Algorithmic modifications to the Jacobi-Davidson parallel eigensolver to dynamically balance external CPU and memory load *

Richard Tran Mills † Andreas Stathopoulos † Evgenia Smirni †

February 13, 2001

Abstract

Clusters of workstations (COWs) and SMPs have become popular and cost effective means of solving scientific problems. Because such environments may be heterogenous and/or time shared, dynamic load balancing is central to achieving high performance. Our thesis is that new levels of sophistication are required in parallel algorithm design and in the interaction of the algorithms with the runtime system. To support this thesis, we illustrate a novel approach for application-level balancing of external CPU and memory load on parallel iterative methods that employ some form of local preconditioning on each node. There are two key ideas. First, because all nodes need not perform their portion of the preconditioning phase to the same accuracy, the code can achieve perfect load balance, dynamically adapting to external CPU load, if we stop the preconditioning phase on all processors after a fixed amount of time. Second, if the program detects memory thrashing on a node, it recedes its preconditioning phase from that node, hopefully speeding the completion of competing jobs and hence the relinquishing of their resources. We have implemented our load balancing approach in a state-of-the-art, coarse grain parallel Jacobi-Davidson eigensolver. Experimental results show that the new method adapts its algorithm based on runtime system information, without compromising the overall convergence behavior. We demonstrate the effectiveness of the new algorithm in a COW environment under (a) variable CPU load and (b) variable memory availability caused by competing applications.

1 Introduction

In the heart of many scientific and engineering applications lies the numerical solution of either a system of linear equations, $Ax = b$, or of an eigenvalue problem, $Ax = \lambda x$. Advances in computational modeling and computer technology allow scientists to tackle increasingly larger problems for which the matrices (A) are very large. Iterative methods are the only means of dealing with these problems, and therefore they constitute a critical module of scientific software libraries and problem solving environments. Typically, they are used with a preconditioning technique to improve convergence rate and robustness. Parallel computing is also recognized as a powerful way

*Work supported by the National Science Foundation through Grants No. EIA-9974992 and No. EIA-9977030

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

of improving execution time and solvable problem size of these applications. However, the performance of existing methods falls short of the capabilities of today's hardware platforms for scientific computing, especially clusters of workstations (COWs) and symmetric multi-processors (SMPs).

Traditional fine grain implementations of iterative methods, because of their highly synchronous nature, present a scalability hurdle especially in clusters with high overhead networks. A popular way of increasing granularity is to compute a more expensive but more accurate and parallelizeable preconditioner. Domain decomposition is a typical example, where the local domains on each processor are solved independently [26]. In [27], we have described an alternative approach that combines both coarse and fine grain in a parallel, 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 clusters of workstations and of SMPs have access to the whole matrix A. This is often the case, not only because of increasing memory sizes, but especially because many matrix-free applications need only a matrix-vector multiplication function.

Beyond issues that are internal to parallel iterative methods, scalability is often inhibited by the ubiquity of resource imbalances on heterogeneous and/or distributed shared environments, such as COWs and SMPs. Cost efficiency suggests not only the sharing of common resources but also space- and even time-sharing among multiple users. Iterative methods can become the bottleneck of bigger applications running on these platforms, because they cannot utilize effectively the dynamically changing resources. 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]. There is substantial, recent work on supporting infrastructure for such distributed systems [15, 30, 14, 16]. However, to utilize resources efficiently, the program itself should be able to adapt its algorithm while in execution, based on runtime system information.

In this paper, we present a methodology that allows a class of iterative algorithms to adapt on-the-fly to changing system conditions. Invariably, we shift the weight of convergence responsibility from the outer, highly synchronous iteration to the local preconditioning phase. Because local preconditioning operations need not be carried to the same accuracy on each processor, we can achieve perfect load balancing by stopping the preconditioning phase on all processors after a fixed amount of time has elapsed. This preconditioning flexibility can also be used to avoid memory thrashing caused by external, memory intensive jobs. A process that detects thrashing recedes its preconditioning phase from that node, hopefully speeding the completion of competing jobs and hence the relinquishing of their resources. Typically, there is a trade-off between the parallelism of the preconditioner and its ability to reduce the number of iterations. Through the appropriate algorithmic modifications, our CPU and memory balancing strategies do not compromise and even improve the overall convergence of the method.

