Scalability Study of the KSR-1 *

Appeared in Parallel Computing, Vol 22, 1996, 739-759

Umakishore Ramachandran *Gautam Shah* *S. Ravikumar*
Jeyakumar Muthukumarasamy

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Phone: (404) 894-5136
e-mail: rama@cc.gatech.edu

Abstract

Scalability of parallel architectures is an interesting area of current research. Shared memory parallel programming is attractive stemming from its relative ease in transitioning from sequential programming. However, there has been concern in the architectural community regarding the scalability of shared memory parallel architectures owing to the potential for large latencies for remote memory accesses. KSR-1 is a commercial shared memory parallel architecture, and the scalability of KSR-1 is the focus of this research. The study is conducted using a range of experiments spanning latency measurements, synchronization, and analysis of parallel algorithms for three computational kernels and an application. The key conclusions from this study are as follows: The communication network of KSR-1, a pipelined unidirectional ring, is fairly resilient in supporting simultaneous remote memory accesses from several processors. The multiple communication paths realized through this pipelining help in the efficient implementation of tournament-style barrier synchronization algorithms. Parallel algorithms that have fairly regular and contiguous data access patterns scale well on this architecture. The architectural features of KSR-1 such as the poststore and prefetch are useful for boosting the performance of parallel applications. The sizes of the caches available at each node may be too small for efficiently implementing large data structures. The network does saturate when there are simultaneous remote memory accesses from a fully populated (32 node) ring.

Key Words:
Shared memory multiprocessors, Scalability, Synchronization, Ring interconnect, Latency, NAS benchmarks, Performance

*This work is supported in part by an NSF PYI Award MIP-9058430 and an NSF Grant MIP-9200005. A preliminary version of this paper appeared in the International Conference on Parallel Processing, 1993, Vol I:237-240

1 Introduction

Understanding the performance potential of parallel systems is an important research endeavor. From an application's perspective, a reasonable expectation would be that the efficiency of the parallel system at least remain the same (if not improve) as the problem size is increased. Similarly, from an architecture's perspective, the system should deliver increased performance with increased computational resources. The term *scalability* is used to make the connection between these two views of system performance. In our work we use this term to signify whether a given parallel architecture shows improved performance for a parallel algorithm with increased number of processors. System architects are primarily concerned with the performance of parallel machines. Performance improvement with added processors is thus used as a measure of scalability. The notion of *speedup*, defined as the ratio of execution time of a given program on a single processor to that on a parallel processor, is perhaps the earliest metric that has been widely used to quantify scalability. A drawback with this definition is that an architecture would be considered non-scalable if an algorithm running on it has a large sequential part. There have been several recent attempts at refining this notion and define new scalability metrics (see for instance [12, 8, 14]).

With the evolution of parallel architectures along two dimensions, namely shared memory and message passing, there has been a considerable debate regarding their scalability given that there is a potential for large latencies for remote operations (message communication and remote memory accesses respectively in the two cases) in both cases as the size of the machine grows. The communication latency in message passing machines is explicit to the programmer and not implicit depending on the memory access pattern of an algorithm as in the case of shared memory machines. Thus there is a tendency to believe that message passing architectures may be more scalable than its architectural rival as is evident from the number of commercially available message passing machines [5, 11, 15]. Yet, there is considerable interest in the architectural community toward realizing scalable shared memory multiprocessors. Indeed the natural progression from sequential programming to shared memory style parallel programming is one of the main reasons for such a trend.

KSR-1 is a shared memory multiprocessor introduced by Kendall Square Research (Section 2). The focus of this paper is to study the scalability of KSR-1. We first obtain low level latency measurements of read/write latencies for the different levels of the memory hierarchy (Section 3.1); we then implement and measure (Section 3.2) the costs of shared memory style synchronization primitives (locks and barriers); and finally we implement three parallel kernels and an application from the NAS benchmark suite and report on their observed performance (Section 3.3).

We found from our experiments on a 32-node KSR-1 that the network tends to saturate (as determined from observed network latencies) when all processors try to access data remotely at the same time. The omission of read shared locks in the hardware architecture may be a performance bottleneck for applications that exhibit considerable read-sharing. From our barrier experiments on a 64-node KSR-2, we show

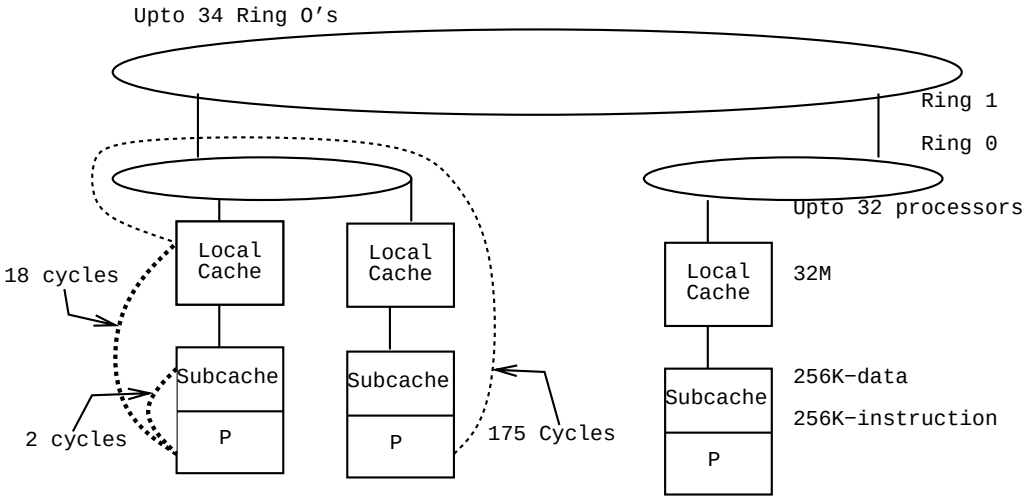


Figure 1: Architecture of the KSR

that a tournament style barrier synchronization with a global wake up flag performs the best for barrier synchronization. Of the kernels we implemented – Embarrassingly Parallel (EP), Conjugate Gradient (CG), and Integer Sort (IS) – EP and CG scale very well on the machine, while IS does not and we do hit the saturation point of the ring for this algorithm on a 32-node KSR-1. The Scalar Pentadiagonal (SP) application scales well and this application provides us an insight on techniques for shared memory application structuring that takes into account the multi-level memory hierarchy of the KSR architecture. Concluding remarks are presented in Section 4.

2 Architecture of the KSR-1

The KSR-1 is a 64-bit cache-only memory architecture (COMA) based on an interconnection of a hierarchy of rings [16, 17]. The ring (see Figure 1) at the lowest level in the hierarchy can contain up to 32 processors. These "leaf" rings connect to rings of higher bandwidth through a routing unit (ARD). Current implementations of the architecture support two levels of the rings and hence up to 1088 processors (34 leaf rings can connect to the level 1 ring).

Each processing node (called a *cell* in KSR-1 and used interchangeably for a node in the rest of this paper) consists of four functional units namely the cell execution unit (CEU), the I/O processor (XIU), the integer processing unit (IPU), and the floating point unit (FPU). There are 32 MBytes of second-level cache (called *local-cache* in KSR-1 and in the rest of this paper) and 0.5 MBytes (0.25 MBytes for data and 0.25M Bytes for instructions) of first-level cache (called *sub-cache* in KSR-1 and in the rest of this paper) present at each node. Additionally each node contains cell interconnect units which interface the node to the ring and cache control units which help maintain cache coherence. Each of the functional units is pipelined.

