Performance Evaluation of A Seismic Data Analysis Kernel on The KSR Multiprocessors[1]

Weiming Gu

GIT–CC–94–43

September 25, 1994

Abstract

The paper investigates the effective performance attainable for a specific class of application programs on shared memory supercomputers. Specifically, we are to investigate how seismic data analysis applications behave on the Kendall Square Research Inc.'s KSR multiprocessors. The computational kernel of seismic computation algorithms is parallelized and its performance is analyzed. Three approaches for parallelizing the g5 kernel are analyzed: column-based, row-based, and grid-based parallelizations. All three approaches result in well balanced decompositions, but differ significantly in data locality. In general, the column-based approach has the best data locality, while the small grid-based approach has the worst. These results clearly indicate that data locality is one of the critical factors for attaining high performance for the g5 kernel. The best parallelized g5 kernel code achieves about 44% of both the KSR-1 and KSR-2 machines' peak computational performance.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

1 Introduction

Current trends indicate that massively parallel processors (MPPs) will emerge as the mainstream supercomputers, with current machines including Intel's Paragon, IBM's SP-2, Cray's T3D, Thinking Machine Inc.'s CM5, Kendall Square Research's KSR-2, etc. Manufacturers of these machines have posted impressive performance numbers, some of which even exceed those of much more expensive vector supercomputers. However, these numbers often capture peak machine performance, and can not be achieved by most application programs. In addition, an application may perform very differently on different MPPs depending on their memory architectures and processor/memory interconnects [BBDS92].

The topic of this work is the investigation of the effective performance attainable for a specific class of application programs on shared memory supercomputers. Specifically, we are to investigate how seismic data analysis applications behave on the Kendall Square Research Inc.'s KSR multiprocessors. In addition, we attempt to understand how easy or how difficult it is to parallelize these applications on shared memory machines. Toward this end, the computational kernel of seismic computation algorithms is parallelized and its performance is analyzed. The so called *g5 kernel* accounts for as much as 95% of total execution time of typical seismic analysis programs. As a result, the parallelizability and resulting performance of the g5 kernel is a good predicator of the class of applications' performance. In fact, the g5 kernel is constantly used by engineers at Schlumberger, one of the world's largest oil service companies, to evaluate new supercomputing machines.

The machine on which the g5 kernel code is parallelized and studied is a single ring 32-node KSR-1 machine and a dual-ring 64-node KSR-2 machine available at the Georgia Institute of Technology. The KSR machines have an ALLCACHE memory architecture, in which all memory units on all processors are treated as caches at different levels. The machine is time shared, but processors can be dedicated to the execution of a specific application.

The rest of this paper is organized into four sections. In the next section, an overview of the KSR system is given. In Section 3, the g5 kernel and its parallelization are discussed. In Section 4, the performance results of the g5 kernel and its data input/output on the KSR machines are presented and evaluated. Conclusions are presented last.

[1]This work was performed when the author was a summer intern at the Schlumberger Laboratory for Computer Science at Austin, Texas, during the Summer of 1993.

2 Overview of the KSR Multiprocessors

KSR architecture. The KSR multiprocessors (both KSR-1 and KSR-2) are NUMA (non-uniform memory access) shared memory cache-only architectures with an interconnection network that consists of hierarchically interconnected rings [Ken93c]. Each first level ring can support up to 32 nodes, and up to 32 first level rings can be connected by a second level ring. Each node consists of four major components: (1) a 64-bit RISC based processor, with a clock rate of 20 MHz for the KSR-1 and 40 MHz for the KSR-2 machines respectively, (2) 32 MB of main memory, (3) a higher performance 512 KB sub-cache, half of which is used for caching instructions and the other half for caching data, and (4) a ring interface. Instruction execution is pipelined at each processor, and up to one integer or floating point operation instruction can be issued during each clock cycle, resulting in a peak computational performance of 20 million integer or floating point operations per second for each KSR-1 processor and 40 for each KSR-2 processor. There is parallelization inside each processor at the instruction level; control and memory access instructions execute concurrently with the computational instructions on different functional units. Hence the overall peak performance is 40 MIPS (million instructions per second) for each KSR-1 processor and 80 MIPS for each KSR-2 processor.

The KSR platform, on which results presented in this paper are obtained, is a 32-node KSR-1 machine. All 32 nodes are linked by a single ring. Additional results are obtained on a later version of the machine, a 64-node KSR-2 with 2 rings, each connecting 32 processors.

KSR's ALLCACHE memory model. The KSR's memory architecture, called *ALLCACHE* [Ken93c], is a three-level hierarchy. At the top is each node's high performance cache, called *sub-cache* in ALLCACHE's terminology. At the second level is each node's main memory, termed *local cache*. The lowest level is the disk storage providing conventional virtual memory. ALLCACHE implements a *sequentially consistent* shared memory model. In this memory model, data moves to the point of reference on demand. The unit of data movement is always a subpage of 128 bytes. Specifically, if a datum (e.g., a character, an integer, a floating point number, etc.) is requested by a processor and found in neither the processor's subcache nor its local cache, a request for the subpage that contains the datum is sent to other processors on the same ring as the requesting

processor. If none of the processors on the same ring have the requested subpage, the request message is propagated to other rings. When the subpage is finally brought in to the requesting node, it is replicated in its local cache. A write request to a subpage goes through a similar process, with additional complexity that a write request invalidates all replicated copies of that subpage on other nodes' local caches and subcaches. Therefore, the cost of accessing a shared variable depends on where it is located. In general, a subcache memory access only takes 2 clock cycles, while it takes 10 clock cycles to access a word in a processor's own local cache. Accessing a subpage in other processor's memory takes 175 clock cycles if it is on the same ring as the requesting processor, and about 600 clock cycles if on other rings.

