A Study of Collective Communication on the KSR-1[*]

Sanjay Goil Albert Wang Sanjay Ranka
Department of Computer Science and
Northeast Parallel Architectures Center
Syracuse University
4-116, Center for Science and Technology
Syracuse, NY, 13244-4100
sgoil@top.cis.syr.edu
Corresponding and Presenting author : Sanjay Goil

Abstract

In this paper we study the primitives for structured communication on the Kendall Square Research Multiprocessor KSR-1. Many parallel applications require operations that involve all the participating processors or some subset of the processors. We have studied the primitives for collective communication and modeled the cost for each on the ALLCACHE memory . We find that some algorithms with a large requirement for remote data from processors on the same ring perform almost as well as the ones that require little communication. We observe very little node and link contention by multiple processors referencing data from the same processor. We also study the use of multiple threads on a single processor and observe a significant overlap between computation and communication.

[*]This research was performed in part using 128-node KSR-1 multiprocessor at the Theory Center at Cornell University.

1 Introduction

The parallelization of most data parallel problems on distributed memory machines involve partitioning of data onto the available memory of processors. In a distributed memory model, it is intuitive to observe that processors need to communicate with each other to access off-processor data and in certain cases perform global operations, either for synchronization or for accumulating results. Several collective data movement (communication) primitives have been identified in the literature: broadcast, cyclic shift, reduction, concatenate-one and concatenate-all (all-to-all broadcast).

Traditional distributed memory machines have little hardware support for shared memory. Processors exchange data using message passing. Several algorithms have been developed in the literature for performing the above communication primitives on parallel machines. The performance of the applications on these parallel machines is determined by how well these primitives can be implemented on these machines. These algorithms are developed keeping the following factors in mind:

1. *node contention*: For most architectures, a node can receive only one (or a limited number) of messages at a particular time. Thus if two nodes are trying to send a message to the same node, then one or more of the sending nodes will be delayed.

2. *link contention*: When multiple messages are being transferred over the communication network, the paths of the two messages may overlap on a few links. This may increase the total time spend on communication by one or more of the processors involved in the communication.

Unlike other commercial machines, KSR1 has a distributed shared memory architecture, in which the global memory is partitioned among processors which are connected by a hierarchy of slotted rings (explained in a later section). There exists no concept of a processor owning data. Data is brought into a processor's memory when it is referenced. Hence, it is not obvious whether the same primitives are relevant for this architecture.However, efficient parallelization of many applications on distributed shared memory machines require similar optimizations as are required by machines which do not have support for shared memory [7] [8]. This is because local accesses are still at least order of magnitude faster than nonlocal accesses. Further, the same issues of link contention/node contention are relevant if data needs to be transferred from local memory of one processor to another processor.

In this paper, we evaluate the performance of these primitives on the KSR and study the effect of node/link contention on their performance.

The basic mode of parallelism on the KSR-1 is through the use of threads. The overheads of thread creation, scheduling and synchronization must be taken into account when using multiple threads on a processor. We study the effect of using threads on computation, communication and then combine the two operations to study overlap of computation with communication.

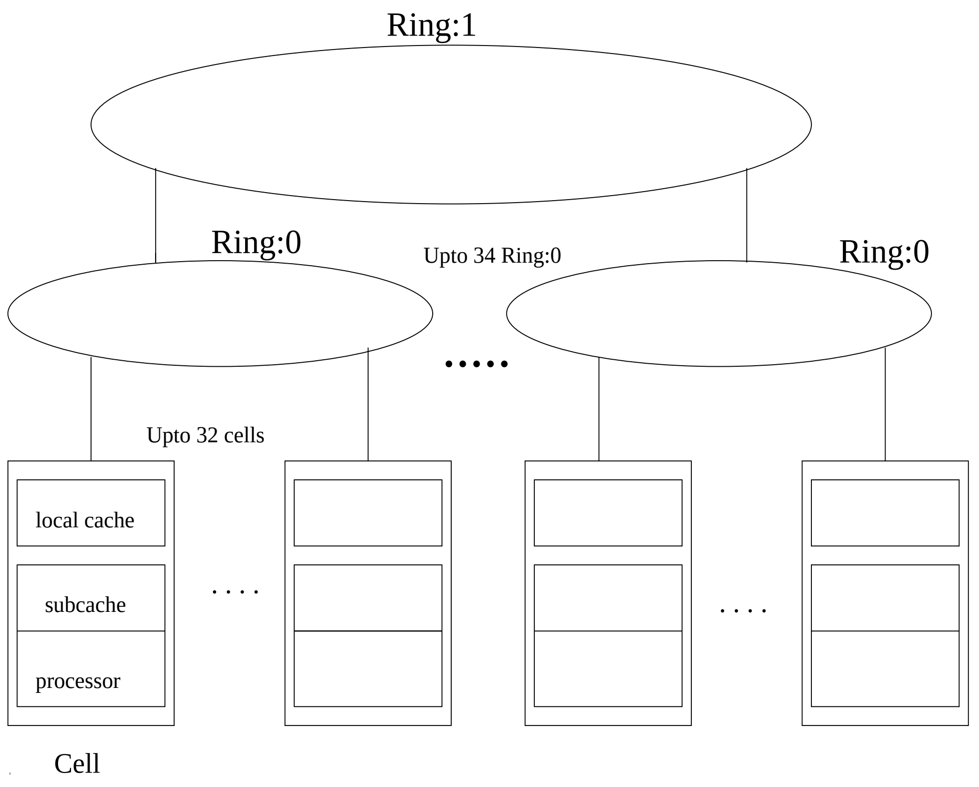


Figure 1: Two levels of the KSR-1 architecture.

Section 2 introduces the architecture of Kendall Square Research KSR-1, and the programming model it supports. Section 3 describes the operations which we are interested in studying. Section 4 discusses the node-to-node communication primitives that includes neighbor and non-neighbor communications. Section 5 presents the structured collective communication primitives, broadcast, cyclic shift, reduction, concatenate-one and concatenate-all (all-to-all broadcast). In Section 6 we analyze the performance of using multiple threads for overlapping computation and communication. Section 7 concludes the paper.