The processing node issues two instructions per cycle (one for the CEU or the XIU and one for the FPU or the IPU). The CPU clock speed is 20MHz and the machine has a peak performance of 40 MFLOPS per processing node.

The architecture provides a sequentially consistent shared memory model. This shared memory constitutes the System Virtual Address (SVA) space that is global to the entire multiprocessor system. The programmer however sees the shared memory in terms of a Context Address (CA) space that is unique for each process. The mapping between a CA space and the SVA space is done through Segment Translation Tables (STT). The COMA model is implemented using the ALLCACHE^{TM} memory in the KSR-1. A distinguishing characteristic of the ALLCACHE^{TM} memory is that there is no fixed location for any System Virtual Address. An invalidation-based cache coherence protocol is used to maintain sequential consistency. The unit of consistency maintenance and transfer on the ring is a *sub-page* of 128 bytes (the cache line size in the local-caches). Each sub-page can be in one of *shared*, *exclusive*, *invalid*, or *atomic* state. The atomic state is similar to the exclusive state except that a node succeeds in getting atomic access to a sub-page only if that sub-page is not already in an atomic state. An explicit release operation on a sub-page removes the sub-page from the atomic state. Even though the transfer size is 128 bytes, allocation is done on a 16KByte page basis in the local-cache. Upon allocation only the accessed sub-page is brought into the local-cache. All other sub-pages of a page are brought into the local-cache on demand. The unit of transfer between the local-cache and the sub-cache is in terms of sub-blocks of 64 bytes. The allocation in the sub-cache is in terms of blocks of 2 KBytes, and once the allocation is done the sub-blocks are brought into the sub-cache from the local-cache on demand. The local-cache is 16-way set-associative and the sub-cache is 2-way set associative and both use a random replacement policy. The *get_sub_page* instruction allows synchronized exclusive access to a sub-page, and the *release_sub_page* instruction releases the exclusive lock on the sub-page. The KSR-1 also provides *prefetch* (into the local-cache) and *poststore* instructions. The poststore instruction sends the updated value on the ring and all place holders for this sub-page in the local-caches receive the new value. The processor that issues the poststore can however continue with its computation. The architecture also supports *read-snarfing* which allows all invalid copies in the local-caches to become valid on a re-read for that location by any one node.

The interconnection network uses a unidirectional slotted pipelined ring with 24 slots in the lowest level ring (two address interleaved sub-rings of 12 slots each), and hence there could be multiple packets on the ring at any given instance. The ring protocol ensures round-robin fairness and forward progress. The lowest level ring has a capacity of 1 GBytes/sec. The packets are used to transfer data and maintain cache coherence. Any address generated by the CEU is first checked in the sub-cache. If it is not present in the sub-cache, the local-cache is checked. On a local-cache miss, a request is sent out on the ring. If any of the nodes at that level contain the sub-page in one of the three valid states, it responds back appropriately. Otherwise the ARD determines that the sub-page is not present on the current level and propagates the request to the next level in the ring hierarchy. Each node in the KSR-1 has a hardware *performance monitor*

that gives useful information such as the number of sub-cache and local-cache misses and the time spent in ring accesses. We used this piece of hardware quite extensively in our measurements. Most of our experiments are run on a one level 32-node KSR-1. We also performed synchronization experiments on a two-level 64-node KSR-2. The only architectural difference between KSR-1 and KSR-2 is that the CPU in KSR-2 is clocked at twice the speed compared to KSR-1. The interconnection network and the memory hierarchies are identical for the two.

3 Experimental Study

We have designed experiments aimed at evaluating the low-level architectural features of KSR-1. These experiments include measuring the effectiveness of the hardware cache coherence scheme in servicing read write misses at different levels of the cache hierarchy; the utility of get_sub_page instruction for implementing read (shared) and write (exclusive) locks; and the usefulness of poststore instruction for implementing barrier synchronization algorithms. These experiments and the observed results are presented in the following subsections.

The notion of scalability is meaningful only in the context of an algorithm-architecture pair. An approach to determining the scalability of any given architecture would be to implement algorithms with known memory reference and synchronization patterns and measure their performance. The measurements would allow us to determine how the features of the underlying architecture interacts with the access patterns of the algorithms, and thus give us a handle on the scalability of the architecture with respect to these algorithms. For this purpose we have implemented three computational kernels and an application that are important in several numerical and scientific computations. We analyze their performance and explain the observed results with respect to the architectural features of KSR-1 in Section 3.3.

3.1 Latency Measurements

The published latencies for accesses from the three levels of the memory hierarchy (within a single ring) are shown in Figure 1. We have conducted simple experiments to verify them as detailed below. For the sub-cache the measured latency agrees with the published one, namely 2 cycles[1]. To measure the local-cache (which is the second level in the hierarchy) latency we do the following: we maintain two data structures A and B, both private to each processor, and each 1 MByte in size (i.e. both are guaranteed to be in the local-cache but are too large to fit in the sub-cache). We first fill the sub-cache with only B by repeatedly reading it[2]. Accessing A now guarantees that the measured access times will be the latencies for the local-cache. Figure 2 shows our measured latencies. The measured values are close to the published

[1]Cycle time is 50ns.

[2]Since a random replacement policy is used in the 2-way set associative sub-cache, we cannot be sure that B ever fills the sub-cache. So we read B repeatedly to improve the chance of the sub-cache being filled with B.

values, but the writes are slightly more expensive than reads. The plausible explanation for this result is that writes incur replacement cost in the sub-cache. As expected, varying the number of processors does not increase the latency.

To measure remote cache access times (latency of an access going out on the ring), we allocate a private data structure on each processor (since there is no concept of ownership we achieve this by making the processor access the data structure and thus making it locally cached). Next, each processor accesses the data cached in its neighboring processor thus ensuring network communication[3]. The resulting remote access latency measurements are shown in Figure 2. Again our measurements are mostly consistent with the published numbers, but the writes are slightly more expensive than the reads as observed in the local-cache access measurements. One may expect that for higher number of processors, the contention to use the ring might cause higher latencies. One of the outcomes of this experiment in terms of scalability is that the ring latency increases (about 8% for 32 processors) when multiple processors simultaneously access remote data. We have ensured in this experiment that each processor accesses distinct remote data to avoid any effects of false sharing, since the purpose of this experiment is simply to determine remote access times. We have also measured the overhead of block (2 KBytes) allocation in the sub-cache, by changing the stride of access such that each access is to a new 2K block, instead of sub-blocks within the same block. This access pattern causes a 50% increase in access times for the local-cache. When the stride of access exhibits more spatial locality the allocation cost gets amortized over the multiple sub-blocks that are accessed within the same block. Similar increase (60%) in remote access times are observed when these accesses cause page (16 KBytes) allocation in the local-cache.

3.2 Synchronization Measurements

The second set of measurements on the KSR-1 involve implementing synchronization mechanisms. Typical synchronization mechanisms used in multiprocessors shared memory style parallel programming are locks and barriers. We discuss the performance of these two mechanisms on the KSR in the next two subsections.

3.2.1 Locks

A lock ensures read or write atomicity to a shared data structure for concurrent processes; and a barrier guarantees that a set of processes comprising a parallel computation have all arrived at a well-defined point in their respective execution. As we mentioned in Section 2, the KSR-1 hardware primitive *get_sub_page*, provides an exclusive lock on a sub-page for the requesting processor. This exclusive lock is relinquished using the *release_sub_page* instruction. The hardware does not guarantee FCFS to resolve lock contention but does guarantee forward progress due to the unidirectionality of the ring.

[3]Note that accessing *any* remote processor would be equivalent to accessing the neighboring processor in terms of latency owing to the unidirectional ring topology.