A key issue is the application-centric view of system information for load and resource balancing. Traditionally, system information has been used at the scheduler level, or at best at the outset of a program for original problem partitioning and allocation of resources [1, 31]. With recent performance monitoring and prediction libraries such as Network Weather Service [30], AppLeS [2], and PAPI [3, 4], this information can be used dynamically to set appropriately the variable granularity of our algorithms. In this paper, we demonstrate the effectiveness and robustness of our approach even with simpler tools, on both a SUN and a Pentium-based COW. Our experiments indicate that the new code efficiently balances the load and provides better throughput when it competes for resources with both CPU- and memory-intensive jobs.

This paper is organized as follows. After a summary of related work in section 2, section 3 outlines the coarse-grain implementation of the Jabobi-Davidson solver. Section 4 presents our load balancing balancing scheme, and describes a model for evaluating its performance. Experimental results on the load balancing scheme follow in section 5. Section 6 presents the anti-thrashing scheme, with experimental results following in section 7. Finally, section 8 provides concluding remarks, and outlines future work.

2 Load balancing in heterogeneous and shared resource environments

A significant amount of research has been invested in methods and software for load balancing scientific computing applications from within the application. Usually, the objectives are multiple (for example, split the problem into equal subproblems and minimize communication) and always they are resource and problem dependent. Yet, we can distinguish two main approaches.

The first approach assumes that the work per data item is known, and thus tries to equipartition the data onto processors. This can be done statically with packages such as METIS or Chaco [18, 17], during execution with repartitioning packages such as ParMETIS and Zoltan [20, 8], or by diffusing data into less loaded processors [19]. The second approach assumes the amount of work per data is variable and possibly unknown. In this case, the master-worker, or pool of tasks paradigm provides good load balancing [13]. However, many applications do not fit naturally this paradigm. Traditionally, these approaches, and especially the former one, have dealt only with load imbalances internal to the application.

Computational Grids consisting of heterogeneous networks are rapidly becoming important computing platforms for resource intensive parallel applications [15]. Besides CPU cycles, resources include memory, network, storage, and even data. On COWs and networks of SMPs effective resource sharing among competing applications becomes critical for high performance. Schedulers have been quite successful managing these resources under sequential workloads, but they still cannot effectively cater for parallel programs with dynamically varying resource requirements [9, 10, 11]. Therefore, application level scheduling of resources is important [2]. However, little is known in this direction, since current research has focused on providing interfaces to system information and not on how to use it to dynamically change the algorithm for better resource utilization.

Although in principle the above load balancing approaches could take into account system load information, there are two problems; first, repartitioning is often too expensive to be performed frequently, and second, system information must be obtained accurately and inexpensively. Recently, a variety of performance monitoring and prediction libraries such as Network Weather Service (NWS) [30], and PAPI [3, 4] have been developed. NWS provides measurements and forecasts of CPU and bandwidth on a heterogeneous network, while PAPI provides local processor information (memory, CPU, paging, etc.). There is a small number of related, new projects focusing on middleware that allows the application to specify alternate configurations, and the system to manage the shared resources accordingly [6, 7, 21]. AppLeS [2], relies on NWS to perform application level scheduling, but currently only as an initial partitioning of the problem.

However, the above projects do not suggest ways of dynamic algorithmic modifications. In addition, the few existing methods that use domain decomposition preconditioning as a means of load balancing, focus mainly on small imbalances caused by the partitioner [23]. In this paper, we present a different, unifying approach to numerical algorithms, parallelism, and system aware implementations. Using inexpensive memory and CPU measurements, we achieve good resource balancing, suggesting that a seamless interaction of the numerical application with the system is a viable and effective solution.

3 Coarse grain Jacobi-Davidson implementation

We have chosen a recent implementation of a coarse-grain parallel, block Jacobi-Davidson iterative eigensolver [27] as our driving application. However, any iterative method with independent, local preconditioners on each processor is also a good candidate for this research.

