Multigrain parallelism for eigenvalue computations on networks of clusters *

James R. McCombs † Andreas Stathopoulos †

March 1, 2002

Abstract

Clusters of workstations have become a cost-effective means of performing scientific computations. However, large network latencies, resource sharing, and heterogeneity found in networks of clusters and Grids can impede the performance of applications not specifically tailored for use in such environments. A typical example is the traditional fine grain implementations of Krylov-like iterative methods, a central component in many scientific applications. To exploit the potential of these environments, advances in networking technology must be complemented by advances in parallel algorithmic design. In this paper, we present an algorithmic technique that increases the granularity of parallel, block iterative methods by inducing additional work during the preconditioning (inexact solution) phase of the iteration. During this phase, each vector in the block is preconditioned by a different subgroup of processors, yielding a much coarser granularity. The rest of the method comprises a small portion of the total time and is still implemented in fine grain. We call this combination of fine and coarse grain parallelism *multigrain*. We apply this idea to the block Jacobi-Davidson eigensolver, and present experimental data that shows the significant reduction of latency effects on networks of clusters of roughly equal capacity and size. We conclude with a discussion on how multigrain can be applied dynamically based on runtime network performance monitoring.

1 Introduction

Commodity components such as desktop workstations and high-speed networking media have made clusters of workstations (COWs) a cost-effective way of performing scientific computing [8]. COW configurations vary from a small number of compute nodes attached to a hub to collections of clusters. Such collections are interconnected via a hierarchy of network switches yielding a heterogeneous networking environment where not all nodes incur the same latency to communicate with each other. Grids are a further generalization of this scheme that include various computational environments that are possibly geographically dispersed [6]. Providing an easy, integrative way for users to access these environments has been the focus of much Grid research recently [5, 7, 22]. Instead, our research focuses on the challenge of devising methods that can harness effectively the power of these environments.

*Work supported by the National Science Foundation (ITR/ACS-0082094 and ITR/AP-0112727), a grant from the Virginia Space Grant Consortium, and performed using computational facilities at the College of William and Mary which were enabled by grants from the National Science Foundation (EIA-9977030) and Sun Microsystems (SAR EDU00-03-793).

†Department of Computer Science, College of William and Mary, Williamsburg, Virginia 23187-8795, (mccombjr/andreas@cs.wm.edu).

Iterative methods for the solution of linear systems and eigenvalue problems are an important, and often the most computationally intensive, kernel of many scientific applications. As these applications require the use of various distributed resources, high performance implementations of iterative methods are crucial to avoid bottlenecks. Traditionally, iterative methods have been implemented in a fine grain way, with inner products requiring a global reduction and matrix-vector multiplication based on nearest neighbor communications [4]. However, global reductions do not scale with the number of processors [20], and are especially sensitive to network overheads which have not kept up with the explosive growth of bandwidth in recent networks.

Beyond COWs, the latency problem in fine grain methods is exacerbated on collections of clusters that employ different switching technologies, and where multiple parallel jobs compete for network bandwidth. In a Grid environment, the significantly higher overheads can completely incapacitate these methods.

Block iterative methods increase granularity by having each processor apply the same fine grain operations on a block of vectors, thus increasing the computation/communication ratio [3, 13]. As a side benefit, block operations also demonstrate better cache performance. However, the granularity increase is usually marginal, and because block methods increase the total number of floating point operations, the benefits from such implementations of these algorithms are limited.

In [21], we described a novel parallelization approach of a block Jacobi-Davidson eigenvalue solver. Each vector from the block is assigned on a different processor that executes the preconditioning step independently, thus improving granularity significantly. In fact, the accuracy of the preconditioning could be set arbitrarily so that prolonged parallel execution on each processor reduces any impact of interprocessor latency. However, in this original implementation the number of processors used had to be equal to the block size (usually less than 4-8), and each processor had to store the whole matrix.

