Dynamic Load Balancing of an Iterative Eigensolver on Grids of Heterogeneous Clusters [*]

James R. McCombs
Department of Computer Science
College of William and Mary
Williamsburg, Virginia
23187-8795
mccombjr@cs.wm.edu

Richard Tran Mills
Department of Computer Science
College of William and Mary
Williamsburg, Virginia
23187-8795
rtm@cs.wm.edu

Andreas Stathopoulos
Department of Computer Science
College of William and Mary
Williamsburg, Virginia
23187-8795
andreas@cs.wm.edu

ABSTRACT

Clusters of homogeneous workstations built around fast networks have become popular and cost effective means of solving scientific problems. Users commonly have access to several such clusters, either within the same or in different computational environments. Harnessing the collective power of these clusters to solve a single, challenging problem is highly desirable, and the focus of much recent Grid research. However, huge network latencies, heterogeneity of different clusters and resource sharing impede the efficient utilization of Grids except by some naturally parallel applications. It is our thesis that the complexity of these environments requires commensurate advances in parallel algorithm design and in the interaction of the algorithms with the runtime system.

We support this thesis by first illustrating the use of an application-level load balancing technique that applies to a wide class of iterative methods. The key characteristic of this class is that each processor can vary the amount of work it performs during the iteration without detriment to the end result. Dynamic load balancing is achieved at each iteration by stopping each processor after a fixed amount of time. We develop a library of functions that facilitate the implementation of the load balancing features on any method in the class. Second, we introduce *multigrain*, a novel algorithmic technique that induces coarse granularity to parallel iterative methods by modifying appropriately their preconditioning phase. Multigrain allows traditionally highly synchronous iterative methods to tolerate the large communication latencies in Grids, and it also enables the use of the application-level load balancing library. We implement both algorithmic techniques on the popular Jacobi-Davidson eigenvalue iterative solver. Our experiments on a Grid-like environment show the effective use of the heterogeneous resources, something that cannot be achieved by traditional implementations of the method.

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

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 high performance 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 [15, 19].

Unlike MPPs and most COWs, Grid environments are usually heterogeneous, consisting of several COWs of relatively homogeneous processors. This heterogeneity arises naturally in realistic situations. For example, different COWs may be purchased for specific applications or tasks. It is also common that hardware upgrades occur in stages, replacing only specific COWs, or subsets of them. In other cases, the Grid is simply a network of loosely coupled workstations. Finally, even when such an environment is not locally available, the user often has access to some geographically dispersed, heterogeneous COWs. The challenge is twofold. First, provide an easy, integrative way for users to access these environments, which has been the focus of much Grid research recently [14, 17, 36]. Second, devise methods that can harness effectively the power of these environments. This second challenge has received little attention in the literature and it is the focus of our research.

Iterative methods are a critical part of many scientific and engineering applications. As theoretical and technological advances allow scientists to tackle increasingly larger problems, the performance and robustness of iterative methods becomes of central importance in high performance scientific computing. In this paper, 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. To improve convergence rate and robustness, eigenvalue iterative methods typically use a preconditioning technique [12, 30]. Parallel computing is the other main way of improving execution time and solvable problem size of these applications. However, achieving high performance with iterative methods can be challenging on todays computational platforms.

Traditionally, iterative methods have been implemented on MPPs [10, 30] and COWs in a fine grain way. Every iteration requires a matrix-vector multiplication, 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 explosive growth of bandwidth in recent networks. In case of large number of processors in MPPs or high overhead interconnection networks in COWs, such costs can limit the scalability of the application [23, 33, 35]. In a Grid environment, the significantly higher overheads can completely incapacitate these methods. Block iterative methods and preconditioners with higher degree of parallelism, such as domain decomposition, are often employed to increase granularity and thus scalability [10]. However, the granularity of these methods is still too fine to be useful on Grids.

