Preconditioned Iterative Methods on Virtual Shared Memory Machines

by

Harriet Roberts

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Calvin J. Ribbens

Advisor: Dr. Calvin Ribbens

D C Allison

Dr. Donald Allison

Layne Watson

Dr. Layne Watson

December, 1994

Blacksburg, Virginia

c.2

Preconditioned Iterative Methods on Virtual Shared Memory Machines

by

Harriet Roberts

Project Advisor: Dr. Calvin Ribbens

Computer Science

(ABSTRACT)

The Kendall Square Research Machine 1 (KSR1) is a virtual shared memory (VSM) machine. Memory on the KSR1 consists primarily of shared, physically distributed caches. Effective memory utilization of the KSR1 is studied within this thesis. Special emphasis is laid upon how best to optimize iterative Krylov subspace methods using domain decomposition preconditioning. The domain decomposition preconditioner used was developed by J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The Krylov subspace method used was the conjugate gradient algorithm. The linear systems being solved are derived from finite difference discretization of elliptic boundary value problems. Most of the focus of this thesis is upon how data structures affect performance of the algorithm on the KSR1. Implications for other iterative methods and preconditioners are also drawn.

ACKNOWLEDGEMENTS

I would like to thank:

- Dr. Ribbens for his patience and guidance.
- Oak Ridge National Laboratory for the use of their KSR1.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 The Problem

This thesis addresses how best to effectively utilize memory on a virtual shared memory (VSM) parallel machine whose memory is physically distributed. The Kendall Square Research 1 (KSR1) is such a machine. Below is a general description of parallel machines including a brief discussion of the drawbacks and assets of each of the different categories. In order to understand the problem addressed by this thesis, an understanding of memory paradigms on parallel computers is essential.

Parallel machines fall into two categories in Flynn's taxonomy [16]. On single instruction stream multiple data stream (SIMD) computers, all the processors execute the same instruction to evaluate different data in lock step. Multiple instruction stream multiple data stream (MIMD) computers allow different processors to execute different instructions on different data in parallel. The ability to solve different problems in parallel has made MIMD computers more popular than SIMD computers. MIMD computers are further categorized by memory organization. The two main memory paradigms for these parallel computers are shared memory and distributed memory.

On shared memory parallel machines, all processors read from and write to the same shared memory through a shared memory bus. To maintain memory coherence, only one processor on a shared memory machine may write to a given data item at any time. Shared memory allows a programmer to port programs from a single processor machine to a parallel processor machine with a minimum of code adjustments. An obvious bottleneck on a shared memory machine is the shared memory since processors must stall whenever another

processor is writing to a given data item that they need to access. However, the shared bus presents an even larger bottleneck in that all the processors must contend with each other for use of the shared bus whenever they need to access an item in memory. Snooping techniques have been developed to make more efficient use of the shared memory bus (Eggers and Katz [15]). In snooping, the processors check whether data items being retrieved by other processors (and hence, "cheaply" available on the shared memory bus) are required for their own calculations. Because of physical limitations, increasing the bandwidth of the shared memory bus is not a feasible solution—especially for a large number of processors. Due to memory bus constraints, shared memory computers tend not to be "scalable." Scalability is a measure of how easy it is to add processors to a parallel machine while maintaining proportional increases in performance—the easier it is to add processors and increase performance, then the more scalable the machine is said to be.

Distributed memory moves much of the burden of memory management from the operating system to the programmer. Each processor has its own memory that it exclusively accesses. Data is shared through message passing on a network that connects all the processors together. The interconnection network is usually designed so that multiple paths are available between processors in case one section of the network is busy or has relatively high use. Distributed memory computers have sophisticated packet switching devices or software that scan the network to find which path is shortest in terms of communication time for any given message or packet between processors. The bottleneck on a distributed memory parallel machine is message passing. The advantage of distributed memory is that there are fewer layers of control to read and write to memory, since all memory is local, and hence peak attainable performance may be increased if communication between processors is held to a minimum. Additionally, distributed memory parallel machines tend to be highly scalable. The distributed use of the interconnection network allows processors to be added to the network without significantly degrading network communication performance.

The KSR1 architecture attempts to exploit the advantages of both memory paradigms through a virtual shared, physically distributed memory architecture. Shared memory on

the KSR1 consists of distributed local caches. Each processor can only read from and write to its own local cache. In order to share information, the operating system manages data movement between the local caches. The interconnection network between processors is organized as rings of 32 processors that are further connected together by a higher level of rings. Because data transfer between local caches is distributed, the probability of a bottleneck on the interconnection network is minimized. Thus, the architecture of the KSR1 is highly scalable.

The goal of this thesis is to investigate how the unique architecture of the KSR1 cache system affects performance of algorithms that require large amounts of memory. Numerical linear algebra algorithms, in particular, meet this requirement. Because the operating system maintains exclusive control of the cache system, the only performance "tweaking" that can be performed is on the algorithms themselves. The question is whether any changes are necessary at all or whether data structures need to be modified to improve algorithm performance through more efficient use of the cache system on the KSR1. The algorithms used to evaluate performance on the KSR1 include the three levels of Basic Linear Algebra Subprograms (BLAS) and the conjugate gradient method (an iterative Krylov subspace method) to solve linear systems arising from finite difference discretizations of elliptic boundary value problems. The BLAS performance provides insight into how best to attack the larger problem of solving elliptic boundary value problems. In particular, how best to implement a preconditioned conjugate gradient method was studied. Because the preconditioner occupies as much space as the linear system derived from a finite difference discretization of the partial differential equation, efficient use of memory becomes a significant issue on a system with physically distributed but virtually shared memory.

1.2 The Kendall Square Research 1 (KSR1) Machine

This section provides an overview of the basic memory architecture of the KSR1 and a few of the more useful software features available to the KSR1 FORTRAN compiler.

The software features discussed are the PRESTO parallel directives which simplify the task of parallelizing code on the KSR1 and dynamic memory allocation which enables the programmer to make more efficient use of the KSR1's unique memory organization.

1.2.1 KSR1 memory architecture

Kendall Square Research [27] describes the KSR1 as a virtual shared memory (VSM) parallel computer. VSM allows a program to reference all of memory, as if the architecture were shared memory, even though the memory is physically distributed. Memory is allocated on the KSR1 on a "need to use" and "availability" basis.

The "need to use" basis means that memory has not been allocated until it has been referenced at least once. A FORTRAN program that declares an array `A(1024,1024)` does not have memory allocated to it until it has been referenced. This has a subtle but important affect upon initialization of data structures. Data structures on the KSR1 should first be referenced in a contiguous fashion rather than noncontiguous for increased performance. Sections 2.2.1, 2.2.2 and 2.3.5 discuss this issue of first reference cost.

The "availability" of memory to an executing program is dependent on two factors. One factor, naturally, is maximum memory size. On the KSR1, all memory (which the exception of disk memory) is cache. Each processor has a local cache of 32MB which it reads from and writes to. Thus within a parallel loop of a program, no processor can access more than 32MB of cache memory without cache-to-cache or cache-to-disk paging. In reality, this hard limit is smaller than 32MB because the operating system is going to occupy some of this space. Memory availability is also restricted by other programs executing on a given processor set and hence using cache memory on that processor set.

Memory on the KSR1 consists of 32MB local caches connected together through a hierarchy of ALL-Cache Engines. The AllCache Engine:0 connects up to 32 local caches in a ring. To increase the number of processors on the KSR1 up to a maximum of 1088, the Allcache Engine:1 connects up to 34 AllCache Engine:0s in a ring. In addition to a local cache, each processor has 64 64-bit floating point registers, 32 64-bit integer registers, and

Table 1.1: KSR1 Capacity and Peak Performance [27].

Processor Configurations	Peak MIPS	Peak MFLOPS	Memory (MB)	Max. Disk Capacity (GB)	Max. I/O Capacity MB/Sec
KSR1-8	320	320	256	210	210
KSR1-16	640	640	512	450	450
KSR1-32	1,280	1,280	1,024	450	450
KSR1-64	2,560	2,560	2,048	900	900
KSR1-128	5,120	5,120	4,096	1,800	1,800
KSR1-256	10,240	10,240	8,192	3,600	3,600
KSR1-512	20,480	20,480	16,384	7,200	7,200
KSR1-1088	43,520	43,520	34,816	15,300	15,300

a .5MB subcache that is divided into equal .25MB segments (one for instruction handling and the other to manage data movement).

Cache pages are 16KB. The pages are further broken down into 128-byte subpages. The unit of allocation in cache is a page and the unit of transfer between local caches is a subpage. When a processor requests a read of a subpage that is not in its local cache but the cache page that the subpage belongs to has been allocated in its local cache, then the subpage is copied to that cache page. Otherwise, a cache page is allocated in the processor's local cache and the subpage is then copied to it. Copies of the same subpage may exist on multiple local caches if the subpage is only being read by the processors. Once a write occurs, the only valid copy of the subpage is located on the writing processor's local cache and all other copies of the subpage are invalidated.

Table 1.1 shows the various KSR1 configurations allowable. A KSR1-64 located at Oak Ridge National Laboratory was used to conduct the experiments reported in this thesis.

1.2.2 KSR1 parallel directives

Pthreads

The KSR1 accomplishes parallelism by creating *pthreads* (i.e., units of sequential work) and assigning each *pthread* to a different processor. *Pthreads* may be directed to attack problems both as a team or individually.

Pthreads are controlled either directly through special calls from a program or indirectly through the KSR1 run time system called PRESTO. Using special calls to control *pthreads* involves making more code changes and thus destroys a key advantage of a shared memory parallel machine. PRESTO makes it easy to create and assign work to *pthreads* through high level parallel directives. When controlled directly through special calls, *pthreads* are assigned attributes that allow the programmer to control mutual exclusion when executing a routine and manage synchronization through barriers and control variables. All programs written in conjunction with this thesis make use of PRESTO and hence *pthread* attribute assignment is not managed by the programmer.

To create *pthreads* on the KSR1, the programmer has three choices (see references [25] and [26] for more detail):

- Call from a program `pthread_create(thread_id,iattr,rtn,iarg,istatus)`. This call creates a new *pthread* with identification number `thread_id` and attribute `iattr`. The *pthread* upon successful creation will begin executing `rtn` passing that routine argument `iarg`. The `istatus` variable contains the status of `pthread_create` with 0 meaning success. This method of *pthread* creation would be used if the programmer were directly managing the *pthreads* in a program.

- Call from a program `ipr_create_team(n,team_id)` where `n` is the desired number of *pthreads* and `team_id` is the returned system team identification number for the team of *pthreads*.

- Set the UNIX environment variable `PL_NUM_THREADS` to the desired number of *pthreads*. This method is used in lieu of a program call to request a *pthread* set. The compiled program has no *pthread* creation calls, instead *pthreads* are created prior to program execution through the UNIX `setenv PL_NUM_THREADS` command.

When creating *pthreads* for use by PRESTO, either the second or third method of *pthread* creation should be used; `pthread_create` is used only for direct control of *pthreads* through special calls.

PRESTO

PRESTO parallel directives may be called from C or FORTRAN.[1] PRESTO provides a simple means of converting sequential code into parallel code. The following paragraphs describe the different PRESTO parallel directives and the syntax for each. See [26] for more examples of PRESTO directives.

Parallel region. `Parallel region` directs the KSR1 to create multiple copies of a code segment to execute in parallel. Typically, the means of making each segment "look" at different data is through indexing data by processor number which can be queried. The syntax to create a `parallel region` is as follows:

```
c*ksr* parallel region(teamid=Team_id,private=(Myprocessor))
      Myprocessor = ipr_mid()
      call code(myprocessor)
c*ksr* end parallel region
```

`Team_id` is the system assigned team number for a group of *pthreads* initialized earlier during program execution by a call to `ipr_create_team`. `Myprocessor` is the processor number that a given *pthread* is running on (returned by the function `ipr_mid`). `Private` indicates that the variables listed in parenthesis are local on each processor. All other variables not listed within the private list are considered shared. The exception to this rule

[1] Within this thesis, FORTRAN is exclusively used.

is subroutine calls. Variables within subroutines called from within **parallel regions**, **parallel sections**, or **tiles** are considered private or local variables.

Parallel sections. **Parallel sections** directs the KSR1 to execute multiple segments of code in parallel. To create a group of **parallel sections** use the following general template:

```
c*ksr* parallel sections(team_id=Team_id,private=(variable list))
c*ksr* section
      call section1
c*ksr* section
      call section2
c*ksr* end parallel sections
```

Each segment of code is identified by a **c*ksr* section** preceding it. The last code segment end is identified by **c*ksr* end parallel sections**.

Tiling. The **tile** directive is based upon Wolfe's tiling concept [53]. Tiling directs the KSR1 to divide work performed in a program loop into iteration groups called tiles. The tiles are then assigned to *pthreads*. There are three main tiling controls—fully automatic, semi-automatic, and manual tiling. To use fully automatic tiling:

```
c*ksr* tile(i,teamid=Team_id)
     Do i = 1,100
       Do j = 1,100
         code
       End do
     End do
c*ksr* end tile
```

Semi-automatic tiling allows the user to specify a tilesize, the number of iterations in a single tile:

```
c*ksr* ptile(i,teamid=Team_id,tilesize=(i:10))
      Do i = 1,100
        Do j = 1,100
          code
        End do
      End do
```

Manual tiling also allows user selection of the tilesize but it differs from semi-automatic tiling by providing the programmer total control over whether variables within a tiled loop are shared or private:

```
c*ksr* user tile(i,private=(j),teamid=Team_id,tilesize=(i:10))
      Do i = 1,100
        Do j = 1,100
          code
        End do
      End do
c*ksr* end tile
```

In both semi-automatic and fully automatic tiling, program correctness is ensured. For example in semi-automatic and fully automatic tiling, the programmer need not specify that j in the above inner loops is a private variable because the compiler will recognize it as such. In manual tiling, all private variables must be specified by the programmer. The advantage of manual and semi-automatic tiling over fully automatic tiling is that the programmer has more control over tiling assignments. When fully automatic tiling is used, the KSR1 will not always use the total number of *pthreads* if it "determines" that there is insufficient work to be performed by the specified number of *pthreads*. Typically, the KSR1 assigns work to a *pthread* in iteration multiples of 32 so the tilesize parameter is ignored regardless of whether it is specified in fully automatic tiling. Manual and semi-automatic tiling, as shown in the above examples, allow the programmer to specify tile "width" through the tilesize control parameter.

In all three tiling methods, the programmer can specify a tiling strategy or means of dividing up work between *pthreads*. There are four possibilities:

Slice strategy. The slice strategy is supposed to create as many tiles as there are *pthreads*. In fully automatic tiling, the KSR1 will create as many tiles as it determines worthwhile. In manual and semi-automatic tiling, the programmer should calculate a tilesize that evenly slices up the iteration space of the loop into p slices where p is the number of *pthreads*. This strategy is the default.

Modulo strategy. The modulo strategy assigns the nth tile to processor n mod p.

Wavefront strategy. The wavefront strategy is similar to the modulo strategy, but it provides a control parameter (`ORDER`) to ensure loop integrity in two or more dimensional tiling. `ORDER` ensures loop integrity by specifying an order in which tiles are executed. The following example taken from KSR [26] would be executed in the order shown in Figure 1.1 where there are 3 *pthreads* and the tilesize is 32×32 where possible.

```
c*ksr* tile(i,j,order=(i,j))
       do i = 1,99
         do j = 1,99
           a(i,j) = a(i+1,j+1)
         end do
       end do
c*ksr* end tile
```

	Pthread 1 $j = 1,32$	Pthread 2 $j = 33,64$	Pthread 3 $j = 65,96$	Pthread 1 $j = 97,99$
$i = 1,32$	1	2	3	4
$i = 33,64$	2	3	4	5
$i = 65,96$	3	4	5	6
$i = 97,99$	4	5	6	7

Figure 1.1: Wavefront tile execution order.

Grab strategy. The grab strategy assigns tiles to *pthreads* on a first-come, first-serve basis.

Affinity region. `Affinity region` enhances tiling efficiency by indicating to PRESTO that all tile families within an **affinity region** are assigned to *pthreads* using the same *pthread*-to-tile assignment. A tile family is a group of tiles that make up one tiled loop. **Affinity regions** are used to ensure that the same *pthread* accesses the same data. This reduces data movement between processors or *pthreads*. The following example outlines the syntax of the **affinity region** directive and the affect it has in *pthread* to tile assignment:

```
c*ksr* affinity region((i:1,192),teamid=Team_id) ! Let # pthreads = 2
c*ksr* tile(i,teamid=Team_id)
      do i = 1,192                                ! Loop 1
        a(i) = b(i)
      end do
c*ksr* end tile
c*ksr* tile(i,teamid=Team_id)
      do i = 1,128                                ! Loop 2
        c(i) = a(i)*alpha
      end do
c*ksr* end tile
c*ksr* end affinity region)
```

Loop 1			Loop 2	
Pthread 1	Pthread 2	Pthread 1	Pthread 1	Pthread 2
Tile 1	Tile 2	Tile 3	Tile 1	Tile 2

Figure 1.2: Mapping *pthreads* to tiles using **affinity region**.