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 (Figure 1). The lowest level, ring:0, consists of a 34 slot backplane connecting 32 processing cells and two cells responsible for routing to the next higher layer ring - ring:1. A fully populated ring:1 is composed of the interconnecting cells from 32 ring:0 rings. Current implementations of the architecture support two levels of the rings, hence up to 1088 processors can be supported.

2.1 System Hardware

Each processing node (called a *cell* in KSR-1 and used interchangeably for a node in the rest of this paper) contains 32 MBytes of second-level cache, *local cache*, and 0.5 MBytes (0.25 MBytes for data and 0.25 MBytes for instructions) of first-level cache, *sub-cache* and consists of four functional units:

- the Co-Execution Unit (CEU) fetches all instructions, controls data fetch and store, controls instruction flow, and does arithmetic required for address calculations.
- the eXternal I/O Unit (XIU) performs DMA and programmed I/O.
- the Integer Processing Unit (IPU) executes integer arithmetic and logical instructions.
- the Floating Point Unit (FPU) executes floating point instructions.

and two types of control units:

- four Cache Control Units (CCU) are the interface between the 0.5 MBytes sub-cache and the 32 MBytes local-cache.
- four Cell Interconnect Units (CIU) are the interface between a processing cell and the ring:0 ring.

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

In addition to the 32 processing cells, each ring:0 also contains 2 ALLCACHE Routing and Directory (ARD) cells. One of the ARD cells is an uplink from the ring:0 to ring:1. The other ARD is a downlink from the ring:1 to ring:0. The ARDs participate in the transfer of shared memory between ring:0s across ring:1.

2.2 Memory Organization

All of the local-caches, together with the interconnecting rings make up the ALLCACHE memory system. The architecture provides a sequentially consistent and strongly ordered shared memory model. This shared memory constitutes the System Virtual Address (SVA) space that is global to the entire multiprocessor system. The programmer sees the shared memory in terms of a Context Address (CA) space that is unique for each process. Addressing in the KSR architecture is based on the translation of a CA into a SVA. Context addresses are composed of a segment and offset and are translated into SVA via fully associative hardware Segment Translation Tables (STTs) on each processor. There are two STTs, one for data and one for instructions.

The SVA space consists of all of the local caches. The ALLCACHE memory and the organization and management of SVA space is the major difference between the KSR architecture and other architectures. A distinguishing characteristic of the ALLCACHE memory is that there is no fixed location for any SVA. When a processor references a SVA, a *search engine*, which is the collection of CIUs and the ARD on each ring:0 along with the ring interface, locates the SVA and moves its contents to the local cache of the referencing processor.

ALLCACHE stores data in units of pages and subpages. Each local cache can hold 2,048 pages, each contains 16K bytes divided into 128 subpages of 128 bytes each. The memory system allocates storage in the local-caches on the basis of pages and each page of SVA space is either entirely allocated in the caches or not allocated at all. An invalidation-based cache coherence protocol is used to maintain sequential consistency. The unit of consistency maintenance and transfer on the ring is a subpage. Even though the transfer size is 128 bytes (size of subpages), allocation is done on a 16K page basis in the local-cache. Upon allocation only the accessed subpage is brought into the local-cache. All other subpages 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 *blocks* of 2K bytes, 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.

Whenever a page of SVA space is allocated in the system, there may be more than one copy present. This would be the case when several threads running on different processors are all referencing shared memory. In the cache directory of each cell, additional information is maintained that represents the *state* of each subpage in the local-cache. There are four states that a sub-page can be in:

- exclusive-owner: this is the only valid copy of the sub-page in all of the local caches (i.e., in the entire system). The contents can be read or modified.

- atomic: like exclusive, this is the only valid copy and the subpage can be modified. This state also provides a flag to allow synchronization by multiple processors. Thus, this state provides for locks.

- shared (read-only): indicates that there are two or more valid copies of this subpage among all of the local caches. The contents of this subpage cannot be modified until its state is changed to exclusive or atomic.

- invalid: the contents of this subpage are not to be accessed (ie., read or modified). Newly allocated pages set all subpage descriptors to invalid. This state is analogous to the setting of a ”dirty bit.”

The instruction sub-cache allows each sub-block to be in either the invalid state or the shared

Location of subpage	Total capacity (MB)	Latency in cycles (5ns)
Local sub-cache	0.5	2
Local cache	32	18
Ring:0	1,024	175
Ring:1	34,816	600
Disk	-	400,000

Table 1: Latencies and Peak Performance given by KSR

state. In addition to invalid and shared state, the data sub-cache allows a block to be in the exclusive-owner state to allow for modification.

When a processor references a SVA it continues execution for two cycles, which is the latency of the sub-cache. If the address is not contained in the sub-cache, a request is presented to the CCUs to locate the subpage containing the requested address in the ALLCACHE memory. If the subpage containing the address is not present in the local-cache, the the CCUs make a request of the local CIUs to format a request message and place it on ring:0. The ring:0 communication interconnect is a slotted pipelined ring with a total bandwidth of 1GBytes. There are 13 slots on the ring:0 ring. Each message on the ring consists of a 16 byte header followed by one subpage of data. As a request message passes each processing cell, the cell's CIU determines if the request can be satisfied from its local cache. If it can be satisfied, the request message is extracted from the ring and a response message is inserted. Also attached to each ring:0 is an ALLCACHE ARD cell that contains a directory of the entire ring:0 cache (i.e., all of the local-caches). If the ARD determines that a request message cannot be satisfied on the local ring:0, it extracts the message and inserts a request on the next higher ring in the hierarchy, ring:1. Once a response message to the original request is inserted on the ring, the requesting processor copies the message and fills the original request from the local CCU. If a request message returns to the requesting processor unanswered, a hard page fault is generated and the subpage is brought in from the disk. The latency and capacity (from KSR Corporation) of the ALLCACHE memory system hierarchy is shown in Table 1.