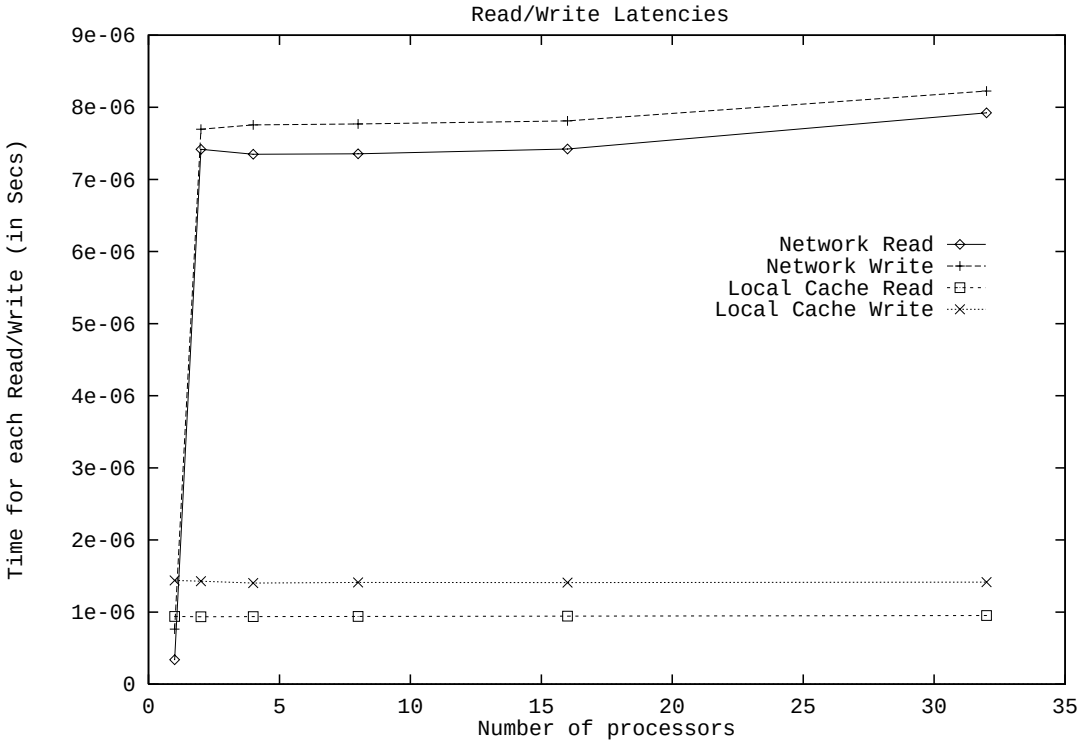


Figure 2: Read/Write Latencies on the KSR

The exclusive lock is not efficient in applications that exhibit read sharing, i.e., when a sub-page may be accessed in both exclusive and shared mode. The hardware primitive essentially serializes all lock requests regardless of whether they are shared or exclusive. We have implemented a simple read-write lock using the KSR-1 exclusive lock primitive. Our algorithm is a modified version of Anderson's ticket lock [1]. A shared data structure can be acquired in read-shared mode or in a write-exclusive mode. Lock requests are granted tickets atomically using the *get_sub_page* primitive. Consecutive read lock requests are combined by allowing them to get the same ticket. Concurrent readers can thus share the lock and writers are stalled until all readers (concurrently holding a read lock) have released the lock. Fairness is assured among readers and writers by maintaining a strict FCFS queue. We have experimented with a synthetic workload[4] of read and write lock requests and the result is as shown in Figure 3. The time for lock acquisition increases linearly with the number of processors requesting exclusive locks. However, when the percentage of read-shared lock request increases, our algorithm performs much better than the hardware exclusive lock for such synthetic work-loads. This experiment shows that in applications that have quite a bit of read-sharing it may be better not to use the hardware exclusive lock of KSR-1 but instead use a software algorithm[5] for

[4]Each processor repeatedly accesses data in read or write mode, with a delay of 10000 local operations between successive lock requests. The lock is held for 3000 local operations.

[5]The software algorithm may itself be implemented using any hardware primitive that the architecture provides for mutual

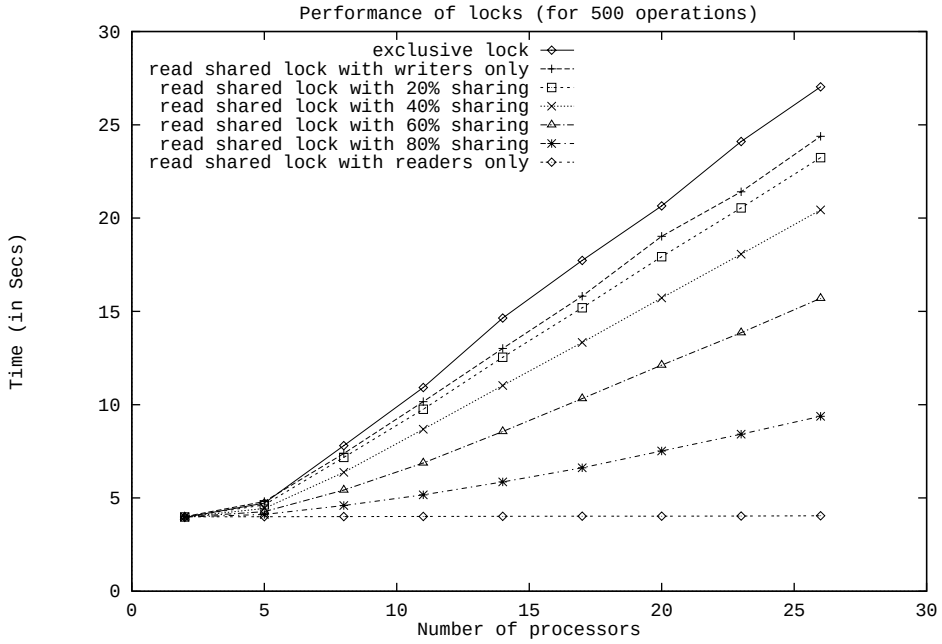


Figure 3: Read/Write and Exclusive locks on the KSR

lock acquisition. Besides, the software algorithm guarantees FCFS while the hardware primitive does not. Also, we observe that even when the workload has writers only, our implementation performs better than the naive hardware exclusive lock. This result is surprising considering the software overhead (maintaining the queue of requesters) in our queue-based lock algorithm. The result can be partly explained due to the interaction of the operating system with the application as follows [7]. While threads of a parallel program can be bound to distinct processors, there is no guarantee that they will be co-scheduled. More importantly, from the point of view of our results, the timer interrupts on the different processors are not synchronized. For our queue-based lock, threads can continue to join the queue of requesters even when the processor that currently owns the lock is servicing a timer interrupt. While this reason may explain why the software algorithm may do as well as the system lock, it is still unclear why it does better as can be seen in the results.