Beyond issues that relate to the communication primitives of the algorithm and the underlying network, scalability, and often usability, is 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. Powerful load balancing techniques such as master-worker, and pool of tasks [13] are not applicable in the context of iterative methods. A common approach is to partition the data according to the relative speeds of the processors, either statically [18, 20] or during execution with expensive repartitioning packages [9, 21]. 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 [11].

We maintain that the complexity of these environments gives rise to problems that cannot be addressed solely through hardware and operating systems advances. New levels of sophistication are required in parallel algorithm design and in the interaction of the algorithms with the runtime system. Current research has focused either on middleware between the application and the system [6, 7, 1, 22], or on performance monitoring and prediction libraries such as NWS [36] and PAPI [4, 5]. However, besides some preliminary work in AppLeS [3], no attention has been given on how to use this system information to dynamically change the algorithm for better resource utilization.

In [34], we described a unique parallelization approach that is designed to tolerate high network latencies by combining both coarse and fine grain in a block Jacobi-Davidson eigenvalue solver. Each processor gathers a different vector from the block on which it applies the preconditioning step independently, thus improving granularity and scalability. The underlying assumption is that individual nodes in a COW have access to the whole matrix A. This is often the case because of increasing memory sizes, and also because many applications compute the matrix-vector multiplication on the fly. In [24], we modified this coarse-grain code to be able to adapt to external CPU and memory load. The first key idea is to allow each node to perform its local preconditioning to different accuracy, stopping after a fixed amount of time, and thus achieving ideal load balance. The second key idea tries to reclaim CPU cycles lost to memory thrashing caused by competing jobs, by receding the preconditioning phase on a node when memory thrashing is detected. The scope of both [34] and [24] is limited to a small number of processors, but the positive experimental results show the huge potential of a broader approach.

In this paper, we extend the scope of our previous research to environments of heterogeneous clusters of homogeneous processors. First, we identify a model for a wide class of iterative methods that can self adapt to achieve resource balancing. Our previous coarse grain modifications allow the Jaobi-Davidson method to be described in this model. Similar modifications could broaden the range of methods that can be described in the model. Therefore, we extend the coarse grain idea to the notion of multigrain, where an arbitrary number of processors can be split into groups, each group performing a different preconditioning operation. This can alleviate the effects of global synchronization, but more interestingly, it can tolerate high inter-cluster latencies in Grids, if each cluster is represented by a group. To facilitate the porting of the dynamic load/memory balancing capabilities to any method in this class, we also develop a C library. Finally, we describe how the combination of multigrain and application-level load balancing provides the empowering mechanism for effective use of Grids, within the same local area network, in geographically different locations, or with a combination of both. Our experiments on a collection of heterogeneous COWs yield time improvements that cannot be obtained otherwise.

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. The following illustrates synchronous iteration on a single instruction, multiple data computer:

```
while(target state has not been reached) {
    body(my_rank);
    Synchronous interaction(s)
}
```

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 body may be varied arbitrarily without detriment to the end result. With smaller amount of work per iteration, the target can still be reached only 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 body 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.*

Although it is very specific, several important algorithms used in the sciences and engineering fit this model. One class of such algorithms includes stochastic search optimizers such as genetic algorithms and simulated annealing. In their synchronous formulations, processors independently perturb an initial set of configurations to search for more optimal ones. After a certain number of successes are obtained by each processor, they synchronize to decide upon new configurations, and then continue their search. Because synchronization could occur before each processor has had a certain number of successes, the search phase is a flexible phase. The decision phase is the control phase.

Another important class of algorithms that are amenable to a flexible phase iteration structure are Krylov-like iterative methods [30]. These methods are widely employed to solve systems of linear equations, eigenvalue problems, and even non-linear systems. Their main iteration involves vector updates and synchronous dot-products, which allow for little flexibility. However, flexibility can be introduced with preconditioning. At each outer (Krylov) iteration, preconditioning improves the current solution of the method by finding an approximate solution to a correction equation. Flexible variants of Krylov methods can solve this equation iteratively [29]. In parallel implementations, this allows different processors to apply different number of iterations on their local portions of the correction equation [26]. Thus, the preconditioning step is a flexible phase, and the outer iteration, where the processors update their corrections, is a control phase. In this paper, we demonstrate an alternative, multigrain approach for introducing more flexibility during the preconditioning phase of a Krylov method.