In the above example, the loops would be divided into tiles and assigned to *pthreads* as shown in Figure 1.2. The same tilesize is used for each tiled loop in an **affinity region**. The tilesize is determined by dividing the number of *pthreads* into the smallest iteration space. Thus, tilesize is 64 for both loops using the *affinity region* parallel directive. The **affinity region** ensures that *Pthread*1 and *Pthread*2 in Loops 1 and 2 access the same segments of **a**, thus reducing data movement of **a** between *pthreads*.

Loop 1		Loop 2	
Pthread 1	Pthread 2	Pthread 1	Pthread 2
Tile 1	Tile 2	Tile 1	Tile 2

Figure 1.3: Mapping *pthreads* to tiles without `affinity region`.

Without the `affinity region` directive, *pthread* to tile assignment would be as shown in Figure 1.3. The tilesize in Loop 1 without `affinity region` becomes 96, hence only two tiles are needed to divide up the iteration space. The tilesize in Loop 2 remains 64.

1.2.3 KSR1 dynamic memory allocation

The KSR1 provides dynamic memory allocation to FORTRAN programs through the `allocate` and `malloc()` directives. `Allocate` allocates arrays whose dimensions are not known at compile time. To use `allocate`, a data item `x` is defined as integer, double precision, or real `x(:)`. When the correct space is known within a program, a call to `allocate` is made giving the desired space. In the example below, `x` is defined as double precision and later allocated to be 20 double precision elements long.

```
      double precision x(:)
c     code
      allocate x(20)
```

The other means of dynamic allocation, `malloc()`, is useful for allocating more than one section of memory to the same variable name at different points in the program. `Malloc()` uses pointers to address data items for which it allocates space. In the following example, `x` is a vector of length 40 that is split into 4 mini-vectors of length 10. The pointers to the newly allocated mini-vectors are saved in the `xptr` integer array in Loop 1 for future reference in Loop 2.

```
double precision x(1)
pointer (px,x)
integer xptr(4), n
parameter(n = 10)
do i = 1,4                          ! Loop 1
  call malloc(16*n,px)              ! 16 is used for double precision
  xptr(i) = px
end do

do i = 1,4                          ! Loop 2
  px = xptr(i)
  do j = 1,n
    x(j) = j
  end do
end do
```

Both `allocate` and `malloc()` allocate space at the top of a cache subpage when called. Two other means of forcing an array to begin at the top of a subpage are through use of the **subpage** KSR1 compiler directive and by declaring the array to be at the beginning of a FORTRAN **common** block. The **subpage** KSR1 compiler directive is invoked after an array has been declared. In the following example, both arrays `a` and `b` are forced to begin on a subpage boundary:

```
      double precision a(10), b(10)
c*ksr* subpage a, b
```

1.3 Related Work

Dongarra, Moler, Bunch, and Stewart [14] managed development a package of routines called LINPACK that solve linear algebra problems under the support and supervision of Argonne National Laboratory. Incorporated into LINPACK are some core linear algebra routines called Basic Linear Algebra Subroutines (BLAS) that perform the most basic linear algebra functions in real, double precision, single complex, and double complex arithmetic. BLAS was written by Lawson, Hanson, Kincaid, and Krogh [31]. Other work involving LINPACK routines may be found in Dongarra, DuCroz, Hammarling, and Hanson [12] and

Dongarra, DuCroz, Hammarling, and Duff [13].

Hestenes and Stiefel [24] developed the conjugate gradient (CG) methods for solving symmetric positive definite linear systems of equations in 1952. It was not until Reid [45] showed that CG methods converged particularly well for certain large sparse systems that these methods became popular to use and study. However, the poor rate of convergence of CG methods when solving poorly conditioned systems (see Luenberger [33]) gave rise to the idea of preconditioning. Concus, Golub, and D. O'Leary [9] proposed that the preconditioner M that approximates A should have the following qualities:

1. M is symmetric positive definite.
2. The system $M\hat{r} = r$ is "easily" solved.
3. The preconditioned matrix $M^{-1}A$ has a much lower condition number than A or is a low rank perturbation of the identity matrix.

In 1977, Meijerink and van der Vorst [35] showed that an incomplete Cholesky factorization of A, which produces an M with the above properties, provides improved CG performance on poorly conditioned problems.

With the 1980s, came the first widely available parallel machines. Parallelization of CG methods became an issue to be studied. In order to increase parallel performance of the preconditioner, some means of breaking it into blocks that could be distributed between processors was needed. O'Leary [40] explored factoring sparse matrices into blocks using Lanczos' algorithm for finding eigenvalues of large sparse matrices to extend to CG methods. This work was important in showing that breaking a matrix up into smaller blocks, when done correctly, could improve performance of CG methods. Concus, Golub, and Meurant [8] provide an excellent outline of how to create block preconditioners factored using incomplete Cholesky. O'Leary [42] provides an outline for a block CG algorithm which performs well on distributed memory parallel machines where communication is costly.

Reordering schemes are important to performance of block preconditioners. Lichnewsky [32] and Schreibner and Tang [49], both in 1982, tested what affect reordering blocks as

well as equations had on a blocked incomplete Cholesky preconditioner and found that using a red-black (even-odd) or 4-color ordering could improve performance. O'Leary [41] showed that reordering mesh points could improve convergence on a 2-D grid of processors. Bramble, Pasciak, and Schatz [6] outlined one of the first methods for constructing a preconditioner based on domain decomposition.[2]

Interest in domain decomposition methods continues to grow as new methods are explored. Important work includes Gropp and Keyes ([20], [21], and [30]), Keyes [28], and Meurant [37]. Chan and Mathew [7] provide an excellent review of different domain decomposition algorithms. A series of conference proceedings is devoted to the study of domain decomposition methods for partial differential equations ([17], [11], [10], [18], and [29]).

For related preconditioned conjugate gradient work on the KSR1, Bagheri, Ilin, and Scott [2] have studied implementation of Brownian dynamics simulation on the KSR1. Their Brownian dynamics simulation involved using a preconditioned conjugate gradient method to solve the linearized Poisson-Boltzmann equation

$$-\nabla \cdot \varepsilon \nabla \phi + \varepsilon k^2 \phi = \rho$$

using a seven-point difference stencil. The preconditioners used were a diagonal preconditioner and a parallel (blocked) incomplete Cholesky preconditioner. Bagheri, Ilin and Scott achieved linear speedup for problems of grid size 160 x 160 x 160. They did not find the KSR1's unique memory architecture to be a problem.

Other work on the KSR1 includes Nurkkala and Kumar [39] who examined performance of natural language parsers. Nurkkala and Kumar recommend avoiding frequent updates to shared data, avoiding blocking due to mutual exclusion (this is due to subpage writes which cause cache misses in other processors reading a subpage), avoiding standard C libraries (the standard C I/O library is not re-entrant), and using dedicated processor sets (due to the degraded performance incurred by context switching on processors shared with other

[2] A variation of the Bramble, Pasciak, and Schatz (BPS) preconditioner is the domain decomposition preconditioner used in this thesis.

users). Baillee, Carr, Hart, Henderson, and Rodriguez [3] found that when comparing the KSR1 to the Intel Paragon (a distributed memory computer), the KSR1 did better on smaller grid-based weather problems than the Paragon did and the Paragon did better on larger grid-based weather problems than the KSR1 did. However, Baillee, et al., state that parallelizing their code on the KSR1 was a trivial problem while parallelizing their code to run on the Intel Paragon was nontrivial.

1.4 Organization Of The Thesis

Chapter 2 describes how first, second, and third level BLAS operations perform on the KSR1 and lays the ground-work for exploring data splitting (i.e., using data structures with noncontiguous memory) on the KSR1.

Chapter 3 lays out the algorithms and data structures used to solve linear elliptic boundary value problems. Performance is evaluated for unpreconditioned conjugate gradient (CG), incomplete Cholesky preconditioned CG, and domain decomposition preconditioned CG.

Chapter 4 contains the summary and conclusions of this thesis. Implications for other iterative methods and preconditioners are described in this chapter.

Chapter 2

LINEAR ALGEBRA KERNELS (BLAS)

This chapter discusses KSR1 performance for several of the basic linear algebra or BLAS operations. By exploring various means of manipulating data storage of arrays on the KSR1 through BLAS operations, a better understanding of how to implement more complex algorithms, such as a partial differential equation solver, is derived. Performance statistics of the various implementations of the BLAS operations indicate that noncontiguous storage is better than contiguous storage when writing to a data structure. Noncontiguous storage, provided that it is implemented correctly, can prevent writes to the same cache subpage by different processors within a parallel loop.

2.1 Level One Operations

This section evaluates performance of the double precision vector dot product (DDOT) and vector update (DAXPY) first level BLAS operations on the KSR1. Results of both operations are divided into two categories, one where vectors of length n are contiguously allocated and one where each vector is broken up into p noncontiguous mini-vectors of length n/p (p being the number of processors).

2.1.1 Vector dot product (DDOT)

The vector dot product operation is defined mathematically as:

$$\sum_{i=1}^{n} x_i \cdot y_i,$$

where x and y are defined to be vectors of length n. On the KSR1, near linear speedup is achieved for noncontiguous mini-vectors with $n = 8192$. Contiguous vectors without the

Table 2.1: DDOT performance results (averaged over 100 runs).

	Noncontiguous			Contiguous			Contiguous/subpage aligned		
P	Time	Speedup	Effic.	Time	Speedup	Effic.	Time	Speedup	Effic.
1	0.166			0.165			0.162		
2	0.080	2.07	1.03	0.084	1.95	0.98	0.083	1.95	0.97
4	0.041	4.08	1.02	0.042	3.90	0.97	0.042	3.89	0.97
8	0.022	7.66	0.96	0.021	7.67	0.96	0.021	7.64	0.95
16	0.012	13.84	0.86	0.014	12.12	0.76	0.013	12.90	0.81

KSR1 FORTRAN compiler directive `c*ksr* subpage x, y`, suffer a slight performance degradation. See Table 2.1 for results.

Because DDOT does not update any elements of a vector, both tests were expected to perform similarly well. In order to eliminate the possibility that reduction was being handled incorrectly, tests were made to simulate correct handling of a reduction variable. PRESTO allows reduction in a tiled loop through the control parameter `reduction=(sum)` where `sum` is a reduction variable. The tests indicate that reduction is being handled correctly by the KSR1 FORTRAN compiler. When the contiguous vectors are forced to begin at the top of a subpage, DDOT performs in a similar manner as the noncontiguous DDOT test (see Table 2.1).

These results show that even for read operations on the KSR1, vectors should be aligned on subpage boundaries to reduce cache paging. Section 1.2.3 describes four methods of forcing arrays to begin at the top of a cache subpage.

2.1.2 Vector update (DAXPY)

The vector update operation is defined mathematically as follows:

$$z = x + \alpha y$$

where x, y, and z are vectors of length n and α is a scalar. Superlinear speedup is achieved when updating noncontiguous mini-vectors with total vector length $n = 8192$. Performance

Table 2.2: DAXPY performance results (averaged over 100 runs).

	Noncontiguous			Contiguous		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	0.216			0.210		
2	0.088	2.45	1.23	0.106	1.98	0.99
4	0.044	4.90	1.22	0.052	4.01	1.00
8	0.021	10.06	1.26	0.029	7.30	0.91
16	0.013	17.04	1.07	0.016	13.28	0.83

degradation is encountered, as expected, on the contiguous vector update code. See Table 2.2 for test results.

The superlinear performance of the noncontiguous vector update is due to paging of the cache. When the number of processors used is low, then there will be more paging per processor as the virtual memory system brings data back from disk into cache. When the number of processors is high, less paging occurs on each processor. Because paging entails a certain amount of system time, performance suffers whenever paging occurs. Hence, superlinear speedup is achieved. The superlinear performance is not achieved in contiguous vector update because the vectors are not aligned on subpage boundaries and as a result different processors may be trying to update the same cache subpage.

2.2 Level Two Operations

This section evaluates the performance of matrix initialization and two second level double precision BLAS operations. Both BLAS operations, matrix-vector product and triangular solve, involve vector updates. Because matrices are also involved in the operations, both row and column oriented algorithm performance are examined. Algorithm performance is further broken down by whether the vector being updated is stored as contiguous, nonsubpage aligned or noncontiguous, subpage aligned.

Table 2.3: Matrix initialization (averaged over 10 runs).

P	Contiguous/Column Time	Speedup	Efficiency	Noncontiguous/Row Time	Speedup	Efficiency
Using exact storage (`A(1024,1024)`)						
1	5.196			9.240		
2	2.611	1.99	0.99	6.372	1.45	0.73
4	1.306	3.98	0.99	3.289	2.81	0.70
8	0.653	7.96	1.00	2.190	4.22	0.53
16	0.330	15.76	0.98	1.279	7.22	0.45
Using more storage than necessary (`A(10240,10240)`)						
1	5.541			17.128		
2	2.789	1.99	0.99	14.069	1.22	0.61
4	1.393	3.98	0.99	6.820	2.51	0.63
8	0.701	7.90	0.99	2.793	6.13	0.77
16	0.354	15.65	0.98	2.509	6.83	0.43

2.2.1 Matrix initialization

Matrix initialization may be done in either column or row order. On the KSR1, matrix initialization should be done contiguously. In FORTRAN, matrices are stored in column order. Hence, to initialize a matrix contiguously, the initialization must be done in column order. On the KSR1, execution time is degraded if the matrix has never been referenced before and the initialization is done noncontiguously. Defining matrices larger than necessary adds additional time to the first reference initialization execution time. Table 2.3 gives performance results for a 1024 x 1024 matrix initialization where the matrix A is defined for one set of tests using exact storage and in another using more storage than necessary.

2.2.2 Matrix-vector product

Row oriented matrix-vector product

The algorithm for a row oriented ($z = y + Ax$) is as follows:

```
c*ksr* user tile(i,private=(sum,j),teamid=iteam)
      do i = 1,n
        sum = y(i)
        do j = 1,m
          sum = sum + A(i,j) * x(j)
        end do
        z(i) = sum
      end do
c*ksr* end tile
```

The algorithm is row oriented in that a row of A is referenced for each calculation of z_i. The dimensions of the matrix A are $n \times m$. The vectors y and z are of length n and x is of length m. The algorithm may be parallelized over the outer loop as shown above in the `c*ksr* user tile` directive.

General row oriented case. The full form of row oriented matrix-vector product ($z = y + Ax$) is evaluated for vectors of length 512 and $A(512, 512)$ where z is stored contiguously and noncontiguously (see Table 2.4). The contiguous results are unexpectedly good given that more than one processor might have to wait to update z. However, the quantity of "work" that each processor must perform before updating z overwhelms the time waiting

Table 2.4: Row oriented matrix-vector product performance results (averaged over 100 runs).

	Noncontiguous			Contiguous		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	5.898			5.813		
2	2.994	1.97	0.98	2.972	1.96	0.98
4	1.500	3.93	0.98	1.600	3.63	0.91
8	0.772	7.64	0.96	0.890	6.53	0.82
16	0.432	13.65	0.85	0.426	13.64	0.85

for exclusive access when writing to a subpage. The following contrived example shows at what point contiguous vector update ceases to be a significant problem in a matrix-vector product.

Accessing A noncontiguously. As shown by the level one BLAS experiments, cache paging can cause degraded system performance in both read and write operations, although write operations are potentially more damaging. Because FORTRAN stores matrices in column order, a row oriented matrix algorithm might present a cache performance problem. In order to measure the affect of accessing A by rows, tests were made of a more limited form of row oriented matrix-vector product ($y = Ax$). A is of dimension 512×512 and vectors x and y are of length 512. Storage of A is varied in the following three ways:

1. Store A as double precision $A(1024, 1024)$ in nontransposed order.
2. Store A in transposed order as double precision $A(1024, 1024)$.
3. Store A in transposed order as $A(512, 512)$.

In the first two storage schemes for A, the full potential matrix is not used. This tests whether performance is degraded by defining a matrix to be significantly larger than necessary. Table 2.5 shows that the second and third methods which access A contiguously perform slightly better than the first method where A is accessed noncontiguously.

Table 2.5: Methods of accessing A in $y = Ax$ (averaged over 10 runs).