3.2.2 Barriers

We have implemented five barrier synchronization algorithms [13] on the KSR-1. In our implementation, we have aligned (whenever possible) mutually exclusive parts of shared data structures on separate cache lines so that there is no false sharing. The results are shown in Figure 4. There are two steps to barrier synchronization: arrival and completion. Algorithm 1 (labeled *counter* in Figure 4) is a naive counter method where a global counter is decremented by each processor upon arrival. The counter becoming zero

exclusion.

is the indication of barrier completion, and this is observed independently by each processor by testing the counter. Algorithm 2 (labeled *tree* in Figure 4) is a tree combining barrier that reduces the hot spot contention in the previous algorithm by allocating a barrier variable (a counter) for every pair of processors participating in the barrier. The processors are the leaves of the binary tree, and the higher levels of the tree get constructed dynamically as the processors reach the barrier thus propagating the arrival information. The last processor to arrive at the barrier will reach the root of the arrival tree and becomes responsible for starting the notification of barrier completion down this same binary tree. Both these algorithms assume an atomic fetch_and_ϕ instruction, which is implemented using the *get_sub_page* primitive. The next algorithm (labeled *dissemination* in Figure 4) is a dissemination barrier, which involves exchanging messages for $log_2 P$ rounds as processors arrive at the barrier (P is the total number of processors). In each round a total of P messages are exchanged. The communication pattern for these message exchanges is such that (see Reference [10] for details) after the $log_2 P$ rounds are over all the processors are aware of barrier completion. Algorithm 4 (labeled *tournament* in Figure 4) is a tournament barrier (another tree-style algorithm similar to Algorithm 2) in which the winner in each round is determined statically. The final algorithm (labeled *MCS* in Figure 4) is another (statically determined) tree-style algorithm proposed by Mellor-Crummey and Scott. The two main differences between MCS and tournament algorithms are: a 4-ary tree is used in the former for arrival; and "parent" processors arrive at intermediate nodes of the arrival tree in the former while processors are only at the leaf of the arrival tree in the latter. Notification of barrier completion is done very similar to Algorithm 2 in both of these algorithms.

KSR-1 owing to its pipelined ring interconnect has multiple communication paths. The latency experiments clearly show that simultaneous distinct accesses on the ring do not increase the latency for each access. However, the counter algorithm performs poorly (see Figure 4) since every arrival results in at least 2 ring accesses (one for fetching the current counter value, and second for sending it to the current set of processors spinning on the counter). Since these accesses are for the same location they get serialized on the ring, and the pipelining is of no help for this algorithm. Note that this poor performance is despite the fact that KSR-1 supports *read-snarfing* (i.e. one read-request on the ring results in all invalidated place holders for that location getting updated). The dynamic-tree algorithm overcomes the hot-spot problem and thus there is a potential for communication overlap exploiting the pipelined ring. We do observe it to perform better than the counter algorithm. But since this algorithm requires mutual exclusion lock for the fetch_and_ϕ operation, the performance degrades quickly as the number of processors is increased. However, modifying the completion notification (as suggested in [13]) by spinning on a global wakeup flag (instead of tree-based notification) set by the last arriving processor produces remarkable performance enhancement (see line labeled *tree(M)* in Figure 4). This is due to two reasons: one the wakeup tree is collapsed thus reducing the number of distinct rounds of communication, and two read-snarfing helps this global wakeup flag notification method tremendously. Read-snarfing is further aided by the use of poststore in our implementation of these algorithms.

The dissemination barrier does not perform as well on the KSR because it involves $O(PlogP)$ distinct communication steps. Yet, owing to the pipelined ring this algorithm does better than the counter algorithm. The MCS and tournament algorithms have almost identical performance. This result was counter-intuitive at first. But a careful analysis reveals the veracity of this result. Both algorithms determine the arrival tree statically. By using a 4-ary arrival tree, MCS cuts the height of the tree by 2 compared to tournament, thus effectively reducing the length of the critical path in half. Both algorithms require $O(P)$ communication steps. The parents at each level wait for their respective 4 children to arrive at the barrier by spinning on a 32-bit word, while each of the children indicate arrival by setting a designated byte of that word. Thus each node in the MCS tree incurs 4 sequential communication steps in the best case, and 8 sequential communication steps in the worst case (owing to false sharing). The tournament algorithm on the other hand incurs only 1 communication step for a pair of nodes in the binary tree in the best case, and 2 in the worst case.

In a machine such as the KSR-1 which has multiple communication paths all the communication at each level of the binary tree can proceed in parallel. However, for the MCS tree the communication steps from the 4 children to the parent is necessarily sequential and thus cannot exploit the potential parallelism that is available in the communication network. This limitation coupled with the potential false sharing that ensues as a result of the node structure is a detrimental effect in the MCS algorithm which is not present in the tournament algorithm. On the KSR-1, every such false sharing access results in one ring latency. Thus while the height of the MCS tree is reduced by 2 for any given P compared to the binary tree, the cost of the communication is at least quadrupled for each level of the tree compared to the binary tree. Note that this effect is necessarily true so long as the cache coherence protocol is invalidation-based. The notification of completion in the MCS algorithm uses a binary tree but since each node wakes up two children this is faster than the corresponding wake up tree used in tournament. This is the reason why the two perform almost similarly for the number of processors we tested. When a global wakeup flag is used instead of the wakeup tree (labeled *tournament(M)* and *MCS(M)* in Figure 4), it helps the tournament algorithm more than MCS (since the MCS wakeup tree is slightly faster as observed earlier). Thus the tournament barrier with global wakeup flag method produces the best performance among all the barrier implementations. We observe a fairly flat curve, as the number of processors participating in the barrier is increased, indicating scalability of this barrier algorithm on the KSR-1 architecture. The performance of the system library provided pthread barriers (labeled *System* in Figure 4) is also shown for comparison with other barrier algorithms. Its performance is almost similar to that of the dynamic-tree barrier with global wakeup flag.

Although it is stated that the MCS algorithm may be the best for large-scale cache coherent multiprocessors in Reference [13], the detrimental effects discussed above and our results on the KSR-1 indicate that the tournament(M) algorithm is better than the MCS(M) algorithm. This is due to the fact that there are multiple communication paths available in the architecture and due to the false sharing inherent in the MCS algorithm.