It is evident from the discussion above that excessive remote memory accesses (i.e., reading or writing data items located in other processors' local caches) can slow down a parallel program's execution. Potential 'subpage thrashing' can exacerbate performance problems: if a shared variable (e.g., a global counter) is constantly accessed by different processors and many of shared accesses are write requests, then the subpage that contains the variable will thrash among the accessing processors' local caches. The KSR architecture offers two special instructions, a `prefetch` and a `poststore`, to deal with some such remote access problems. A prefetch instruction can be issued to a subpage before it is accessed in later execution, avoiding potential processor stalling due to remote memory access. A program that updates a variable can issue a poststore instruction to ask the memory system to broadcast the new value to other processors who may need it. Compiler support is needed to fully realize the benefits of these two special instructions.

While the KSR machine's shared memory architecture is designed for ease of use by application programmers, programmers still need to carefully distribute both computation and data among participating processors in order to achieve high performance on this machine. The experiences and results from parallelizing the g5 kernel (presented in the next two sections) demonstrate not only that such careful parallelization is necessary for high performance, but also that it is achievable.

Parallel programming support on the KSR machines. Parallel programming on the KSR machines is supported at the low level by the pthreads library and at the high level by three types of parallel constructs [Ken93b, Ken93a]: parallel regions, parallel sections, and tile families. The pthreads library, supported at the KSR operating system's kernel level, is the basic lowest-

level parallel construct, upon which higher level parallel constructs are eventually implemented. Parallelism on the KSR machines is achieved by concurrently running multiple pthreads on different processors. A pthread can serve as a *logical processor* for higher level parallel constructs. A *parallel region* is a segment of code in a program that executes concurrently on several logical processors. *Parallel sections* in a program are code segments that execute in parallel with each other on different logical processors. Tiles are directives to parallelize loops; a loop is divided into multiple segments, each is called a *tile* and is assigned to a logical processor. The iterative nature of the g5 kernel makes tiles the most suitable parallel construct for parallelization of the code. Following is a simple example of parallelizing a loop using a tile directive:

```
C*KSR*USER TILE(j:1,256, TILESIZE=(32), PRIVATE=(i))
      DO j = 1, 256
          DO i = 1, 256
              C(i,j) = A(i,j) + B(i,j)
          END DO
      END DO
C*KSR*END TILE
```

The directive at the first line of the sample code informs the compiler to divide the outer loop into 8 chunks, each containing 32 loops (specified by `TILESIZE`). The directive also specifies that each processor consider the inner loop variable i a private variable. By default, variables are shared if not otherwise specified. If the inner variable i is *not* specified as a private variable, sharing it among processors will produce incorrect computations.

In addition to these parallel constructs, a user level threads package, called *Cthreads* library (developed at Georgia Institute of Technology [Muk91]), is also available for programming on the KSR machines. User level threads avoid the overheads of kernel calls experienced by pthreads programs. A program and performance monitoring system, called *Falcon*[GEK+95], also developed at Georgia Tech, assists programmers in the process of performance and correctness debugging of their Cthreads programs.

3 The g5 Kernel and Its Parallelization on KSR

Seismic data analysis is an important step in oil exploration. Data is first acquired from surface survey. It is then processed and interpreted in order to understand the subsurface structure of a region, and to determine whether there are oil deposits in the surveyed area [Hal90, HP91]. Such seismic data processing typically takes months on the fastest computing machines available today, and it involves terabytes of data. The majority of computational cost in seismic data processing is captured by the g5 kernel, which is described next.

3.1 The g5 Kernel

The g5 kernel code is the computational core of many numerical analysis algorithms, such as convocation and finite element methods, commonly used in seismic analysis applications and in many other scientific computation programs. Typically, such algorithms operate on a two or three dimensional computation space, represented by a two or three dimensional array of discrete points. Given some initial values of these points, the algorithm iteratively computes new values for each point from its previous values and from the values of neighboring points. The algorithm terminates when either convergence or divergence is detected. Divergence indicates that the algorithm fails on the problem with the given initial values.

The g5 kernel's computation space is a two dimensional array of points layered on a plane. Figure 1 illustrates how the center point (at row i and column j) is calculated from itself and its neighbors. All of the shaded neighboring points take part in computing the new value of the central point during each iteration. Constants c_1, c_2, c_3, c_4, and c_5 are weights associated with the corresponding points in the calculation of the new value of the central point. The actual values of these constants are defined in the context of specific applications, and typically not disclosed by companies providing seismic data processing services.

The specific computation used by the g5 kernel is described by the following formula (Equation 1),