	Method 1			Method 2			Method 3		
P	Time	Spdup	Effic	Time	Spdup	Effic	Time	Spdup	Effic
1	5.932			5.336			5.056		
2	3.576	1.66	0.83	2.758	1.93	0.97	2.581	1.96	0.98
4	2.336	2.54	0.63	1.372	3.89	0.97	1.130	3.90	0.98
8	1.180	5.03	0.63	0.684	7.80	0.98	0.651	7.76	0.97
16	0.540	10.985	0.69	0.364	14.66	0.92	0.347	14.58	0.91

First reference cost. The KSR1 does not physically allocate space to a variable in memory until that variable is first referenced. For large data structures this translates to a hidden cost when they are first referenced. This hidden cost is the time it takes for the KSR1 to allocate space for the data structure. In a contrived experiment where $y = Ax$ is calculated, A is not initialized. Storage of A is varied in a manner similar to the experiment "Accessing A noncontiguously." Table 2.6 shows that if A is not initialized (i.e., never referenced before):

- Storing A in transposed order is significantly better.
- Defining A to be no larger than necessary and storing it in transposed order is the best of the three storage methods.

These results show that storage should be initially referenced in a contiguous manner on the KSR1.

Table 2.6: First reference cost of accessing A in $y = Ax$ (averaged over 10 runs).

	Method 1			Method 2			Method 3		
P	Time	Spdup	Effic	Time	Spdup	Effic	Time	Spdup	Effic
1	22.48			6.628			8.036		
2	15.68	1.43	0.72	3.705	1.79	0.89	4.204	1.91	0.96
4	14.32	1.57	0.39	1.936	3.42	0.86	2.148	3.74	0.94
8	14.86	1.51	0.19	1.140	5.81	0.73	1.000	7.31	0.91
16	19.29	1.17	0.07	1.124	5.90	0.37	0.736	10.92	0.68

Example to show benefits of noncontiguous storage. To clearly show the benefits of noncontiguous storage, a simplified matrix-vector product of the form $y = Ax$ is used. Using double precision, the variables are defined as $A(1024, n)$, $x(1024)$, and $y(n)$ where n is varied from 4 to 64 by increments of 4. To minimize any paging of the matrix A that might occur in the row oriented matrix vector product, A is stored in transposed order so that its true dimension was $A(n, 1024)$. In two tests of the matrix-vector product, y is stored once

Table 2.7: Benefits of noncontiguous storage in $y = Ax$ (averaged over 100 runs).

n	Base Time	4 Processor Time	Speedup	Efficiency
Noncontiguous storage of y				
4	0.103	0.023	4.43	1.11
8	0.177	0.046	3.88	0.97
12	0.26	0.062	4.27	1.07
16	0.345	0.085	4.04	1.01
20	0.433	0.108	4.02	1.01
24	0.519	0.128	4.08	1.02
28	0.605	0.148	4.08	1.02
32	0.692	0.169	4.09	1.02
Contiguous storage of y				
4	0.086	0.080	1.07	0.27
8	0.170	0.158	1.08	0.27
12	0.255	0.164	1.56	0.39
16	0.335	0.170	1.97	0.49
20	0.418	0.180	2.32	0.58
24	0.502	0.185	2.72	0.68
28	0.587	0.185	3.17	0.79
32	0.671	0.186	3.60	0.90

contiguously and once noncontiguously as four mini-vectors of length $n/4$. In both tests, speedup is measured on 4 processors. Table 2.7 shows that:

- The noncontiguous version exhibits linear speedup regardless of n.
- The contiguous version does not behave well until $n = 32$, where y can be evenly divided into 4 subpages.

Table 2.8: Column oriented matrix-vector product performance results (averaged over 100 runs).

	Noncontiguous			Contiguous		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	6.133			5.501		
2	3.557	1.72	0.86	3.168	1.74	0.87
4	1.921	3.19	0.80	1.720	3.20	0.80
8	1.428	4.30	0.54	1.227	4.49	0.56
16	0.993	6.18	0.39	1.005	5.47	0.34

Column oriented matrix-vector product

The column oriented matrix-vector ($z = y + Ax$) works as follows:

```
c*ksr* user tile(i,teamid=iteam)
      do i = 1,n                          ! z = y
        z(i) = y(i)
      end do
c*ksr* end tile
      do j = 1,n                          ! z = z + Ax
c*ksr* user tile(i,teamid=iteam)
        do i = 1,m
          z(i) = z(i) + A(i,j) * x(j)
        end do
c*ksr* end tile
      end do
```

A column of A is processed for each iteration of the outer loop in the second loop, hence the algorithm is said to be column oriented. Because only the inner loop may be parallelized, there is less work per processor before an update occurs to z than in the row oriented matrix-vector product. As a result, performance of the column oriented matrix-vector product on the KSR1 is expectedly poor regardless of how z is stored (either contiguously or noncontiguously). See Table 2.8 for results of the algorithm where $n = 512$ and $m = 512$.

The column oriented matrix-vector performance might be improved by making use of n temporary vectors (t) of length m that collect the product of A with x and then summing the temporary vectors to get z. The modified algorithm would be as follows:

```
for j = 1, n                    ! This loop may be parallelized
    z_j = y_j
end
for j = 1, n                    ! This nested loop may be parallelized
    for i = 1, m
        t_i^(j) = A_{i,j} x_j
    end
end
for i = 1, m                    ! This nested loop may be parallelized
    sum = 0
    for j = 1, n
        sum = sum + t_i^(j)
    end
    z_i = sum
end
```

2.2.3 Triangular solve

Consider the solution of a linear system of equations $Ax = b$, given the factorization of $A = LU$ where L is unit lower triangular and U is upper triangular. The unknown vector x can be found by solving two triangular systems, one forward solve ($Ly = b$) and one backward solve ($Ux = y$).

The noncontiguous versions of the row and column oriented triangular solve use n mini-vectors of length 1. Only the inner loops of row or column oriented solves are parallelizable. Furthermore, the iteration space of the inner loops is dependent on the value of the outer loop control variable. Hence, the iteration space varies and in order to spread work evenly across processors, the tilesize parameter must vary accordingly. The smallest tilesize is 1 where the iteration space of the inner loop is less than or equal the number of processors being used. The mini-vectors are made to be length 1 so that when the iteration space of the inner loop is less than or equal to the number of processors, no processor is updating the same subpage.

Row oriented triangular solve

The row oriented triangular solve algorithm is as follows:

```
      do i = 1,n                          ! Forward Solve
        sum = b(i)
        if (i .ne. 1) then
          itile = int(real(i-2)/real(P)) + 1
c*ksr* user tile(j,reduction=(sum),tilesize=(j:itile),teamid=iteam)
          do j = 1,i-1
            sum = sum - a(i,j) * x(j)
          end do
c*ksr* end tile
        end if
        x(i) = sum
      end do
      do i = n,1,-1                       ! Backward Solve
        sum = x(i)
        if (i .ne. 1) then
          itile = int(real(i-2)/real(P)) + 1
c*ksr* user tile(j,reduction=(sum),tilesize=(j:itile),teamid=iteam)
          do j = i-1,1,-1
            sum = sum - a(i,j)*x(j)
          end do
c*ksr* end tile
        end if
        x(i) = sum/a(i,i)
      end do
```

In the above algorithm, A contains both L and U. Because L is unit lower triangular and the diagonal is known to be all 1s, storage of the diagonal for L is not necessary. Thus, both L and U fit within the storage of a single matrix (A). The tilesize is varied because the iteration space of the inner loops vary dependent on the outer loops.

Both the contiguous, nonsubpage aligned and noncontiguous, subpage aligned row oriented solves do not parallelize well. The amount of work performed by the inner loop varies dependent on the outer loop. Eventually, there is insufficient work in the inner loop to be divided up between many processors. Hence, performance degrades as the number of processors grows. Table 2.9 shows poor performance of the row oriented solve where A is a

Table 2.9: Row oriented triangular solve performance results (averaged over 100 runs).

	Noncontiguous			Contiguous		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	7.457			7.052		
2	4.416	1.69	0.84	4.702	1.50	0.75
4	2.409	3.09	0.77	2.509	2.81	0.70
8	2.039	3.66	0.46	2.144	3.29	0.41
16	2.082	3.58	0.22	2.109	3.34	0.21

512×512 double precision matrix and x and b are double precision vectors of length 512.

Column oriented triangular solve

The column oriented triangular solve of $Ax = b$ works as follows:

```
      nvect = n/P
c*ksr* user tile(i,tilesize=(i:nvect),teamid=iteam)
      do i = 1,n                              ! x = b
        x(i) = b(i)
      end do
c*ksr* end tile
      do j = 1,n-1                            ! Forward Solve
        itile = int(real(n-j-1)/real(P)) + 1
c*ksr* user tile(i,tilesize=(i:itile),teamid=iteam)
        do i = j+1,n
          x(i) = x(i) - a(i,j)*x(j)
        end do
c*ksr* end tile
      end do
      do j = n,1,-1                           ! Backward Solve
        x(j) = x(j)/a(j,j)
        if (j .ne. 1) then
          itile = int(real(j-2)/real(p)) + 1
c*ksr* user tile(i,tilesize=(i:itile),teamid=iteam)
          do i = j-1,1,-1
            x(i) = x(i) - a(i,j)*x(j)
          end do
c*ksr* end tile
        end if
      end do
```

Just as in the row oriented version of the triangular solve, the LU factorization is stored in A. The inner loops of the forward and backward solves access a column of A for each iteration of the outer loop—hence, the algorithm is column oriented. Tilesize must be varied in both inner loops because the iteration space changes dependent on the outer loop control variable j.

The column oriented version of the triangular solve does not parallelize well for the same reason that the row oriented version does not. There is insufficient work when j is small for a larger number of processors to keep busy. This is reflected in Table 2.10.

Table 2.10: Column oriented triangular solve performance results (averaged over 100 runs).

	Noncontiguous			Contiguous		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	7.201			6.571		
2	4.529	1.59	0.79	4.528	1.45	0.73
4	2.666	2.70	0.68	2.945	2.23	0.56
8	2.025	3.56	0.44	2.257	2.91	0.36
16	2.942	3.71	0.23	2.052	3.20	0.20

2.3 Level Three Operation—Matrix-matrix Product

The only level three BLAS operation tested was the matrix-matrix product. We consider four different versions of code to evaluate the equation

$$C = \beta C + \alpha AB$$

on the KSR1 where A, B, and C are $n \times n$ matrices and α and β are scalars. These three versions are:

Table 2.11: Nonblocked matrix-matrix product performance results (averaged over 10 runs).

	jik Column			*jki* Column			*ijk* Row		
P	Time	Spdup	Effic	Time	Spdup	Effic	Time	Spdup	Effic
1	478.3			210.9			494.9		
2	240.5	1.99	0.99	105.7	2.00	1.00	248.7	1.99	0.99
4	120.5	3.97	0.99	53.1	3.97	0.99	125.3	3.95	0.99
8	60.0	7.97	1.00	27.0	7.81	0.98	62.9	7.87	0.98
16	30.0	15.92	0.99	13.5	15.58	0.97	31.4	15.74	0.98

Table 2.12: Blocked matrix-matrix product performance results (averaged over 10 runs).

P	Time	Speedup	Efficiency
1	11.03		
2	5.92	1.86	0.93
4	3.45	3.19	0.80
8	1.76	6.25	0.78
16	0.94	11.73	0.73

1. *jik* column oriented

2. *jki* column oriented

3. *ijk* row oriented

4. Optimized Block oriented[1]

For testing purposes, the three matrices are defined in single precision as 512 x 512.

[1]The optimized block oriented code is KSR proprietary and is written using single precision arithmetic. For this reason, the column and row oriented versions are also implemented in single precision so that the three versions are comparable.

2.3.1 *jik* Column oriented matrix-matrix product

The *jik* column oriented matrix-matrix product algorithm is:

```
        nvect = n/nbr_of_processors
c*ksr* user tile(j,private=(i,k,sum),tilesize=(j:nvect),teamid=iteam)
        do j = 1, n
          do i = 1, n
            sum = 0.0
            do k = 1,n
              sum = sum + a(i,k)*b(k,j)
            end do
            c(i,j) = beta*c(i,j) + alpha*sum
          end do
        end do
c*ksr* end tile
```

A column of C is calculated at each iteration of the outer loop. Because the outer loop can be parallelized and because of the three level loop nesting, the amount of work to be performed at each iteration of the outer loop is sufficient to make the algorithm parallelize well. Table 2.11 contains performance statistics.

2.3.2 *jki* Column oriented matrix-matrix product

The *jki* column oriented matrix-matrix product algorithm is:

```
      nvect = n/nbr_of_processors
c*ksr* user tile(j,private=(i),tilesize=(j:nvect),teamid=iteam)
      do j = 1,n
        do i = 1,n
          c(i,j) = beta*c(i,j)
        end do
      end do
c*ksr* end tile
```

```
c*ksr* user tile(j,private=(i,k,temp),tilesize=(j:nvect),teamid=iteam)
        do j = 1, n
          do k = 1, n
            temp = alpha*b(k,j)
            do i = 1,n
              c(i,j) = c(i,j) + a(i,k)*temp
            end do
          end do
        end do
c*ksr* end tile
```

Just as in the jik column oriented algorithm, the jki column oriented algorithm processes a column of C for each iteration of the outer loop. The major difference between the two algorithms is that the jki column oriented version reads A in a contiguous fashion rather than noncontiguously. The difference is reflected in algorithm performance. The jki column oriented matrix-matrix multiply outperforms the jik version by a factor of 2 (see Table 2.11).

2.3.3 ijk Row oriented matrix-matrix product

The ijk row oriented matrix-matrix product algorithm is:

```
        nvect = n/nbr_of_processors
c*ksr* user tile(i,private=(j,k,sum),tilesize=(i:nvect),teamid=iteam)
        do i = 1, n
          do j = 1, n
            sum = 0.0
            do k = 1,n
              sum = sum + a(i,k)*b(k,j)
            end do
            c(i,j) = beta*c(i,j) + alpha*sum
          end do
        end do
c*ksr* end tile
```

The algorithm calculates a row of C for each iteration of the outer loop. Like the column oriented version, the row oriented algorithm parallelizes well on the KSR1. See Table 2.11.

2.3.4 Optimized block matrix-matrix product

The optimized block matrix-matrix product algorithm, blocks C into p blocks where p is the number of processors. The code that implements the algorithm is KSR proprietary and uses the parallel region PRESTO directive to give each processor a block of C to update. Local variables are extensively used to prevent writes to the same subcache page while calculations are being performed. The code is extremely convoluted. The main subroutine which calculates $C = \beta C + \alpha AB$ for a block of C is 8426 lines long. The subroutine works roughly as follows:

- Outer loop:

    1. Read in an up to 8 $\times$ 6 element section of C into local variables and multiply these local variables by β.
    2. Inner loop:
        (1) Read in up to 8 elements of a column of A and up to 2 elements of a row of B into local variables.
        (2) Update the appropriate local variables for C by adding $\alpha \times fa_n \times fb_n$ where fa_n and fb_n are the appropriate column and row local variables for A and B respectively.
        (3) Continue this process until the local variables for the section of C have been fully updated (i.e., $C = \beta C + \alpha AB$ for the section of C being evaluated).
    3. Update the section of C with the values in the appropriate local variables.
    4. Continue this process until the block of C has been updated for this processor.

By reading A and B in a little bit at a time to calculate and update elements of the 8×6 section of C, the designers of this code were attempting to interleave operations that would require cache paging with arithmetic operations. On a sophisticated machine like the KSR1, data retrieval can be performed at the same time that arithmetic operations take place as long as the data being retrieved is not being used within the arithmetic operation.

Table 2.13: Matrix-matrix product first reference performance results (averaged over 10 runs).

	Column oriented			Row oriented		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	541.634			543.482		
2	272.313	1.99	0.99	272.111	2.00	1.00
4	139.041	3.90	0.97	137.030	3.97	0.99
8	70.311	7.70	0.96	68.551	7.93	0.99
16	36.311	14.92	0.93	35.529	15.30	0.96

This code tries to make these kinds of optimizations in an effort to keep the machine as busy as possible. Because of the extreme "messiness" of the code, perhaps the designers of the KSR1 need to develop a "smarter" compiler which can perform this kind of optimization automatically.[2]

The results in Table 2.12 show that performance degrades as the number of processors used grows for matrices A, B, and C of size 512×512. However, the execution times show this algorithm to be more than an order of magnitude faster than the other two.

2.3.5 First reference costs of matrix-matrix product

The KSR1 does not physically allocate memory to a data item until after the first reference to that item. For this reason, performance results of a row oriented matrix-matrix product where $C = \alpha AB$ (and C has not been referenced before running the algorithm) were expected to be poor in comparison to the column oriented version. However, performance of the two algorithms is similar (see Table 2.13). The quantity of work involved in multiplying a row of A by a column of B where matrix sizes are all 512×512 outweighs the time spent by the system physically allocating space to C as each $C_{i,j}$ is computed.

[2]The optimized block oriented matrix-matrix product code is available in the /usr/local/examples/KSRlib/MXM directory of the KSR1 at Oak Ridge National Laboratories.

Chapter 3

PARTIAL DIFFERENTIAL EQUATION APPLICATION

A differential equation is an equation involving an unknown function and its derivatives. A differential equation involving only one independent variable is called an "ordinary differential equation" or ODE. When two or more independent variables are involved in the differential equation, the equation is called a "partial differential equation" or PDE. The order of a differential equation refers to the order of the highest derivative appearing in the equation.