In this paper, we extend our previous approach to any number of processors, and especially to collections of small numbers of COWs connected with different networks. Within individual clusters (or subgroups of processors) the iterative methods are implemented in a fine grain way, while a coarse grain distribution of block vectors between clusters minimizes intercluster communications. This *multigrain* technique reduces the effects of latency either when the number of processors in a COW is very large or when the link between clusters is slow or highly variable, as is the case in Grids.

First, we describe a class of block iterative methods on which the multigrain technique is applicable. These methods are formulated as an inner-outer iteration, where the inner iteration (preconditioning) solves approximately a correction equation to accelerate the convergence of the outer method [19, 16]. We then show the benefits of multigrain in the context of a Jacobi-Davidson eigensolver on our collection of COWs. We conclude with a discussion on how to apply multigrain dynamically based on runtime network performance monitoring.

2 A multigrain paradigm for preconditioned, block iterative methods

Many applications involve the solution of a system of linear equations $A\tilde{x} = b$ for the unknown vector $\tilde{x}$, or the solution of the eigenvalue problem $A\tilde{x}_i = \tilde{\lambda}_i\tilde{x}_i$, for the $i = 1, \dots l$ smallest or largest eigenvalues $\tilde{\lambda}_i$ and the corresponding eigenvectors $\tilde{x}_i$. Because the matrix A is usually sparse and of large dimension, iterative methods provide the only means of solving these problems. Many block iterative methods such as block FGMRES [16] for linear systems and block Jacobi-Davidson [19] for eigenvalue problems follow the inner-outer iterative structure of figure 1.

Considering a block size of k, these methods build a subspace V, k vectors at a time, from where they extract their approximate solutions. At each iteration, they build the next k vectors by solving approximately

```
while ( x_1 ... x_k are unconverged) {
    if (solving linear systems)
        Set s_i = v_i, for i = 1, ..., k
    elseif (solving eigenvalue problem)
        Set s_i = x_i, for i = 1, ..., k
    endif
    Apply m inner iterations on s_i to obtain t_i
    Append t_i to V and orthogonalize V
    Compute the new approximations x_1 ... x_k
}
```

Figure 1: A class of inner-outer methods that can be implemented with multigrain parallelism. The inner iterations are often themselves a preconditioned iterative solver.

k different correction equations, one per block vector. The approximate solutions are then orthogonalized and appended to the subspace V. The correction equations are usually solved using m steps of another preconditioned iterative solver. The more accurately the inner systems are solved the fewer outer iterations the algorithm performs. However, there is usually an optimal number of inner iterations beyond which the actual time, as measured also by the total number of matrix vector multiplications, slowly increases.

Block methods are known to accelerate convergence in linear systems with multiple right-hand sides [13], and to improve robustness when solving for eigenvalues that occur close together (clustered) [14]. However, it is known that larger block sizes increase the total number of floating point operations. This is manifested as an increase in the number of total matrix vector multiplications performed by both inner and outer iterations. Despite improved cache efficiency and a relatively coarser granularity in fine grain parallel implementations, block algorithms are only competitive when access to the matrix is expensive so that it pays off to block several matrix-vector operations per matrix access.

The multigrain technique uses the block in a vertical rather than horizontal way. Extending the key ideas from [21], we note that the correction equations are independent for each of the block vectors and each may take an arbitrarily long amount of time. If we assign them on different subgroups of processors, each subgroup should be able to compute the majority of its computation without communicating with other groups. Thus, we effectively reduce the latencies in our cluster to the latencies of a cluster of $1/k$ the size. This is particularly beneficial when each subgroup represents a COW with high performance intra-cluster, but not inter-cluster, networks. Note that each subgroup (COW) should be able to keep the whole matrix A to solve the correction equation. In contrast to our [21] method, this is not a scaling limitation any more, as each processor in the subgroup needs to store only k times more rows than its fine grain partition — a small amount considering the rapid growth of DRAM sizes.

