Dynamic Load Balancing of an Iterative Eigensolver on Networks of Heterogeneous Clusters *

James R. McCombs Richard Tran Mills Andreas Stathopoulos
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187-8795
{mccombjr, rtm, andreas}@cs.wm.edu

Abstract

Clusters of homogeneous workstations built around fast networks have become popular means of solving scientific problems, and users often have access to several such clusters. Harnessing the collective power of these clusters to solve a single, challenging problem is desirable, but is often impeded by large inter-cluster network latencies and heterogeneity of different clusters. The complexity of these environments requires commensurate advances in parallel algorithm design.

We support this thesis by utilizing two techniques: 1) multigrain, a novel algorithmic technique that induces coarse granularity to parallel iterative methods, providing tolerance for large communication latencies, and 2) an application-level load balancing technique applicable to a specific but important class of iterative methods. We implement both algorithmic techniques on the popular Jacobi-Davidson eigenvalue iterative solver. Our experiments on a cluster environment show that the combination of the two techniques enables effective use of heterogeneous, possibly distributed resources, that cannot be achieved by traditional implementations of the method.

1. Introduction

The power and low-cost of todays workstations and the introduction of inexpensive high-speed networking media have made clusters of workstations (COWs) a cost-effective means of parallel processing for an increasing number of scientific applications. Massively parallel processors (MPPs) are based on the same design philosophy, targeting a higher performance albeit at a higher cost. The emergence of Grids promises to deliver this higher performance to a large number of applications by enabling the collective use of various existing computational environments [11, 15].

Grids present a twofold challenge: First, provide an easy, secure and integrative way for users to access these environments, which has been the focus of much research recently [10, 13, 30]. Second, devise methods that can harness effectively the power of these environments, through new parallel algorithmic designs and adaptation to the runtime system [1, 3, 4]. This challenge has received little attention in the literature and it is the focus of our research.

In this paper, we concentrate on iterative methods, which are of central importance in many scientific and engineering applications. Specifically, we focus on iterative methods for the numerical solution of large, sparse, eigenvalue problems, although much of our discussion applies to a wider class of iterative methods. Traditionally, iterative methods have been implemented on MPPs [7, 23] and COWs in a fine grain way. Every iteration requires a matrix-vector product, an application of the preconditioner operator, and several inner products. Inner products require a global reduction, an operation that does not scale with the number of processors. But more importantly, communication overheads have not kept up with the rapid growth of bandwidth in recent networks. In the case of large number of processors in MPPs or high overhead interconnection networks in COWs, such costs can limit the scalability of the application [18, 27, 29]. In a Grid environment, the significantly higher overheads can completely incapacitate these methods. Block iterative methods and preconditioners with high degree of parallelism, such as domain decomposition, are often employed to increase granularity and thus scalability [7]. However, their granularity is still too fine to be useful on Grids.

Beyond issues that relate to the communication primi-

*Work supported by the National Science Foundation (ITR/ACS-0082094 and ITR/AP-0112727), a DOE computational sciences graduate fellowship, 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).

tives of the algorithm and the underlying network, scalability and often usability are inhibited by the resource imbalances on heterogeneous and/or distributed shared environments. Most parallel implementations of iterative methods assume homogeneous parallel processors. Even when such implementations scale well within a cluster, there may be little gain in speedup and possibly a performance degradation if heterogeneous clusters are linked together, as is often the case in collections of clusters and Grid environments. A common approach is to partition the data according to the relative speeds of the processors, either statically [14, 16] or during execution with expensive repartitioning packages [6, 17]. However, this approach is not effective in the presence of dynamic external load on some of the COWs. On the other hand, scheduling parallel programs on shared environments is also intrinsically difficult, because the system cannot predict the variable requirements of programs [8].

In [28], we showed how a combination of coarse and fine grain parallelism on a block Jacobi-Davidson eigenvalue solver can provide tolerance for high network latencies. In [20], we modified this code to adapt to external CPU and memory load, by allowing each node to perform its local preconditioning to different accuracy. The scope of both [28] and [20] is limited to a small number of processors, but in [19], we extended the approach of [28] to a large number of processors by utilizing the notion of multigrain, where an arbitrary number of processors is split into subgroups, each performing a different preconditioning operation. The multigrain approach is effective, but when used by itself in heterogeneous environments, or ones in which network topology necessitates that the subgroups be of different sizes, it can accentuate or even introduce load imbalance. In this paper we show that the combination of multigrain with a simple, though non-traditional, load balancing technique eliminates these shortcomings and provides an empowering mechanism for the efficient use of Grid environments.