A general solution to a partial differential equation is a set of all possible solutions. In order to find a unique particular solution for a set or domain (D) of independent variables, certain additional conditions must be known about the function and/or its derivatives.

The application described in this chapter solves the Dirichlet boundary value problem

$$\begin{aligned} Lu &= g \text{ in } D, \\ u &= h \text{ on } \partial D, \end{aligned}$$

where D is a rectangular domain in $\Re^2$ with boundary ∂D, L is a linear second order, self-adjoint elliptic operator, and g and h are known functions of x and y.

3.1 Basic Algorithms And Data Structures

The Dirichlet boundary value problem is solved by using a finite difference discretization to generate a linear system which is then solved through some iterative or direct method. The linear system solver implemented in this thesis is the conjugate gradient method (an iterative Krylov subspace method). From Birkhoff and Lynch [4], a standard second order

accurate finite difference discretization yields a linear system $Ax = b$ where A is symmetric and positive definite. The symmetric and positive definite properties of A are required for the conjugate gradient method to work. Additionally, the generated matrix A is sparse, a property which Reid [45] showed could be exploited by the conjugate gradient method.

To improve performance of the conjugate gradient algorithm, a preconditioner may be applied to the problem. A preconditioner is a matrix that approximates the inverse of the original matrix but also improves the conditioning and/or clusters the eigenvalues of the preconditioned system facilitating a faster convergence rate. Two different preconditioners are examined in this chapter, the incomplete Cholesky (IC) preconditioner (taken from Golub and Van Loan [19]) and a variation of Bramble, Pasciak, and Schatz (BPS) [6] domain decomposition preconditioner.

The ELLPACK software package is used to generate the linear systems of equations. ELLPACK is an elliptic boundary value solver written in FORTRAN. Documentation for ELLPACK may be found in Rice and Boisvert [47]. The conjugate gradient algorithm and the preconditioner code implemented for this thesis is incorporated into the ELLPACK package.

3.1.1 Finite differences

Finite difference methods may be applied to find a numerical solution to a boundary value problem. Instead of attempting to find the continuous solution u, finite difference methods return a numerical solution in the form of a vector U which approximates the solution u at a discrete set of points within the domain D.

The first step in a finite difference discretization of a boundary value problem is to select a discrete set of points within D. These points are chosen so that they are spaced across D to form a grid of points. We consider only uniform grids in this thesis. For rectangular D with independent variables x and y, each grid point is a distance h_x from its neighbors on the x-axis of the grid and h_y from its neighbors on the y-axis. Thus, the rectangular domain D is overlaid with a $N_x \times N_y$ grid of points. On the edge of the grid, the points are

on ∂D (the boundary of D) and the solution u is known at these points. For the interior $(N_x - 2) \times (N_y - 2)$ grid points, the solution is not known. The interior grid points are then given some ordering. Typically, the grid point ordering is from left to right, bottom to top.

After a discrete set of grid points is selected and ordered, finite difference formulas are used to approximate the partial derivatives of u at the interior grid points. Again using a rectangular domain with independent variables x and y, the linear system $Ax = b$ is built where each row of the system is the finite difference approximation $L_n(U_{i,j}) = g(x_i, y_j)$ of the PDE at the point (x_i, y_i), and where $U_{i,j}$ approximates the true solution $u(x_i, y_j)$ at each interior grid point (x_i, y_j) for $i = 2, ..., N_x - 1$ and $j = 2, ...N_y - 1$. The discrete linear operator L_n is a finite difference approximation of the continuous operator L. The equations and unknowns of $Ax = b$ are ordered using the grid point ordering.

The standard second order finite difference discretization that ELLPACK uses to generate the linear systems is called five-point star. Five-point star discretizes equations of the form

$$Lu = au_{xx} + cu_{yy} + du_x + eu_y + fu = g \tag{3.1}$$

or

$$Lu = (au_x)_x + (cu_y)_y + fu = g \tag{3.2}$$

defined on a rectangular domain D, subject to the boundary conditions

$$Mu = s(\partial u/\partial n) + ru = pu_x + qu_y + ru = t \text{ on } D_1 \tag{3.3}$$

and

$$\text{periodic on } D_2, \tag{3.4}$$

where $D_1 \cup D_2$ is the boundary of D and $a, c, d, e, f, g, p, q, r, s,$ and t are smooth functions of x and y.

To discretize Equation 3.1 at grid point (x_i, y_j), five-point star approximates the first and second derivatives using the following formulas (for a nonuniform grid):

$$
\begin{aligned}
u_x &= \frac{U_{i+1,j} - U_{i-1,j}}{h_x(1+\theta x)}, \\
u_{xx} &= \frac{\theta x U_{i+1,j} - (1+\theta x)U_{i,j} + U_{i-1,j}}{h_x^2 \theta x(1+\theta x)/2}, \\
u_y &= \frac{U_{i,j+1} - U_{i,j-1}}{h_y(1+\theta y)}, \\
u_{yy} &= \frac{\theta y U_{i,j+1} - (1+\theta y)U_{i,j} + U_{i,j-1}}{h_y^2 \theta y(1+\theta y)/2},
\end{aligned}
$$

where $h_x = x_{i+1} - x_i$, $h_y = y_{i+1} - y_i$, $\theta x = (x_i - x_{i-1})/h_x$, and $\theta y = (y_i - y_{i-1})/h_y$.

For Equation 3.2, five-point star approximates the second derivatives using:

$$
\begin{aligned}
(au_x)_x &= \frac{z_1 U_{i+1,j} - (z_1+z_2)U_{i,j} + z_2 U_{i-1,j}}{h_x^2 \theta x(1+\theta x)/2}, \\
(cu_y)_y &= \frac{z_3 U_{i,j+1} - (z_3+z_4)U_{i,j} + z_4 U_{i,j-1}}{h_y^2 \theta y(1+\theta y)/2},
\end{aligned}
$$

where h_x, h_y, θx, and θy are as defined above and

$$
\begin{aligned}
z_1 &= a((x_{i+1} + x_i)/2, y_j), \\
z_2 &= a((x_i + x_{i-1})/2, y_j), \\
z_3 &= c(x_i, (y_{j+1} + y_j)/2), \\
z_4 &= c(x_i, (y_j + y_{j-1})/2).
\end{aligned}
$$

The result of discretizing Equations 3.1 and 3.2 at each of the $n = (N_x - 2) \times (N_y - 2)$ grid points is an $n \times n$ linear system $Ax = b$. The matrix A is always symmetric for Equation 3.2. For Equation 3.1, A is symmetric if a and b are constant and $c = d = 0$.

3.1.2 Sparse matrix representation

ELLPACK uses a sparse matrix representation to store matrices. The actual number of columns used to store a matrix is the maximum number of nonzero entries in any row of the matrix A. When five-point star discretization is used, there are at most five nonzero

entries in any row, so the physically stored coefficient matrix will be $n \times 5$ where n is the number of equations and unknowns.

In order to differentiate which element in a row of the coefficient matrix belongs to which "true" column, ELLPACK uses a column matrix of the same size as the coefficient matrix. Each element of the column matrix stores the "true" column position of the corresponding element in the coefficient matrix. For example, let A be a 4×4 matrix:

$$A = \begin{pmatrix} 4 & -1 & -2 & 0 \\ -1 & 4 & 0 & -2 \\ -3 & 0 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{pmatrix}.$$

One possible representation of the ELLPACK coefficient and column matrices for A is:

$$Coef = \begin{pmatrix} 4 & -1 & -2 \\ 4 & -1 & -2 \\ 4 & -3 & -1 \\ 4 & -1 & 0 \end{pmatrix} \quad \text{and} \quad Jcoef = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 1 & 4 \\ 4 & 3 & 0 \end{pmatrix},$$

respectively.

Should the ordering of bottom to top, left to right of equations not be desired, ELLPACK provides two vectors to facilitate reordering without physically moving the data elements in the matrix A. One vector ($iendx$) stores the new order of the rows. The other vector ($iundx$) stores the new column reordering. These vectors are defined so that the ith row of the reordered matrix is actually stored in row $iendx(i)$ of the coefficient matrix and the jth unknown in the original system is the $iundx(j)th$ unknown in the reordered system. Hence, if A is the original (full) matrix and B the reordered matrix, then $B(i, iundx(j)) = A(iendx(i), j)$.

3.1.3 Conjugate gradients

The conjugate gradient method solves the equation $Ax = b$ by minimizing the convex function

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b,$$

where $b \in \Re^n$ and $A \in \Re^{n \times n}$ is a positive definite symmetric matrix. The minimum value of ϕ at $x = A^{-1}b$ is

$$\phi(A^{-1}b) = -b^T A^{-1}b/2.$$

Calculating A^{-1} is extremely expensive for general matrices, so the conjugate gradient method employs a modified steepest descent method to minimize ϕ along linearly independent A-conjugate search directions. A vector p_k is A-conjugate to vectors p_1, ..., p_{k-1} if

$$P_{k-1}^T Ap_k = 0$$

where $P_{k-1} = [p_1, ..., p_{k-1}] \in \Re^{n \times k-1}$. A more in depth description of the conjugate gradient method may be found in Golub and Van Loan [19].

The conjugate gradient method works well when either A is well conditioned or has just a few distinct eigenvalues. It can be shown that at the *kth* iteration, the error is bounded by

$$\|x - x_k\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1)}{\sqrt{\kappa} + 1}\right)^k$$

where $\|x\|_A = \sqrt{x^T Ax}$ and $\kappa = \kappa_2(A)$, the 2-norm condition number of A. This error bound reflects how the conjugate gradient method is dependent upon the conditioning of the system.

The conjugate gradient algorithm is as follows (Golub and van Loan [19]):

$k = 0; x = 0, r = b; \rho = \|r\|_2^2$
$tol = \epsilon \times \sqrt{\rho}$
while $(\sqrt{\rho} > tol) \wedge (k < n)$
    $k = k + 1$
    if $k = 1$
      $p = r$
    else
      $\beta = \rho/(\text{old } \rho)$
      $p = r + \beta p$
    end
    $w = Ap$
    $\alpha = \rho/p^T w$
    $x = x + \alpha p$
    $r = r - \alpha w$
    old $\rho = \rho$
    $\rho = \|r\|_2^2$
end

The above algorithm requires space for the matrix A and five vectors (r, b, w, p, and x). The expensive operations within the loop are one matrix-vector multiply, three vector updates, and two vector dot products. The variable n refers to the dimension of A where A is an $n \times n$ matrix. The variable ϵ contains the degree of accuracy that is desired.

3.2 Preconditioning

Preconditioners are used to improve the conditioning of the linear system $Ax = b$. Ideally, a preconditioner is a "cheaply" produced "good" estimate of A^{-1}. In order to cheaply derive the preconditioner, the preconditioner is typically a approximation of A which facilitates a solve to be performed at minimal cost. The solve implicitly applies A^{-1} to the problem. The preconditioner, when applied to the problem, should improve the conditioning of the problem thus reducing the number of iterations for convergence to the solution.

Because the conjugate gradient method only applies to symmetric, positive definite systems, the preconditioned system must also be symmetric positive definite. Following Golub

and van Loan [19], if we apply a symmetric positive definite preconditioner C to the linear system $Ax = b$, the preconditioned system becomes $\tilde{A}\tilde{x} = \tilde{b}$ where $\tilde{A} = C^{-1}AC^{-1}$, $\tilde{x} = Cx$, and $\tilde{b} = C^{-1}b$. By replacing A, x, and b with $\tilde{A}, \tilde{x}$, and $\tilde{b}$, respectively, the preconditioned conjugate gradient method is derived. This complicates the conjugate gradient algorithm by introducing a number of solves since C^{-1} is not known. However, by setting $M = C^2$, the following version of the preconditioned algorithm may be derived (Golub and van Loan [19]):

$k = 0; x = 0, r = b, \rho = \|r\|_2^2$
$tol = \epsilon \times \sqrt{\rho}$
while $(\sqrt{\rho} > tol) \wedge (k < n)$
    Solve $Mz = r$
    $\varrho = r^T z$
    $k = k + 1$
    if $k = 1$
      $p = z$
    else
      $\beta = \varrho/(\text{old } \varrho)$
      $p = z + \beta p$
    end
    $w = Ap$
    $\alpha = \varrho/p^T w$
    $x = x + \alpha p$
    $r = r - \alpha w$
    old $\varrho = \varrho$
    $\rho = \|r\|_2^2$
end

The preconditioned algorithm adds one solve with M and one extra dot product operation to the cost of the conjugate gradient algorithm. The two preconditioners implemented in this thesis are the incomplete Cholesky (IC) preconditioner and the Bramble-Pasciak-Schatz (BPS) domain decomposition preconditioner. The IC preconditioner does not parallelize well, while the BPS preconditioner is ideal for a parallel machine.

3.2.1 Incomplete Cholesky preconditioning

Incomplete Cholesky (IC) preconditioning works well on single processor systems. It can significantly reduce the number of iterations required for the conjugate gradient algorithm to converge to a solution. However, neither the factorization of the preconditioner M nor the forward and backward solves needed to solve $Mz = r$ parallelize well. The general idea of IC preconditioning is to use Cholesky factorization without "fill in" to build a preconditioner for A. The Cholesky factorization of A is $A = LL^T$ where L is a lower triangular matrix with positive diagonal elements. The following algorithm generates an IC preconditioner (Golub and van Loan [19]):

```
for k = 1, n
    A_{k,k} = sqrt(A_{k,k})
    for i = k + 1, n                ! This loop may be parallelized.
        if A_{i,k} ≠ 0
          A_{i,k} = A_{i,k}/A_{k,k}
        end
    end
    for j = k + 1, n                ! This nested loop may be parallelized.
        for i = j, n
          if A_{i,j} ≠ 0
            A_{i,j} = A_{i,j} - A_{i,k}A_{j,k}
          end
        end
    end
end
```

As shown above, only two loops may be parallelized during the factorization. These two loops do not parallelize well because the iteration space does not remain constant. When k is close to n, very few iterations are required to complete the loop and so there will be more processor idle time.

The performance of the solve ($Mz = r$) is also bad on a parallel machine. The solve involves one forward solve ($Lw = r$) and then one backward solve ($L^T z = w$) where $M = LL^T$. As seen in Section 2.2.3, only the inner loops of the two solves may be parallelized and the performance of the solves on a parallel machine degrades rapidly as the number

of processors grows. Five-point star discretization produces a sparse banded matrix of bandwidth $\sqrt{n}$ where n is the number of equations and unknowns in the system. On a sparse banded matrix, there is even less work for the inner loop of the solves to do because of the bandwidth of the system. Heath, Ng, and Peyton [23] concur with this observation and add that the pipelining techniques used to improve solve performance for dense matrices do not work well on sparse matrices. For the reasons cited in this paragraph, the IC solves were not parallelized.

An incomplete block Cholesky preconditioner would parallelize much better than a non-blocked IC preconditioner in terms of both the factorization and solve steps as shown by Bagheri, Ilin, and Scott [2]. In [2], the authors use a strip decomposition of the grid to block the domain and then factor the block subsystems using IC factorization on the KSR1. When applied to the PDE, Bagheri, Ilin and Scott achieve superlinear speedup on the KSR1 for a grid of size $160 \times 160 \times 160$ with up to 64 processors. Poole and Ortega [44] and Van der Vorst [52] show that the incomplete block Cholesky preconditioner proposed by Concus, Golub, and Muerant [8] works well on vector supercomputers. Because incomplete block Cholesky does not take advantage of the information gained from a coarse discretization of the problem, a domain decomposition preconditioner should perform better than a incomplete block Cholesky preconditioner in terms of reducing the number of iterations to converge.

As a side note, the ELLPACK sparse matrix storage makes implementation of the column oriented solve more complex than that of the row oriented solve. In a column oriented solve, for each iteration of the outer loop, a column of A is examined. With ELLPACK sparse matrix storage, the physical column position in the coefficient matrix bears little relation to the true column position in the full matrix. In order to find the correct column element for a row of A, a search of the corresponding row in the ELLPACK column matrix must be performed. The row oriented solve does not suffer from this problem in that it processes a row of A for each iteration of the outer loop and the order in which the row elements are processed is not important—hence no search is necessary.

3.2.2 Domain decomposition preconditioning

Domain decomposition preconditioners parallelize well because they consist of submatrices that when put together make up most of the preconditioner matrix. The submatrices are built based on a decomposition of the original problem domain D into rectangular subdomains. Each submatrix corresponds to the equations and unknowns of a single subdomain of D. The submatrices may be distributed among processors when performing the most time expensive step, the solve, in the preconditioned conjugate gradient algorithm. Additionally, forming and factoring the preconditioner parallelizes well for the same reason.

The domain decomposition preconditioner used in this thesis is a simplified version of the well known preconditioner developed by Bramble, Pasciak, Schatz (BPS) [6]. Although this simplified version of BPS does not reduce the number of iterations as effectively as the original algorithm, it does require the same data structures and operations as the original algorithm. For the purpose of this thesis—studying parallelism and data movement—the version implemented here is sufficient.