Since each processor need not do the same amount of work as its peers, perfect load balancing can be achieved during the flexible phase: If all processors are limited to the same time T for an execution of the flexible phase, any differences in the speed of the processors—even due to changes in the external load—cannot cause load imbalance.

3. THE COARSE GRAIN JACOBI-DAVIDSON 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 [32, 31]. 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 $\mathbf{x}_i$. These vectors are then used to extend the basis V. Below we show a block variant of JD that extends V by computing k correction vectors at each iteration.

```
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 processor.
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
```

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 or GMRES [30].

Block methods improve robustness for difficult eigenproblems where the sought eigenvalues occur in multiplicities or are clustered (very close) together [25]. In general, the total number of outer JD iterations reduce with larger block sizes, but the total number of matix-vector operations increase [16, 28, 34]. However, larger blocks introduce repetitive patterns of computation yielding 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 require no communication, while dot products require a global reduction. Matrix-vector multiplications with A are performed in parallel by user-provided subroutines. Fine grain implementations can scale well when synchronization during global reductions is performed efficiently. However, in many COWs, scalability is impaired by high overheads, despite the sometimes high bandwidth of the network.

For high latency/overhead environments, it is possible to modify the JD algorithm to introduce coarser granularity. The basic assumption is that the memory capacity of each compute node is large enough to store A in its entirety. This is a reasonable assumption because the matrices are large (of dimension 10^5 to 10^6) but sparse, or they are not stored explicitly but represented by a function that applies the matrix-vector multiply. Under this assumption, we have developed a hybrid coarse-grain implementation, which we call JDcg [34]. The 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 our previous work [34], we have demonstrated the effectiveness of JDcg in hiding the communication latencies of slow networks.

4. 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. Also, some vectors ϵ_i may be computed to lower accuracy, but 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 [12]. Using classical convergence bounds for Conjugate Gradient [30], 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 phase proceeds as follows:

Load-balanced correction phase of JDcg

1. In the first JDcg iteration, do no load balancing. Each processor performs $maxits$ BCGSTAB iterations, calculates the rate at which it performed them, and communicates 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. In the all-to-all communication of step 8 of JDcg, faster processors gather the extrememost eigenpairs and residuals ensuring numerical progress.

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

In previous work [24], we obtained good results with this scheme in the presence of external loads introduced by sequential jobs. Here, we show that our load balanced JDcg adapts well even against parallel jobs, and especially jobs that are themselves dynamically load balanced. In particular, we present experiments in which JDcg jobs compete with each other for processors. The experiments are run on nine Sun Ultra-5 Model 333 machines with 256 MB of RAM, connected via switched Fast Ethernet. Three 4-processor JDcg jobs execute simultaneously, seeking the lowest eigenvalue of NASASRB[1], with $m = 150$ and ILUT(20,0) used as the preconditioner. The table below shows the overlap of the jobs on the nodes:

Job#	Node ids used by each job								
1	1	2	3	4					
2			3	4	5	6			
3				4			7	8	9

To quantify the load imbalance during execution, we have instrumented all MPI communication functions used by JDcg, to time-stamp their beginning and end. Because these functions are synchronous, we can calculate the imbalance on a node as the time each communication took to complete minus the minimum time the same one took to complete on any of the processors. By 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.

Table 1 summarizes the experimental results. Note that we expect node 3 to run its two jobs twice as slowly as node 1, and node 4 to run its three jobs 3 times as slowly. Considering that the non-load balanced code is limited by the speed of the slowest node, and that running the code on four otherwise unencumbered nodes takes 1530 seconds, the timings for the non-load balanced code in table 1 are in line with our expectations. The observed load imbalance is around 50%, close to the theoretically expected values. The load balanced code fares much better, displaying dramatically lower runtimes and load imbalance of around only 5 or 6%. Due to the non-work conserving nature of the JD algorithm, the load balanced code decreases the number of matvecs, actually speeding even algorithmic convergence.