$$N_{i,j} = c_1 * (P_{i-2,j-2} + P_{i-2,j+2} + P_{i+2,j-2} + P_{i+2,j+2}) +$$

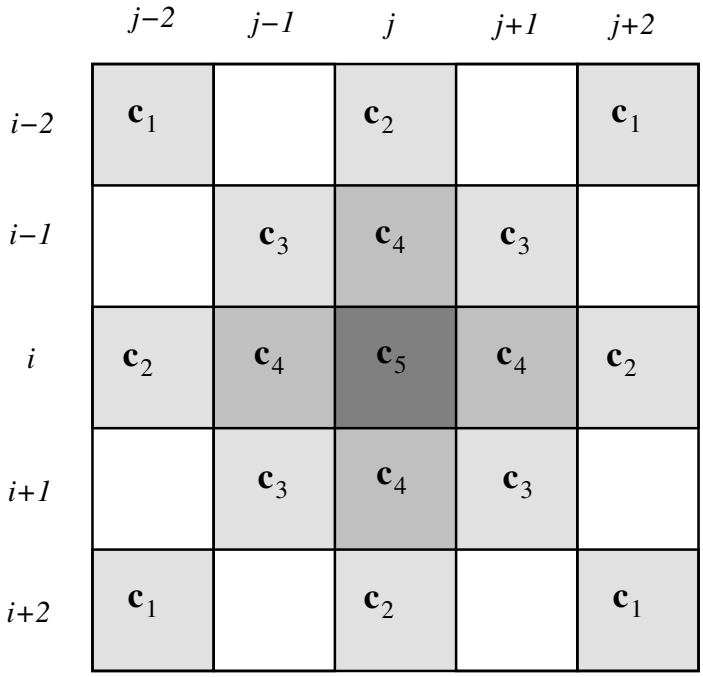


Figure 1: The central point at row i and column j is computed from the previous values of itself and its neighbors (shaded). Constants c_1, c_2, c_3, c_4, and c_5 are weights in the calculation.

$$
\begin{aligned}
& c_2 * (P_{i,j-2} + P_{i,j+2} + P_{i-2,j} + P_{i+2,j}) + \\
& c_3 * (P_{i-1,j-1} + P_{i-1,j+1} + P_{i+1,j-1} + P_{i+1,j+1}) + \\
& c_4 * (P_{i,j-1} + P_{i,j+1} + P_{i+1,j} + P_{i+1,j}) + \\
& c_5 * P_{i,j} \qquad (1)
\end{aligned}
$$

where $P_{i,j}$ is the old value of the point at row i and column j from the previous iteration, and $N_{i,j}$ holds the new value to be calculated. For the boundary points, the values of their neighbors outside the plane boundary are always assumed to be 0. All coefficients are real numbers and all point values are complex numbers (represented by two floating point words). From the formula, the calculation for each point per iteration requires 21 operations of complex and real number additions or multiplications, resulting in 42 actual floating point operations. For a problem size of an $1,024 \times 1,024$ plane, the total computation required for each iteration of the g5 kernel code is 42 million floating point operations.

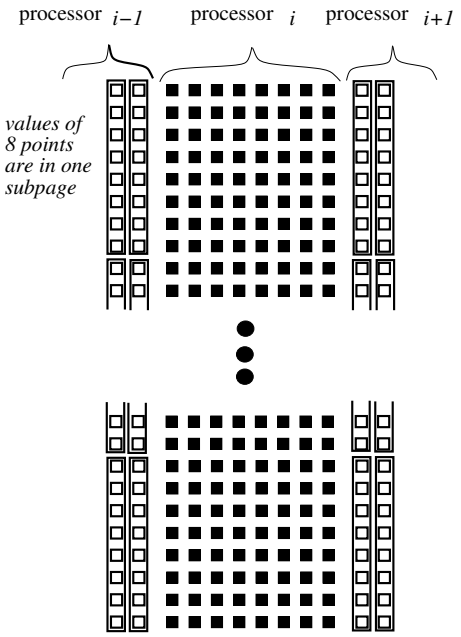
A straightforward implementation of the g5 kernel algorithm requires two arrays, one containing

the old values from the previous iteration ('previous value array'), the other storing new values calculated from the current iteration ('new value array'). The roles of these arrays are interchanged between iterations. Namely, the 'new value array' becomes the 'previous value array' in the next iteration, and vice versa. To make the calculation of the boundary points the same as inside points, we extend the arrays by two rows and two columns outside each plane boundary, and set these extended array elements to 0. Thus there is no need for special treatment for boundary points during the computation.

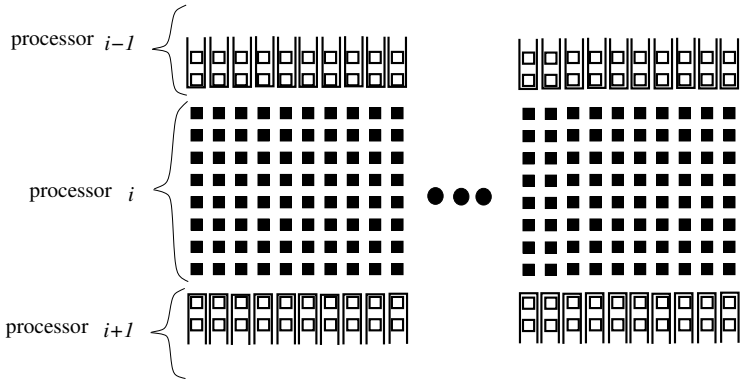
3.2 Parallelization of the g5 Kernel

The attainment of high performance on parallel machines require that programs exhibit balanced workloads and high data locality on all participating processors. Balanced workloads are not difficult to achieve for the g5 kernel code because the required computation for each point is always the same. The following three decompositions of the kernel's computation space result in balanced work load: (1) *row-based* parallelization divides the plane into blocks of rows and assigns each row block to one processor, (2) *column-based* parallelization divides the problem space into blocks of columns and assigns each column block to one processor, and (3) *grid-based* parallelization divides the plane into grids and distributes the grids evenly among processors.

