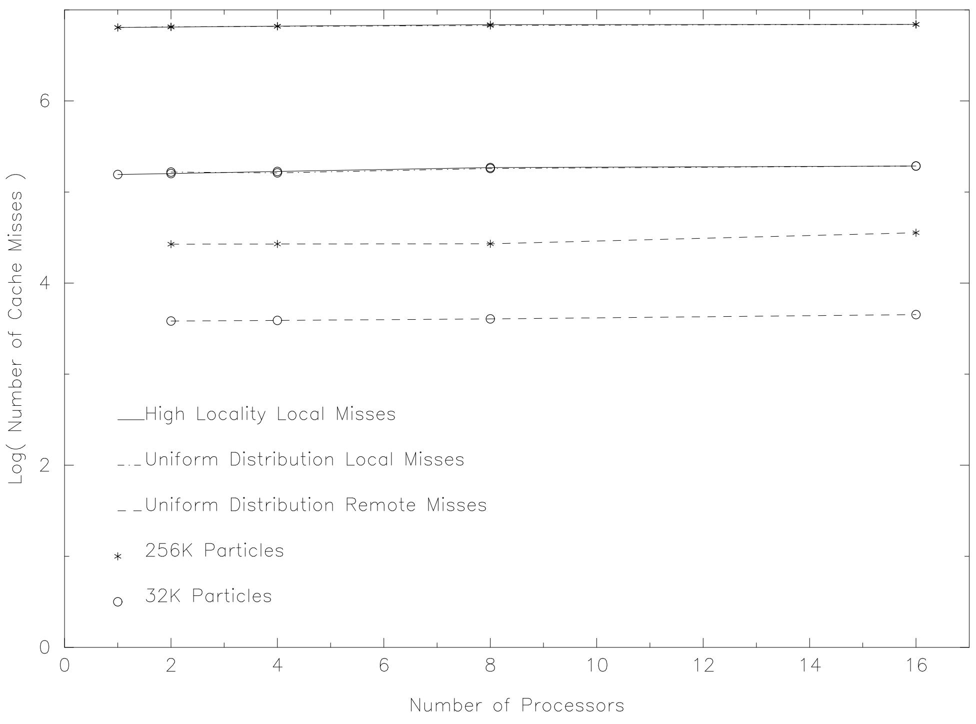

Figure 7: N-Body Cache Misses

of the tree structure for the application is distributed approximately equally between the two hypernodes. The force calculation for each simulated gravitational body on one hypernode requires mass data about the bodies on the other hypernode. But because this distribution also reflects simulated distance, not all the information on one node needs to be copied to the other hypernode. Rather, only higher levels of the tree are required.

The first set of experiments is performed to determine how the cache miss rate varies for a fixed size problem as the number of processors is increased. Measurements were taken of both misses local to the hypernodes and misses across the SCI based interconnect. The experiments were run in two ways as was the case with the scaling tests. In one set of tests, the first 1 to 8 processors run were all in a single hypernode. This is the high locality test. Only the 16 processor run included processors from both hypernodes. In another set of tests, processors are taken evenly from both hypernodes; when 6 processors were running, 3 were from one hypernode while 3 more were from the other hypernode. This series is the uniform tests. Finally, the tests were performed for two problem sizes: 32K particles and 256K particles.

The results of these experiments are shown in Figure 7. This semi-log plot shows the number of misses (log scale) with respect to the number of processors used to perform the computation. There are six curves there, all almost straight lines and two pairs almost completely overlapping. It is immediately clear that the miss rate is highly insensitive to the number of processors employed for the computation. All curves are essentially straight and horizontal. There is a gradual rise of within a few percent, but this is well within the range of measurement error.

A more detailed examination reveals additional behavior. Not only are the local miss rates insensitive to the number of processors used, but they are also insensitive to the locality of the processors. The number of local misses, those occurring only within the hypernodes, is the same whether all the processors (for 8 or less) are in a single cluster or shared between the two hypernodes used. This is poorly demonstrated by the overlap of the solid and double-dashed lines near the top of the figure. Where the uniform tests are performed employing processors from both hypernodes, we find that the number of misses is there too insensitive to the number of processors. As might be expected, the number of cache misses is sensitive to the problem size, in this case represented by the number of particles simulated. Local misses are almost two orders of magnitude more for the 256K particle simulation than the 32K particle simulation while that difference for remote misses is less than one order-of-magnitude. It is also seen that the local misses are almost a thousand times more frequent than the remote misses for the 256K particle case and more than a factor of 10 for the 32K particle problem.

The findings show that for a given problem size, the scaling characteristics of the cache misses with respect to system size are very favorable. For this application, cache misses do not increase as the system grows, at least within the experimental context and constraints. Also, most by far of the cache misses are local with only a small fraction of the total misses being the more costly remote inter-hypernode accesses. This too is a favorable outcome with the system exhibiting good scaling properties. The cache miss scaling properties with respect to problem size is less favorable. The remote misses grow roughly proportionally with problem size while local misses may scale $O(n^2)$ with problem size n. This is explainable by the nature of the application where nearby bodies tend to require $O(n^2)$ computations while distant force calculations converge on $O(n)$ in the limit or $O(n \log(n))$ in general. For the range of values considered, these are approximately the same. Using available measurement resources, some estimates of the local and remote cache miss latencies are possible. These differed somewhat across the range of processors and to a great degree between local and global misses. The average local cache miss latency is 0.54 microseconds and the average remote cache miss latency is 3.7 microseconds. For only a single processor, the cache miss penalty is about 20% less at 0.46 microseconds. While small changes in remote latency times were observed, no discernible correlation between number of processors and average remote latency time was identified. However, the experimental system employed only two hypernodes. It is expected that the nature of the SCI implementation may cause a sensitivity of remote access latency time to number of hypernodes used. Further study is required here.