Consider the standard eigenvalue problem $Ax_i^* = \lambda_i^* x_i^*$, where A is a large matrix, and a few extreme (λ_i^*, x_i^*) eigenpairs are required. For simplicity of presentation we assume A is symmetric. Starting from an initial vector, the JD algorithm successively builds a basis for a space from where the approximations of the eigenpairs are obtained, usually, by the Rayleigh-Ritz procedure [12]. The JD expands the basis by adding the approximate solution ϵ_i of the correction equation (1) for some approximate eigenpair (Ritz pair) (λ_i, x_i), with residual $r_i = (A - \lambda_i I)x_i$.

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

A block version of the JD starts with a block of k initial vectors, and expands the basis by a block of k vectors at a time. These vectors are the approximate solutions of k correction equations, each for a different Ritz pair.

This algorithm is easily parallelizeable in a data parallel way. Each processor keeps a subset of the rows for each long vector, requiring a global reduction (summation) for each inner product. The user provides a parallel matrix vector multiplication and preconditioning operations. Below, we outline the data parallel JD algorithm:

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

end while

The easiest way to apply the $(I - x_i x_i^T)$ projections in eq. (1) is to use an iterative method for linear systems and perform the projections before and after the matrix vector multiplication. In our implementation we use BCGSTAB because of indefiniteness of the systems, and because of its short recurrence, which allows many steps without having to store a large number of vectors. Preconditioners can be applied directly to the BCGSTAB for solving the correction equation.

Consider the situation where every processor has access to the entire matrix A. With increasingly large memories available today, a 12-th order, 3-D finite difference matrix of 1 million unknowns can be stored on a PC with 512 MB. But more importantly, many applications do not store the matrix, providing instead a function for computing the matrix vector product. A typical example comes from the area of materials science [29], where the matrix vector product consists of two vector updates and two Fast Fourier Transforms. Another typical example is from the area of Kronecker-based Markov chains [5]. The matrix is represented as the product of few, small vectors that any processor can store.

Based on this observation we have developed a hybrid fine/coarse grain JD (JDcg) [27, 28]. With the matrix A available on all processors, the basis vectors V of the JD are still partitioned by rows. The main JD step (*projection phase*) is applied in the traditional fine grain fashion, oblivious to the matrix redundancy. The coarse grain parallelism is induced by having each node gather all the rows of one of the block vectors and apply the matrix vector product (and preconditioner) in BCGSTAB independently from other processors (*correction phase*). The following is an example of this coarse grain interface.

6. **Coarse grain correction equation**
   All-to-all: send local pieces of x_i, r_i to proc i, receive a piece for x_{myid}, r_{myid} from proc i
   Apply m steps of (preconditioned) BCGSTAB on eq. (1) with the gathered x_{myid}, r_{myid}
   All-to-all: send the i-th piece of ϵ_{myid} to proc i, receive a piece for ϵ_i from proc i

Despite the expensive all-to-all communication, the power of this interface lies in the independent solution of the correction equations. By increasing the BCGSTAB steps (m), more parallel work occurs between communications, and thus, we can improve arbitrarily the parallel speedup of the method. Typically, larger values of m and larger block sizes k reduce the number of outer JDcg iterations, but increase the total number of matrix-vector products, making the parallel algorithm non work-conserving. However, large values of m are required for the solution of numerically hard problems, and in addition this coarse granularity is ideal for hiding the latencies of slow networks.

4 Dynamic load balancing of JDcg

Traditional iterative methods are particularly susceptible to load imbalances because of the fine grain partitioning and frequent synchronizations. For numerically hard problems, the vast majority of the execution time of JDcg is spent in the correction phase. We can eliminate load imbalance during this phase by limiting more heavily loaded processors to fewer iterations of BCGSTAB. Imbalances persist during the brief projection phase, but the overall imbalances are virtually eliminated. Some correction vectors ϵ_i are then computed to lower accuracy but this only increases the number of outer JDcg iterations — in fact, it usually reduces the total amount of work. To ensure progress, during the all-to-all communication the fastest processor gathers the most critical vector (e.g., the extreme eigenvector), while the most loaded processor gathers the least critical one (innermost).