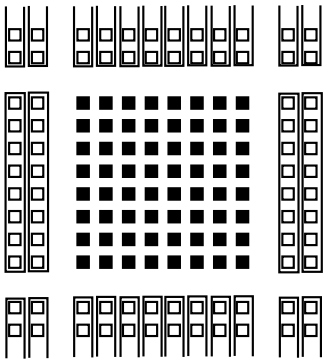
However, data locality on the KSR machines, and therefore, performance varies significantly among these three methods due to the machine's 'move-on-demand' shared memory policy. Specifically, the g5 implementation uses two arrays to store the old values from the previous iteration ('previous value array') and new values from the current iteration ('new value array'). After all points on the plane are computed, the distribution of the 'new value array' is exactly the same as the computational space distribution, because all replicated copies of array elements have been invalidated by storing new values into the array. During the next iteration of computation, however, the 'new value array' becomes the 'previous value array'. Therefore, some elements of the 'previous value array' must be accessed remotely. Figure 2 illustrates the distribution of the computational space and 'previous value array', and lists the required remote subpage accesses for each parallelization method for a sample problem of size 128×128.



(a) Column−based parallelization of an 128 x 128 plane. Each column block has 8 columns. Each processor needs to access 64 remote subpages per iteration.



(b) Row−based parallelization of an 128 x 128 plane. Each row block contains 8 rows. Each processor needs to access 256 remote subpages per iteration.



(c) Grid−based parallelization. Grid size is 8 x 8 points. For each grid, the responsible processor needs to access 28 remote subpages.

Figure 2: Distribution of computation and required remote subpage accesses for a sample problem of size 128×128. Narrow vertical boxes surrounding every 8 points indicate that values of these points are stored in one 128-byte subpage.

In Figure 2, the small black boxes represent points assigned to the same processor, while the empty boxes represent points on other processors. Since Fortran uses column major array storage, values of every 8 points in the same column are stored in a subpage (16 bytes is needed to store each point). Each processor needs to access some 'previous value array' elements located in other processors' local caches. As in the example shown in the figure, with problem size 128×128 computed on 16 processors, the column-based parallelization method requires each processor to access 64 remote subpages per iteration, while the row-based and grid-based (each processor is responsible for 4 grids of size 8×8) methods require each processor to access 256 and 448 subpages respectively, both of which are significantly higher than with the column-based approach.

If we increase the problem size to a 256×256 plane (still running on 16 processors), the numbers of remote subpage accesses per iteration for the column-based, row-based, and 8×8 grid-based parallelization methods increase to 128, 512, and 1,792 respectively. And if the problem size is further increased to $1,024 \times 1,024$, the remote subpage accesses per iteration increase to 512, 2,048, 28,672 subpages for the three parallelization methods respectively. Note that the number of remote subpage accesses for the column-based parallelization approach is determined by the number of rows in the computation space, while the number of the row-based approach is only affected by the number of columns in the computation space. The grid-based approach is affected by both factors, however. But if the grid size increases as the problem size grows, the total number of remote subpage accesses drops drastically. For example, if the grid size increases to 256×256 for problem size $1,024 \times 1,024$, each processor only needs to access 648 remote subpages, which is quite close to the column-based parallelization approach.

The access pattern of the 'new value array' is similar to that of the 'previous value array'. After each iteration, those elements of the 'previous value array' accessed by multiple processors are replicated on these processors' local caches. During the next iteration, the 'previous value array' becomes the 'new value array'. Writing new values into the 'new value array' has to invalidate those replicated subpages, a reversal of the process describe above. The column-based approach invalidates the least amount of subpages, while the small grid-based approach invalidates the most.

In summary, the three parallelization techniques, column-based, row-based, and grid-based decompositions, produce balanced work loads for the g5 kernel. However, column-based parallelization

results in the best data locality, whereas small grid-based parallelization behaves the worst for small size grids. Performance results concerning g5 parallelization appear in the next section.

4 Evaluation of the g5 Kernel on KSR

In this section, the performance of the parallelized g5 kernel is evaluated by comparison of results obtained with the column-, row-, and grid-based methods of parallelization. The KSR-1 and KSR-2 machines are used in the evaluations. Last, the I/O throughput and performance of the g5 kernel are characterized and evaluated. This is important because typical seismic analysis applications involve extraordinarily large amounts of data, so that the speed of I/O can determine the overall program performance.

Computational performance of the g5 kernel on the KSR-1 machine. Table 1 and Figure 3 depict the average execution time of each iteration of the g5 kernel code (parallelized with the

No. of Processors	1	2	4	8	16	24
$1,024 \times 1,024$	5.03	2.31	1.14	0.552	0.280	0.191
512×512	1.10	0.552	0.279	0.142	0.0736	0.0524
256×256	0.276	0.141	0.0723	0.0380	0.0206	0.0152

Table 1: Average execution time (in seconds) of each iteration of different problem sizes and on different numbers of processors. The g5 kernel is parallelized with the column-based approach.

column-based parallelization technique) on a 32-node KSR-1. Execution times are presented for three different problem sizes, $1,024 \times 1,024$, 512×512, and 256×256. In the case of a problem size of an $1,024 \times 1,024$ plane computing on 24 processors, the average computation time of each iteration is less than 0.2 seconds. Therefore, the actual computational throughput of the application is at least 210 millions operations per second. Since the peak computational performance of 24 KSR-1 processors is 480 MFLOPS, the g5 kernel parallelized with the column-based technique realizes 44% of the machine's peak performance.