6 Discussion and Conclusions

This paper has presented the first published results of an evaluation of the Convex SPP-1000 global communication and cache consistency mechanisms. The Convex SPP-1000 multiprocessor incorporates hardware supported runtime mechanisms to achieve full cache coherency across 128 processors organized in a two level hierarchy of 8 processor hypernodes. Directory based cache consistency mechanisms are employed within the cluster while consistency across hypernodes is managed by an implementation of the SCI protocol. A two hypernode system was employed in a series of experiments to evaluate the performance properties of the global shared memory management system. These experiments examined the operating characteristics of the system performing basic global control operations including barrier synchronization and thread scheduling. System scalability properties were evaluated using a full Earth and space science application program, the N-body gravitation simulation with a tree code algorithm. A detailed study of the scaling of cache misses was performed for both levels of the cache consistency structure and measurements of cache miss penalty were conducted for both local and global cache misses.

It was found that the system architecture supports global mechanisms at low enough performance penalty that medium grained parallelism can be exploited efficiently. The global cache miss penalty was determined to be only a few times that of a local cache miss while executing a real-world application code. Scaling of this code was seen to be good for a reasonable problem size. In particular, the remote access communication and SCI mechanisms were observed to cause only minor performance degradation over purely local hypernode execution. Cache miss rate did not vary significantly with the number of processors and for this problem the global cache misses were greatly surpassed by the local misses.

The system tested is still in beta test and the software system provided is in a state of flux. A number of performance characteristics have been altered from the original study carried out on a single hypernode due to changes both in hardware and software. For example, the performance of the barrier synchronization mechanism has been degraded while that of the fork-join thread scheduling was seen to improve. The overall performance of this application improved for a number of reasons including better compiler technology, enhanced operating system functionality, and modifications to the algorithm used. Additional changes are anticipated in the near future, affecting the degree to which these findings accurately reflect the properties of this system in the future.

While other applications have been ported to the Convex SPP-1000, their limited maturity precluded reporting of their performance behavior at this time. Therefore, generalizations about the system operation as a whole must wait confirmation from a larger test set. This will be done shortly. Scaling properties at the global level require access to systems comprising four or more hypernodes. Such systems are now just being assembled at customer sites and a new set of experiments will be conducted. Cache misses were discussed without relating them to total number of access requests providing cache miss ratios. This proves to be difficult as there is no hardware support for enumerating total memory accesses. More detailed study of memory reads and writes will have to be made of the application code using estimates of total memory access requests from analysis and simulation trace studies of the application. Only then will the true effectiveness of caches as a latency avoidance mechanism be determined.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, et al. "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," *M. Dubois and S.S. Thakkar, editors, Scalable Shared Memory Multiprocessors,* Kluwer Academic Publishers, 1992, pp. 239-261.

[2] J.E. Barnes and P. Hut, "A Hierarchical O(n log n) Force Calculation Algorithm," *Nature,* vol. 342, 1986.

[3] CONVEX Computer Corporation, "Camelot MACH Microkernel Interface Specification: Architecture Interface Library," Richardson, TX, May 1993.

[4] D. Chaiken, J. Kubiatowitz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV),* 1991, pp. 224-234.

[5] CONVEX Computer Corporation, "Exemplar Architecture Manual," Richardson, TX, 1993.

[6] CONVEX Computer Corporation, "Exemplar Programming Guide," Richardson, TX, 1993.

[7] Cray Research, Inc., "CRAY T3D System Architecture Overview," Eagan, Minnesota.

[8] L. Hernquist, "Vectorization of Tree Traversals," *Journal of Computational Physics,* vol. 87, 1990.

[9] Hewlett Packard Company, "PA-RISC 1.1 Architecture and Instruction Set Reference Manual," Hewlett Packard Company, 1992.

[10] Intel Corporation, "Paragon User's Guide," Beaverton, Oregon 1993.

[11] IEEE Standard for Scalable Coherent Interface, IEEE, 1993.

[12] Kendall Square Research Corporation, "KSR Technical Summary," Waltham, MA, 1992.

[13] K. Olson and J. Dorband, "An Implementation of a Tree Code on a SIMD Parallel Computer," *Astrophysical Journal Supplement Series,* September 1994.

[14] Thinking Machines Corporation, "Connection Machine CM-5 Technical Summary," Cambridge, MA, 1992.

[15] T. Sterling, D. Savarese, P. Merkey, J. Gardner, "An Initial Evaluation of the Convex SPP-1000 for Earth and Space Science Applications," *Proceedings of the First International Symposium on High Performance Computing Architecture,* January 1995.

[16] M.S. Warren and J.K. Salmon, "A Parallel Hashed Oct-tree N-Body Algorithm," *Proceedings of Supercomputing '93,* Washington: IEEE Computer Society Press, 1993.