3 Programming Model

3.1 Pthreads

C programming on KSR-1 is basically using the *pthread*, a low level parallel-control mechanism. A pthread is a sequential flow of control within a process, that cooperates with other pthreads to solve a problem. A program begins with one pthread and creates others to perform work in parallel. Typically, an application creates multiple pthreads that execute simultaneously. The programmer determines what data the pthreads share and what data is private to particular pthreads.

Processor sets are groups of processors setup by the system administrator. Processes are assigned to a processor set when they start. A process cannot move from one processor set to another. The threads in a process can only work on those processors assigned to the processor set.

3.2 private and shared Qualifiers

__private and __shared are qualifiers used in declarations to specify whether the declared variable is pthread-private or shared. If the declared variable is private, each pthread binds the variable name to a different storage location, giving the pthread a private copy of the variable. Otherwise, all pthreads bind the variable's name to the same storage location.

3.3 Data Alignment

To prevent excess data movement, the programmer should align data on subpage boundaries. This minimizes the possibility of thrashing when writing data. **__align128** is a qualifier used in declarations to specify that the declared variable should be subpage aligned, i.e. aligned to a 128-byte boundary.

3.4 Prefetch and Poststore

To maximize throughput, the programmer needs to consider network bandwidth and memory latency. If the programmer can anticipate sufficiently far in advance the program's need for data, memory latency needs not be a limiting factor either. KSR-1 provides *prefetch* and *poststore* which are mechanisms for minimizing memory-latency delay [1] by overlapping communication with computation.

Prefetch instructions allow a processor to fetch data from another processor's local cache before it is needed. The goal of prefetch is to maximize the overlap of computation and asynchronous remote data access. Prefetch is controlled by the processor that needs to read the data.

Poststore instructions allow a processor to broadcast data needed by another processor(s), before the other processor(s) requests the data. A program that updates a location can use poststore to ask the memory system to broadcast the new value to local caches that contain that address. Poststore is controlled by the processor that writes the data.

3.5 Test Programs

In order to measure the communication time needed for transferring data from one node to the other node, we make the required data locally cached in the source node. The destination node either reads from or writes to the source node. One node can only access global variables that are locally cached in the other nodes. Atomic variables are invisible to other nodes. We run a single thread on each processor and explain in the next section each of the operations and their requirements for communication. To avoid false sharing we take care to access data across subpage boundaries while projecting costs per 128 byte subpage transfers. The data we present has been obtained by executing these test programs on the 128 node KSR1 at the Cornell Theory Center using the C Compiler and KSR's implementation of the OSF-1 Mach threads.

4 Node-to-Node Communication

The time needed for node-to-node communications is higher than the local access due to the network search and access. Nodes in a ring:0 are connected with a unidirectional ring. The data rates are as high as 8 million packets (16 byte header + 128 byte data) per second in ring:0. Therefore, when one node communicates with another node, the distance between the two nodes is not significant unless they are in the different ring:0's (Figure 2). Since the units of cache coherence maintainance and transfer on the ring is subpage (128 bytes), the smallest size of communications between nodes is a subpage. Upon referencing only the accessed subpage is brought into the local-cache, all other subpages of a page are brought into the local-cache on demand. We use the following method to get the timing for remote access.

1. Declare a global integer array aligned on a 128 byte boundary.
2. Have each node write to an element of the global array. This will cause the subpage that contains the element exclusively cached in the node.
3. A remote access can be achieved by having node i read from (or write to) the element of the global array which is cached in node j, where $i \neq j$. Each access makes sure that a new subpage is fetched. This is done by using the appropriate stride while accessing the array.

Figure 2 shows us that remote read within ring:0 is independent of the distance between two processors. Only when we move to processors that involve processors over ring:1, i.e more than 32 processors we observe a jump in the cost and it remains the same for all the processors. The increase is attributed to the overhead of routing through the ARD cell when going from ring:0 to ring:1.

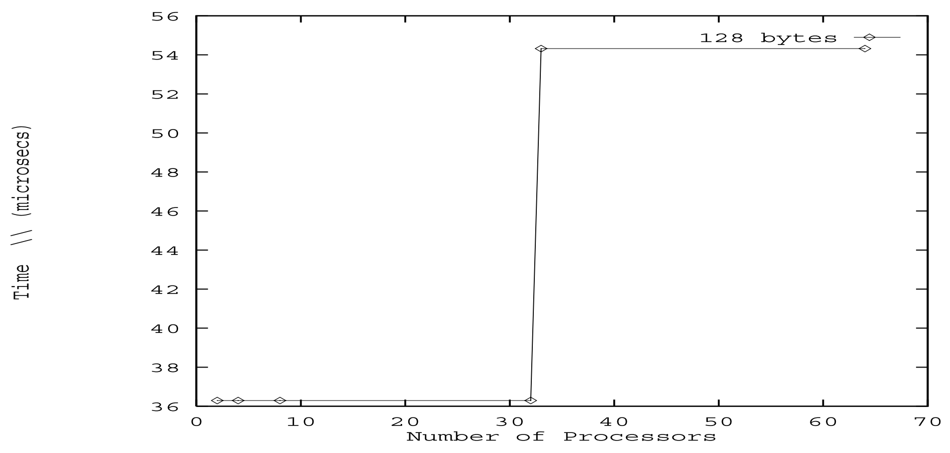


Figure 2: Remote Read from one node by other processors

5 Structured Collective Communication

For solving a large number of regular and synchronous applications (e.g. matrix multiplication, Gaussian elimination), the communication required is structured and collective. In this section we will look at some of the global communication primitives needed for this kind of applications: synchronization, broadcast, cyclic shift, reduction, and concatenate (all-to-all broadcast).