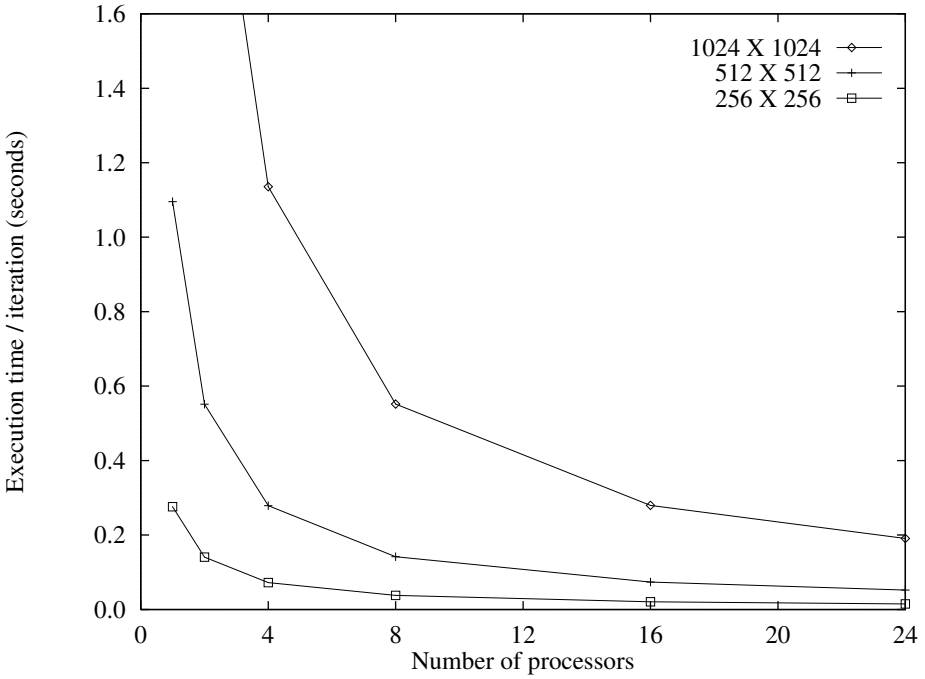


Figure 3: Execution time of each iteration of the g5 kernel with the column-based parallelization technique.

From the results in Table 1 and Figure 3, it is clear that the g5 kernel scales well both in problem size and in number of processors. Furthermore, as expected, speedup improves when problem sizes are increased. For example, the total amount of computation of problem size $1,024 \times 1,024$ is 16 times larger than that required for problem size 256×256. However, the execution time of each iteration of the former is only 12 times that of the smaller problem size when running on 24 processors. The speedup results presented next demonstrate that the g5 kernel on the KSR-1 also scales well in machine size.

Speedups. Figure 4 depicts the speedups when running the g5 kernel with problem sizes $1,024 \times 1,024$, 512×512, and 256×256, respectively, with different numbers of processors. Super-linear speedups in the results are due to differences in the speed of subcaches, local caches, and remote memory accesses on the machine. In the case of the g5 kernel of problem size $1,024 \times 1,024$, each of the 'previous value array' and the 'new value array' requires 16 MB of storage. Therefore, when the g5 kernel code runs on a single processor, it cannot fit its code, stack, and local data into the processor's 32 MB local cache. As a result, additional time is spent on moving pages and

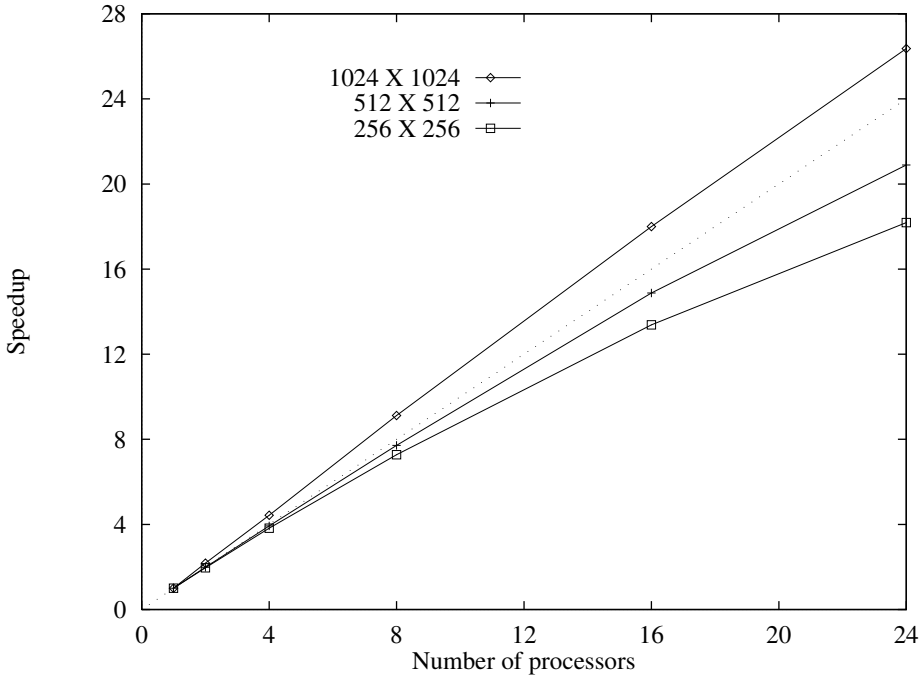


Figure 4: Speedups of the g5 kernel on the KSR-1.

subpages in and out of the processor's local cache. On the other hand, if the same program and problem size run on a larger number of processors, all code, stack, and local data can be kept in the processors' local caches, thereby avoiding or reducing the extra costs of page and subpage movement. This hypothesis is confirmed by examination of the total number page misses, subpage misses, and subcache misses obtained from the KSR's hardware monitor[2] (shown in Table 2).