To define the BPS preconditioner, the equations and unknowns of the discrete system are reordered. Instead of using the natural grid point ordering of bottom to top, left to right, the grid is divided into $nsubx \times nsuby$ blocks or subdomains where $nsubx$ is the number of subdomains in the x direction and $nsuby$ is the number of subdomains in the y direction. The grid points that fall within the interior of the subdomains are numbered first, then the grid points on the subdomain boundaries are numbered, and lastly the crosspoints or intersection points of subdomain boundaries are numbered. The overall order of numbering within subdomains and boundaries is bottom to top, left to right. Figure 3.1 illustrates the grid point reordering of a 7×7 grid broken into four subdomains. In this example, the subdomain interiors are {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}}, the subdomain west boundaries are {{17,18},{19,20}}, the subdomain south boundaries are {{21,22},{23,24}}, and the crosspoint is {25}. Points beginning with a "b" are domain boundary points.

b18	b19	b20	b21	b22	b23	b24
b16	11	12	20	15	16	b17
b14	9	10	19	13	14	b15
b12	21	22	25	23	24	b13
b10	3	4	18	7	8	b11
b8	1	2	17	5	6	b9
b1	b2	b3	b4	b5	b6	b7

Figure 3.1: Subdomain grid point numbering example.

The grid point reordering produces a reordered A of the form:

$$A = \begin{pmatrix} A_I & A_{IB} & 0 \\ A_{IB}^T & A_B & A_{BC} \\ 0 & A_{BC}^T & A_C \end{pmatrix}. \tag{3.5}$$

A_I in Equation 3.5 represents the subdomain interiors, A_B the subdomain boundaries, and A_C the crosspoints. Both A_B and A_I are block diagonal with $(nsubx-1) \times nsuby + nsubx \times (nsuby-1)$ blocks and $nsubx \times nsuby$ blocks, respectively. The remaining submatrices A_{IB} and A_{BC} represent the interactions between the interiors and the boundaries and between the boundaries and the crosspoints, respectively. Because five-point star discretization is used, the subdomains do not interact with the crosspoints and hence the upper right and lower left corners in Equation 3.5 are zero matrices.

Once A has been reordered, the preconditioner M is defined as follows:

$$M = \begin{pmatrix} A_I & A_{IB} & 0 \\ A_{IB}^T & \tilde{A}_B + A_{IB}^T A_I^{-1} A_{IB} & 0 \\ 0 & 0 & \tilde{A}_C \end{pmatrix} \tag{3.6}$$

where $\tilde{A}_B$ is an approximation of A_B and $\tilde{A}_C$ is derived from a coarse grid discretization of

the crosspoints.[1] M^{-1} is as follows:

$$M^{-1} = \begin{pmatrix} I & -A_I^{-1}A_{IB} & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} A_I^{-1} & 0 & 0 \\ 0 & \tilde{A}_B^{-1} & 0 \\ 0 & 0 & \tilde{A}_C^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ -A_{IB}^T A_I^{-1} & I & 0 \\ 0 & 0 & I \end{pmatrix}. \quad (3.7)$$

BPS preconditioner storage

The BPS preconditioner is stored in three arrays using symmetric band LINPACK storage format. The three arrays are factored using the standard LINPACK band Cholesky factorization.

The subdomain interiors are stored in an array of size

$$(nx - 2 + 1) \times (nx - 2)(ny - 2) \times (nsubx \times nsuby)$$

where there are $nsubx \times nsuby$ subdomains of dimension $nx \times ny$. The bandwidth of the subdomain interiors is $2(nx - 2) + 1$, but because of symmetry storage for only $nx - 2 + 1$ diagonals is required. The subdomain interiors have $(nx - 2)(ny - 2)$ equations.

The subdomain boundaries are stored in the remaining two arrays. One array stores the "south" or lower boundary and the other stores the "west" or left boundary of the subdomains. The south boundary array is of dimension

$$2 \times (nx - 2) \times (nsubx \times nsuby)$$

where 3 is the bandwidth (due to symmetry only 2 rows are required) and $nx - 2$ is the number of equations. Although there are only $(nsubx - 1) \times nsuby$ south subdomain boundaries, $nsubx \times nsuby$ is used to simplify the identification of which south boundary goes with which subdomain. The west subdomain boundaries are similarly stored, except that there are $ny - 2$ equations instead of $nx - 2$ equations.

A_{IB} and A_C are implicitly derived from the original A. For example, if a subdomain has a south boundary, then the first $nx - 2$ equations or rows will have corresponding elements

[1] $\tilde{A}_B$ and $\tilde{A}_C$ are not implemented and instead A_B and A_C are used.

in A_{IB}. Using the ELLPACK reordering vectors, the appropriate elements of A_{IB} can easily be found in A. A_C is found in the lower right $(nsubx-1)(nsuby-1)\times(nsubx-1)(nsuby-1)$ corner of the reordered A and it consists only of the diagonal.

BPS preconditioner solve implementation

The implementation of the solve ($Mz = r$) in the preconditioned conjugate gradient algorithm is:

1. Solve $A_I z_I^{(1)} = r_I$ for $z_I^{(1)}$.
2. Solve $A_B z_B = r_B - A_{IB}^T z_I^{(1)}$ for z_B.
3. Solve $A_C z_C = r_C$ for z_C.
4. Solve $A_I z_I^{(2)} = -A_{IB} z_B$ for $z_I^{(2)}$.
5. Set $x_I = x_I^{(1)} + x_I^{(2)}$.

The two solves involving the subdomain interior dominate the preconditioner implementation since the number of equations or grid points within the subdomains is much larger than those on the boundaries or crosspoints.

For coding purposes, the second and third steps can be combined into one loop and the fourth and fifth steps combined into another loop. The preconditioner pseudo code is as follows:

1. $z = r$
2. In parallel, solve $A_I z_I = z_I$.
3. In parallel,
   (1) Update right hand side of boundary systems (i.e., $z_B = z_B - A_{IB}^T \times z_I$).
   (2) Solve $A_B z_B = z_B$.
   (3) Solve $A_C z_C = z_C$.

4. In parallel,

   (1) Update right hand side of interior systems (i.e., $w_I = -A_{IB} \times z_B$).

   (2) Solve $A_I w_I = w_I$.

   (3) $z_I = z_I + w_I$.

Noncontiguous vector implementation

To improve performance of the preconditioner solve, noncontiguous vectors are broken up by subdomain interiors, boundaries, and crosspoints. The basic structure for a noncontiguous vector broken up by subdomains is as follows:

- For each subdomain interior, there will be a mini-vector of length $(nx-2)(ny-2)$ where $nx \times ny$ is the size of the subdomain.
- For each subdomain with a south boundary, there will be a mini-vector of length $nx-2$ where the subdomain south boundary is of length nx.
- For each subdomain with a west boundary, there will be a mini-vector of length $ny-2$ where the subdomain west boundary is of length ny.
- For each subdomain with a south-west crosspoint, there will be a mini-vector of length 1.

3.3 Performance

Performance of the preconditioned and unpreconditioned conjugate gradient method was measured for the most expensive operations. These operations include the matrix-vector product, dot product, daxpy, and solve (for the preconditioned conjugate gradient). The performance is further broken down by whether vectors are noncontiguous or contiguous.

Noncontiguous vectors are broken up across the total number of processors for the IC preconditioned and unpreconditioned conjugate gradient. For domain decomposition, noncontiguous vectors are broken up over subdomain interiors, boundaries, and crosspoints.

Table 3.1: Conjugate gradient (averaged over 5 runs in nonpeak system hours).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid (129 its.)						
1	61.612			66.952		
2	32.757	1.88	0.94	36.405	1.84	0.92
4	17.540	3.51	0.88	20.904	3.20	0.80
8	10.688	5.76	0.72	12.828	5.22	0.65
16	7.524	8.19	0.51	9.284	7.21	0.45
32	7.408	8.32	0.26	9.380	7.14	0.22
89 x 89 grid (248 its.)						
1	401.97			436.27		
2	211.16	1.90	0.95	235.94	1.85	0.93
4	104.48	3.85	0.96	127.51	3.42	0.86
8	56.02	7.18	0.90	74.37	5.87	0.73
16	32.89	12.22	0.76	49.73	8.77	0.55
32	23.47	17.13	0.54	36.60	11.92	0.37

Performance is compared between a grid of dimension 49×49 and one of 89×89. These grid sizes were selected because both could be broken up into 4×4 and 8×8 subdomains both of which may be evenly distributed among powers of 2 processors.

The problem being solved is:

$$Lu = (e^{xy}u_x)_x + (e^{-xy}u_y)_y - 1/(1+x+y)u = g(x,y)$$

where

$$\begin{aligned} g(x,y) &= .75(e^{xy}e^{xy}\sin(\pi y)((2y^2-\pi^2)\sin(\pi x) + 3\pi y\cos(\pi x)) \\ &\quad +\pi\sin(\pi x)(x\cos(\pi y) - \pi\sin(\pi y)) - e^{xy}\sin(\pi x)\sin(\pi y)/(1+x+y)) \end{aligned}$$

and $u(x,y) = 0$ for $x = 0$ or 1 and $y = 0$ or 1.

This problem is a good test problem because it takes the conjugate gradient method a relatively long time to converge.

Tables 3.1—3.4 give the overall performance of the unpreconditioned and preconditioned conjugate gradient methods. Although both preconditioned methods significantly reduce

Table 3.2: Incomplete Cholesky preconditioned conjugate gradient (averaged over 5 runs in nonpeak system hours).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid (37 its.)						
1	336.48			333.06		
2	196.18	1.72	0.86	189.09	1.76	0.88
4	120.65	2.79	0.70	113.46	2.94	0.73
8	81.47	4.13	0.52	74.53	4.47	0.56
16	63.63	5.29	0.33	57.21	5.82	0.36
89 x 89 grid (72 its.)						
1	3675.0			3670.6		
2	1982.5	1.85	0.93	1964.0	1.87	0.93
4	1131.9	3.25	0.81	1071.4	3.43	0.86
8	696.8	5.27	0.66	641.9	5.72	0.72
16	483.9	7.59	0.48	433.8	8.46	0.53

the iteration count, neither surpasses the unpreconditioned conjugate gradient in providing a faster total execution time. The BPS preconditioner would have reduced the number of iterations more if we had used better choices for $\tilde{A}_B$ and $\tilde{A}_C$.

Unpreconditioned conjugate gradient does not parallelize well on the KSR1. The amount of storage required to store the ELLPACK coefficient and column matrices as well as the 5 vectors is $3 \times N - 2 \times 5$ for a $N \times N$ grid. On the KSR1, this translates to an increased likelihood that paging will occur. Furthermore, the matrix-vector product is a row oriented algorithm which means that most of A must be in memory while the product operation is occurring. To improve the algorithm, A should be broken up into smaller pieces so that the distance from one row element to the next is not so far—thus reducing paging activity.

Table 3.3: BPS preconditioned conjugate gradient on a 49 x 49 grid (averaged over 5 runs in nonpeak system hours).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
4 x 4 subdomains (35 iterations)						
1	117.03			119.48		
2	59.87	1.96	0.98	62.34	1.93	0.96
4	30.89	3.79	0.95	32.49	3.68	0.92
8	16.37	7.15	0.89	17.78	6.72	0.84
16	9.19	12.74	0.80	10.63	11.24	0.70
8 x 8 subdomains (51 iterations)						
1	109.53			110.16		
2	56.33	1.95	0.97	57.99	1.90	0.95
4	29.37	3.73	0.93	30.48	3.61	0.90
8	16.17	6.77	0.85	17.28	6.37	0.80
16	9.27	11.82	0.74	11.00	10.02	0.63
32	7.648	14.32	0.45	7.99	13.79	0.43

IC preconditioning significantly increases execution time. The poor performance is attributable to two causes:

1. The solve in the IC preconditioner is not parallelized for reasons discussed in Section 3.2.1.

2. The space requirements for IC appear to be a large bottle-neck on the KSR1.

The preconditioner is stored in a matrix of the same size as the ELLPACK coefficient matrix for A. This extra (contiguous) storage can and does seem to affect the preconditioned algorithm adversely due to cache paging.

BPS domain decomposition preconditioning provides the greatest speedup and efficiency of the three methods tested on the KSR1. With a correct version of BPS, BPS preconditioned conjugate gradient should prove to be the best of the three methods to use on poorly conditioned problems in terms of reducing execution time.

The code versions using noncontiguous vectors for BPS domain decomposition preconditioning and unpreconditioned conjugate gradient outperform the contiguous vector versions.

Table 3.4: BPS preconditioned conjugate gradient on a 89 x 89 grid (averaged over 5 runs in nonpeak system hours).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
4 x 4 subdomains (46 iterations)						
1	816.62			836.02		
2	413.97	1.97	0.99	416.69	2.01	1.00
4	209.62	3.90	0.97	211.84	3.95	0.99
8	108.43	7.53	0.94	110.37	7.58	0.95
16	56.30	14.51	0.91	59.77	13.99	0.87
8 x 8 subdomains (68 iterations)						
1	696.11			702.81		
2	351.04	1.98	0.99	357.93	1.96	0.98
4	178.69	3.90	0.97	187.27	3.75	0.94
8	91.96	7.57	0.95	98.33	7.15	0.89
16	50.01	13.92	0.87	54.61	12.87	0.80
32	29.22	23.83	0.75	34.81	20.19	0.63

The IC preconditioned conjugate gradient, however, exhibits the opposite results. Because the performance of the IC was so poor and obviously affected by cache thrashing,[2] the performance results can and should be disregarded except as an example of a poor preconditioner to implement on a parallel computer distributed memory computer.

The interesting result from the BPS domain decomposition preconditioning is that performance for the 8 x 8 subdomain decomposition is consistently better than the 4 x 4 subdomain decomposition for the larger grid size. This reflects how the KSR1 cache system works best with small chunks of data. The larger the chunk of data, even if it is read only data, the more likely that cache paging will adversely affect algorithm performance.

Sections 3.3.1—3.3.5 discuss performance of the four most expensive operations: the matrix-vector product, dot product, vector sum, and preconditioner factor and solve operations. Performance of the less expensive basic operations (i.e., vector dot product and

[2]Thrashing is when the computer spends more time paging information in and out of the cache than in computation.

vector sum) is better when vectors are broken up across processors than when vectors are broken up by subdomain. This is due to the uneven load balancing when vectors are broken up by subdomain. For example, the typical driver for a basic operation (matrix-vector product, vector dot product, and vector sum) is as follows:

```
c*ksr* user tile(ksub,private=(isub,jsub),tilesize=(ksub:itile),
c*ksr*&          teamid=iteam_id)
      do 10 ksub = 1, nsuby*nsubx
        jsub = (ksub-1)/nsubx + 1
        isub = 1 + mod(ksub-1,nsubx)
c       do basic operation on subdomain interior(isub,jsub)
        if (jsub .gt. 1) then                    ! South boundary
c         do basic operation on subdomain south boundary(isub,jsub)
        end if
        if (isub .gt. 1) then                    ! West boundary
c         do basic operation on subdomain west boundary(isub,jsub)
          if (jsub .gt. 1) then                  ! Crosspoint
c           do basic operation on crosspoint
          end if
        end if
      end do
c*ksr* end tile
```

Because not every subdomain has a south or west boundary or (less importantly) a crosspoint, work is unevenly distributed across the processors. Distributing vectors across subdomains enhances the performance of the BPS preconditioner solve. In the preconditioned conjugate gradient algorithm, the preconditioner solve is the most expensive operation. Hence, sacrificing performance of the less expensive operations is beneficial to overall algorithm performance.

3.3.1 Matrix-vector product

Performance results show that the code version using noncontiguous vectors outperform the code versions using contiguous vectors. Furthermore, the performance of the matrix-vector product operation using noncontiguous vectors broken up by subdomain slightly outperforms that of noncontiguous vectors broken up across processors. This last result is

Table 3.5: Nonreordered matrix-vector product (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid.						
1	0.245			0.241		
2	0.126	1.95	0.97	0.123	1.96	0.98
4	0.0655	3.74	0.94	0.0651	3.71	0.93
8	0.0370	6.62	0.83	0.0359	6.72	0.84
16	0.0209	11.76	0.74	0.0207	11.64	0.73
89 x 89 grid.						
1	0.836			0.827		
2	0.432	1.94	0.97	0.427	1.94	0.97
4	0.214	3.91	0.98	0.220	3.77	0.94
8	0.111	7.56	0.95	0.114	7.23	0.90
16	0.0597	14.01	0.88	0.0643	12.87	0.80

most likely because the special implementation of code for noncontiguous vectors broken up across subdomains does not use the ELLPACK column matrix or reordering vectors. Hence, no indirect references occur and less memory is necessary during the matrix-vector product.