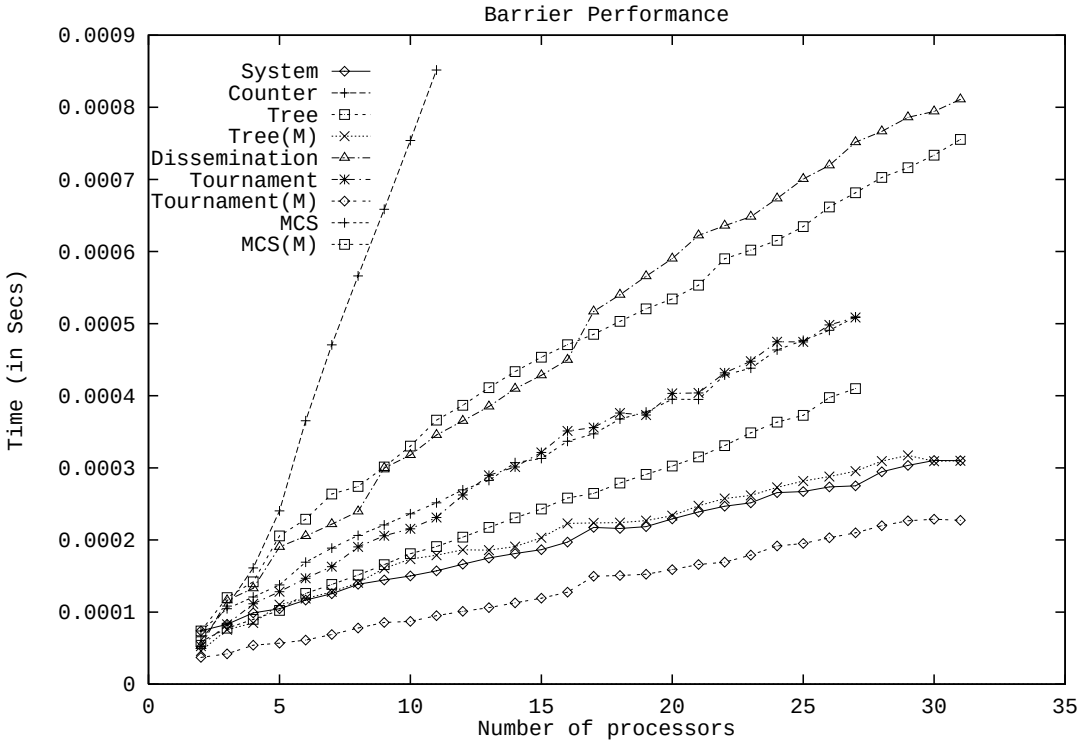


Figure 4: Performance of the barriers on 32-node KSR-1

3.2.3 Comparison with Other Architectures

It is illustrative to compare the performance of the barriers on the KSR-1 with their performance on the Sequent Symmetry and the BBN Butterfly reported in Reference [13]. The Symmetry is a bus-based shared memory multiprocessor with invalidation-based coherent caches. The Butterfly is a distributed shared memory multiprocessor with a butterfly multistage interconnection network, and no caches. The counter algorithm performs the best on the Symmetry. This result is to be expected as the bus serializes all the communication and hence algorithms which can benefit in the presence of parallel communication paths (such as dissemination, tournament, and MCS) do not perform well. MCS with global wakeup flag method does better than the other tree-style algorithms. The false sharing effect that affects the MCS performance on the KSR-1 is present on the Symmetry as well. However, since there are no parallel communication paths this effect is not an issue on the Symmetry. Both tournament and MCS have the same $O(P)$ communication which are all serialized on the Symmetry. Thus the fact that MCS reduces the critical path in half by using the 4-ary arrival tree is the reason why it does better than tournament on the Symmetry. On the BBN Butterfly, we do have parallel communication paths. However, since there are no (hardware) coherent caches the global wakeup flag method cannot be used on this machine. So the determinant here in deciding the winner is the number of rounds of communication (or the critical path for the arrival + wakeup tree) as pointed out

by Mellor-Crummey and Scott [13]. Thus the dissemination algorithm does the best ($(log_2 P)$ rounds of communication), followed by the tournament algorithm ($2log_2 P$ rounds of communication), and then the MCS algorithm ($(log_4 P + log_2 P)$ rounds of communication). This is in spite of the fact that dissemination incurs the most number of messages ($O(P log_2 P)$ compared to $O(P)$ for the other two). The importance of the parallel communication paths is evident from these results. Comparing our KSR-1 results, we can see that even though the Butterfly network can handle multiple simultaneous requests, the non-availability of coherent caches hurts the performance of barrier algorithms that are best suited for shared memory multiprocessors.

Thus the synchronization experiments on the KSR-1 demonstrate that the slotted pipelined ring overcomes the communication disadvantage observed on the Symmetry and the hardware cache coherence overcomes the inability to perform global broadcast using shared variables on the BBN Butterfly.

3.2.4 Results on 64-node KSR-2

As we mentioned earlier (see Section 2), the only architectural difference between KSR-1 and KSR-2 is that the processor in KSR-2 is clocked at twice the speed compared to KSR-1. The interconnection network and the memory hierarchies are identical for the two. Qualitatively, there are some very minor differences in the performance of the barrier algorithms from KSR-1 to KSR-2. In particular, the *tournament* algorithm performs a little worse (between 10% and 15% more for completion times) than the *MCS* algorithm on the KSR-2. With faster processors, the detrimental effect due to false-sharing which we alluded to for the *MCS* algorithm on KSR-1 (see Section 3.2.2) is less of an issue, thus reversing the observed performance for these two algorithms on KSR-2. The detrimental effects do become important for the modified versions of these two algorithms. Thus, *tournament(M)* is still the best performer on KSR-2 closely followed by *System* and *tree(M)*.

We also conducted the barrier experiments on a 64-node KSR-2 which involves communication using the second level ring. As can be seen from Figure 5 the same performance trends continue for the larger system as for the 32-node system.

3.3 NAS Benchmark Suite

The Numerical Aerodynamic Simulation (NAS) parallel benchmark [3, 2] consists of five kernels and three applications which are considered to be representative of several scientific and numerical applications. We have implemented three of the five kernels and one application on the KSR-1 as part of our scalability study. The first one is the Embarrassingly Parallel (EP) kernel, which evaluates integrals by means of pseudo-random trials and is used in many Monte-Carlo simulations. As the name suggests, it is highly suited for parallel machines, since there is virtually no communication among the parallel tasks. Our implementation [6] showed linear speedup, and given the limited communication requirements of this kernel, this result was

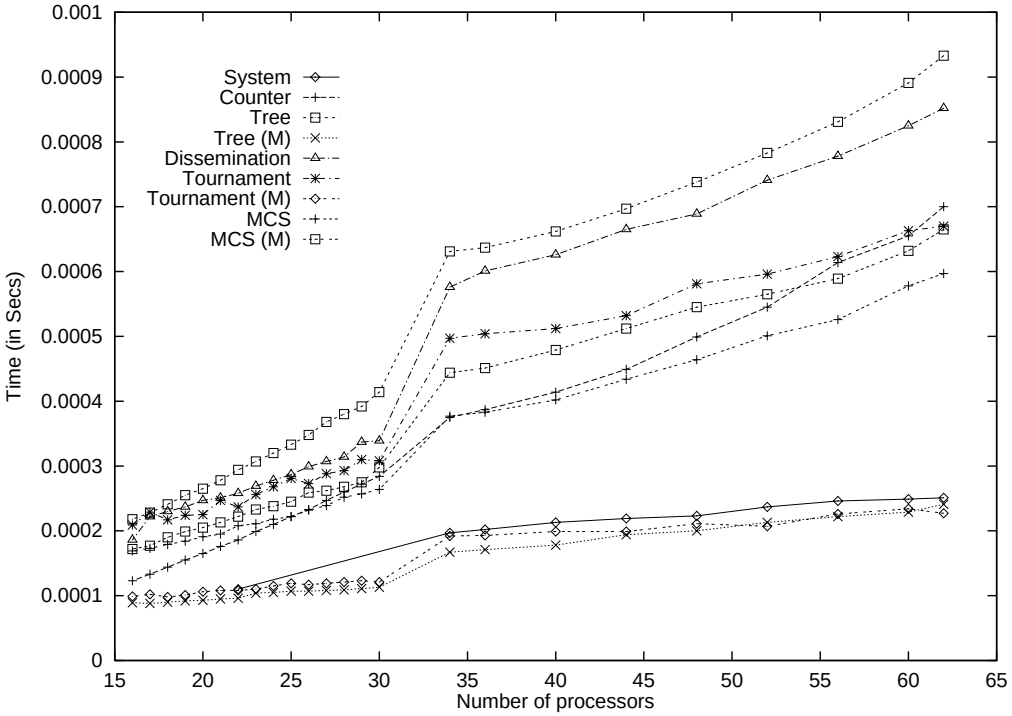


Figure 5: Performance of the barriers on 64-node KSR-2

not surprising.

For the rest of this section we focus on the other two kernels, namely, the conjugate gradient method (CG) and the integer sort (IS) since both these kernels revealed interesting insight into the scalability of the KSR-1 architecture. In both the kernels, there is considerable amount of true data sharing among the processors; and further in both kernels one data structure is accessed in sequence but another is accessed based on the value in the first data structure. Thus a considerable part of the data access patterns in both the kernels is data dependent.