5.1 Synchronization

Data parallel problems require barrier synchronization prior to most communications steps to achieve accuracy and efficiency. Typical synchronization mechanisms used in multiprocessors include¡ the use of lock and barriers. We have used global barriers to implement synchronization of all processors. On KSR-1, synchronization cost for 8-node, 16-node, and 32-node processor set are all about 270 microseconds because all the nodes are in the same ring:0. Synchronization for shared variables can be achieved by the *get_sub_page* instruction which allows synchronized exclusive access to a sub-page, and the *release_sub_page* instruction releases the exclusive lock on the sub-page.

5.2 Broadcast

In a broadcast, a message is sent from a single node to all the other nodes. The broadcast operation is frequently needed for matrix algorithms. On KSR-1, the broadcast communication from node 0 to all the other nodes can be achieved by two steps:

1. have a global array exclusively cached in node 0,
2. have all the other nodes read this global array at the same time.

The communication network of KSR-1 supports the simultaneous remote memory accesses from several nodes. However, when 64 processors are used the broadcast takes a little longer, now that some communication takes over Ring:1. When using 64 processors communication generated for the processors on the other ring:0 have to be routed through ring:1 and the ARD cell becomes a bottleneck.

The following equation can be used to arrive at the broadcast time required for a given processor set and the number of subpages :

$$T_{broadcast} = C_0 + C_1 * N_{subpages}$$

Figure 3 shows the cost of the broadcast for different sizes of variables. Our experiments estimate $C_0 \approx 0$ and $C_1 = 0.054$ when all the processors are on the same ring $N_{procs} <= 32$. The broadcast is independent of the number of processors used on ring:0. Using more than 32 processors involve communication over ring:1 and in this case $C_0 = 0.419$ and $C_1 = 0.069$.

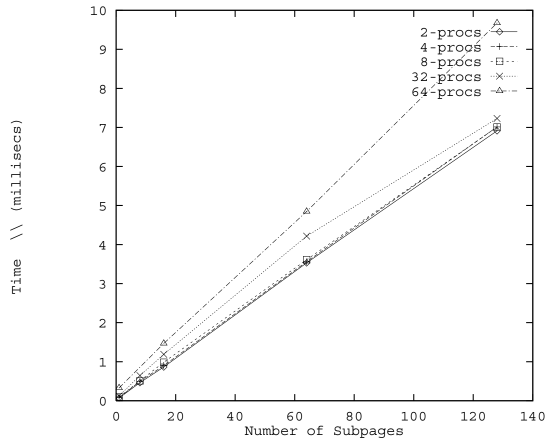


Figure 3: Broadcast

5.3 Node Contention

Each processor can either read the same subpage from processor 0 or different subpages. If the same subpage is read then a processor P_i which requests data from P_0 can get it's request satisfied from a processor P_j on the ring, which may have a copy of the page it has read from P_0. and we are not accounting for the node contention that arises if many processors read from the same node. Keeping the subpages distinct achieves our objective of studying node contention by making sure all read requests are satisfied by one processor , in this case P_0. Table 2 studies the results for upto 16 processors. We do not observe much degradation in performance for more processors. This shows that node contention is not a major problem that we need to address.

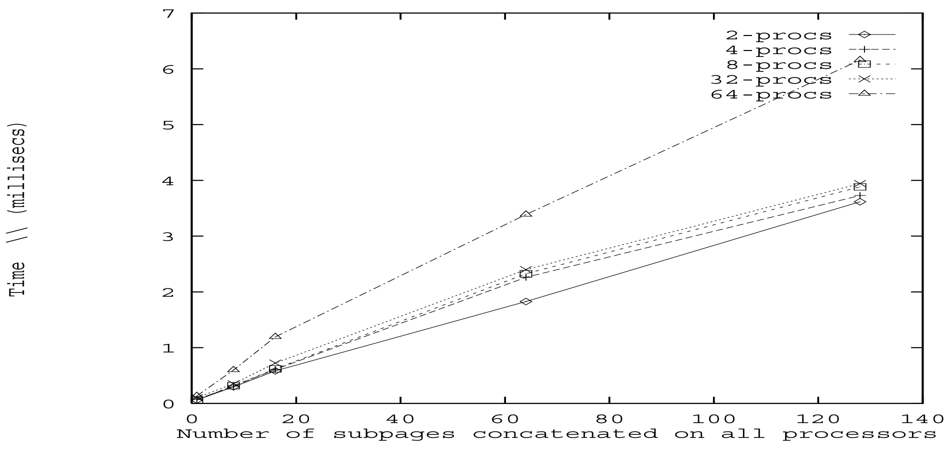


Figure 4: Cyclic Shift costs

	Processors							
	2	4	6	8	10	12	14	16
Time	6.07	6.26	6.45	6.50	6.85	6.92	7.04	7.31

Table 2: Broadcast cost per subpage when reading distinct subpages (Total CPU time in microsecs)

Processors	Distance						
	1	8	16	24	31		
32	3.88	3.64	3.66	3.63	3.88		
	1	8	16	32	48	56	63
64	3.97	4.41	5.15	7.94	5.20	4.38	3.92

Table 3: Cshift by different distances to study link contention (Total CPU time in milliseconds)

5.4 Cyclic Shift

The ring structure of the nodes on KSR-1 can be used to do cyclic shift operation, in which each node communicates with its right (or left) neighbor, the last (first) node reads from the first (last) node. Figure 4 shows the cost of a cyclic shift for different sizes of variables. The cost of cyclic shift can be modeled by the following equation with shifts at distance 1 :

$$T_{cshift} = C_1 * N_{procs} + C_2 * N_{subpages}$$

where $C_1 = 0.007$ and $C_2 = 0.029$. Because there is no link contention in the ring, the cost for cyclic shift is close to that of a single node-to-node read. Our experiments also show that the direction (clockwise or counter-clockwise) does not affect the time to perform cyclic shift. In the counter-clockwise direction the packets need to travel all around the ring to communicate to the right neighbor.