2. Load balancing for a class of iterative methods

A common algorithmic paradigm for parallel programs is that of *synchronous iteration*, in which processors perform local work to complete an iteration of a loop, and then synchronize before proceeding to the next iteration. We are interested in a very specific but important case of synchronous iteration: one in which the amount of work completed by each processor during an execution of the loop may be varied arbitrarily without detriment to the end result. With smaller amount of work per iteration, the target can still be reached, though with more iterations. We decompose algorithms in this category into two phases: 1) A *control phase*, during which synchronous interactions update global knowledge of the current state, allowing each processor to make better decisions later. 2) A *flexible phase*, during which local execution of the loop occurs. It is "flexible" insofar as each processor can vary the amount of work that it does during this phase. We designate this type of parallelism *flexible phase iteration*. This flexibility provides a unique opportunity for load balancing: Since each processor need not do the same amount of work as its peers, perfect load balance can be achieved in the flexible phase by limiting all processors to the same time T during its execution.

Although it is very specific, several important algorithms used in the sciences and engineering can fit this model. One class of such algorithms includes stochastic search optimizers such as genetic algorithms and simulated annealing. Another important class of algorithms amenable to a flexible phase iteration structure are Krylov-like iterative methods [23], which are widely employed to solve systems of linear equations, eigenvalue problems, and even non-linear systems. These methods usually utilize a preconditioner, which, at each outer (Krylov) iteration, improves the current solution of the method by finding an approximate solution to a correction equation. In parallel implementations, if this correction is found iteratively, different processors can apply a different number of iterations on their local portions of the correction equation [24]. Thus, the preconditioning step is a flexible phase, and the outer iteration, where the processors update their corrections, is a control phase.

3. The coarse grain JD method

Many applications involve the solution of the eigenvalue problem, $A\tilde{\mathbf{x}}_i = \tilde{\lambda}_i\tilde{\mathbf{x}}_i$ for the extreme (largest or smallest) eigenvalues, $\tilde{\lambda}_i$, and eigenvectors, $\tilde{\mathbf{x}}_i$, of a large, sparse, symmetric matrix A. One such method that has attracted attention in recent years is the Jacobi-Davidson (JD) method [26, 25]. This method constructs an orthonormal basis of vectors V that span a subspace $\mathcal{K}$ from which the approximate *Ritz values*, λ_i, and *Ritz vectors* $\mathbf{x}_i$ are computed at each iteration. These approximations and the residual $\mathbf{r}_i = A\mathbf{x}_i - \lambda\mathbf{x}_i$ are then used to solve the correction equation:

$$(I - \mathbf{x}_i\mathbf{x}_i^T)(A - \lambda_i I)(I - \mathbf{x}_i\mathbf{x}_i^T)\epsilon_i = \mathbf{r}_i, \qquad (1)$$

for the vector ϵ_i, an approximation to the error in $\mathbf{x}_i$. The basis V is then extended with these vectors. Below we show a block variant of JD that extends V by computing k correction vectors at each iteration.

During the projection phase (steps 1-7), the block algorithm finds the k smallest Ritz eigenpairs and their residuals. During the correction phase, k different equations (1) are solved approximately for ϵ_i, usually by employing an iterative solver for linear systems such as BCGSTAB [23].

```
Algorithm: Block JD
starting with $k$ trial vectors $\epsilon_i$
While not converged do:
1. Orthogonalize $\epsilon_i, i = 1 : k$. 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 $H\mathbf{y}_i = \lambda_i \mathbf{y}_i,\ i = 1 : k$ (all procs)
6. $\mathbf{x}_i = V\mathbf{y}_i,\ \mathbf{z}_i = W\mathbf{y}_i, i = 1 : k$ (local rows)
7. $\mathbf{r}_i = \mathbf{z}_i - \lambda_i \mathbf{x}_i,\ i = 1 : k$ (local rows)
8. Correction equation: Solve eq. (1) for each $\epsilon_i$
end while
```

Block methods improve robustness for difficult eigenproblems where the sought eigenvalues occur in multiplicities or are clustered together [21]. In general, the total number of outer JD iterations reduces with larger block sizes, but the total number of matrix-vector products increases [12, 22, 28]. However, larger blocks yield better cache efficiency, and better computation to communication ratio (coarser granularity) in parallel programs.