3.3.1 Conjugate Gradient

The CG kernel computes an approximation to the smallest eigenvalue of a sparse symmetric positive definite matrix. On profiling the original sequential code (available from NASA Ames), we observed that most of the time (more than 90%) is spent in a sparse matrix multiplication routine of the form $y = Ax$, wherein x and y are n-element vectors and A contains the non-zero elements of an $n \times n$ sparse matrix. Since most of the time is spent only in this multiplication routine, we parallelized only this routine for this study. The sequential code uses a sparse matrix representation based on a column start, row index format. The original loop to perform the multiplication is as shown in Figure 6. In this loop, a is a linear array containing the non-zero data elements of the original matrix A. As can be seen the elements of y are

```
      do 200 j = 1, n
          xj = x(j)
          do 100 k = column_start(j) , column_start(j+1)-1
              y(row_index(k)) = y(row_index(k)) + a(k) * xj
100       continue
200   continue
      return
      end
```

Figure 6: Matrix Multiplication Loop

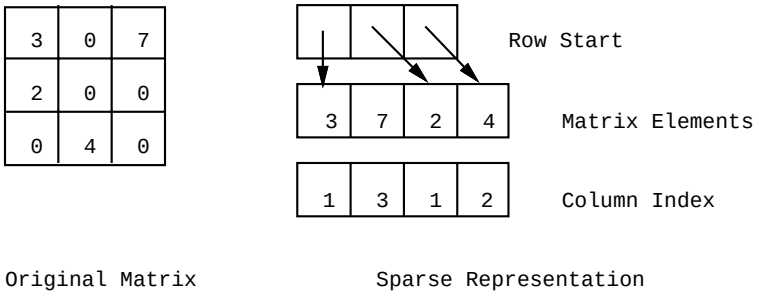


Figure 7: Spare Matrix Representation

computed in a piece-meal manner owing to the indirection in accessing the y vector. Therefore, there is a potential for increased cache misses in carrying out this loop. We can expect better locality for cache access if an element of y (say $y(i)$) is computed in its entirety before the next element (say $y(i+1)$) is computed. Thus we modified the sparse matrix representation to a row start column index format (see Figure 7). This new format also helps in parallelizing this loop. If a distinct set of rows is assigned to each processor, then the processor with row i becomes responsible for generating the i-th element of the y vector, without any need for synchronization among the processors for updating the successive elements of the vector y. With the original format, the obvious way of parallelizing by assigning a set of distinct columns of the sparse matrix to each processor could result in multiple processors writing into the same element of y necessitating synchronization for every access of y.

Each processor of the KSR-1 is rated at 40 MFlops peak. For the EP kernel, we observed a sustained single processor performance of about 11 MFlops. Given that the sub-cache latency is 2 cycles, and local-cache latency is 18 cycles, this performance is quite impressive. This performance is achieved due to the very limited number of data accesses that the algorithm has to perform, and the extremely compute intensive nature of the EP kernel. On a single processor, the CG algorithm yields 1 MFlop for a problem size of n = 14K, with 2M non-zero elements. The relatively poor absolute performance of the algorithm can be attributed to the data access pattern that may not be utilizing the sub-cache and local-cache efficiently, and to the limited size of the two caches. This reason was also confirmed by measuring the number of accesses

Processors	Time (in seconds)	Speedup	Efficiency	Serial Fraction
1	1638.85970	1.00000	-	-
2	930.47700	1.76131	0.881	0.135518
4	565.22150	2.89950	0.725	0.126516
8	259.55210	6.31418	0.789	0.038141
16	126.51990	12.95340	0.810	0.015680
32	72.00830	22.75930	0.711	0.013097

Table 1: Conjugate Gradient, datasize=$n = 14000, nonzeros = 2030000$

(local-cache and remote caches) using the hardware performance monitor.

Table 1 gives the speedup, efficiency, and the measured serial fraction[6] [12] for the CG algorithm on the KSR-1. Figure 8 shows the corresponding speedup curve for the algorithm. Up to about 4 processors, the insufficient sizes of the sub-cache and local-cache inhibits achieving very good speedups. However, notice that relative to the 4 processor performance the 8 and 16 processor executions exhibit superunitary[7] [9] speedup. This superunitary speedup can be explained by the fact that the amount of data that each processor has to deal with fits in the respective local-caches, and as we observed earlier the algorithm design ensures that there is very limited synchronization among the processors. The superunitary speedup is confirmed by the fact that the efficiency remains almost a constant while the serial fraction decreases. We notice that there is a drop in speedup when we go from 16 to 32 processors. We attribute this drop to the increase in the number of remote memory references in the serial section of the algorithm. With 32 processors, since each processor now works on a smaller portion of the data, the processor that executes the serial code has more data to fetch from all the processors thus increasing the number of remote references. To prove that our hypothesis is indeed the case, we modified the implementation using poststore to propagate the values as and when they are computed in the parallel part. Since poststore allows overlapping computation with communication, the serial code would not have to incur the latency for fetching all these values at the end of the serial part. Using poststore improves the performance (3% for 16 processors), but the improvement is higher for lower number of processors. For higher number of processors, the ring is close to saturation due to the multiple (potentially simultaneous) poststores being issued by all the processors thus mitigating the benefit.

Given this result, we can conclude that the CG algorithm scales extremely well on this architecture.

[6]Serial fraction represents the ratio of the time to execute the serial portion of the program to the total execution time of the program. The measured serial fraction is a ratio derived using the execution times and incorporates the above algorithmic as well as any other architectural bottlenecks.

[7]Speedup using n processors $S(n)$ is said to be superunitary when $\lim_{n\to\infty} S(n)/n$ is greater than 1.