No. of Processors	1	2	4	8	16	24
data subcache misses	593,000	295,000	148,000	76,000	40,000	28,000
subpage misses	93,000	11,000	1,000	423	164	149
page misses	1,100	147	7	3	1	0

Table 2: Average number of page misses, subpage misses, and data sub-cache misses on each processor of the KSR-1.

Table 2 shows the average number of page misses, subpage misses, and data subcache misses per iteration for each processor when running the g5 kernel code (with column-based parallelization)

[2] The KSR's hardware monitor only monitors one half of each processor's subcache, which is used to cache data; it does not monitor the other half that is used to cache instructions. The hardware monitor also does not monitor the total numbers of page hits, subpage hits, or subcache hits. Hence hit ratios of pages, subpages, or subcaches are not available.

of problem size $1,024 \times 1,024$. Note that the numbers of subpage misses and page misses of local caches drop very quickly as the number of processors increases.

From the results shown in Figure 4, it is evident that the g5 kernel code is scalable in terms of problem size as well as in terms of machine size. Larger problem sizes result in increased speedups compared to smaller problem sizes, and if the problem size is suitably large, the speedup increases almost linearly as the number of processors increases.

Comparing the performance of different parallelization techniques. Section 3.2 describes several alternative parallelization techniques. Figure 5 depicts measurements of the g5 kernel

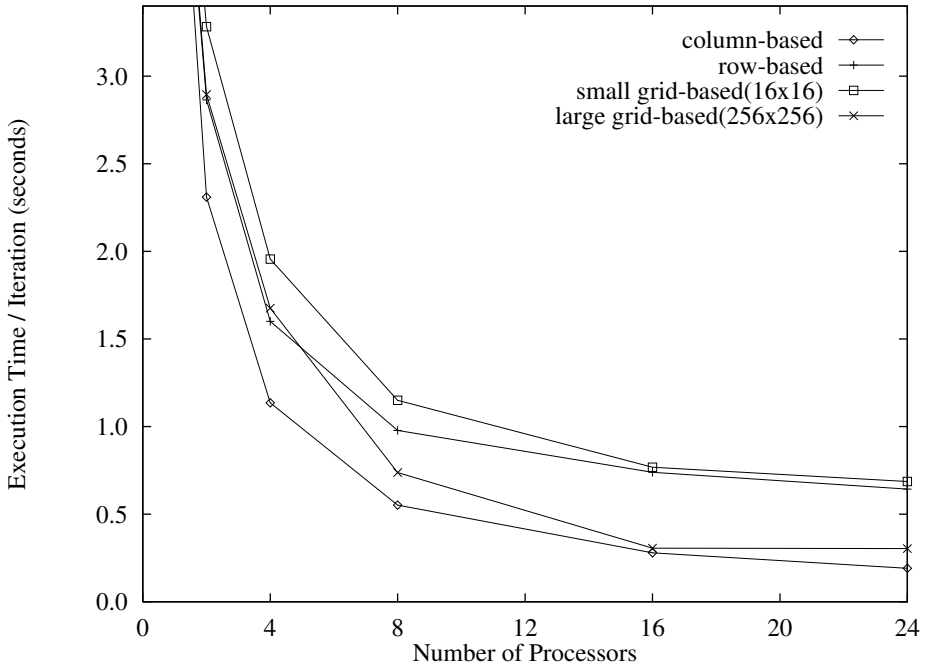


Figure 5: Execution times of the g5 kernel using different parallelization techniques on the KSR-1. The problem size is an $1,024 \times 1,024$ plane in all cases.

parallelized with these parallelization techniques: row-based, column-based, small grid-based (grid size 16×16), and large grid-based (grid size 256×256). The problem size for all the measurements is the same $1,024 \times 1,024$ plane. Corresponding speedups are presented in Figure 6.

From the results shown in these two figures, it is clear that column-based parallelization produces the best performance on any number of processors, while the small grid-based parallelization

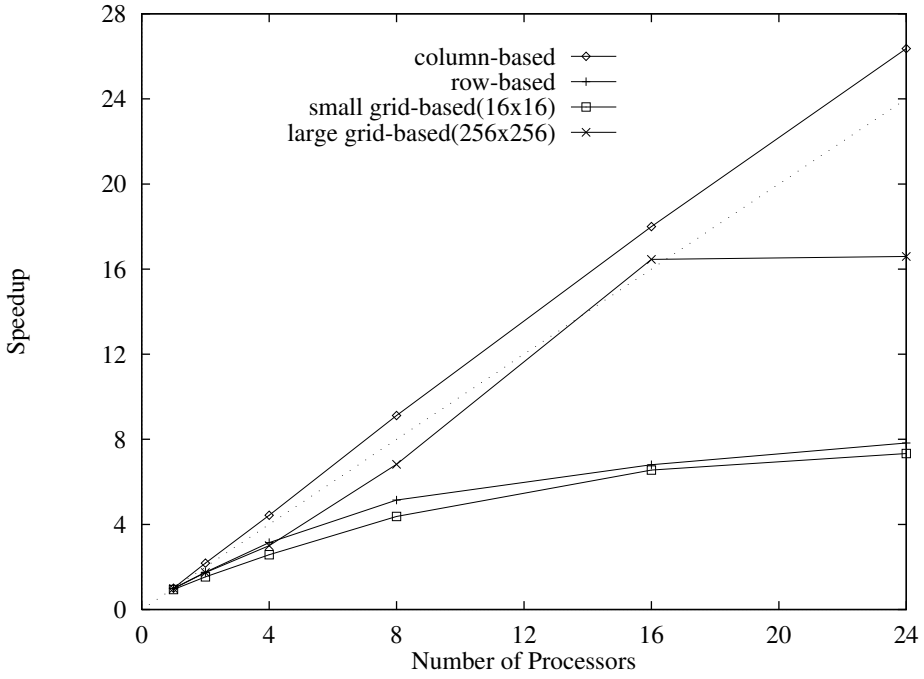


Figure 6: Speedups of the g5 kernel for different parallelization techniques on the KSR-1.

technique using small grids yields the worst performance. Row-based parallelization is generally close to the small grid-based approach. However, the large grid-based approach approximates the performance of the column-based approach, except on 24 processors. This anomaly is due to the fact that there are only 16 grids of size 256×256 for a problem size of $1,024 \times 1,024$, and therefore at most 16 processors are needed.