Except the correction equation, the rest of the steps of the algorithm cannot be efficiently performed in the above coarse grain setting. Fortunately, they comprise only a small portion of the total execution time. Therefore, multigrain follows the traditional fine grain partitioning for all the other steps, and switches to coarse grain only for the correction phase. An all-to-all operation is required to transition each of the fine grain vectors s_i and t_i to their coarse partitioned counterparts on each processor, so that each subgroup has its respective vector. Despite the high cost of the all-to-all operation, the number of inner iterations (and thus granularity) can be increased arbitrarily to diminish the associated latencies. Finally, we show next that in the case of homogeneous processors with a fast interconnect (such as MPPs) a faster all-to-all is possible.

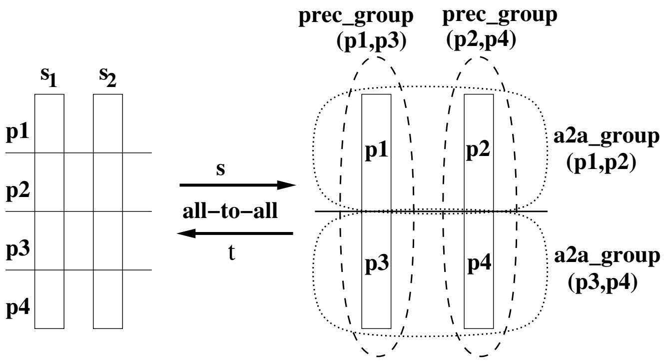

Figure 2: Example of MPP multigrain, with more processors than block vectors. Before the preconditioning phase, nodes in the same all-to-all group receive the coarse-grain portions of the vectors $\mathbf{s}_i$ they are responsible for. Each preconditioning group then performs its respective preconditioning. After this phase, each node distributes its coarse-grain portion of $\mathbf{t}_i$ amongst its fellow all-to-all members.

2.1 Multigrain algorithm for MPPs

To avoid a global all-to-all exchange between all processors, we can consider the following hierarchical partitioning. Assume for simplicity that the block size k divides the number of processors P, First, we obtain the coarse grain partitioning of the matrix onto P/k processors using partitioning software [15, 10]. Second, each processor partitions its local, coarse grain rows into k subdomains, and designates one of those as its fine-grain partitioning.

This hierarchical partitioning of rows reduces the complexity of the all-to-all communications that now involve only groups of P/k processors. Before the preconditioning phase, each member of an all-to-all group sends its fine-grain portions of the k vectors $\mathbf{s}_i$ to the P/k members of its subgroup, and receives the P/k pieces that compose its coarse-grain portion of $\mathbf{s}_i$. Figure 2 illustrates this for $P = 4$ and $k = 2$. After each preconditioning group finishes its inner iterations, the all-to-all is reversed and each processor's coarse-grain portion of $\mathbf{t}_i$ is distributed across all the processors in the all-to-all group.

2.2 Multigrain algorithm for collections of clusters and Grids

In collections of clusters, the preconditioning groups may be of different sizes and are chosen by the user to correspond to the physical boundaries of the COWs, or to those processor boundaries where inter-boundary communication is expensive. In a multigrain implementation, all nodes compute a fine grain partitioning of A, and each solve group computes an independent coarse grain partitioning based on the group size. The independence of the two partitionings makes it impossible to use all-to-all groups, as one processor may be involved in a total exchange with all other processors in the cluster. Therefore, the all-to-all communications must involve all P processors. In the next section we show how we applied the cluster algorithm to a block Jacobi-Davidson method for solving eigenvalue problems.