The matrix-vector product for contiguous vectors is complicated by the ELLPACK sparse matrix storage. The algorithm for a reordered A is as follows:

```
c*ksr* user tile(i,private=(sum,jj,j),teamid=iteam_id,tilesize=(i:mtile))
      do 10 i = 1, n
        sum = 0.0
        do 9 j = 1, i1ncoe
          jj = i1idco(i,j)
          if (jj .eq. 0) goto 9
          jj = i1undx(jj)
          sum = sum + a(i,j) * x(jj)
    9   continue
        y(i1undx(i)) = sum
   10 continue
c*ksr* end tile
```

Table 3.6: Reordered matrix-vector product for 4 x 4 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid.						
1	0.220			0.274		
2	0.116	1.90	0.95	0.141	1.95	0.97
4	0.0591	3.73	0.93	0.0780	3.51	0.88
8	0.0324	6.79	0.85	0.0457	6.00	0.75
16	0.0183	12.02	0.75	0.0299	9.16	0.57
89 x 89 grid.						
1	0.755			0.909		
2	0.387	1.95	0.98	0.444	2.05	1.02
4	0.198	3.82	0.95	0.227	4.01	1.00
8	0.102	7.41	0.93	0.119	7.63	0.95
16	0.0563	13.40	0.84	0.0642	14.16	0.89

The algorithm for a nonreordered A is:

```
c*ksr* user tile(i,private=(j,sum,jj),teamid=iteam_id,tilesize=(i:mtile))
      do 20 i = 1, n
        sum = 0.0
        do 19 j = 1, i1ncoe
          jj = i1idco(i,j)
          if (jj .ne. 0) sum = sum + a(i,j)*x(jj)
   19   continue
        y(i) = sum
   20 continue
c*ksr* end tile
```

In the above two algorithms, A is an $n \times n$ matrix which is stored as $n \times i1ncoe$, `i1idco` is the ELLPACK column matrix, and `i1undx` is the ELLPACK column reordering vector. The IC preconditioned and unpreconditioned conjugate gradient methods use the nonreordered version of matrix-vector product. BPS domain decomposition preconditioning uses the reordered matrix-vector product. See Tables 3.5—3.7 for performance results.

Just as in the contiguous case, there are two versions of the noncontiguous matrix-vector product code. The version that uses a nonreordered A (for IC preconditioned and

Table 3.7: Reordered matrix-vector product for 8 x 8 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid.						
1	0.222			0.246		
2	0.114	1.95	0.97	0.128	1.92	0.96
4	0.0604	3.67	0.92	0.0647	3.81	0.95
8	0.0339	6.55	0.82	0.0370	6.65	0.83
16	0.0185	12.02	0.75	0.0224	10.97	0.69
89 x 89 grid.						
1	0.758			0.859		
2	0.385	1.97	0.98	0.435	1.97	0.99
4	0.198	3.84	0.96	0.226	3.79	0.95
8	0.103	7.38	0.92	0.116	7.43	0.93
16	0.0672	11.29	0.71	0.0624	13.77	0.86

unpreconditioned conjugate gradient methods) uses a driver to set up the pointers correctly for the mini-vectors and then calls code that looks similar to the code for the matrix-vector product using contiguous vectors and a nonreordered A. The version that uses a reordered A (for domain decomposition preconditioning) uses a driver across subdomains to call a special version of matrix-vector product code.

To simplify the matrix-vector product code when BPS preconditioning and noncontiguous vectors are used, A is physically reordered prior to starting the preconditioned conjugate gradient code. The physical reordering of A makes it simpler to identify which portions of A belong to which subdomain interior, boundary, or crosspoint. The code relies on a static method of column ordering in ELLPACK's sparse matrix storage by five-point star's discretization of the problem. The contribution from the grid point around which a discretization occurs (i.e., the diagonal of A) is expected to be found in the first column of the ELLPACK coefficient matrix. The contribution from the west, east, south, and north grid points to an equation (or row of A) are expected to be stored in the second, third, fourth, and fifth columns, respectively. A copy of the matrix-vector product code for non-

Table 3.8: Nonreordered vector dot product (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
	49 x 49 grid.					
1	0.0853			0.1307		
2	0.0455	1.88	0.94	0.0722	1.81	0.91
4	0.0250	3.42	0.85	0.0353	3.70	0.93
8	0.0143	5.98	0.75	0.0275	4.75	0.59
16	0.0110	7.76	0.49	0.0177	7.39	0.46
	89 x 89 grid.					
1	0.2931			0.4500		
2	0.1549	1.89	0.95	0.2418	1.86	0.93
4	0.0759	3.86	0.95	0.1270	3.55	0.89
8	0.0405	7.24	0.91	0.0706	6.38	0.80
16	0.0245	11.98	0.75	0.0455	9.89	0.62

contiguous vectors broken up over subdomains may be found in Appendix A.1. By relying on a static column ordering method, the column matrix is not needed. This improves algorithm performance on the KSR1 by decreasing the likelihood that cache paging will occur through the use of less memory.

3.3.2 Dot products

Performance for the dot product was evaluated for three different versions of code. One code version handles contiguous vectors, another noncontiguous vectors broken up across processors, and the last noncontiguous vectors broken up across subdomain boundaries. Tables 3.8—3.10 contain performance statistics for the different versions of dot operation.

As demonstrated in Section 2.1.1, the noncontiguous versions of vector dot product perform better than contiguous versions. The noncontiguous version where vectors were broken up across processors outperform the noncontiguous version where vectors where broken up by subdomain.

Even load balancing for the dot product operation where noncontiguous vectors are broken up by subdomain is not possible because not every subdomain has a south or west

Table 3.9: Reordered vector dot product for 4×4 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
			49 x 49 grid.			
1	0.1323			0.1290		
2	0.0688	1.92	0.96	0.0685	1.88	0.94
4	0.0377	3.51	0.88	0.0385	3.35	0.84
8	0.0244	5.41	0.68	0.0217	5.93	0.74
16	0.0181	7.33	0.46	0.0211	6.12	0.38
			89 x 89 grid.			
1	0.4327			0.4509		
2	0.2200	1.97	0.98	0.2247	2.01	1.00
4	0.1152	3.76	0.94	0.1164	3.87	0.97
8	0.0638	6.78	0.85	0.0639	7.05	0.88
16	0.0376	11.51	0.72	0.0391	11.53	0.72

boundary or a south-west crosspoint. For a grid broken up by $nsubx \times nsuby$ subdomains, only $(nsubx - 1) \times nsuby$ subdomains have a west boundary, $nsubx \times (nsuby - 1)$ subdomains have a south boundary, and $(nsubx - 1) \times (nsuby - 1)$ subdomains have a south-west crosspoint. Because most equations involve subdomain interiors, the uneven load balancing due to uneven distribution of subdomain boundary and crosspoint equations is not too detrimental to performance. Furthermore, the dot product operation is very "cheap" in terms of floating point operations. So, if sacrificing performance of this simple $O(n)$ operation improves the performance of the more complex operations (i.e., preconditioner solve and matrix-vector product), the sacrifice is beneficial to overall preconditioned conjugate gradient performance.

Table 3.10: Reordered vector dot product for 8 x 8 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid.						
1	0.1444			0.1291		
2	0.0757	1.91	0.95	0.0708	1.82	0.91
4	0.0416	3.47	0.87	0.0436	2.96	0.74
8	0.0206	7.01	0.88	0.0196	6.58	0.82
16	0.0168	8.57	0.54	0.0179	7.22	0.45
89 x 89 grid.						
1	0.4531			0.4414		
2	0.2291	1.98	0.99	0.2266	1.95	0.97
4	0.1185	3.83	0.96	0.1202	3.67	0.92
8	0.0565	8.01	1.00	0.0663	6.65	0.83
16	0.0381	11.88	0.74	0.0394	11.20	0.70

3.3.3 DAXPYS

The performance results of the daxpy operation for the three code versions (contiguous vector (CV), noncontiguous vector broken up across processors (NCVP), and noncontiguous vector broken up across subdomains (NCVS) show the NCVP version to be best (see Tables 3.11—3.13). The CV versions, in general, outperform the NCVS versions.

P13	P14	P15	P16
P9	P10	P11	P12
P5	P6	P7	P8
P1	P2	P3	P4

Figure 3.2: 4 x 4 subdomain mapping to 16 processors.

The performance statistics illustrate the point that NCVS performs better when the number of subdomains is larger than the number of processors. The work load is better balanced across processors when this condition is true. For example when 16 processors are used: NCVS for 4 x 4 subdomains will have one processor that works only on a subdomain

Table 3.11: Nonreordered daxpy (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
			49 x 49 grid.			
1	0.1345			0.1352		
2	0.0714	1.88	0.94	0.0723	1.87	0.94
4	0.0384	3.50	0.88	0.0451	3.00	0.75
8	0.0230	5.85	0.73	0.0238	5.69	0.71
16	0.0148	9.07	0.57	0.0155	8.70	0.54
			89 x 89 grid.			
1	0.4663			0.459		
2	0.2465	1.89	0.95	0.2432	1.89	0.94
4	0.1194	3.91	0.98	0.1273	3.60	0.90
8	0.0643	7.25	0.91	0.0691	6.64	0.83
16	0.0373	12.49	0.78	0.0431	10.64	0.67

interior; while the same processor for NCVS with 8 x 8 subdomains will work on four subdomain interiors, two west boundaries, two south boundaries, and one crosspoint. Figures 3.2 and 3.3 illustrate the subdomain to processor mapping for 16 processors.

P13	P13	P14	P14	P15	P15	P16	P16
P13	P13	P14	P14	P15	P15	P16	P16
P9	P9	P10	P10	P11	P11	P12	P12
P9	P9	P10	P10	P11	P11	P12	P12
P5	P5	P6	P6	P7	P7	P8	P8
P5	P5	P6	P6	P7	P7	P8	P8
P1	P1	P2	P2	P3	P3	P4	P4
P1	P1	P2	P2	P3	P3	P4	P4

Figure 3.3: 8 x 8 subdomain mapping to 16 processors.

Table 3.12: Reordered daxpy for 4 x 4 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
			49 x 49 grid.			
1	0.1343			0.1348		
2	0.0727	1.85	0.92	0.0731	1.84	0.92
4	0.0404	3.32	0.83	0.0400	3.37	0.84
8	0.0248	5.42	0.68	0.0251	5.36	0.67
16	0.0179	7.51	0.47	0.0160	8.43	0.53
			89 x 89 grid.			
1	0.4590			0.4626		
2	0.2374	1.93	0.97	0.2369	1.95	0.98
4	0.1272	3.61	0.90	0.1219	3.79	0.95
8	0.0660	6.95	0.87	0.0672	6.88	0.86
16	0.0415	11.07	0.69	0.0369	12.52	0.78

Table 3.13: Reordered daxpy for 8 x 8 subdomains (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
			49 x 49 grid.			
1	0.1445			0.1355		
2	0.0763	1.89	0.95	0.0735	1.84	0.92
4	0.0418	3.46	0.86	0.0397	3.41	0.85
8	0.0275	5.26	0.66	0.0254	5.33	0.67
16	0.0141	10.26	0.64	0.0142	9.55	0.60
			89 x 89 grid.			
1	0.4692			0.4506		
2	0.2384	1.97	0.98	0.2347	1.92	0.96
4	0.1238	3.79	0.95	0.1235	3.65	0.91
8	0.0674	6.97	0.87	0.0688	6.55	0.82
16	0.0398	11.80	0.74	0.0397	11.34	0.71

Table 3.14: Incomplete Cholesky preconditioner factorization (averaged over 5 runs).

	49 x 49 grid			89 x 89 grid		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	288.296			3375.750		
2	153.264	1.88	0.94	1731.667	1.95	0.98
4	82.057	3.51	0.88	867.191	3.89	0.97
8	44.933	6.426	0.80	454.566	7.43	0.93
16	28.620	10.073	0.63	252.184	13.39	0.84

Table 3.15: BPS preconditioner factorization (averaged over 5 runs).

	4 x 4 Subdomains			8 x 8 Subdomains		
P	Time	Speedup	Efficiency	Time	Speedup	Efficiency
			49 x 49 grid			
1	6.020			2.468		
2	3.084	1.95	0.98	1.312	1.88	0.94
4	1.588	3.79	0.95	0.668	3.69	0.92
8	0.824	7.31	0.91	0.360	6.86	0.86
16	0.432	13.94	0.87	0.208	11.87	0.74
			89 x 89 grid			
1	52.493			17.848		
2	26.904	1.95	0.98	9.060	1.97	0.99
4	13.416	3.91	0.98	4.624	3.86	0.97
8	6.912	7.59	0.95	2.368	7.54	0.94
16	3.592	14.61	0.91	1.244	14.35	0.90

3.3.4 Preconditioner factorization

Factorization of the BPS domain decomposition preconditioner is significantly less expensive and parallelizes better than that of the IC preconditioner. On an $N \times N$ grid where $N = N_x - 2 = N_y - 2$, the time complexity of IC factorization is $O\left(\frac{7}{2}N^2(N^2+1)\right)$. For the same grid using an $m \times m$ decomposition, there are approximately $N/m \times N/m$ equations in a subdomain interior. Thus, the time complexity of the BPS factorization is

$O\left(m^2\left(\frac{1}{2}\left(\frac{N}{m}\right)^4 - \frac{1}{3}\left(\frac{N}{m}\right)^3\right)\right)$.[3] The highest order term in IC factorization $\left(\frac{7}{2}N^4\right)$ is considerably larger than the highest order term of BPS factorization $\left(\frac{N^4}{2m^2}\right)$, especially as m grows. Additionally, IC preconditioning suffers on the KSR1 by requiring a large, contiguous memory space to store the preconditioner. The BPS preconditioner stores only A_I, $A_{B_{south}}$, and $A_{B_{west}}$ each of which is stored noncontiguously in separate matrices. Furthermore, BPS preconditioner factorization works on one subdomain at a time, and so does not need to "look" at the entire matrix—only little pieces of it.

Because BPS preconditioner factorization processes one subdomain at a time, work can be fairly evenly distributed across processors when the number of subdomains is divisible by the number of processors. The BPS parallelization is not "perfectly" distributed because there are $(nsubx-1)\times nsuby$ west boundary sub-matrices in $A_{B_{west}}$ and $nsubx\times(nsuby-1)$ south boundary sub-matrices in $A_{B_{south}}$ for a $nsubx \times nsuby$ subdomain decomposition. In IC, the iteration space varies and hence does not parallelize well (see Section 3.2.1).

Tables 3.14 and 3.15 contain performance statistics for factorization of the IC and BPS domain decomposition preconditioner, respectively.

3.3.5 Preconditioner solve

The IC preconditioner solve is not parallelized for reasons given in 3.2.1. The IC solve is fairly "cheap" in comparison to the BPS preconditioner solve. On an $N_x \times N_y$ grid with $m \times m$ decomposition for BPS preconditioning where $N = N_x - 2 = N_y - 2$, the time complexities of the IC and BPS solves are $O(10N^2)$ and $O\left(4\frac{N^3}{m} - 2N^2\right)$, respectively.[4] However, since the BPS preconditioner solve parallelizes well, the BPS solve outperforms the IC solve given a sufficient number of processors.

Just as in preconditioner factorization, the BPS preconditioner solve does not parallelize

[3] Derived from the LINPACK [14] band factorization time complexity formula of the subdomain interiors which dominate the BPS factorization.

[4] The BPS solve time complexity is derived from the two LINPACK [14] band solves of the subdomain interiors.

Table 3.16: Incomplete Cholesky preconditioner solve (averaged over the iterations).

Time on a 49 x 49 grid	Time on a 89 x 89 grid
0.663	2.276

Table 3.17: BPS preconditioner solve on a 49 x 49 grid (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
4 x 4 subdomains						
1	2.579			2.598		
2	1.305	1.98	0.99	1.338	1.94	0.97
4	0.662	3.89	0.97	0.676	3.84	0.96
8	0.339	7.60	0.95	0.353	7.36	0.92
16	0.175	14.73	0.92	0.185	14.01	0.88
8 x 8 subdomains						
1	1.528			1.541		
2	0.774	1.97	0.99	0.795	1.93	0.97
4	0.392	3.90	0.97	0.402	3.83	0.96
8	0.175	14.73	0.92	0.185	14.01	0.88
16	0.105	14.59	0.91	0.120	12.81	0.80

perfectly. Given a mapping of subdomains to processors, not every subdomain has a west or south boundary or a south-west crosspoint. From the pseudo code implementation of the solve in Section 3.2.2, the second and third parallel loops involve subdomain boundaries. In the second loop, the steps

1. Update right hand side of boundary systems (i.e., $z_B = z_B - A_{IB}^T \times z_I$).

2. Solve $A_B z_B = z_B$.

each involve subdomain boundaries. In the third loop, only the step $w_I = -A_{IB} \times z_B$ involves a subdomain boundary. Because the first loop (solve $A_I z_I = z_I$) dominates preconditioner solve, the BPS preconditioner solve parallelizes extremely well for noncontiguous vectors broken up over the subdomains (never dropping below 90% efficiency).