[1]available from URL: `http://math.nist.gov/MatrixMarket`

Without load balancing

Job id	Time	Mvecs	%Imbal
1	4515	9571	50.2
2	4448	9571	49.8
3	4292	9571	49.2

With load balancing

Job id	Time	Mvecs	%Imbal
1	1848	7335	4.6
2	1896	7625	5.8
3	1852	8843	6.4

Table 1: Performance when running three 4 processor JDcg jobs on nine compute nodes, with some competition for nodes. "Time" denotes wall-clock time in seconds, "Mvecs" the total number of matrix-vector products computed, and "%Imbal" the percentage of time wasted by all processors of a job due to load imbalance.

In addition to CPU balancing, the correction phase also presents an opportunity for balancing memory load. When the memory requirements of JDcg and any external jobs far exceed the available memory, a significant number of CPU cycles are wasted as the system swaps pages to/from disk. Receding JDcg during the correction phase allows the competing job to use 100% of the CPU and memory resources, hopefully speeding its completion and hence relinquishment of resources. We have demonstrated the viability of this approach in previous work [24].

5. A GENERAL, USER-LEVEL, LOAD BALANCING LIBRARY

To facilitate general use of our load balancing strategy, we have written an object-based C library, LBLIB, that hides much of the required bookkeeping from the application programmer. The library is simple to use and provides support for both CPU and memory balancing.

To simplify data management and provide information hiding, data required for resource balancing are stored in a variable of the defined type `LBS`. An application programmer can access the data within an `LBS` structure through LBLIB functions. Data encapsulation is ensured by appropriate use of `void *` pointers in the internal implementation. A unique `LBS` structure is explicitly associated with a group of processors, and implicitly with a particular flexible section that these processors must load balance. A description of the basic functionality of the CPU balancing support follows:

`LBS lb_new_lbstruct(MPI_Comm communicator)`
This is the `LBS` constructor function, where `communicator` is the MPI communicator with which the `LBS` is to be associated. Note that a node of an MPI job could participate in the load balancing of nested, but different, flexible sections.

`void lb_section_start(LBS lbs)`
`double lb_section_end(LBS lbs, double ops_completed)`
These functions designate the beginning and end of the flexible section associated with the `lbs` structure, that is to be load balanced. `lb_section_start()` begins timing the execution of the section and, if memory balancing is used, it also begins tracking the amount of page swapping and CPU idling. Once all operations within the flexible section are completed, `lb_section_end()` stops the timing and calculates the rate at which the local processor performed operations since `lb_section_start()`. This rate is returned by the function for convenience, though it is also stored in the `lbs`. `ops_completed` is the number of operations that were completed during the flexible section. It is up to the application programmer to decide on a suitable way of counting the number of operations. In our eigensolver application, we use the number of iterations performed by the linear system solver during the correction phase.

```
double lb_decide(LBS lbs, double ops, int method,
                 [function])
```
Before entering a section we must determine the time T that all processors will spend in it. This is accomplished by calling `lb_decide()`, where `ops` is the number of operations that we ideally want each processor to complete. `lb_decide()` is the only synchronous LBLIB call that causes all processors in the communicator associated with `lbs` to gather the computation rates observed by each processor during the most recent execution of the flexible section. The rates are then sorted, and are used to determine T using the method specified by the `method` argument. In the current version of LBLIB, the possible values of `method` and the corresponding procedure for determining T are as follows:

- `LB_USE_FASTEST`: T is the predicted time for the fastest processor to compute `ops` operations.
- `LB_USE_SLOWEST`: T is the predicted time for the slowest processor to compute `ops` operations.
- `LB_USER_DEFINED`: A pointer to a user-defined function for calculating T is passed as a fourth argument. `lb_decide()` calls this function, passing `ops` and the array of sorted rates to it.