3 A multigrain, block Jacobi-Davidson method for use in Grid-like environments

Jacobi-Davidson (JD) is a popular method for computing eigenvalues of large, sparse matrices [19, 18]. A block version is also possible that follows the general iterative model of figure 1. At each iteration the method computes the current approximate eigenvalues λ_i, the approximate eigenvectors x_i, and the associated residuals $r_i = Ax_i - \lambda x_i$ which are then used to solve the correction equation,

$$(I - x_i x_i^T)(A - \lambda_i I)(I - x_i x_i^T) t_i = r_i, \tag{1}$$

for the vector t_i, an approximation to the error in x_i. These k approximate correction vectors t_i are then used to extend the basis V and improve future approximations. The block JD algorithm is given below.

Algorithm: **Block JD**
starting with k trial vectors t_i
While not converged do:
1. Orthogonalize $t_i, i = 1 : k$. Add them to V
2. **Matrix-vector** $W_i = AV_i,\ i = 1 : k$
3. $H = V^T W$ (local contributions)
4. Global_Sum(H) over all processors.
5. Solve $Hy_i = \lambda_i y_i,\ i = 1 : k$ (all procs)
6. $x_i = Vy_i,\ z_i = Wy_i, i = 1 : k$ (local rows)
7. $r_i = z_i - \lambda_i x_i,\ i = 1 : k$ (local rows)
8. **Correction equation** Solve eq. (1) for each t_i

end while

During the projection phase (steps 1-7), the block algorithm finds the k smallest Ritz eigenpairs and their residuals. During the correction/preconditioning phase, k different equations (1) are solved approximately for the t_i, usually by employing an iterative solver for linear systems such as BCGSTAB or GMRES [17]. In the multigrain adaptation, the coarse-grain subgroups (solve groups) will solve these correction equations. Preconditioners such as sparse approximate inverse or incomplete LU factorization may be used to accelerate the convergence of the corrections.

3.1 Adapting JD for use with multigrain

Steps 1-7 of the block JD algorithm are still performed in fine grain. However, the vectors x_i and r_i must now be gathered onto their respective solve group with an all-to-all operation amongst the P processors before the correction equations are solved. The new multigrain version of step 8 is as follows:

8. **Multi grain correction phase in JD**
   All-to-All: **send** local fine-grain rows of x_i, r_i to each proc
   receive coarse-grain rows for $x_{mygroup}, r_{mygroup}$ from each proc
   Apply m steps of (preconditioned) BCGSTAB on
   eq.(1) with the gathered $r_{mygroup}, x_{mygroup}$
   All-to-All: **send** coarse-grain rows of $t_{mygroup}$ to proc i
   receive fine-grain rows for t_i from proc i

Note that the cluster version of all-to-all is used. Next, we present experiments that show the benefit of our multigrain implementation on workstation clusters.