The above block JD method is given in a data parallel (fine grain) form. The rows of each of the vectors $\mathbf{x}_i$, $\mathbf{r}_i$, $\mathbf{z}_i$, and ϵ_i as well as of the matrices A, V, and W are partitioned evenly among the processors. Thus, vector updates are local, while dot products require a global reduction. Matrix-vector products with A are performed in parallel by user-provided subroutines. Fine grain implementations can scale well when synchronization during the reductions is efficient. However, in many COWs, scalability is impaired by high overheads, despite the sometimes high network bandwidth.

For high latency/overhead environments, coarser granularity is needed. In [28], we developed a hybrid coarse-grain JD algorithm, which we call JDcg. It is based on the assumption that each compute node can store A in its entirety, which is reasonable when the matrices are sparse, and especially when they are not stored explicitly but represented only as a matrix-vector product function. Our method attempts to improve upon the performance of the fine grain implementation by eliminating communication between processors during the correction phase. We do this by requiring the number of processors to be equal to the block size and having each processor solve a distinct correction equation independently of the other processors.

Steps 1-7 of the algorithm are performed as before in a data parallel manner involving all the processors. However, just before the start of the correction phase, each processor gathers all the rows of one of the block vectors via an all-to-all operation. Each processor then solves its respective correction equation independently with BCGSTAB. The coarse-grain version of step 8 is summarized as follows:

```
8. Coarse grain correction equation
   All-to-all: send local pieces of $\mathbf{x}_i, \mathbf{r}_i$ to proc $i$,
          receive a piece for $\mathbf{x}_{myid}, \mathbf{r}_{myid}$ from proc $i$
   Apply $m$ steps of (preconditioned) BCGSTAB on
          eq. (1) with the gathered $\mathbf{x}_{myid}, \mathbf{r}_{myid}$
   All-to-all: send the $i$-th piece of $\epsilon_{myid}$ to proc $i$,
          receive a piece for $\epsilon_i$ from proc $i$
```

The parallel speedup of JDcg can be improved arbitrarily by increasing the number of BCGSTAB iterations (m). However, the total number of matrix-vector multiplications increases if m is chosen too large. Fortunately, large values for m are often necessary to solve numerically difficult eigenproblems. In [28], we have demonstrated the effectiveness of JDcg in hiding the communication latencies of slow networks.

4. JDmg: the multigrain JDcg

The requirement of JDcg that each processor has access to the entire matrix may be too stringent in environments with a large number of processors, where memory demanding applications need to scale their problem size with the number of nodes. Furthermore, even if the memory is available a large block size (equal to the number of processors) is expected to significantly increase the total number of matrix-vector multiplications. This non work conserving behavior limits the use of JDcg to small clusters of 4-8 processors.

Yet, the same principle can be used to introduce coarser granularity on MPPs. Assume an MPP with 256 processors, and the JD algorithm with a block size of 4 executing in fine grain on this MPP. We can envision the MPP split in four groups, which we call solve groups, of 64 processors each, and during the JD correction phase each solve group gathers a distinct residual and solves a distinct correction equation. The only difference from the JDcg is that the correction equation is solved by a data parallel linear solver on 64 processors. The benefits stem from the lower communication latencies associated with a cluster of one fourth the size of the original. In a similar situation, a fine grain JD method running on four COWs (possibly heterogeneous to each other), could assign a different correction equation to each COW, effectively hiding the latencies of the network.

We use the term *multigrain* to refer to this extension of our coarse-grain technique, where the number of processors P is greater than the block size k. The only memory requirement posed by multigrain is that each processor stores k times more rows than fine grain alone. With typical block sizes of 4-8, this does not limit the memory scalability of the method. In multigrain, matrix-vector multiplications occur at two levels of granularity, so A is partitioned both in fine grain over all processors and in coarse grain over each solve group. Similarly to JDcg, an all-to-all transfers information between the two levels. In the particular case where k divides P, or k is small compared to P, the all-to-all can be made more efficient if the solve groups are chosen to have P/k processors, as described in [19]. This is typically the case with MPPs or COWs with large numbers of homogeneous processors. In clusters of heterogeneous processors or simply clusters with different processor numbers, the solve groups generally are not of size P/k, but 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.