To obtain truly dynamic load balancing of the JDcg, we should cater for changes in the external load during the correction phase. We achieve this by iterating BCGSTAB for a fixed time T rather than for a predefined number of iterations m. For JD, it is often suggested that BCGSTAB is iterated to a convergence threshold of 2^{-iter}, where $iter$ is the number of outer iterations [12]. In our method, T should correspond to the time needed by the fastest processor to perform the required m iterations or meet the required convergence threshold for the most critical vector. To estimate T dynamically, we need first the following definitions:

- m is that "optimal number" of iterations that the *fastest* processor should compute.
- R_p is the rate at which processor p performed iterations during the previous correction phase.
- max is the maximum number of iterations that a processor should compute.
- T_m is the predicted time for the *fastest* processor to perform m iterations.
- T_{max} is the predicted time for the *fastest* processor to perform the max iterations.
- $iter$ is the number of outer (Jacobi-Davidson) iterations computed so far.

To set T, we need a rough estimate of how soon BCGSTAB will meet the required residual threshold. Using the classical convergence bound for Conjugate Gradient [25] we correlate the optimal number of iterations m with the 2^{-iter} threshold:

$$m = \frac{-iter}{\log_2 \rho}, \tag{2}$$

where $\rho = (\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$, and κ the condition number of the matrix can be estimated from the approximations of λ_{max} and λ_{min} available in JDcg. The algorithm proceeds as follows:

Load balancing of the correction phase

1. The first time through the correction phase, do no load balancing. Each processor computes the number of iterations per unit time R_p, and communicates its rate with all other processors.

2. For subsequent iterations, use the R_p measured in the previous iteration to rank the processors from fastest to slowest. A modified all-to-all communication then ensures that the slower processors gather the residuals associated with the more interior eigenpairs.

3. Determine the optimal number of iterations m using (2) and broadcast it to all processors.
   If $m < max$, then each processor iterates for time T_m
   If $m \geq max$, then each processor iterates for time T_{max}

By design, the above algorithm achieves perfect load balance during the correction phase, because all processors spend an equal amount of time in it. In addition, it ensures that reducing the load on some processors does not impede the convergence of the original algorithm, and often, as shown in our experiments, it improves it.

4.1 A model for quantifying load imbalance

To evaluate our load balancing scheme, we need to quantify the total load imbalance. This can be interpreted as the total number of wasted CPU cycles over all processors. For the JDcg process i running on processor i, the total wall-clock time is:

$$T_i = u_i + c_i + b_i, \quad \text{where} \tag{3}$$

- u_i is the total amount of CPU time (user time) spent on the JDcg process i,
- c_i is the total amount of time spent by process i in communication,
- b_i is the total amount of time wasted by process i due to load imbalance.

We assume any other system time to be negligible. The user time u_i is measured simply by looking at kernel counters (in most UNIX systems under the `/proc` pseudo-filesystem). The total load imbalance over all processors is then $B = \sum_{i=1}^{p} b_i$. Summing both sides of eq. (3) we obtain:

$$B = \sum_{i=1}^{p} (T_i - u_i) - \sum_{i=1}^{p} c_i. \tag{4}$$

It is thus sufficient to obtain a method for measuring the total communication time $C = \sum_{i=1}^{p} c_i$.

For measuring C, we run an experiment on a non loaded system. The total wall clock time W_i, the user time w_i, and the communication time g_i are now different for each process, and it holds:

$$W_i = w_i + g_i.$$

In this case, the total time spent in communication is measurable: $G = \sum_{i=1}^{p} g_i = \sum_{i=1}^{p} (W_i - w_i)$.