We have also studied the link contention that is caused by doing circular shifts of greater distances. Each processor shifts data to another processor at a certain distance on the ring. With increasing distance more paths overlap and cause link contention. A processor i communicates with a processor *(i + k) mod n*, where k is the distance of the destination processor and n is the number of processors. Table 3 presents our results for such shifts. In case of 32 processors the time is independent of the shift distance. When using 64 processors the time depends on the shift distance because the number of messages traversing over the ARD cell decides on the cost. At distance 32 all processors on one ring:0 shift data to a processor on the other ring:0. Thus maximum communication takes place over ring:1 and again the ARD cell becomes a bottleneck.

5.5 Concatenate-One

Some applications require that each processor receives data from all other processors. For instance, in the classical N-body algorithm of arithmetic complexity $O(N^2)$, every particle interacts with every other particle. Concatenation appends the value from each processor to the values of all preceding processors (in processor identifier order). Assume that each processor has

N elements, and there are P processors. Suppose processor i contains a vector $V_i[0 \cdots N-1]$. The global concatenate operation computes a concatenation of the local list in each of the processors. The resultant vector $R[0 \cdots NP-1]$ is stored in one node (Concatenate-One) or every node (Concatenate-All).

$$R[j] = V_{j\ div\ N}[j\ mod\ N]$$

Concatenate-one appends the value from every other nodes to the value of a node. The result is stored in one node. We use two different algorithms to achieve concatenate-one as follows:

1. Node 0 reads from all other nodes one by one to get all the variables into node 0. This takes O(N_{procs}) steps, but all the steps require the same size (the size of the initial variable) of data movement.
2. Use tree structure to gather all the values into node 0. The size of data movement in an iteration is twice as that at the previous iteration. This takes O($log_2 N_{procs}$) steps.

The number of processors collecting data is halved at each stage. In the first step the first $N_{procs}/2$ processors read $N_{subpages}/N_{procs}$ from the other processors. At the next stage the data doubles as half of these processors gather data and finally after $logN_{procs}$ steps the data is finally concatenated in one processor. The performance of the above algorithms is given in Figure 5. In algorithm 1 the size of data movement in each step are the same, therefore, we use the following formula for the timing of concatenate-one:

$$T_{concatenate} = C_0 + C_1 * N_{procs} + C_2 * N_{subpages}$$

where C_0 is close to 0, C_1 and C_2 are 0.050 and 0.030 respectively. More processors mean a deeper binary tree of processors doing operations on lesser amount of data. Figure 6 shows that performance for 64 processors does not degrade much from that of 32 processors. As compared to algorithm 1, where the size of data is small and uniform but is fetched more often, N times, in algorithm 2 the size of data doubles at each stage and is fetched only $log_2 N$ times. We do not see the expected performance gain by using algorithm 2. Although the number of startups are lower in this algorithm, the amount of bandwidth utilized is the same.

The cost of concatenate is given by the following equation :

$$T_{concatenate} = C_0 + C_1 * log_2 N_{procs} + C_2 * N_{subpages}$$

We have estimated $C_0 \approx 0, C_1 = 0.442$ and $C_2 = 0.029$ from our experiments.

5.6 Concatenate-All (All-to-All Broadcast)

Concatenate-all is the concatenate-one followed by a broadcast operation. An alternative algorithm for concatenate-all is to do cyclic shift $N-1$ times where N is the number of nodes. The three algorithms for which we measure performance are :