5. Load balancing JDcg through algorithmic modifications

JDcg fits the flexible-phase iteration model: The corrections ϵ_i need not be computed to the same accuracy, so the correction phase is flexible. The highly-synchronous projection phase is the control phase. Thus, we can load-balance JDcg by restricting each processor to a fixed time T in the correction phase. Even though imbalances will persist during the brief projection phase, this virtually eliminates overall load imbalance, because the correction phase dominates the execution time. Although some vectors ϵ_i may be computed to lower accuracy, this only increases the number of outer iterations and often decreases the amount of total work.

To determine an appropriate T, we follow the commonly used guideline that BCGSTAB be iterated to a convergence threshold of 2^{-iter}, where $iter$ is the number of outer iterations [9]. Using classical convergence bounds for Conjugate Gradient [23], we determine heuristically an "optimal" number of iterations m that corresponds to the 2^{-iter} threshold. To avoid hurting convergence by too large an m, we set a maximum bound $maxits$ for m. T is then the time required by the fastest processor to complete m BCGSTAB steps. The algorithm for the load-balanced correction follows:

Load-balanced correction phase of JDcg

1. In the first JDcg (outer) iteration, do no load balancing. Each processor performs $maxits$ BCGSTAB iterations, calculates the rate at which it performed them, and sends its rate to all other processors.

2. In subsequent JDcg iterations, use the rate measured in the previous iteration to rank the processors from fastest to slowest. To ensure numerical progress, faster processors gather the more extreme eigenpairs and residuals during the all-to-all of step 8 of JDcg.

3. Use the highest rate to determine T, and then iterate on the correction equation for this time.

```
Algorithm: Load balanced JDcg
lbs = lb_new_lbstruct(MPI_COMM_WORLD);
Until convergence do:
// Control phase
    Perform projection phase, steps 1–7 of JDcg
    Determine optimal number of iterations $optits$
    lb_decide(lbs,$optits$,LB_USE_FASTEST);
    ordering = lb_get_index(lbs);
    All-to-all: faster procs receive more critical residuals
// Flexible Phase
    lb_section_start(lbs);
        for ($ops = 0$; lb_continue(lbs,$ops$,1) ; $ops$++)
            Perform one BCGSTAB step on eq. (1)
    lb_section_end(lbs,$ops$);
end do
```

Figure 1. Load balancing JDcg with LBLIB. `lb_new_lbstruct()` is the constructor for the LBS type. `lb_decide()` exchanges processor rates and estimates the time T for the fastest processor to perform $optits$ iterations. `lb_get_index()` returns an array of processor id's, sorted from slowest to fastest. `lb_section_start()` and `lb_section_end()` denote the beginning and end of the flexible section. `lb_continue()` returns TRUE if, using the current rate, it predicts that there is enough of the allotted time T remaining for the processor to perform another BCGSTAB iteration.

In [20], we obtained good results using this scheme to balance CPU load in the presence of loads introduced by external jobs. Additionally, we demonstrated the viability of a method of avoiding thrashing when the memory requirements of JDcg and any external loads exceed the available memory: we recede JDcg during the correction phase, allowing the competing job to use 100% of the CPU and memory, hopefully speeding its completion and hence relinquishing the resources.

To facilitate general use of our CPU and memory load balancing strategies, we have written an object-based C library, LBLIB, that hides much of the required bookkeeping from the application programmer. To simplify data management and provide information hiding, data required for resource balancing are stored in a variable of the defined type LBS, and are accessed through LBLIB functions. Figure 1 illustrates the use of LBLIB to balance CPU load in JDcg.

6. Enabling Grid computations

Multigrain parallelism hides communication latencies, but, used by itself, it can accentuate or even introduce load imbalance. For instance, if three identical processors are used to run JD in fine-grain, there is perfect load balance. However, if the same three processors are used to run multigrain JD with a block size of two, one solve group will contain two processors and the other only one. Since the latter group is only half as fast as the former, the load imbalance is now 33.33%! Clearly, the utility of multi-grain is limited if not used in conjunction with a load balancing scheme.

Fortunately, JDmg also fits the flexible-phase iteration model and can be load balanced in the same manner as JDcg. The only difference is that solve groups, rather than individual processors, are the entities that work independently during the correction phase. Processor 0 of each solve group is responsible for coordinating the load balancing, calling the LBLIB functions, much as in figure 1. Prior to the correction phase, each processor 0 uses the execution rates of the independent solve groups to determine which block its group should receive, and broadcasts this information to the other members. During the correction phase, the 0th processors determine whether to perform another iteration or to halt, and broadcast this information to their group.