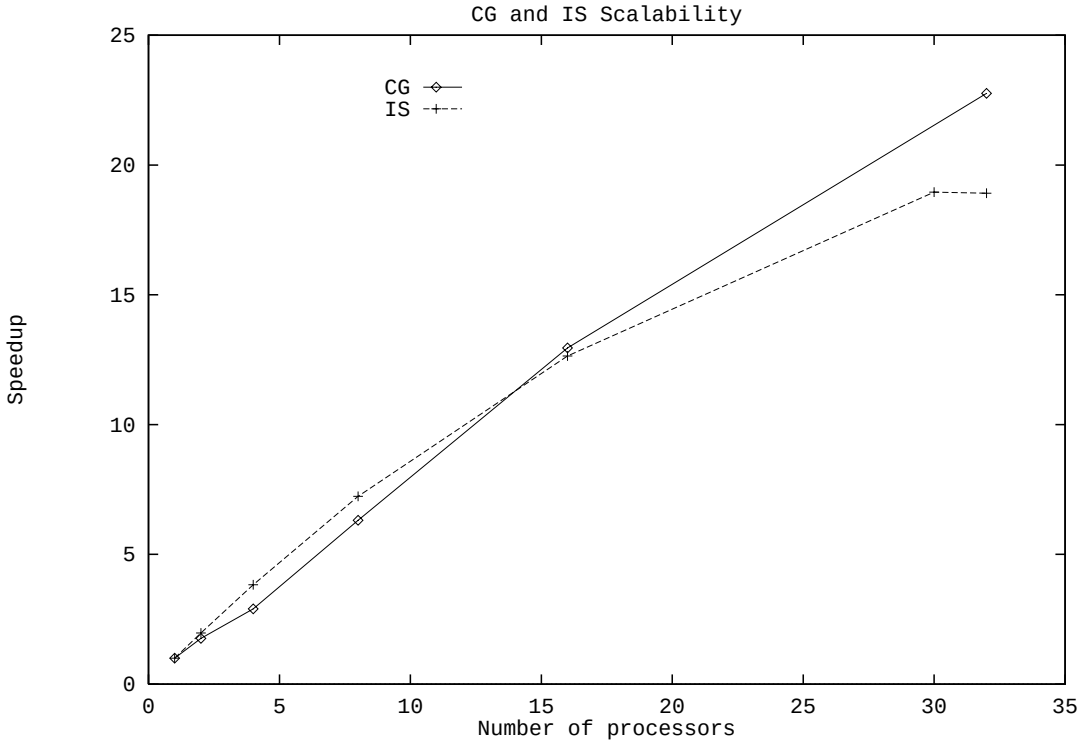


Figure 8: Speedup for CG and IS

KSR-1 has an architectural mechanism for selectively turning off sub-caching of data. Given that the CG algorithm manipulates three huge vectors, it is conceivable that this mechanism may have been useful to reduce the overall data access latency and thus obtain better performance. However, there is no language level support for this mechanism which prevented us from exploring this hypothesis. The "prefetch" primitive of KSR-1 allows remote data to be brought into the local-cache. There is no architectural mechanism for "prefetching" data from the local-cache into the sub-cache. Given that there is an order of magnitude difference in the access times of the two, such a feature would have been very useful.

3.3.2 Integer Sort

Integer Sort (IS) is used in "particle-in-cell method" applications. The kernel is implemented using a bucket sort algorithm. In this algorithm, each key is read and count of the bucket to which it belongs is incremented. A prefix sum operation is performed on the bucket counts. Lastly, the keys are read in again and assigned ranks using the prefix sums.

This sequential algorithm proves to be quite challenging for parallelization. A simple method to parallelize the algorithm is to distribute the input keys among all the processors and maintain a global bucket count. Since multiple input keys could belong to the same bucket, this parallel method now requires synchronization to access the global bucket count. In order to avoid this synchronization, we replicate

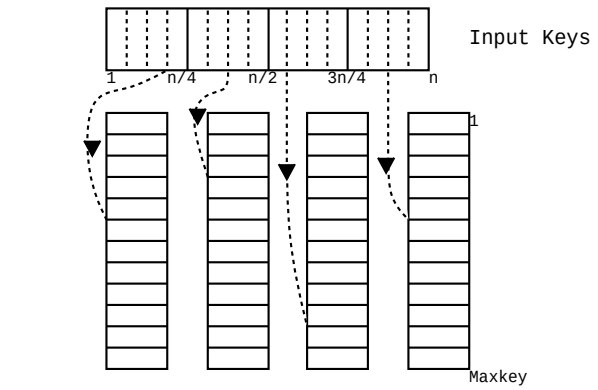
Processors	Time (in seconds)	Speedup	Efficiency	Serial Fraction
1	692.95492	1.00000	-	-
2	351.03866	1.97401	0.987	0.013166
4	180.95085	3.82952	0.957	0.014839
8	95.79978	7.23337	0.904	0.015141
16	54.80835	12.64320	0.790	0.017700
30	36.56198	18.95290	0.632	0.020099
32	36.63433	18.91550	0.591	0.022314

Table 2: Integer Sort, Number of Input Keys=2^{23}

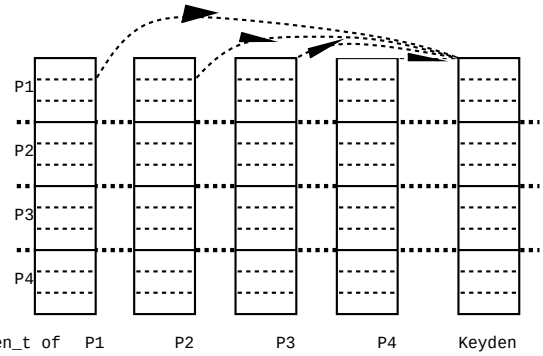
the bucket count data structure at each processor. Maintaining such local counts requires accumulating them once they have been computed, in order to compute the prefix sums. This step is not present in the sequential algorithm. Computation of the ranks would also necessitate synchronization, since multiple processors need to update the data structure holding the prefix sums. To avoid this synchronization, each processor copies the prefix sums data structure atomically and updates it to reflect the ranks it would assign. This step, which is also not present in the sequential version, serializes access to the prefix sums data structure. However, since this data structure is large (roughly 2 MBytes), only a portion of it is locked at a time by each processor allowing for pipelined parallelism. Since 32 MBytes of local-cache is available on each processor, replicating the bucket count data structure is not a problem in terms of space on the KSR-1. Figure 9 shows the six phases of a parallel algorithm for implementing bucket sort. For simplicity of exposition the figure shows the phases of the algorithm using four processors P1 through P4.

As can be seen from Figure 9, phase 4 is a sequential phase where one processor accumulates the partial prefix sums computed by all the processors in the previous phase. In fact the time for this phase T_s increases as the number of processors increases since the number of partial sums to be accumulated increases. Moreover, since the partial sums have been computed in parallel, accumulating them in one processor results in remote accesses. Finally, phase 6 requires access to the global prefix sums by all processors. Thus the time for this phase also increases with the number of processors.

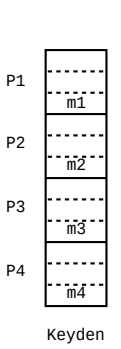
Table 2 shows the speedup, efficiency and serial fraction measurements for this algorithm on KSR-1. We notice that the serial fraction increases with the number of processors. The amount of simultaneous network communication increases with the number of processors in phase 2. As we observed in Section 3.1, the network is not a bottleneck when there are simultaneous network transfers until about 16 processors. Barrier synchronization (using the system barrier) is used at the conclusion of the phases and we know from the synchronization measurements (see Section 3.2) that barrier times increase with the number of processors. However, from the measurements we know that the time for synchronization in this algorithm

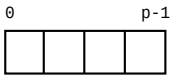


Phase 1: Each Processor reads its portion of Key and updates its local bucket count (keyden_t)



Phase 2: Each processor reads in appropriate portions of the local bucket counts from all processors and accumulates it in appropriate portions of the global bucket count (keyden)

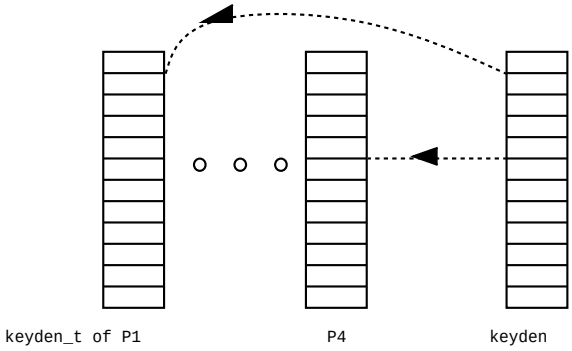




Phase 3: Each processor computes partial prefix sums for its portion of keyden. At the end of this phase the maximum value computed by each processor is m1,m2,m3,m4 respectively

Phase 4: Processor P1 computes prefix sums of m1 to m4 in tmp_sum

Phase 5: Each processor i reads in tmp_sum[i-1] and adds it to each element in its portion of keyden. At the end of this phase the global prefix sums has been computed in keyden.



Phase 6: Each processor atomically reads in keyden into its local copy keyden_t and appropriately decrements keyden

Phase 7: Again each processor reads in its portion of the input from key and assigns ranks based on its count in the appropriate bucket in keydent_t, which is then decremented.