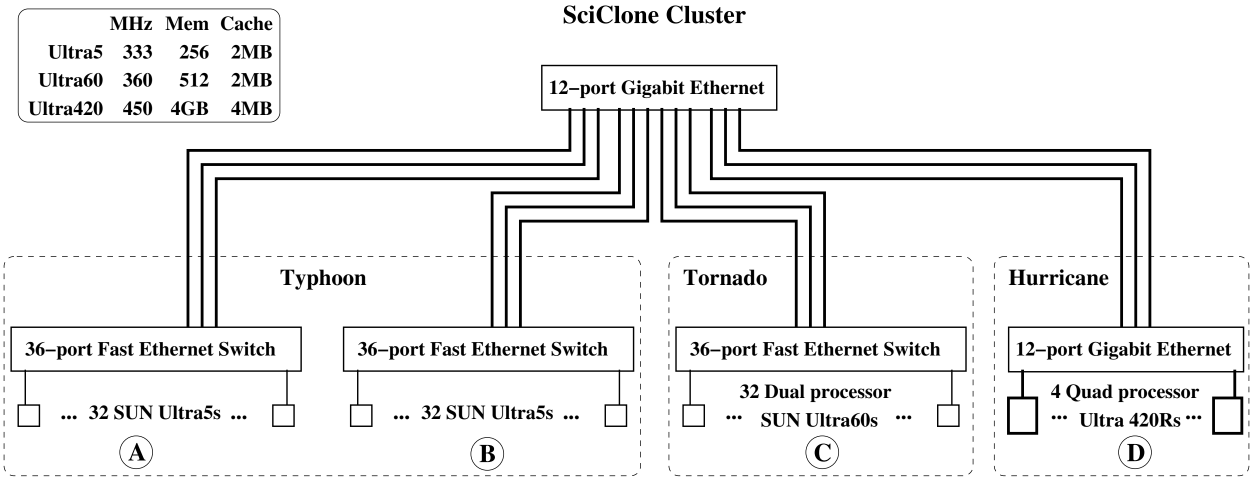

Figure 3: SciClone: The William and Mary heterogeneous cluster of three homogeneous clusters: Typhoon, Tornado (also called C), and Hurricane (also called D). We distinguish between A and B, the subclusters of Typhoon, because their intercommunication passes through the Gigabit switch.

4 Experiments

We conducted experiments with our JD code using both the fine grain and multigrain capabilities. The experiments were performed on the SciClone (Figure 3), a collection of clusters used for scientific computing and computer science research at the College of William and Mary. SciClone is highly a suitable environment for our experimentation in cluster and Grid computing because it is composed of various networking technologies and organized as a cluster of subclusters.

For the experiments, we test the fine grain implementation of JD with block sizes 1 and 4, and our multigrain JD with block size of 4. The test matrix is derived from a 3-D finite element problem [1] of dimension $268,515$ and containing $3,926,823$ non-zero elements. In all tests we compute the smallest 50 eigenvalues using a maximum size of 108 for the basis V and a restart size of 60 vectors. The correction equation is solved approximately by a maximum of 40 steps of preconditioned BCGSTAB. The preconditioner is an approximate inverse from the ParaSails library [2]. Note that we use this preconditioner because it does not depend on the partitioning and thus provides a common reference for both fine and multigrain methods. In practice, multigrain will have an additional numerical advantage over fine grain, because it can use larger local domain preconditioners.

To better ensure load balance of multigrain during the correction phase, we choose solve groups of equal size. The fine grain partitions of the matrix are computed using the weighted METIS library [9]. The weights account for the relative speed differences of the processors. An exception to this is in cases where more than one process runs on a two or four-way SMP, which has been observed to degrade processor performance. In these cases, no load balanced partitioning was used.

Examination of the fine-grain experiments with a block size of 1 show the speedup from 16 to 32 and 64 Ultra 5's to be about 1.7 and 1.4 respectively. However, despite having more cache and faster processors, the speedup on the Ultra 60's was only about 1.08. In general, our test cases seem to perform more poorly when more than one MPI process is executed on the SMPs. This may be due to a combination of contention for the network interface and memory bus. Fortunately, the use of multigrain does alleviate these problems in some cases.

Nodes	Time	Mvecs	Iterations
A_{16}	7513	67819	1849
A_{32}	4401	64196	1754
$A_{32}B_{32}$	3022	62838	1725
$A_{32}B_{32}C_{64}$	2837	62879	1721
$A_{32}B_{32}D_{16}$	3360	65541	1790
$A_{32}C_{64}D_{16}$	2912	64465	1766
$A_{32}D_{16}$	4412	66479	1813
$A_{32}C_{32}$	3041	69185	1877
$A_{32}C_{64}$	2854	61936	1702
C_{32}	3423	64196	1754
C_{64}	3163	62838	1725
$C_{32}D_{4}$	2410	66065	1799
$C_{32}D_{8}$	3411	67500	1838
$C_{32}D_{16}$	4133	66479	1813
$C_{64}D_{16}$	3388	65541	1790

Table 1: Performance of the fine-grain JD algorithm with block size of 1. "Time" indicates run time in seconds, "Mvecs" is the number of matrix-vector multiplications performed, "Iterations" is the number of iterations performed. The clusters used are denoted by the letters A, B, C, and D. The subscripts indicate how many MPI processes were run on that particular cluster. For instance, C_{32} indicates that one process was run on each of the 32 dual processor Ultra 60's and C_{64} indicates that two processes were run on each node. Combinations of clusters are indicated by two or more letters.