Tables 3.16 and 3.17—3.18 contain performance statistics for the IC and BPS domain decomposition preconditioner solves, respectively.

Table 3.18: BPS preconditioner solve on a 89 x 89 grid (averaged over the iterations).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
			4 x 4 subdomains			
1	14.571			14.763		
2	7.363	1.98	0.99	7.340	2.01	1.01
4	3.714	3.92	0.98	3.705	3.99	1.00
8	1.909	7.63	0.95	1.884	7.83	0.98
16	0.965	15.10	0.94	0.963	15.33	0.96
			4 x 4 subdomains			
1	8.107			8.134		
2	4.076	1.99	0.99	4.108	1.98	0.99
4	2.063	3.93	0.98	2.120	3.84	0.96
8	1.053	7.70	0.96	1.057	7.70	0.96
16	0.543	14.92	0.93	0.548	14.85	0.93

Chapter 4

SUMMARY AND CONCLUSIONS

4.1 Summary

To best use the KSR1 cache system, the distribution of data to processors should be in small chunks. Memory that is written should be distributed to processors in noncontiguous chunks that are physically located in separate cache subpages. The KSR1 pointer feature in FORTRAN allows the programmer to divide a data structure into separate, cache subpage aligned pieces that are referenced by the same data name. Memory that is read should also be distributed to processors in small chunks, but need not be stored in noncontiguous data structures. For example, within the PDE application, no performance enhancement was exhibited when the BPS preconditioner subdomain interior and boundary arrays were physically separated so that each interior and boundary array for a subdomain was aligned on a separate cache subpage. Instead, performance using read memory was only enhanced when the number of subdomains for a given problem was increased and hence the size of each preconditioner array for a given subdomain was decreased.

The pointer feature in the KSR1 FORTRAN compiler, when utilized, destroys program portability. Pointers are a unique data type within the KSR1 FORTRAN compiler (i.e., not included within the standard FORTRAN compiler implementation). Although program portability does suffer, pointer implementation is not difficult and only a few minor modifications need to be made in order to change a program to use pointers. Because the pointer feature does not require too many code changes to utilize and enables the programmer to divide arrays into noncontiguous segments without changing data-reference names, the feature is on the whole beneficial and worth the cost of destroying program portability

through these minor changes.

The PRESTO directives give a simple means of parallelizing a program without requiring many code changes. The tiling directive is an especially powerful tool in that it allows the programmer to easily map the iteration space to processors in a wide variety of methods. The fully automatic tile directive is the only feature that is not particularly useful because of the way it divides up the iteration space of a program loop. In fully automatic tiling, the iteration space is divided up into segments that are divisible by 32. Thus any loop that has ≤ 32 iterations will not be parallelized while a loop that has > 32 iterations will be cut up into iteration segments that are divisible by 32 hence full parallelism is not always derived (i.e., if there are more processors than iteration segments, some processors will not be used and will be idle). The KSR1 manual recommends using semi-automatic tiling or manual tiling instead of fully automatic tiling. Manual tiling is generally best because it gives the programmer full control of which variables within a loop are to be private (i.e., local) and which are shared.

Another feature within PRESTO is that all arrays are considered shared. Because pointers typically are used to point to arrays, they are also considered shared by the KSR1 FORTRAN compiler. This provides a small hurdle for using pointers to divide up an array into noncontiguous memory chunks in a program loop. To overcome this hurdle, the value of a pointer is passed to a subroutine. The subroutine defines the pointer and the array that it points to solely within the subroutine and the pointer value passed to the subroutine is then assigned to the pointer defined there. All subroutine variables (with the exception of those which are passed by argument) are considered private by the compiler.

The code for the preconditioned iterative methods was originally written on a Sequent and then transferred to a KSR1. PRESTO made the transition from one machine to the next particularly simple. The DO ACROSS loops on the Sequent easily translate to PRESTO tile loops. Altering the code to use noncontiguous vectors (i.e., pointers) was not particularly difficult and proved to be slightly beneficial.

Of the two preconditioners implemented, the preconditioner based on domain decompo-

sition, the Bramble-Pasciak-Schatz (BPS) preconditioner, parallelized much better than the incomplete Cholesky (IC) preconditioner. Using many small subdomains in the domain decomposition preconditioner resulted in faster execution times than fewer large subdomains on the same number of processors, given that there were at least as many subdomains as processors in either case. This illustrates the KSR1 cache system property of working best when each processor only works on a small piece of information rather than a larger chunk. The larger chunk, because of its size, is more likely to be partially paged out of cache to the detriment of algorithm performance.

The BPS preconditioned conjugate gradient method could be further improved on the KSR1 by splitting A into submatrices in a similar manner to the preconditioner. This would allow A to be distributed in small chunks to the processors during the matrix-vector multiply step. Some careful thought would need to be given about how to handle A_{IB} and A_{BC}. These two submatrices are very sparse since A is dominantly block diagonal. Hence, little storage space should be required for either submatrix.

$$\begin{pmatrix} A_I & A_{IB} & 0 \\ A_{IB}^T & A_B & A_{BC} \\ 0 & A_{BC}^T & A_C \end{pmatrix} \begin{pmatrix} x_I \\ x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_I \\ y_B \\ y_C \end{pmatrix} \tag{4.1}$$

To facilitate the matrix-vector multiply as shown in Equation 4.1, A_{IB} and A_{BC} should be implemented in such a way that it makes it easy to identify which segments of the vector x multiply with the appropriate sections of A_{IB}, A_{BC}, and their transposes. Through an intelligent use of pointers, this implementation would not be too difficult.

The worst feature on the KSR1 involves getting accurate timing statistics. Because the KSR1 is a multi-user system, parallel jobs tend to suffer due to cache paging when the system is busy. The timing statistics for a single problem run on the same number of processors can differ by as much as a factor of 100. All of the "larger" PDE problems were run on the KSR1 in nonpeak hours to avoid this problem. Also, in an effort to avoid inaccurate statistics, multiple runs were made of each problem, the outliers discarded, and the results averaged.

4.2 Conclusions

- Arrays or vectors that need to be repeatedly written to by multiple processors should be stored in noncontiguous, cache subpage aligned storage in such a manner that no processor needs to write to the same subpage as another processor. The KSR1 FORTRAN compiler provides a pointer data type that enables the programmer to split up an array into multiple cache subpage aligned segments while keeping the data reference name to the array.

- Arrays or vectors that are repeatedly read by multiple processors should be divided into smaller sub-arrays that are easily distributed across the processors. For this reason, domain decomposition preconditioning works particularly well on the KSR1 since the preconditioner is broken by subdomains, each of which is much smaller than original problem matrix.

- Arrays should be accessed contiguously—especially when first-referenced and in algorithms processing more than one array.

- When using the PRESTO tiling directive, never use fully automatic tiling. Instead, use either semi-automatic or manual tiling. Manual tiling requires programmer specification of all variables that are to be considered private—hence it provides the greatest programmer control. Semi-automatic tiling makes some decisions as to which variables must be considered shared (i.e., ensures program correctness).

- Getting accurate timing statistics on the KSR1 can be difficult. Results can differ wildly from one run of a problem to the next using the same number of processors. Averaging the results is not a sufficiently accurate means of getting timing data (especially when the number of runs to average is low); the outliers must also be discarded.

4.3 Implications For Future Iterative Methods

On shared, physically distributed memory architecture machines, vectors should be broken up into mini-vectors in a manner that enhances the parallelism of the most expensive operations. By breaking the vectors up into mini-vectors, write conflicts to the same subpage can be prevented—thus improving the parallelism of the operation.

Matrices should be broken into smaller pieces to reduce paging activity by promoting data locality. In a row oriented algorithm using a FORTRAN compiler, the distance between row elements is the length of a column. If a matrix is partioned into smaller pieces so that the number of rows in a submatrix is smaller than the number of rows in the original matrix, the distance between row elements is decreased and data locality is improved. Furthermore, when using a coefficient and column matrix to store a sparse matrix, the column matrix should be partitioned in the same manner as the coefficient matrix.

Indirect referencing should not be used in a manner that destroys data locality. For this reason, vectors that reorder the rows of matrix should be avoided. Instead of using reordering vectors, matrices should be physically reordered.

When using domain decomposition preconditioning, the original matrix should be re-ordered and broken up in a manner similar to that used by the preconditioner. The solve step is the most expensive step in an iterative method like the preconditioned conjugate gradient method. For this reason, the vectors should be broken up "over the subdomains" (see Section 3.2.2, Noncontiguous vector implementation) to prevent cache write conflicts during the solve. However, subdomain vector partitioning increases the complexity of the matrix-vector product operation. Breaking the original matrix up in a manner similar to that of the preconditioner would make the implementation of the matrix-vector product operation simpler and more efficient than an implementation where the original matrix was not partitioned in such a manner.

REFERENCES

[1] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal on High Speed Computing*, V. 1, pp. 73–95, 1989.

[2] B. Bagheri, A. Ilin, and L. R. Scott, "A comparison of distributed and shared memory scalable architectures. 1. KSR shared memory," *Proc. of the IEEE Scalable High-Perf. Comp. Conf.*, pp. 9–16, May 1994.

[3] C. F. Baillie, G. Carr, L. Hart, T. Henderson, and B. Rodriguez, "Comparison of shared memory and distributed memory parallelization strategies for grid-based weather forecast models," *Proc. of the IEEE Scalable High-Perf. Comp. Conf.*, pp. 560–567, May 1994.

[4] G. Birkhoff and R. E. Lynch, *Numerical Solution of Elliptic Problems*, SIAM, Philadelphia, 1984.

[5] J. G. Blom and J. D. Verwer, "Vectorizing matrix operations arising from PDE discretization on 9-point stencils," *The J. of Supercomputing*, V. 8, No. 1, pp. 29–51, March 1994.

[6] J. H. Bramble, J. E. Pasciak, and A. H. Schatz, "The construction of preconditioners for elliptic problems by substructuring," *Math. Comp.*, V. 47, pp. 103–134, 1986.

[7] T. F. Chan and T. P. Mathew, "Domain decomposition algorithms," *Acta Numerica*, pp. 61–143, 1994.

[8] P. Concus, G. H. Golub, and G. A. Meurant, "Block preconditioning for the conjugate gradient method," *SIAM J. on Scient. and Stat. Computing*, V. 6, No. 1, pp. 220–252, Jan. 1985.

[9] P. Concus, G. H. Golub, and D. O'Leary, "A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations," in *Sparse Matrix Computations*, J. R. Bunch and D. Rose, eds., Academic Press, NY, pp. 309–322, 1976.

[10] T. F. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1990.

[11] T. F. Chan, G. A. Meurant, J. Périaux, and O. Widlund, eds., *Domain Decomposition Methods*, SIAM, Philadelphia, 1989.

[12] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, V. 14, pp. 1–17, 1988.

[13] J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, V. 16, pp. 1–17, 1990.

[14] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[15] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," *Proceedings of the 16th International Symposium on Computer Architecture*, IEEE Catalog Number 89CH2705-2, pp. 2–15, June 1989.

[16] M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, V. 54, pp. 1901–1909, Dec. 1966.

[17] R. Glowinski, G. H. Golub, G. A. Meurant, and J. Périaux, eds., *First International Conference on Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1988.

[18] R. Glowinski, Y. A. Kuznetsov, G. A. Meurant, J. Périaux, and O. Widlund, eds., *Fourth International Conference on Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1991.

[19] G. H. Golub and C. F. Van Loan, *Matrix Computations, 2nd edition*, The Johns Hopkins University Press, Baltimore and London, 1989.

[20] W. D. Gropp and D. E. Keyes, "Domain decomposition on parallel computers," *Research Report YALEU/DCS/RR-723*, August 1989.

[21] W. D. Gropp and D. E. Keyes, "Domain decomposition with local mesh refinement," *Research Report YALEU/DCS/RR-726*, August 1989.

[22] J. P. Hayes, *Computer Architecture and Organization*, 2nd edition, McGraw-Hill Publishing Co., 1988.

[23] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems," *Parallel Algorithms for Matrix Computations*, SIAM, Philadelphia, pp. 83–124, 1990.

[24] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Standards Sect. B*, V. 49, pp. 409–436, 1952.

[25] Kendall Square Research, *KSR Fortran Programming*, Waltham, MA, 1993.

[26] Kendall Square Research, *KSR Parallel Programming*, Waltham, MA, 1993.

[27] Kendall Square Research, *KSR Technical Summary*, Waltham, MA 1993.

[28] D. E. Keyes, "Domain decomposition: a bridge between nature and parallel computers," *Proceedings of the Symp. on Adaptive, Multilevel and Hierarchical Computational Strategies*, ASME, 1992.

[29] D. E. Keyes, T. F. Chan, G. A. Meurant, J. S. Scroggs, and R. G. Voigt, eds., *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1992.

[30] D. E. Keyes and W. D. Gropp, "A comparison of domain decomposition techniques for elliptical partial differential equations and their parallel implementation," *SIAM J. on Scient. and Stat. Computing*, V. 8, No. 2, pp. s166–s202, March 1987.

[31] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Trans. Math. Softw.*, V. 5, pp. 308–323, 1979.

[32] A. Lichnewsky, "Sur la resolution de systèmes linéares issus de la méthode des élements finis par une multiprocessor," *INRIA Report 199*, 1982.

[33] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, New York, 1973.

[34] U. Meier and A. Sameh, "The behavior of conjugate gradient algorithms of a multivector processor with a hierarchical memory," *J. of Computational and Applied Math.*, V. 24, pp. 13–32, Nov. 1988.

[35] J. Meijerink and H. A. Van der Vorst, "An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix," *Math. Comp.*, V. 31, pp. 148–162, 1977.

[36] R. Melham, "Toward efficient implementation of preconditioned gradient methods on vector supercomputers," *Int'l. J. of Supercomputer Applications*, V. 1, No. 1, pp. 70–98, 1987.

[37] G. A. Meurant, "Domain decomposition methods for partial differential equations on parallel computers," *Int'l J. of Supercomputer Applications*, V. 2, No. 4, 1988.

[38] G. A. Meurant, "The block preconditioned conjugate gradient method on vector computers," *BIT*, V. 24, pp. 623–633, 1984.

[39] T. Nurkkala and V. Kumar, "The performance of a highly unstructured parallel algorithm on the KSR1," *Proc. of the IEEE Scalable High-Perf. Comp. Conf.*, pp. 215–220, May 1994.

[40] D. O'Leary, "The block conjugate gradient algorithm and related methods," *Linear Algebra and its Applications*, V. 29, pp. 293–322, 1980.

[41] D. O'Leary, "Ordering schemes for parallel processing of certain mesh problems," *SIAM J. on Scient. and Stat. Computing*, V. 5, No. 3, pp. 620–632, Sept. 1984.

[42] D. O'Leary, "Parallel implementation of the block conjugate gradient algorithm," *Parallel Comp.*, 5, pp. 127–139, 1987.

[43] J. M. Ortega and R. G. Voigt, "Solution of partial differential equations on vector and parallel computers," *SIAM Review*, V. 27, No. 2, pp. 149–240, June 1985.

[44] E. L. Poole and J. M. Ortega, "Multicolor ICCG methods for vector computers," *SIAM J. on Numerical Analysis*, V. 24, pp. 1394–1418, 1987.

[45] J. Reid, "On the method of conjugate gradients for the solution of large sparse systems of linear equations," *Proc. Conf. Large Sparse Sets of Linear Equations*, Academic Press, NY, 1971.

[46] C. J. Ribbens, "On adaptive domain decomposition with moving subdomains," *Fifth Int'l Symp. on Domain Decomposition Methods for Partial Differential Equations*, D. E. Keyes, et al. (eds.), SIAM, Philadelphia, pp. 322–329, 1992.

[47] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1985.

[48] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM J. on Scient. and Stat. Computing*, V. 10, No. 6, pp. 1200–1232, Nov. 1989.

[49] R. Schreibner and W. Tang, "Vectorizing the conjugate gradient method," in *Control Data Corp.*, 1982.

[50] H. S. Stone, *High-Performance Computer Architecture*, 2nd edition, Addison-Wesley, 1990.

[51] X. Sun and J. Zhu, "Shared virtual memory and generalized speedup," *NASA Contractor Report 191592*, ICASE Report No. 94-2, 1994.

[52] H. A. Van der Vorst, "A vectorizable variant of some ICCG methods,", *SIAM J. of Sci. and Stat. Comp.*, V. 3, pp. 350–356, 1982.

[53] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Cambridge, MA, the MIT Press, 1989.

Appendix A

PDE CODE

A.1 NCVS Matrix-Vector Multiply

NCVS stands for noncontiguous vectors broken up over the subdomains.