`lb_decide()` returns the time T, but it also stores it in `lbs` so that the application programmer does not need to know it. Note that before the first call to `lb_decide()`, one execution of the flexible section should have already been timed through calls to `lb_section_start()` and `lb_section_end()`. Otherwise, no computation rates are available for `lb_decide()` to determine the time T.

```
int lb_continue(LBS lbs, double ops_completed,
                double ops_needed)
```
This function allows the programmer to check whether the allotted time T is about to be exceeded, in which case the program must exit the flexible section. `ops_completed` is the number of operations completed so far during the current flexible section and `ops_needed` is the number of operations that will be completed before another call to `lb_continue()` can be made. `lb_continue()` calculates the execution rate of the local processor during the current flexible section, and then uses this rate to predict the time required to do `ops_needed` more operations. If this time falls within the allotted time, then `lb_continue()` returns 1, indicating that execution of the flexible section should continue. Otherwise it returns 0, indicating that we should exit the flexible section to avoid load imbalance. In our eigensolver application, the value of `ops_needed` is one, because we

call `lb_continue()` after each iteration of the linear system solver.

`double *lb_get_rates(LBS lbs)`
`int *lb_get_index(LBS lbs)`
It is often necessary for the application programmer to know the ordered rates of the processors, so that the critical tasks can be assigned to the most appropriate processors. `lb_get_rates()` returns a pointer to an array containing the rates of the processors, in ascending order. The processor indices corresponding to each of these rates can be retrieved through a pointer to an array returned by `lb_get_index()`.

5.1 Using the library

The above functions simplify significantly the load balancing of codes that abide with the general model of section 2. We present the following pseudocode that demonstrates the use of LBLIB to balance CPU load in our coarse-grain JDcg eigensolver.

```
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
```

Note that, the first time through the flexible section, the for-loop should execute a constant number of iterations in order to collect the first set of performance data. The semantics of the flexible section and its associated data dictate certain rules when calling some functions of the LBLIB library. Following the `lb_decide()` semantics, `lb_continue()` must be placed between `lb_section_start()` and `lb_section_end()`. Also, `lb_get_rates()` must be called only after a call to `lb_decide()` has been made, so that the rates have been exchanged between processors. Finally, between a call to `lb_section_start(lbs1)` and a call to `lb_section_end(lbs1)`, we cannot issue another call to `lb_section_start(lbs1)`. However, a call to `lb_section_start(lbs2)` is permissible for a different `LBS` structure.

In order to enforce such rules, an `LBS` keeps track of two fields: `scope` and `context`. `scope` keeps track of whether program execution is currently within the flexible section, while `context` tracks the availability of data for functions such as `lb_decide()`. In case of inappropriate scope or context, LBLIB functions set an error code in the `LBS`, which the programmer can check through **void lb_chkerr(LBS lbs)**.

We should mention that this paper describes only a subset of the functionality available in LBLIB. For example, in the functions we have presented, the processor rates are always computed as average rates over the whole flexible section. In systems with widely varying external loads, it may be better to compute "instantaneous" rates over a user-specified interval. The library provides a mechanism for such an update of rates, and all LBLIB functions that utilize rate information have an "instantaneous" counterpart. Finally, there is a set of functions that implement an anti-thrashing memory balancing scheme.

6. JDmg: A MULTIGRAIN EXTENSION TO 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. However, there is a more subtle reason for why JDcg is not suitable for these environments. Even if the memory is available, or the matrix is computed on the fly, 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 of 64 processors each, and during the JD correction phase, each 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 on a subset of the processors. An all-to-all similar to the JDcg case 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. This is typically the case with MPPs or COWs with large numbers of homogeneous processors.