Figure 9: Parallelization of Integer Sort

is negligible compared to the rest of the computation. The increase in serial fraction due to these two architectural components is quite small compared to that due to the inherent nature of phases of 4 and 6 of the algorithm. Therefore, we contend that the slow-down from 16 to about 30 processors is more due to the algorithm than any inherent architectural limitation. However, we see from the Table that there is a sharper increase in the serial fraction from 30 to 32 processors. This increase we attribute to the ring saturation effect with simultaneous network accesses that we mentioned earlier (see Section 3.1). Thus the only architectural bottleneck that affects the performance of this algorithm at higher number of processors is the bandwidth of the ring.

The reason for the extremely good speedups observed for up to 8 processors can be explained by the efficient utilization of the caches by the algorithm. Each of the data structures *key* (the input to be sorted) and *rank* (the array storing rank for each key) is of size 32 MBytes. Therefore just one of these data structures is enough to fill the local-cache. With fewer processors there is considerable amount of remote accesses due to over-flowing the local-cache. As we increase the number of processors, the data is distributed among the participating processors. Thus each processor has lesser data to manipulate thus reducing local-cache misses. These caching effects dominate initially as we increase the number of processors aiding the performance. However, the inherent communication bottlenecks in the algorithm (phases 4 and 6 in particular) result in a loss of efficiency at higher number of processors. This reasoning was confirmed by the data from the hardware performance monitor which shows that the latencies for remote accesses start increasing as the number of processors increase.

3.3.3 The SP Application

The scalar-pentadiagonal (SP) application is part of the NAS parallel benchmark suite. The SP code implements an iterative partial differential equation solver, that mimics the behavior of computational fluid dynamic codes used in aerodynamic simulation. Thus, techniques used to speed up the SP application can be directly applied to many important applications that have similar data access and computation characteristics.

Table 3 shows the performance of the application on the KSR-1. The table shows the time it takes to perform each iteration in the application (it takes 400 such steps to run the benchmark). Details of the application performance can be found in [4]. The results show that the application is scalable on the KSR-1. However, it is instructive to observe the optimizations that were necessary to get to this level of performance. As shown in Table 4, the initial unoptimized version of the code took about 2.54 seconds per iteration using 30 processors. Given the known data access pattern of the application, we discovered that there was a big disparity between the expected number of misses in the first level cache and the actual misses as observed from an analysis of the data provided by the hardware performance monitor. The large data set of the application, and the random replacement policy of the first level cache combined to cause thrashing of data in the first level cache. By careful re-organization of the data structures in the application the performance was improved by over 15% to 2.14 seconds per iteration. But the data structure organization can only help in

Processors	Time per iteration (in seconds)	Speedup
1	39.02	-
2	19.48	2.0
4	10.02	3.9
8	5.04	7.7
16	2.55	15.3
31	1.40	27.8

Table 3: Scalar Pentadiagonal Performance, data-size=64x64x64

Optimizations	Time per iteration (in seconds)
Base version	2.54
Data padding and alignment	2.14
Prefetching appropriate data	1.89

Table 4: Scalar Pentadiagonal Optimization(using 30 processors), data-size=64x64x64

the reuse of variables. Each iteration is composed of three phases of computation. Communication between processors occurs at the beginning of each phase. By using prefetches, at the beginning of these phases, the performance was improved by another 11%. Thus, the savings from the use of explicit communication primitives available in KSR is substantial. The improvement from 1.89 seconds using 30 processor in Table 3 to 1.40 seconds using 31 processors in Table 4 can be attributed to the load balance that occurs because of the use of one extra processor, in addition to the use of the extra processor itself.

We experimented with the use of poststore mechanism too. But its use caused slowdown rather than improvements. The reason for the slowdown is because of the interaction between the write-invalidate protocol and the exclusive data use required in the application across the phases. A processor issuing a poststore operation is stalled until the data is written out to the second level cache. However this does not help, because even though data might be copied into the caches of the other processors that need the value, it is in a shared state. Since in this application, these other processors need to write into these data items at the beginning of the next phase, they have to pay the penalty of a ring latency to invalidate the other shared copies.

4 Concluding Remarks

We designed and conducted a variety of experiments on the KSR-1, from low level latency and synchronization measurements to the performance of three numerical computational kernels and an application. Thus we believe that this study has been a fairly good assessment of the scalability of the architecture.

There are several useful lessons to be learned as a result of this scalability study. The sizes of the sub-cache and local-cache is a limitation for algorithms that need to manipulate large data structures as is evident from our results. The prefetch instruction of KSR-1 is very helpful and we used it quite extensively in implementing CG, IS and SP. However, the prefetch primitive only brings the data into the local-cache. It would be beneficial to have some prefetching mechanism from the local-cache to the sub-cache, given that there is roughly an order of magnitude difference between their access times. Further, (as we observed in the CG algorithm) the ability to selectively turn off sub-caching would help in a better use of the sub-cache depending on the access pattern of an application. Although allocation is made in blocks of 2 KBytes in the sub-cache and 16 KBytes in the local-cache, for the algorithms implemented (owing to their contiguous access strides), we did not experience any particular inefficiency. For algorithms that display less spatial locality this allocation strategy could pose efficiency problems. The architecture aids computation-communication overlap with the prefetch and poststore instructions both of which resulted in improving the performance in our experiments with the barrier algorithms and the kernels. However, the poststore primitive sometimes causes performance degradation as was observed for SP, because of its implementation semantics. Our results also show that indiscriminate use of this primitive can be detrimental to performance due to the increased communication saturating the network for large number of processors.

We also learned some lessons on structuring applications so that the best performance can be obtained on shared memory machines. For instance, the pipeline nature of the ring network is designed to be resilient to the network load in the presence of simultaneous remote accesses. This feature improves the performance of certain barrier algorithms such as the tournament and MCS. Pipelining of the network can also be exploited with a suitable choice of data structures for shared variables as we demonstrated in the CG and IS algorithms. The SP implementation also showed us the importance to sidestep architectural limitations (in this case cache replacement policy by careful data layout) to improve performance.

Our barrier experiments on a 64-node KSR-2 showed that the performance trends that we observe for a one-level ring carry over to a two-level ring. The increased latency (when we cross the one-level ring boundary) manifests itself as a sudden jump in the execution time when the number of processors is increased beyond 32. The same trend is expected for applications that span more than 32 processors.

Acknowledgments

We would like to thank Steve Frank of Kendall Square Research for clarifying some of the architectural concepts of KSR-1. We also would like to thank Michael Scott of University of Rochester for his comments

on our results of the barrier algorithms on the KSR-1. Karl Dias helped with a preliminary implementation of software locks on KSR-1.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] D. H. Bailey, E. Barszcz, and J. T. Barton et al. The NAS parallel benchmarks - summary and preliminary results. In *Supercomputing*, pages 158–65, November 1991.

[3] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report Report RNR-91-002, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, January 1991.

[4] S. R. Breit and G. Shah. Implementing the NAS parallel benchmarks on the KSR-1. In *Parallel CFD*, May 1993.

[5] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, October 1991.

[6] Steve Breit et al. Implementation of EP, SP and BT on the KSR-1. KSR internal report, September 1992.

[7] Steve Frank. Personal communication, January 1993.

[8] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[9] D. P. Helmbold and C. E. McDowell. Modelling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):.250–6, April 1990.

[10] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, February 1988.

[11] Intel Corporation, Beaverton, Oregon. *Touchstone Delta System User's Guide*, 1991.

[12] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[13] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[14] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, March 1991.

[15] J. F. Palmer and G. Fox. The NCUBE family of high-performance parallel computer systems. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 847–51 vol.1, 1988.

[16] Kendall Square Research. *KSR1 Principles of Operations*, 1992.

[17] Kendall Square Research. Technical summary, 1992.