A.1.1 NCVS matrix-vector driver code

```
      subroutine q5cgmv (n, ix, iy, i1work, firsteqn, firsteqn_s,
     +                   firsteqn_w)
      implicit double precision (a-h,o-z)
c
c Matrix by vector multiplication delivers y=ax, given x.
c
      common  / cksr   /  p, iteam_id, itile
      common  / c4d0iv /  nsubx, nsuby, nx, ny
      common  / ctimes /  tddin, tddbps, tcgmv, tcgvs, tcgvd
      common  / c1param / nvectors, irhs, isol, irr, ipp, izz, iww,
     a                    iint, ifcs, ifcw, icrspt, JC,JW,JE,JS,JN
      integer i1work(nsubx,nsuby,4,nvectors),
     +        firsteqn(nsubx,nsuby), firsteqn_s(nsubx,nsuby),
     +        firsteqn_w(nsubx,nsuby)
      Real etime, times(2), start, end
c
c ELLPACK common blocks
c
      common  / c2coef /  r2coef(1)
      common  / c1ivdi /  i1neqn, i1mneq, i1ncoe, i1mnco
c
```

```
      start = etime(times)
      icrossoff = n - (nsubx-1) * (nsuby-1)  ! Crosspoint offset position
c*ksr* user tile(ksub,private=(isub,jsub,ieqn),
c*ksr*&          tilesize=(ksub:itile),teamid=iteam_id)
      do 10 ksub = 1, nsuby*nsubx
        jsub = (ksub-1)/nsubx + 1
        isub = 1 + mod(ksub-1,nsubx)
        ieqn = firsteqn(isub,jsub)         ! 1st eqn # of subdm. interior
        call q5cgm1(i1mneq,i1ncoe,isub,jsub,ieqn,iint,ix,
     +              i1work(isub,jsub,iint,iy),i1work,r2coef)
        if (jsub .gt. 1) then
          ieqn = firsteqn_s(isub,jsub)     ! 1st eqn # of S. boundary
          call q5cgm1(i1mneq,i1ncoe,isub,jsub,ieqn,ifcs,ix,
     +                i1work(isub,jsub,ifcs,iy),i1work,r2coef)
        end if
        if (isub .gt. 1) then
          ieqn = firsteqn_w(isub,jsub)     ! 1st eqn # of W. boundary
          call q5cgm1(i1mneq,i1ncoe,isub,jsub,ieqn,ifcw,ix,
     +                i1work(isub,jsub,ifcw,iy),i1work,r2coef)
          if (jsub .gt. 1) then
            ieqn = icrossoff +
     +             (jsub - 2)*(nsubx - 1) + (isub - 1)    ! Crosspt eqn#
            call q5cgm1(i1mneq,i1ncoe,isub,jsub,ieqn,icrspt,ix,
     +                  i1work(isub,jsub,icrspt,iy),i1work,r2coef)
          end if
        end if
   10 continue
c*ksr* end tile
      end = etime(times)
      tcgmv = tcgmv + (end - start)

      return
      end
```

A.1.2 NCVS matrix-vector multiply code

```
      subroutine q5cgm1 (i1mneq, i1ncoe, isub, jsub, ieqn, ipart,
     +                   ix, iy, i1work, a)
      implicit double precision (a-h,o-z)
      common  / c4d0iv /  nsubx, nsuby, nx, ny
      common  / c1param / nvectors, irhs, isol, irr, ipp, izz, iww,
     a                    iint, ifcs, ifcw, icrspt, JC, JW, JE, JS, JN
      double precision x(1), y(1), a(i1mneq,i1ncoe)
      integer i1work(nsubx,nsuby,4,nvectors)
      pointer (px,x)
      pointer (py,y)
c
      py = iy
c
      if (ipart .eq. iint) then            ! RHS subdomain interior
        ii = ieqn
        ioff = 1
        px = i1work(isub,jsub,iint,ix)
        do 10 j = 1,ny-2                   ! y_int(isub,jusb)=
          do 10 i = 1,nx-2                 !   A_int(isub,jsub)*
            sum = a(ii,JC) * x(ioff)       !   x_int(isub,jsub)
            if (j .ne. 1)    sum = sum + a(ii,JS) * x(ioff-(nx-2))
            if (j .ne. ny-2) sum = sum + a(ii,JN) * x(ioff+(nx-2))
            if (i .ne. 1)    sum = sum + a(ii,JW) * x(ioff-1)
            if (i .ne. nx-2) sum = sum + a(ii,JE) * x(ioff+1)
            y(ioff) = sum
            ii = ii + 1
            ioff = ioff + 1
   10   continue
        if (jsub .gt. 1) then              ! Add interaction w/ S. boundary
          ii = ieqn
          px = i1work(isub,jsub,ifcs,ix)
          do 20 i = 1,nx-2                 ! y_int(isub,jsub)+=
            y(i) = y(i) + a(ii,JS) * x(i)  !   A_int(isub,jsub)*x
            ii = ii + 1                    !   x_south(isub,jsub)
   20     continue
        end if
```

```
      if (jsub .lt. nsuby) then          ! Add interaction w/ N. boundary
        ioff = (ny-3)*(nx-2)
        ii = ieqn + ioff
        px = i1work(isub,jsub+1,ifcs,ix)
        do 30 i = 1,nx-2                                ! y_int(isub,jsub)+=
          y(ioff+i) = y(ioff+i) + a(ii,JN) * x(i)  ! A_int(isub,jsub)*
          ii = ii + 1                                   ! x_south(isub,jsub+1)
30      continue
      end if
      if (isub .gt. 1) then              ! Add interaction w/ W. boundary
        ii = ieqn
        ioff = 1
        px = i1work(isub,jsub,ifcw,ix)
        do 40 i = 1,ny-2                          ! y_int(isub,jsub)+=
          y(ioff) = y(ioff) + a(ii,JW) * x(i)  !   A_int(isub,jsub)*
          ii = ii + nx-2                          !   x_west(isub,jsub)
          ioff = ioff + nx-2
40      continue
      end if
      if (isub .lt. nsubx) then          ! Add interaction w/ E. boundary
        ii = ieqn + nx-3
        ioff = nx-2
        px = i1work(isub+1,jsub,ifcw,ix)
        do 50 i = 1,ny-2                          ! y_int(isub,jsub)+=
          y(ioff) = y(ioff) + a(ii,JE) * x(i)  !   A_int(isub,jsub)*
          ii = ii + nx-2                          !   x_west(isub+1,jsub)
          ioff = ioff + nx-2
50      continue
      end if
    else if (ipart .eq. ifcs) then       ! RHS subdomain south boundary
      ii = ieqn
      px = i1work(isub,jsub,ifcs,ix)
      do 60 i = 1,nx-2          ! y_south(isub,jsub) =
        sum = a(ii,JC) * x(i)  !  A_south(isub,jsub)*x_south(isub,jsub)
        if (i .ne. 1)    sum = sum + a(ii,JW) * x(i-1)
        if (i .ne. nx-2) sum = sum + a(ii,JE) * x(i+1)
        y(i) = sum
        ii = ii + 1
60    continue
```

```
      ii = ieqn                          ! Add interaction w/ N. interior
      px = i1work(isub,jsub,iint,ix)
      do 70 i = 1,nx-2                   ! y_south(isub,sub)+=
        y(i) = y(i) + a(ii,JN) * x(i)    !   A_south(isub,jsub)*
        ii = ii + 1                      !   x_int(isub,jsub)
70    continue
      ii = ieqn                          ! Add interaction w/ S. interior
      px = i1work(isub,jsub-1,iint,ix)
      ioff = (ny-3)*(nx-2) + 1
      do 80 i = 1,nx-2                     ! y_south(isub,jsub)+=
        y(i) = y(i) + a(ii,JS) * x(ioff)   !   A_south(isub,jsub)*
        ii = ii + 1                        !   x_int(isub,jsub-1)
        ioff = ioff + 1
80    continue
      if (isub .gt. 1) then              ! Add interaction w/ W. crosspt
        ii = ieqn                         ! y_south(isub,jsub)+=
        px = i1work(isub,jsub,icrspt,ix) !   A_south(isub,jsub)*
        y(1) = y(1) + a(ii,JW) * x(1)    !   x_crspt(isub,jsub)
      end if
      if (isub .lt. nsubx) then          ! Add interaction w/ E. crosspt
        ii = ieqn + nx-3                    ! y_south(isub,jsub)+=
        px = i1work(isub+1,jsub,icrspt,ix)  !   A_south(isub,jsub)*
        y(nx-2) = y(nx-2) + a(ii,JE) * x(1) !   x_crspt(isub+1,jsub)
      end if
    else if (ipart .eq. ifcw) then       ! RHS subdomain west boundary
      ii = ieqn
      px = i1work(isub,jsub,ifcw,ix)
      do 90 i = 1,ny-2                                ! y_west(isub,jsub)=
        sum = a(ii,JC) * x(i)                         ! A_west(isub,jsub)*
        if (i .ne. 1) sum = sum + a(ii,JS) * x(i-1) ! x_west(isub,jsub)
        if (i .ne. ny-2) sum = sum + a(ii,JN) * x(i+1)
        y(i) = sum
        ii = ii + 1
90    continue
```

```
      ii = ieqn                          ! Add interaction w/ E. interior
      px = i1work(isub,jsub,iint,ix)
      ioff = 1
      do 100 i = 1,ny-2                    ! y_west(isub,jsub)+=
        y(i) = y(i) + a(ii,JE) * x(ioff)  !   A_west(isub,jsub)*
        ii = ii + 1                       !   x_int(isub,jsub)
        ioff = ioff + nx-2
100   continue
      ii = ieqn                          ! Add interaction w/ W. interior
      px = i1work(isub-1,jsub,iint,ix)
      ioff = nx-2
      do 110 i = 1,ny-2                    ! y_west(isub,jsub)+=
        y(i) = y(i) + a(ii,JW) * x(ioff)  !   A_west(isub,jsub)*
        ii = ii + 1                       !   x_int(isub-1,jsub)
        ioff = ioff + nx-2
110   continue
      if (jsub .gt. 1) then              ! Add interaction w/ S. crosspt
        ii = ieqn                          ! y_west(isub,jsub)+=
        px = i1work(isub,jsub,icrspt,ix)  !   A_west(isub,jsub)*
        y(1) = y(1) + a(ii,JS) * x(1)     !   x_crspt(isub,jsub)
      end if
      if (jsub .lt. nsuby) then          ! Add interaction w/ N. crosspt
        ii = ieqn + ny-3                     ! y_west(isub,jsub)+=
        px = i1work(isub,jsub+1,icrspt,ix) !   A_west(isub,jsub)*
        y(ny-2) = y(ny-2) + a(ii,JN) * x(1) !   x_crspt(isub,jsub+1)
      end if
    else                                 ! RHS subdomain boundary crosspt
      px = i1work(isub,jsub,icrspt,ix)   ! y_crspt(isub,jsub)=
      y(1) = a(ieqn,JC) * x(1)           !   A_crspt(isub,jsub)*
c                                        !   x_crspt(isub,jsub)
c                                        ! Add northern west boundary
      px = i1work(isub,jsub,ifcw,ix)     !   y_crspt(isub,jsub)+=
      y(1) = y(1) + a(ieqn,JN) * x(1)    !     A_crspt(isub,jsub)*
c                                        !     x_west(isub,jsub)
c                                        ! Add eastern south boundary
      px = i1work(isub,jsub,ifcs,ix)     !   y_crspt(isub,jsub)+=
      y(1) = y(1) + a(ieqn,JE) * x(1)    !     A_crspt(isub,jsub)*
c                                        !     x_south(isub,jsub)
```

```
c                                          ! Add southern west boundary
        px = i1work(isub,jsub-1,ifcw,ix)   !   y_crspt(isub,jsub)+=
        y(1) = y(1) + a(ieqn,JS) * x(ny-2) !     A_crspt(isub,jsub)*
c                                          !       x_west(isub,jsub-1)
c                                          ! Add western south boundary
        px = i1work(isub-1,jsub,ifcs,ix)   !   y_crspt(isub,jsub)+=
        y(1) = y(1) + a(ieqn,JW) * x(nx-2) !     A_crspt(isub,jsub)*
      end if                               !     x_south(isub-1,jsub)
      return
      end
```

Appendix B

INCOMPLETE CHOLESKY (IC) PRECONDITIONING

B.1 IC Factorization

Table B.1: IC factorization (averaged over 5 runs).

Time on a 49 x 49 grid	Time on a 89 x 89 grid
3.736	28.780

The IC factorization Section 3.3.4 was inefficiently implemented. The code took advantage of the sparse ELLPACK column structure, but did not take advantage of the banded affect on the rows. Using the ELLPACK column matrix and the symmetry of the matrix, the factorization should be able to immediately go to the rows with data in them rather than traversing the entire column. The revised IC factorization code is:

```
      Do 550 j = 1, i1ncoe
        Do 550 i = 1, n
          if (i1idco(i,j) .le. i) m(i,j) = a(i,j)
  550 Continue
      Do 575 i = 1, n
  575   temp(i) = 0.0D0
```

```
      Do 800 k = 1, n
        kk = colindex(i1idco,i1undx,l1asis,i1mneq,i1ncoe,k,k) ! Diag
        if (kk .eq. 0) then
          print *, 'cannot find diagonal element in IC', k
          Stop
        else if (m(k,kk) .eq. 0.0D0) then
          print *, 'zero diagonal element in IC', k
          stop
        else if (m(k,kk) .lt. 0.0D0) then
          print *, 'diagonal element less than 0 in IC', k
          stop
        end if
        m(k,kk) = dsqrt(m(k,kk))
c This loop may be parallelized:
        Do 600 j = 1,i1ncoe                          ! reduce kth column
          i = i1idco(k,j)                            ! and by symmetry kth
          if ((i .gt. k) .and. (i .ne. 0)) then      ! upper row
            jj = colindex(i1idco,i1undx,l1asis,i1mneq,i1ncoe,i,k)
            m(i,jj) = m(i,jj)/m(k,kk)
            temp(i) = m(i,jj)
          end if
  600   continue
c This loop may be parallelized:
        Do 700 j2 = 1, i1ncoe                        ! reduce remaining lower
          j = i1idco(k,j2)                           ! triangular matrix
          if (j .gt. k) then              ! a(k,j) .ne. 0
            Do 650 j3 = 1, i1ncoe
              i = i1idco(k,j3)
              if (i .ge. j) then          ! a(k,i)=a(i,k) .ne. 0, and i.ge.j
                jj = colindex(i1idco,i1undx,l1asis,i1mneq,i1ncoe,i,j)
                if (jj .ne. 0) then       ! a(i,j) .ne. 0
                  m(i,jj) = m(i,jj) - temp(i) * temp(j)
                endif
              endif
  650       Continue
          endif
  700   Continue
        Do 750 j = 1,i1ncoe                          ! zero out temp
          i = i1idco(k,j)
          temp(i) = 0.0D0
  750   Continue
```

```
800 Continue
```

This revised code is $O(25N^2)$ which is much better than $O\left(\frac{7}{2}N^4\right)$, the time complexity of Section 3.3.4's IC factorization code. The revised code is also better in time complexity than the BPS factorization $\left(O\left(\frac{N^4}{2m^2}\right)\right)$. When the two loops (600 and 700) were parallelized, speed down occurred. This was due to there being insufficient work in these loops for the KSR1 to take advantage of since five-point star discretization produces at most 5 nonzero elements in any row or column (due to symmetry). Table B.1 has performance statistics for the factorization on one processor.

B.2 IC Preconditioned Conjugate Gradient Performance

Table B.2 has the revised IC preconditioned conjugate gradient performance. The execution times of the revised code are better than those reported in Section 3.3. Because neither the factor nor the solve is parallelized, IC preconditioned conjugate gradient performance degrades rapidly as the number of processors grows.

Table B.2: IC preconditioned conjugate gradient (averaged over 5 runs in nonpeak system hours).

	Noncontiguous			Contiguous		
P	Time	Spdup	Effic	Time	Spdup	Effic
49 x 49 grid (37 its.)						
1	54.74			48.45		
2	45.72	1.20	0.60	39.56	1.22	0.61
4	41.00	1.34	0.33	34.56	1.40	0.35
8	38.57	1.42	0.18	33.26	1.46	0.18
16	38.34	1.43	0.09	32.65	1.48	0.09
89 x 89 grid (72 its.)						
1	358.7			329.4		
2	299.1	1.20	0.60	264.3	1.25	0.62
4	264.4	1.36	0.35	231.1	1.43	0.36
8	250.4	1.43	0.18	219.7	1.50	0.19
16	246.6	1.45	0.09	212.7	1.55	0.10

VITA

I was born in Miami, Florida on December 23, 1957. My education consists of a Bachelor or Arts degree in Mathematics from Hollins College, Hollins, VA and a high school diploma from Lexington High School in Lexington, VA. After college, I worked at the Central Intelligence Agency (CIA) as a programmer/analyst for approximately 10 years.

In my latter years at the CIA, I became convinced that I needed to go back to school in order to broaden my education and find new areas of interest within computer science. I am particularly interested parallelism and scientific computation.

Harriet Jane Roberts