Note that the individual communication operations take the same time as with the loaded system (assuming the external load is not network intensive). Thus, although the number of JDcg iterations changes (and thus the g_i's), the communication time per iteration does *not* depend on load. Therefore, we have the equality:

$$\frac{G}{iter_{nonloaded}} = \frac{C}{iter_{loaded}}.$$

By substituting into eq. (4), we can compute the total load imbalance:

$$B = \sum_{i=1}^{p} (T_i - u_i) - \frac{iter_{loaded}}{iter_{nonloaded}} G. \tag{5}$$

5 Experimental evaluation

To evaluate the performance of our load balancing scheme, we conducted a series of experiments in which we loaded processors with additional "dummy" jobs that that perform CPU intensive computations. The experiments were run on four Sun Ultra-5 Model 333 machines with 256 MB of

	Without load balancing				With load balancing			
Load	Its	Mvecs	Time	% Imbal	Its	Mvecs	Time	% Imbal
1-1-1-1	19	9813	1745	4.3	18	9205	1666	3.8
2-1-1-1	19	9813	3509	38.5	19	8517	1783	5.6
2-2-1-1	19	9813	3526	26.4	19	7381	1788	4.4
2-2-2-1	19	9813	3599	15.5	21	6941	1997	3.6

Table 1: Performance when finding the smallest eigenpair of NASASRB on a system subject to various static external loads.

RAM, connected via switched fast Ethernet. Two large, sparse, symmetric test matrices available from MatrixMarket[1] were used. The first matrix is NASASRB, of dimension 54870 with 2677324 non zero elements, and we seek the lowest eigenvalue. Because the lowest eigenvalues of NASASRB are highly clustered, this is a difficult computation requiring a large number of iterations in the correction phase (we use $max = 150$). The second matrix is S3DKQ4M2, of dimension 90449 with 2455670 non zero elements, and we seek the largest eigenvalue. This is an easier problem requiring only a small number of iterations in the correction phase (we use $max = 20$). For both cases, the BCGSTAB is preconditioned with a local ILUT(0,20) preconditioner [24].

To validate our model for quantifying load imbalance, we also implemented a profiling instrumentation of the JDcg code, by timestamping the beginning and end of each communication operation. By synchronizing the initial timestamps, this profiling allows us to determine precisely where JDcg is spending its time and thus to quantify the amount of load imbalance incurred. Our experiments have shown excellent agreement with our model. In addition, we have used these timestamps in Matlab, to create a visual profile of the execution, with zooming capabilities.

5.1 Experiments with static external load

In the first set of experiments, we ran JDcg on a system subjected to static external machine loads. Processors were loaded with additional dummy jobs that execute register-based computations in an infinite loop. Tables 1 and 2 summarize the performance of JDcg on NASASRB and S3DKQ4M2, respectively. In these tables, the column named "Load" indicates the number of jobs (including the JDcg) on each processor. For example, "2-2-1-1" indicates that two processors are running an additional job. "Its" denotes the number of outer JDcg iterations, "Mvecs" the total amount of matrix-vector multiplications, "Time" the total wall clock time in seconds, and "% Imbal" the percentage of the time wasted in load imbalance over the total time for the application (i.e., $B/\sum T_i$ from eq. (5)).

The results for both NASASRB and S3DKQ4M2 show that the load balancing scheme dramatically reduces the overall load imbalance and cuts the execution time in half, even when the system is heavily loaded as in the 2-2-2-1 case. Notice that in the load balanced cases, the experiments with NASASRB display somewhat less load imbalance than the experiments with S3DKQ4M2. This is because S3DKQ4M2 requires fewer BiCGSTAB iterations in the correction phase, and hence spends a greater proportion of time in the projection phase, which is not load balanced. Note also

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

	Without load balancing				With load balancing			
Load	Its	Mvecs	Time	% Imbal	Its	Mvecs	Time	% Imbal
1-1-1-1	24	1959	601	4.0	25	1921	564	5.5
2-1-1-1	24	1959	1096	38.0	25	1720	601	8.3
2-2-1-1	24	1959	1120	27.4	28	1686	712	10.4
2-2-2-1	24	1959	1121	15.5	35	1706	892	10.4

Table 2: Table 2: Performance when finding the largest eigenpair of S3DKQ4M2 on a system subject to various static external loads.

that going from the 2-1-1-1 to the 2-2-1-1 to the 2-2-2-1 cases, the percentage of load imbalance in the non load balanced code decreases: fewer free processors wait for the loaded ones, and thus fewer CPU cycles are wasted. With load balancing, the imbalance stays relatively constant.