Besides dynamic load and memory balancing, multigrain allows also for an adaptive choice of granularity. Based on measured network load either during the first iteration or from performance monitoring libraries [30], the code could decide on the number of solve groups and which nodes each group should be assigned. The slowest links should be assigned between groups or within the same small solve group. We do not explore this possibility in this paper, but we use simple group choices dictated by our experimentation environment.

6.1. Experiments on a Grid-like environment

We conducted a series of experiments with JDmg, using it in fine-grain and multigrain modes (both with and without load balancing). The experiments were run on SciClone, a heterogeneous cluster of workstations at the College of William and Mary. SciClone is an ideal testbed for Grid applications because it employs three different processor configurations, two networking technologies, and is organized as a cluster of subclusters. Thus it effectively captures three levels of heterogeneity that are characteristic of Grid-based computing: node architecture, networks, and number of nodes at a site. Figure 2 details the architecture of the portion of SciClone that we use. In all experiments, we use JDmg with block size $k = 4$ to compute the lowest eigenvalue of a matrix derived from a 3-D finite element problem [2]. The matrix is of dimension $268,515$ and contains $3,926,823$ non-zero elements. BCGSTAB is preconditioned with a sparse approximate inverse preconditioner from the ParaSails library [5].

Nodes	Time	Mvecs
A	2912	12564
AB	1784	11944
C_{32}	2597	12564
C	2087	11944
D_4	23944	13600
D_8	11424	12808
D	6560	13466
AC_{32}	1489	11944
ABC	1714	13104
AD	3378	13178
ABD	1856	12914
$C_{32}D$	2679	13178
CD	1970	12914
$AC_{32}D$	1813	12914
ABCD	1732	14266

Table 1. Performance of the fine-grain JD running on different node configurations. "Time" is wall-clock time in seconds and "Mvecs" is the number of matrix-vector products computed. Strings within the "Nodes" column specify what nodes are used for an experiment: For each subcluster that is utilized, its letter is given. If a subscript n is appended to that letter, it indicates that only n processors of the subcluster are utilized; if no subscript is present, all processors are utilized. For instance, "C" means that all 64 processors of cluster C are used, while $C_{32}D$ indicates that 32 processors from cluster C are used together with all the processors from cluster D.

To enable measurement of load imbalance in the multigrain experiments, we timestamp the beginning and end of synchronous communication calls. Since these calls are synchronous, the imbalance on a node can be calculated as the time each communication took to complete minus the minimum time the same call took to complete on any of the processors. Summing these imbalances over all communications and processors, we obtain an aggregate of all wasted CPU cycles. Dividing by the sum of wall-clock times over all processors yields the percentage of time wasted due to load imbalance. Note that we do not timestamp communications internal to the solve groups during the correction phase, because we are interested only in the imbalance across solve groups. Additionally, we do not timestamp communication calls associated with matrix-vector products because those are performed via ParaSails calls. This causes a slight underestimate of the overall load imbalance during the projection phase. The load imbalance estimates are quite accurate, however, because the formation of the matrix-vector products in the projection phase comprises a negligible part of the execution time.