Comparison of the multigrain results in Table 3 and the fine-grain data with block size 4 in Table 2 shows that the multigrain algorithm provides significant improvement for the vast majority of tests. Thus, multigrain is particularly useful for solving those difficult problems where a block method is necessary. On the other hand, load imbalance induced by the use of two solve groups of differing sizes increases the run time for cluster combinations such as $A_{32}D_{16}$. Our current work in [12] addresses such heterogeneous clusters by considering an application based load balancing solution.

Of further interest are the improvements that the multigrain method provides over the fine-grain method with a block size of 1. This is the most stringent test for multigrain because block methods tend to increase the amount of work. Notice that from block size 1 to block size 4 the number of iterations decreases by more than half, but the total number of matrix-vector multiplications goes up by at most 14%. Yet, the significant reduction in latency caused by the multigrain design enables the block method to outperform its fine-grain counterparts for more than 75% of the test cases.

5 Conclusions and work in progress

The increasing complexity of computing environments, consisting of collections of COWs either in the same local area network or geographically dispersed, necessitates new algorithmic techniques that tolerate high network latencies. The proposed multigrain technique for Krylov-like methods transfers most of the convergence work from the outer fine grain iteration to an inner coarse grain iteration that processors can execute for a long time independently, thus tolerating arbitrary large latencies.

Because block methods increase the total number of operations, the fine grain with block size of one

Nodes	Time	Mvecs	Iterations
A_{16}	10037	71934	631
A_{32}	5513	69608	617
$A_{32}B_{32}$	3789	72112	631
$A_{32}B_{32}C_{64}$	3031	70943	625
$A_{32}B_{32}D_{16}$	3906	72942	636
$A_{32}C_{64}D_{16}$	3313	70483	626
$A_{32}D_{16}$	4775	70835	626
$A_{32}C_{32}$	3602	75082	654
$A_{32}C_{64}$	3654	71400	626
C_{32}	4104	69936	618
C_{64}	3929	71051	630
$C_{32}D_{4}$	5721	70648	621
$C_{32}D_{8}$	5510	71932	628
$C_{32}D_{16}$	4473	70835	626
$C_{64}D_{16}$	3882	73216	640

Table 2: Performance of fine-grain JD algorithm with block size of 4. Each of the correction vectors were solved for in fine-grain. The number of iterations was reduced and the number of matrix-vector multiplications increased as is common with block methods.

			% Improvement			
MG Nodes	FG counterpart	Time	over FG1	over FG4	Mvecs	Iterations
$A_4A_4A_4A_4$	A_{16}	7591	-1.1	24.4	71744	629
$A_8A_8A_8A_8$	A_{32}	3868	12.2	29.9	69894	620
$A_{16}A_{16}$	A_{32}	4396	0.2	20.3	72346	631
$A_{32}B_{32}$	$A_{32}B_{32}$	2532	16.3	33.2	69740	617
$A_{32}C_{32}$	$A_{32}C_{32}$	2562	15.8	28.9	71168	624
$A_{16}A_{16}B_{16}B_{16}$	$A_{32}B_{32}$	2265	25.1	41.3	72510	633
$A_{16}A_{16}C_{16}C_{16}$	$A_{32}C_{32}$	2104	30.9	41.6	69780	617
$A_{32}B_{32}C_{32}C_{32}$	$A_{32}B_{32}C_{64}$	2554	10.0	15.8	72432	634
$C_{16}C_{16}$	C_{32}	3429	-0.02	16.5	72346	631
$C_{32}C_{32}$	C_{64}	2960	6.5	24.7	69740	617
$C_{16}C_{16}C_{16}C_{16}$	C_{64}	2691	15.0	31.6	76450	658
$A_{32}D_{16}$	$A_{32}D_{16}$	5149	-19.8	-7.3	73748	639
$A_{16}A_{16}D_8D_8$	$A_{32}D_{16}$	4387	-5.8	8.2	73946	641