Multigrain algorithm for MPP's

Node homogeneity allows for an easy manipulation of the coarse grain groups. Each such group, called a solve group, has P/k processors. For presentation simplicity and because $k \ll P$, we assume k divides P. To avoid a global all-to-all exchange between all processors, we can consider the fol-

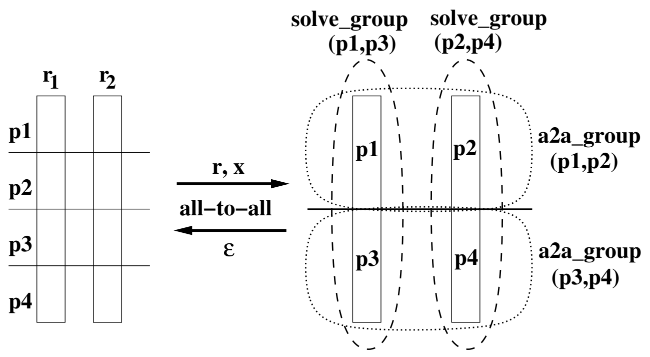

Figure 1: Example of MPP multigrain, with more processors than block vectors. Before the correction phase, nodes in the same all-to-all group receive the coarse-grain portions of the residuals and Ritz vectors they are responsible for. Each solve group then solves its respective correction equation. After this phase, each node distributes its coarse-grain portion of ϵ_i amongst its fellow all-to-all members.

lowing hierarchical partitioning. First, we obtain the coarse grain partitioning of the matrix onto P/k processors using partitioning software [27, 21]. Then, 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 correction phase, each member of an all-to-all group sends its fine-grain portions of the k vectors $\mathbf{r}_i$ (and $\mathbf{x}_i$) to the P/k members of its subgroup, and receives the P/k pieces that compose its coarse-grain portion of $\mathbf{r}_i$ ($\mathbf{x}_i$). Figure 1 illustrates this for $P = 4$ and $k = 2$. After each solve group finishes its correction phase, the all-to-all is reversed and each processor's coarse-grain portion of ϵ_i is distributed across all the processors in the all-to-all group.

Multigrain algorithm for Grids of clusters

In clusters of heterogeneous processors or simply clusters with different processor numbers, the solve groups 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 obviates the use of all-to-all groups, as one processor may be involved in total exchange with all other processors in the cluster. Therefore, the all-to-all communications must involve all P processors.

Besides the more expensive all-to-all communication, additional permutations and bookkeeping are required for the multigrain correction phase. We have observed that the time spent in all-to-alls by the cluster version is usually one order of magnitude or more greater than that taken by the MPP method. Still, our experiments show that JDmg is usually beneficial when the cluster sizes are roughly the same.

7. 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 multi-grain 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, like the coarse grain version, JDmg also fits the flexible-phase iteration model and can be load balanced in the same manner. The only real 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. At the beginning and end of each correction phase, the 0th processors call `lb_section_start()` and `lb_section_end()`, respectively. Before the all-to-all that begins each correction phase, the 0th processors call `lb_decide()`. This function returns the execution rates of each independent solve group. Each processor 0 then uses these rates to determine which block its subgroup should receive, and broadcasts this information to the other members. During the correction phase, the 0th processors call `lb_continue()` independently after each BCGSTAB iteration, and then broadcast to their members whether to perform another iteration or to halt.

7.1 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 [8].

To enable measurement of load imbalance in the multi-grain experiments, we timestamp synchronous communication calls in JDmg, much as we did with JDcg. 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 multiplications 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