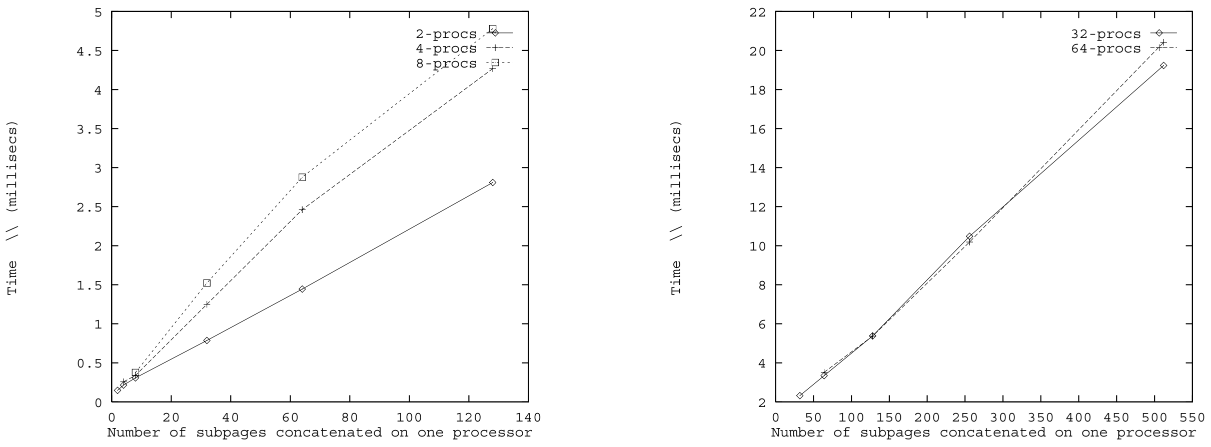


Figure 5: Concatenate on one processor, Algorithm 1

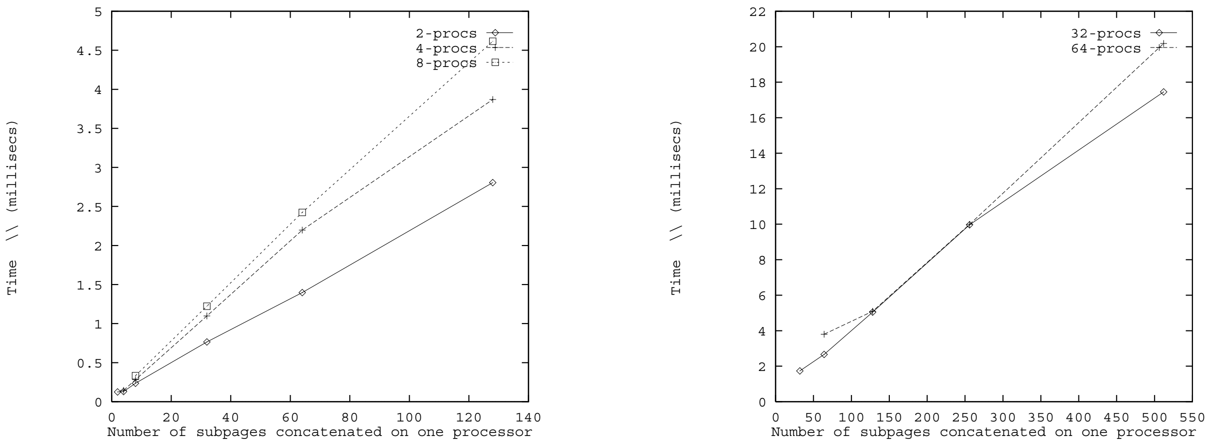


Figure 6: Concatenate on one processor, Algorithm 2

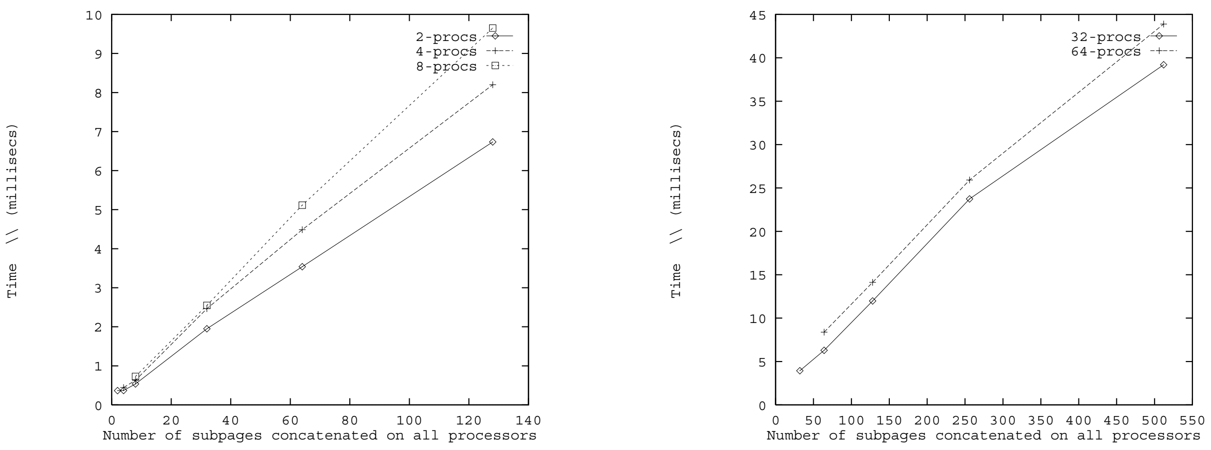


Figure 7: Concatenate on all processors, Algorithm 1

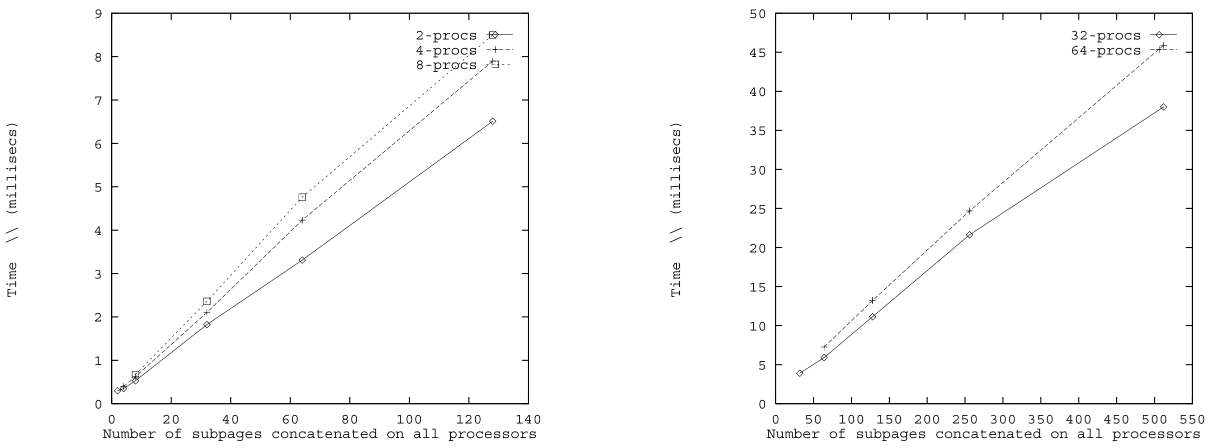
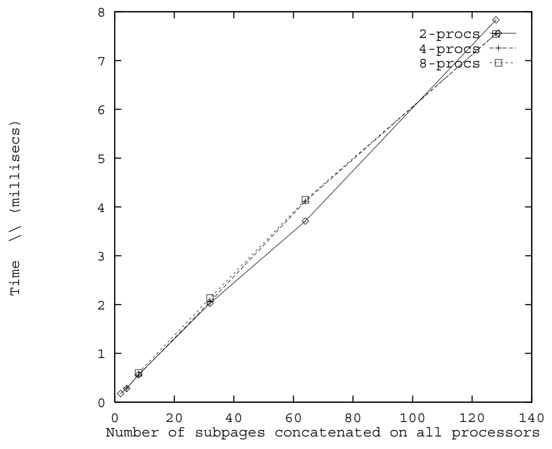