Table 3: Performance of multigrain JD algorithm with block size of 4. Each node description indicates what machines compose the solve groups. For, instance, $C_{16}C_{16}$ indicates two solve groups each consisting of 16 processes on 16 Ultra 60's. If only two solve groups were used, then the four correction equations were solved two at a time. Columns "FG1" and "FG4" indicate the percent decrease/increase in runtime of multigrain over its fine-grain counterpart with block sizes 1 and 4. A negative value represents an increase.

should be preferred when there is a low overhead network. However, in Grids and COWs the network load may vary dynamically as a result of competing parallel applications or due to a few impaired links in a Grid environment.

In such cases, it may be best to dynamically switch from fine grain to multigrain mode. Currently, a system is being installed as part of the SciClone cluster that can mimic arbitrary synthetic network loads. We plan to run our code under various dynamic configurations of this network, and interface it with Remos [11], a network performance monitoring tool. Based on Remos' runtime statistics and on a computational model of our algorithm, we will compute estimates for the execution time of multigrain and fine grain (with a block size of 1) and choose the smaller one dynamically. We plan to report preliminary results in the camera ready copy.

References

[1] L. Bergamaschi, G. Pini, and F. Sartoretto. Parallel preconditioning of a sparse eigensolver. *Parallel Computing*, 27(7):963–76, 2001.

[2] Edmond Chow. ParaSails: Parallel sparse approximate inverse (least-squares) preconditioner. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551, 2001.

[3] P. Concus, G. H. Golub, and D. P. O'Leary. *A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations*. Academic Press, New York, NY, 1976.

[4] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, 1998.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[6] I. Foster and C. Kesselman, editors. *The Grid — Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[7] A. S. Grimshaw and W. A. Wulf et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), 1997.

[8] K. Hwang and Z. Xu. *Scalable Parallel Computing*. WCB/McGraw Hill, 1998.

[9] George Karypis and Vipin Kumar. METIS: unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[10] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–85, 1998.

[11] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *7th IEEE Smposium on High-Performance Distributed Computing, IEEE*, July 1998.

[12] J. R. McCombs, R. T. Mills, and A. Stathopoulos. Dynamic load balancing of an iterative eigensolver on Grids of heterogeneous clusters. 2002. submitted.

[13] Dianne P. O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5:127–139, 1987.

[14] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, 1998.

[15] Y. Saad and K. Wu. Parallel SPARSe matrix LIBrary (P_SPARSLIB): the iterative solvers module. Technical Report 94-008, Army High Performance Computing Research Center, Minneapolis, 1994.

[16] Yousef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, March 1993.

[17] Yousef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1996.

[18] G. L. G. Sleijpen, A. G. L. Booten, D. R. Fokkema, and H. A. van der Vorst. Jacobi-davidson type methods for generalized eigenproblems and polynomial eigenproblems. *BIT*, 36(3):595–633, 1996.

[19] G. L. G. Sleijpen and H. A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, 1996.

[20] A. Stathopoulos and C. F. Fischer. Reducing synchronization on the parallel Davidson method for the large,sparse, eigenvalue problem. In *Supercomputing '93*, pages 172–180, Los Alamitos, CA, 1993. IEEE Comput. Soc. Press.

[21] A. Stathopoulos and J. R. McCombs. A parallel, block, Jacobi-Davidson implementation for solving large eigenproblems on coarse grain environments. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2920–2926. CSREA Press, 1999.

[22] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.