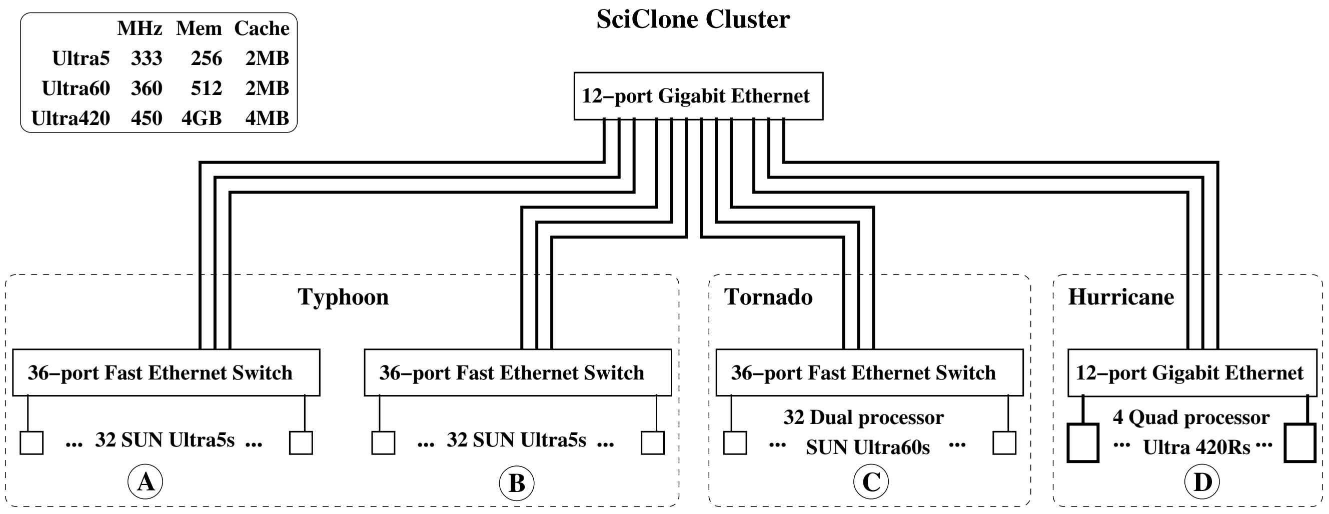

Figure 2. SciClone: The William and Mary heterogeneous cluster of three homogeneous clusters: Typhoon, Tornado (also called C), and Hurricane (also called D). We distinguish between A and B, the subclusters of Typhoon, because their intercommunication passes through the Gigabit switch. There are three levels of heterogeneity: node architecture, number of nodes, and networks.

We have tested the fine-grain implementation on several node combinations from various clusters (Table 1). We mention a few important observations here. The speedup from 32 to 64 Ultra5's (experiments A and AB) is about 1.63, but the speedup on the Ultra60's (experiments C_{32} and C), machines with faster processors and more cache memory, is only about 1.24. We suspect that the poor speedup on cluster C may be the result of two MPI processes on each node contending for the network interface. Similar behavior is observed on the SMPs of the D cluster.

Further improvement in fine-grain speedup can be obtained by using clusters A and C_{32}, with only one processor on each Ultra60. For instance, there is a speedup of 1.74 between experiments C_{32} and AC_{32}, and 1.95 between A and AC_{32}. For this small number of relatively homogeneous nodes the good scalability leaves little room for multigrain improvements. However, multigrain improves performance if the size of the solve groups increases. In Table 2, experiments $(AB)C$ and $ABC_{32}C_{32}$ yield significantly better timings compared to fine-grain test ABC because multigrain is able to hide the latency introduced by the additional processors.

As discussed earlier, multigrain alone may not result in performance improvements if the rates at which each subgroup performs matrix-vector products vary greatly. With the exception of two experiments (AD and $(AB)C_{32}C_{32}D$), when subcluster D is used in conjunction with other subclusters the multigrain code performs much worse than its fine-grain counterpart. This trend is an example of multigrain's tendency to accentuate load imbalance: the much smaller number of processors in subcluster D results in a slower solve group, and thus greater load imbalance.

The experiments using multigrain with load balancing, however, yield much better results. When combining clusters of disparate power (e.g., (AB)D or CD) the load balanced multigrain method outperforms significantly both the unbalanced multigrain and fine grain methods. When the clusters involved are relatively homogeneous (e.g., AC_{32}, (AB)C or $ABC_{32}C_{32}$), load balancing still performs comparably to multigrain and always improves performance over fine grain. Overall, load imbalance is almost always below a tolerable level of 10%, and the problem is solved twice as fast as any combination of clusters using traditional fine grain methods.

7. Conclusions and future work

As computing environments become increasingly complex, consisting of collections of heterogeneous COWs either in the same local area network or geographically dispersed, it becomes increasingly important to devise new algorithmic techniques that tolerate high network tolerances and that adapt to the (often dynamically) varying system load. We have presented two such techniques, multigrain and an application-level load balancing strategy, that apply to iterative methods. The key idea for multigrain is that it transfers the bulk of the convergence work from the outer iteration to an inner iteration that processors can execute for a long time independently, thus tolerating arbitrary large latencies. The key idea for the load balancing technique is to