Figure 8: Concatenate on all processors, Algorithm 2



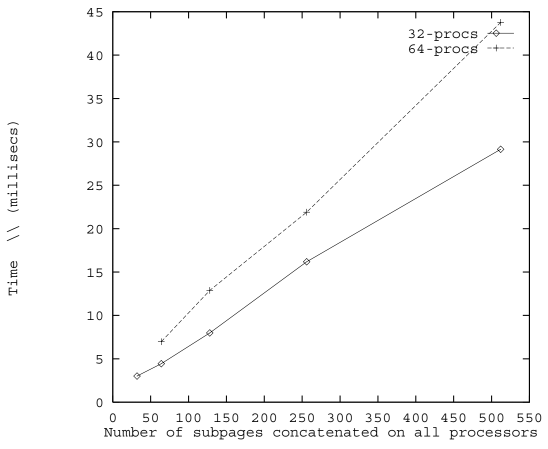


Figure 9: Concatenate on all processors, Algorithm 3

1. *Concatenate-One (algorithm 1) + Broadcast*
2. *Concatenate-One (algorithm 2) + Broadcast*
3. *$N-1$ Cyclic Shifts*

Algorithm 1 generates the most traffic on the ring as it involves one processor reading the value from the other processors and then broadcasting the final result to all the processors. Figure 7 shows us that the cost agrees with the figures for broadcast and the all-to-one concatenation that are used in this algorithm. The following equation is used to model the performance of this algorithm.

$$T_{concatall} = C_0 + C_1 * N_{procs} + C_2 * N_{subpages}$$

where $C_0 \approx 0, C_1 = 0.091$ and $C_2 = 0.064$. Figure 8 plots the performance of the second algorithm we use. In this case we observe a little better performance for higher number of processors as expected. The complexity analysis of this algorithm shown in the following equation shows us that the high data transfer rates on the ring offset any advantage that this algorithm has over the previous one. The time for the gather phase is $C_1 * log_2 N_{procs} + C_2 * N_{subpages}$ and the broadcast phase is $C_3 * N_{procs} + C_4 * N_{subpages}$ for a combined complexity of :

$$T_{concatall} = C_0 + C_5 * log_2 N_{procs} + C_6 * N_{procs} + C_7 * N_{subpages}$$

$C_0 \approx 0$, C_5 is found to be 0.087 and C_6 is 0.060 from our experiments. Algorithm 3 uses cyclic shift to perform the all to all broadcast. Figure 9 shows that this outperforms the previous two approaches. Mainly because we are doing away with the bottleneck of having one processor control the broadcast. All processors shift $N_{subpages}/N_{procs}$ data N_{procs} number of times. Hence the complexity of the algorithm is expressed as :

$T_{concatall} = C_0 + C_1 * N_{procs} + C_2 * N_{subpages}$

We find that $C_0 = 0.074, C_1 = 0.058$ and $C_2 = 0.022$.

The three algorithms are compared against each other in figure 10 for 64 processors These illustrate that Algorithm 3 does marginally better than the other two.

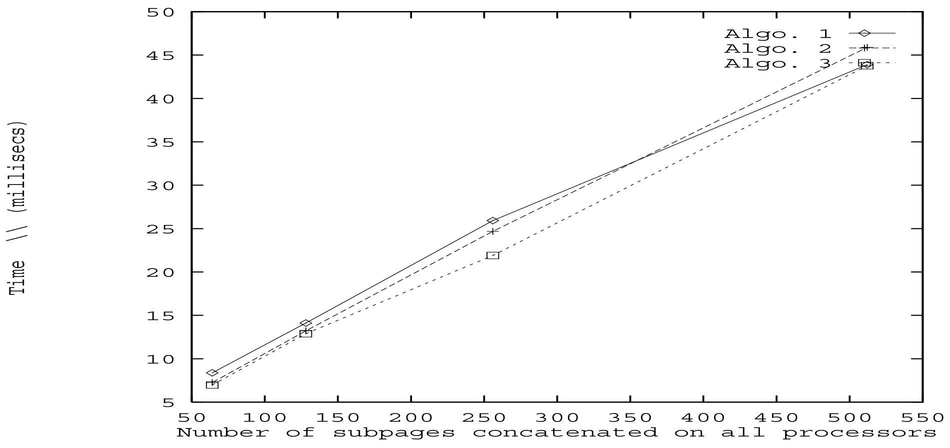


Figure 10: Concatenate on 64 processors, Algorithm 1,2,3

5.7 Reduce

A reduction operation starts with values in every node and ends with a single value in every node. Values may be added, so that the sum of all values is returned; or the largest or smallest value may be chosen. some frequently used reduction operations are: *add*, *multiply*, *max*, *and min*. On KSR-1, we achieved reduction operation in three steps:

1. concatenate-one gathers values from all the other nodes;

2. one node does the arithmetic operation;

3. broadcast the final result to all the other nodes.

Figure 11 illustrates the timing for performing the sum operations on different numbers of processors for the various sizes of arrays.

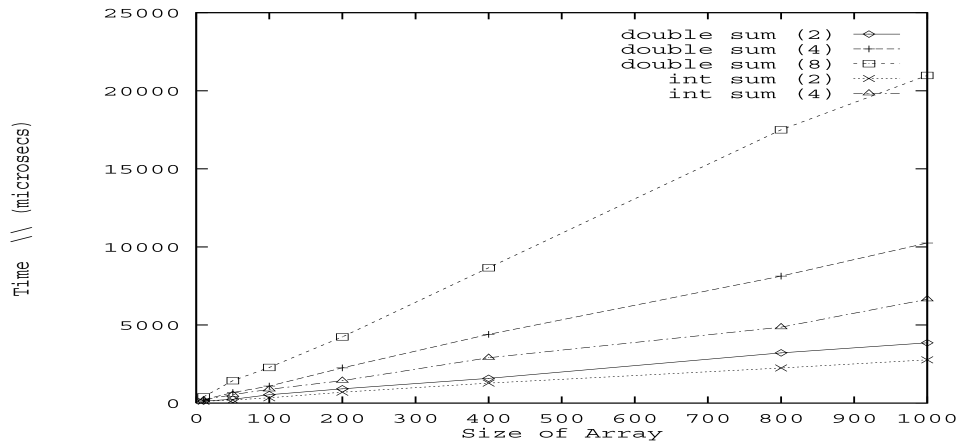


Figure 11: Reduction, (sum) Algorithm 1

	Number of Threads				
	1	2	3	4	5
Time	1.225	2.515	3.791	5.096	6.361

Table 4: Thread Creation time (CPU time in millisecs)