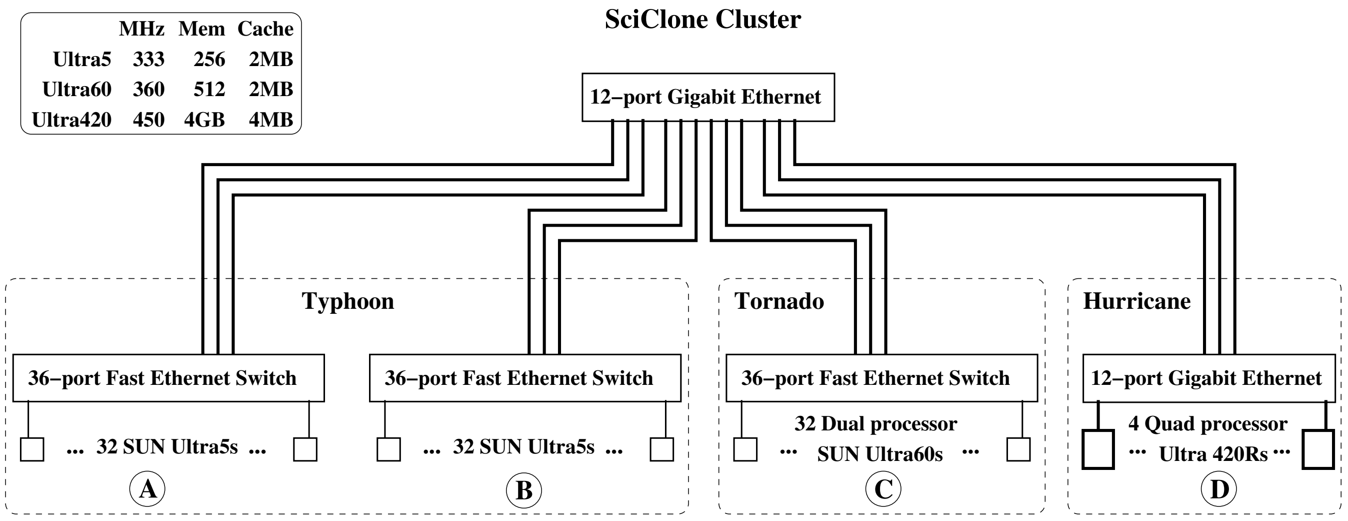

Figure 2: SciClone: The William and Mary heterogeneous cluster of three homogeneous clusters: Typhoon, Tornado (also called C), and Hurricane (also called D). We distinguish between A and B, the subclusters of Typhoon, because their intercommunication passes through the Gigabit switch. There are three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

only a small part of the execution time.

We have tested the fine-grain implementation on several node combinations from various clusters (Table 2). 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 B, 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 improvement through multigrain. However, multigrain can improve performance if the size of the solve groups is increased. Experiments $(AB)C$ and $ABC_{32}C_{32}$ yield significantly better timings compared to fine-grain test ABC (Table 3) because multigrain is able to hide the latency introduced by the additional processors.

As expected, multigrain by itself may not result in improvements if the rates at which each subgroup can perform matrix-vector multiplications vary greatly. In fact, this can result in significant performance degradation due to load imbalance. For example, the multigrain code often performs significantly worse than the fine-grain code when subcluster D is used in conjunction with other subclusters. Only two of the multigrain experiments involving subcluster D (AD and $(AB)C_{32}C_{32}D$) resulted in improvements over their fine-grain counterparts. This trend is an example of multigrain's tendency to accentuate load imbalance: the smaller number of processors in subcluster D results in a smaller solve group, and thus greater load imbalance.

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

Nodes	Time	Mvecs
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 2: 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.

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.

	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 3: Performance of the multigrain JD running on different node configurations, with and without load balancing. "Nodes", "Time" and "Mvecs" are as in Table 2. "%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).

8. 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 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 Grid 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.

9. REFERENCES

[1] A. C. Arpaci-Dusseau and Remzi 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] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. In *The International Journal of High Performance Computing Applications*, volume 14, pages 189–204, Fall 2000.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing 2000*, November 2000.

[6] P. Chandra, Y.-H. Chu, A. Fisher, J. Gao, C. Kosak, T.S. Eugene Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable resource management for value-added network services. 15(1), 2001.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

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

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] P. Keleher, J. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *19th Intl. Conf. on Distributed Computing Systems*, June 1999.

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.