	Without load balancing			With load balancing		
Nodes	Time	Mvecs	%imbal	Time	Mvecs	%imbal
AD	3265	13058	36.17	1746	10515	4.47
$A_{16}A_{16}D_8D_8$	4022	16910	38.96	1692	11208	5.14
$C_{32}D$	3282	13058	39.57	1631	10478	5.01
$A_{16}A_{16}C_{16}C_{16}$	1405	12424	11.02	1546	14698	5.24
$C_{16}C_{16}D_8D_8$	4037	16910	41.71	1544	10711	6.22
AC_{32}	1585	12730	9.46	1450	12833	2.05
CD	3495	13996	52.37	1381	9608	7.68
$C_{32}C_{32}D_8D_8$	3132	13124	58.32	1284	11202	9.94
(AB)C	1198	12656	11.97	1214	13653	5.98
(AB)D	3500	13996	55.42	1126	8870	8.97
$ABC_{32}C_{32}$	981	12240	21.00	991	14167	8.99
ABD_8D_8	3152	13124	61.58	941	8680	11.78
$(AB)C_{32}C_{32}D$	1870	14534	52.64	724	9481	15.05

Table 2. Performance of the multigrain JD running on different node configurations, with and without load balancing. "Nodes", "Time" and "Mvecs" are as in Table 1. "%imbal" is the percentage of time wasted due to load imbalance. When multiple subclusters are assigned to one block vector, they are grouped together with parentheses. E.g., "(AB)" indicates that subclusters A and B work together on the same block vector (are in the same solve group), whereas "AB" indicates that subclusters A and B work on different block vectors (each composing their own solve group).

let every processor execute on the inner iteration for a fixed amount of time, thus achieving ideal load balancing during the dominant phase of the algorithm. Iterative methods for the numerical solution of eigenvalue problems are notoriously synchronous. Yet, by applying our two techniques on such a method, we have managed to significantly improve scalability on a collection of heterogeneous clusters over traditional fine grain implementations.

Future extensions include identifying potential applications that fit into the flexible iteration model, and dealing with the situation of heterogeneous clusters of heterogeneous workstations. The latter case can be addressed by applying two levels of our load balancing library; one inter-cluster and one intra-cluster using a domain decomposition preconditioner and a flexible version of GMRES.

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *18th Symposium on Operating Systems Principles (SOSP '18)*, October 2001.

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

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Supercomputing 1996*, Fall 1996.

[4] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *9th IEEE Inlt. Symp. on High Performance Distributed Computing*, August 2000.

[5] E. 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.

[6] K. Devine, B. Hendrickson, E. Boman, M. St.John, and C. Vaughan. Zoltan: A dynamic load-balancing library for parallel applications; user's guide. Technical Report Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

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

[8] D. G. Feitelson and L. Rudolph, editors. *2000 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1911. LNCS, 2000.

[9] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1), 1998.

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

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

[12] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. In J. R. Rice, editor, *Mathematical Software III*, pages 361–377, New York, 1977. Academic Press.

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

[14] B. Hendrickson and R. Leland. The Chaco useer's guide, Version 1.0. Technical Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM, 1992.

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

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

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

[18] S. Kuznetsov, G. C. Lo, and Y. Saad. Parallel solution of general sparse linear systems. Technical Report UMSI 97/98, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997.

[19] J. R. McCombs and A. Stathopoulos. Multigrain parallelism for eigenvalue computations on networks of clusters. In *Proceedings of the Eleventh IEEE International Symposium On High Performance Distributed Computing*, pages 143–149, July 2002.

[20] R. T. Mills, A. Stathopoulos, and E. Smirni. Algorithmic modifications to the Jacobi-Davidson parallel eigensolver to dynamically balance external CPU and memory load. In *2001 International Conference on Supercomputing*, pages 454–463. ACM Press, 2001.

[21] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, 1998.

[22] Y. Saad. On the rate of convergence of the Lanczos and the block-Lanczos methods. *SIAM J. Numer. Anal.*, 17:687–706, 1980.

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

[24] Y. Saad and M. Sosonkina. Non-standard parallel solution strategies for distributed sparse linear systems. In A. U. P. Zinterhof, M. Vajtersic, editor, *Parallel Computation: Proc. of ACPC'99*, Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A. Stathopoulos, S. Öğüt, Y. Saad, J. R. Chelikowsky, and H. Kim. Parallel methods and tools for predicting material properties. *Computing in Science and Engineering*, 2(4):19–32, 2000.

[30] 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.