Threads	Computation			Communication	Comp. and Comm.		
	DS1	DS2	DS3	Comm	DS1 + Comm	DS2 + Comm	DS3 + Comm
1	27.81	96.65	183.20	30.04	34.64	98.91	186.96
2	27.93	96.70	183.30	29.90	34.80	98.87	187.04
3	27.96	96.75	183.31	30.00	34.88	98.79	187.02
4	27.94	96.92	183.48	29.81	34.73	99.48	187.67
5	28.04	96.79	183.42	30.19	34.45	98.94	187.43

Table 5: Total time taken by different number of threads considering only scheduling overhead (Total CPU time in milliseconds)

6 Computation and Communication Overlap using multiple threads

This section studies the effects of using multiple threads on computation and the overlap that can be achieved for communication. The additional overhead of thread creation and scheduling them must be taken into account when deciding to use more than one thread of control. Any performance gain might be offset by this overhead. We observe that on the KSR the thread creation and synchronization overheads are higher than thread scheduling overheads.

We describe the three experiments that were conducted to look into the above issues. Starting with a single thread of control we repeat the experiments with greater number of threads. The objective here is to calculate the cost of thread creation (given in Table 4) and study the communication overlap that can be achieved when multiple threads are used. The idea is that with multiple threads of control, each of which performs computation on off-processor data (generates communication), the communication requests are not sequentialized. Many threads can generate requests to fetch off-processor data simultaneously. Our experiments vary the amount of computation performed per thread which affects the ratio of computation to communication. In DS1 (Table 5) the ratio of computation to communication is 1:1. In DS2 it is 1:3 and in DS3 1:6. The total time taken is the sum of the time taken by individual threads.

In the first experiment we perform computation on an array by different number of threads. We start by allocating the entire computation to a single thread. Then two, three, four and five threads share the computation. We do not observe any performance degradation by using more threads. This involves local computation and we have masked any communication effects by caching the array locally on the processor beforehand.

We observe very little overhead for scheduling of threads as shown in the Table 5. The time reported for multiple threads is the sum of the time taken by the individual threads. The computation column illustrates three data sets with differing amount of computation performed by different number of threads. The communication column illustrates fetching off-processor data by different number of threads. The last column combines both computation and communication for the three data sets DS1, DS2 and DS3. We observe an overlap between communication and computation by comparing these timings to the sum of the previous two columns. We are only factoring in the scheduling overheads in this table.

In the second experiment we compare communication performed by different number of threads. Each thread reads off-processor data. First only one thread reads all the off-processor data, then two, three, four and five threads share fetching of off-processor data. Each thread now generates a communication request and gets scheduled out while its request is being serviced and the processor is waiting for data. Another thread gets scheduled and since this also needs off-processor data generating a communication request, it gets scheduled out. With more threads the need for scheduling threads arises more frequently to maximize processor utilization.

The third experiment combines the above two experiments. We would like to observe the overlap between generating a communication request for off-processor data and performing local computation. By varying the amount of local computation performed we wanted to study the effects of scheduling, and improved performance achieved by overlap between computation and communication. We have assumed no data dependencies for computation so that maximum overlap can be obtained. Since multiple threads now are fetching off-processor values simultaneously, there is a significant overlap between threads waiting for data which are scheduled out and the one which performs computation.

7 Conclusions

In this paper, we have described and evaluated the important communication primitives for solving a large fraction of data parallel applications on all-cache memory machines, KSR-1. We have characterized the latencies for communication and have modeled the global operations which can be used to design efficient algorithms for applications that use collective communication. We have studied the performance of threads for investigating the overlap of computation and communication.

References

[1] Rosti, E., Smirni, E., Wagner, T., Apon, A., and Dowdy, L., *The KSR1: Experimentation and Modeling of Poststore*, Proc. of the 1993 ACM Sigmetrics Conf. on Measur. and Modelling of Comp. Sys., Santa Clara, California, May 1993, pp 74-85.

[2] Dunigan, T.H., *Kendall Square Multiprocessor: Early Experience and Performance*, Oak Ridge National Laboratory Tech. Report No. ORNL/TM-12065, April 1992.

[3] Singh, J.P., Truman, J., Hennessy, J., and Gupta, A., *An Empirical Comparison of the Square Research KSR-1 and Stanford DASH Multiprocessors*, SuperComputing'93, November 1993.

[4] Ramachandran, U., Shah, G., Ravikumar, S., and Muthukumarasamy, J., *Scalabilty Study of the KSR-1*, 22nd Int. Conf. on Parallel Processing, St. Charles, August 1993.

[5] Ghosh, K., Mukherjee, B., Schwan, K., *Experimentation with Configurable, Lightweight Threads on a KSR Multiprocessor*, Georgia Institute Of Technology, Report No. GIT-CC-93/37, June 1993.

[6] Ranka, S., Wang J. C., *All-to-many communication avoiding node contention* Technical Report, Syracuse University, 1992.

[7] Bolosky, William J., Scott, Michael L., *False Sharing and its effect on Shared Memory Performance*, Technical Report, University of Rochester, 1992.

[8] LeBlanc, Thomas J., Markatos, Evangelos P., *Shared vs. Message Passing in Shared Memory Multiprocessors*, Technical Report, University of Rochester, 1992.

[9] Kendall Square Research Corporation, *KSR Parallel Programming*, 1991.