We can conclude from these results that data locality determines the overall performance of the g5 kernel, especially on large numbers of processor. The column-based parallelization approach is superior in performance because it best preserves data locality. Performance differences due to data locality are significant. For example, when the g5 kernel runs on 24 processors, the best case speedup (column-base parallelization) is more than 3 times as much as the worst case (small grid-based parallelization).

Performance of the g5 kernel on the KSR-2. The performance of the g5 kernel on the 64-node KSR-2 is shown in Figure 7 and 8. Figure 7 shows the average execution time of each iteration of the g5 kernel code with different problem sizes on different numbers of processors. Figure

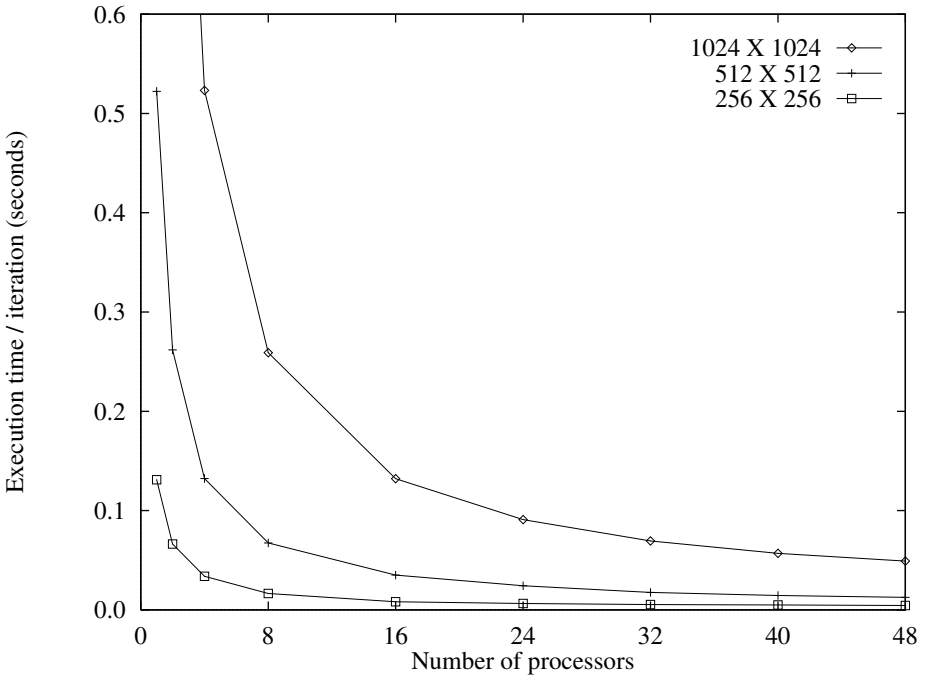


Figure 7: Average execution time of each iteration of the g5 kernel on a 64-node KSR-2. Column-based parallelization is used.

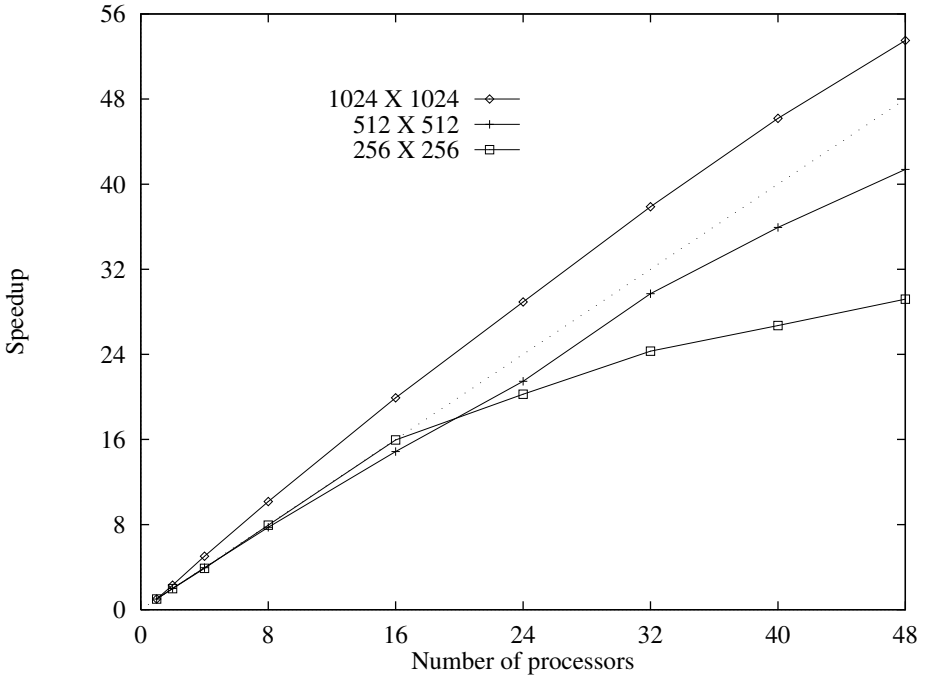


Figure 8: Speedup of the g5 kernel on the KSR-2.

8 shows the corresponding speedups on the KSR-2. By comparing the results on the KSR-2 (Figure 7) with the results on the KSR-1 (Figure 3), it is clear that the execution speed of the g5 kernel on KSR-2 is about twice as fast as on KSR-1, resulting from faster processor and interconnection network speed on the KSR-2. Column-based parallelization is again superior in performance to other techniques. Specifically, the execution time for problem size $1,024 \times 1,024$ on 48 processors is less than 0.05 seconds, resulting in total computational throughput of 840 MFLOPS. While the peak computational performance of 48 KSR-2 processors is 1,920 MFLOPS, the g5 kernel can still realize 44% of KSR-2's peak performance.

From the speedup results in Figure 8, it is clear that the g5 kernel scales very well except for small problem sizes. It is also clear that running on two rings (the 64-node KSR-2 is interconnected by 2 rings) does not have any observable effects on the overall performance of the g5 kernel, thereby indicating good scalability for even larger machine sizes.

Input/Output performance on the KSR machines. Since typical seismic data processing applications involve large amounts of data, the I/O performance of the machine on which the applications are running can affect overall program performance. Only application-level I/O performance is measured in this section. The KSR machine has I/O controllers attached to only some nodes, which implies that I/O performance may be different from node to node. Moreover, despite the availability of independent disks and I/O controllers, parallel I/O is not supported by the KSR machine's OS. As a result, the performance measurements presented here are attained from a simple program that reads from a file and writes to another. This program is run on different processors to obtain realistic averages. Table 3 shows the best and average times and the throughput of reading and writing files of size 8 MB on the KSR-2 installed at Georgia Tech.

From the results, it is clear that the I/O performance of the current KSR configuration is not sufficient for applications that require the input and output of large amounts of data (e.g., seismic data analysis applications). Consider the following example: a program reads initial data from a file, uses the g5 kernel to process them, and then writes the results into another file. The problem size is an $1,024 \times 1,024$ plane, thus the file size is 16 MB(each point is a complex value, which requires 16 bytes to store it). At the best I/O rate, reading the initial values from a file takes about 3.9 seconds and writing the results into another file takes 5.9 seconds. If the computation requires

	best time (seconds)	best throughput (MB/s)	average time (seconds)	average throughput (MB/s)
write file	2.94	2.72	5.67	1.41
read file	1.97	4.06	7.19	1.11

Table 3: I/O performance of the KSR-2: Best and average times of reading and writing an 8 MB file.

100 iterations and the program runs on 48 processors, the total computation time is only 5 seconds. Due to slow I/O, total execution time is lengthened to 14.8 seconds. If reading and writing files at the average rate, the total time is 30.8 seconds. I/O time becomes the dominant factor in this worst case scenario. However, there exist ways to reduce the effects of I/O or to improve I/O performance. For example, initial data and final results can be compressed to reduce the total amounts of I/O, at some cost of extra computation. Support of parallel I/O (i.e. reading and writing at several I/O controllers for a single program) can also improve the KSR's I/O performance.

5 Conclusions

The g5 kernel is a computational kernel extracted from seismic data processing applications. In this paper, the kernel is parallelized and its performance is evaluated on the KSR-1 and KSR-2 multiprocessors. Three approaches for parallelizing the g5 kernel are analyzed: column-based, row-based, and grid-based parallelizations. All three approaches result in well balanced decompositions, but differ significantly in data locality. In general, the column-based approach has the best data locality, while the small grid-based approach has the worst.

These results clearly indicate that data locality is one of the critical factors for attaining high performance for the g5 kernel. The best parallelized g5 kernel code achieves about 44% of both the KSR-1 and KSR-2 machines' peak computational performance. Similar conclusions can be drawn for the g5 kernel on any NUMA shared memory machines, include SGI's multiprocessor workstations.

The I/O performance of the KSR is not adequate for applications that input and output large amounts of data, as typical seismic analysis applications do. Improvements of the performance of the I/O system, such as supporting parallel I/O, are required to make these applications run efficiently on the KSR multiprocessors.

Acknowledgement

This work was performed when I was a summer intern at Schlumberger Laboratory for Computer Science, Austin, Texas, in 1993. I would like to thank Dr. Peter Highnam at Schlumberger Laboratory for Computer Science for sponsoring this project and for many extremely helpful discussions. I would also like to thank Dr. Karsten Schwan at College of Computing, Georgia Tech for his assistance to writing this paper.

References

[BBDS92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. Nas parallel benchmark results. In *Proceedings of Supercomputing'92*, pages 386–393, Minneapolis, Minnesota, November 1992. IEEE Computer Society Press.

[GEK+95] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of FRONTIERS'95*, February 1995. To appear. Also available as Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology.

[Hal90] D. Hale. Stable explicit depth extrapolation of seismic wavefields. In *60th Annual International Meeting and Exposition of the Society of Exploration Geophysicists*, pages 1301–1304, 1990.

[HP91] P. T. Highnam and A. Pieprzak. Implementation of a fast, accurate 3-d migration on a massively parallel computer. In *61st Annual International Meeting and Exposition of the Society of Exploration Geophysicists*, pages 338–340, 1991.

[Ken93a] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154-1379. *KSR FORTRAN Programming*, December 1993.

[Ken93b] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154-1379. *KSR Parallel Programming*, December 1993.

[Ken93c] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154-1379. *KSR Principles of Operations*, December 1993.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

Contents