Notice that the amount of work performed (in terms of matrix-vector multiplications) decreases when load balancing is used on an externally loaded system. This is because block methods tend to increase the total number of work, especially if the correction equation for the innermost eigenpairs is solved more accurately. By assigning the innermost eigenpairs to the more loaded processors, our algorithm becomes more work conserving, resulting in faster convergence.

5.2 Experiments with dynamic external load

In a time-shared environment, one is likely to encounter load imbalances that are dynamic in nature. In our second set of experiments, we run JDcg on a system subjected to come-and-go external loads. Each processor is loaded with an additional dummy job which executes an endless loop that sleeps for a random amount of time and then performs register based computations for a random amount of time. The duration of the sleep and computation phases are sampled from exponential distributions with means λ and μ seconds, respectively.

Tables 3 and 4 summarize our results for matrices NASASRB and S3DKQ4M2 respectively. Since the behavior of the dummy jobs varies between experiments, we run ten trials for each set of parameters, and we report the averages of the timings and the observed standard deviation. The load balancing scheme works also well in the presence of a dynamic load imbalance. Performance of the the scheme generally worsens as λ and μ decrease. This is not surprising, because as the average duration of the computation done by the dummy jobs decreases, the ability to forecast the speed of a node based on its performance during the previous correction phase is lessened. However, because our scheme forces each processor to spend the same amount of time T in the correction phase, this lessened ability does not result in poor load balancing, but only worse overall convergence because of poorer estimation of an optimal value of T.

6 Avoiding memory thrashing from within the algorithm

The above load balancing technique works well when the external load is a CPU intensive job. However, in many cases jobs contend for another limited resource; memory. If the memory requirements of two applications far exceed the memory available on a processor, a significant amount of CPU

		Without load balancing		With load balancing			
λ	μ	Time	% Imbal	Its	Mvecs	Time	% Imbal
400	400	3135 (141)	23.6 (3.1)	21.0 (1.3)	8586 (486)	2094 (192)	5.4 (1.0)
300	300	3256 (165)	25.9 (2.3)	21.8 (1.1)	8800 (568)	2161 (160)	5.2 (0.5)
200	200	3094 (107)	25.4 (1.7)	21.5 (1.4)	8873 (531)	2122 (189)	5.5 (0.8)
200	100	2778 (115)	23.5 (1.5)	20.9 (1.1)	9250 (569)	2010 (124)	4.8 (0.4)
100	200	3356 (113)	21.1 (1.4)	22.7 (0.7)	9019 (328)	2539 (191)	6.4 (0.9)
100	100	2977 (185)	21.6 (1.8)	22.6 (0.9)	9448 (486)	2313 (86)	5.5 (0.6)

Table 3: Performance averages and their standard deviations (in parenthesis) for 10 different runs on NASASRB with come-and-go dummy jobs on each node. Dummy jobs execute a loop in which they sleep for γ seconds and then perform computations for ξ seconds. γ's and ξ's are sampled from exponential probability distributions with averages λ and μ, respectively. In cases without load balancing, 19 outer iterations and 9813 matrix-vector multiplications are always performed.

		Without load balancing		With load balancing			
λ	μ	Time	% Imbal	Its	Mvecs	Time	% Imbal
200	200	1074 (38)	25.2 (3.0)	28.6 (1.3)	1731 (37)	732 (46)	10.3 (0.9)
100	100	1028 (58)	25.6 (3.4)	28.5 (2.8)	1735 (58)	733 (100)	9.9 (1.9)
100	50	866 (67)	24.0 (3.8)	27.8 (1.6)	1800 (86)	684 (42)	10.3 (1.4)
60	60	1018 (47)	25.2 (1.8)	29.3 (1.2)	1775 (59)	751 (43)	10.1 (1.3)
50	100	1089 (32)	21.0 (1.8)	30.2 (3.0)	1792 (141)	857 (100)	10.3 (1.7)
30	30	974 (45)	23.6 (1.7)	30.4 (4.3)	1861 (196)	776 (132)	10.4 (2.3)

Table 4: Performance averages and their standard deviations (in parenthesis) for 10 different runs on S3DKQ4M2 with come-and-go dummy jobs on each node. Dummy jobs execute a loop in which they sleep for γ seconds and then perform computations for ξ seconds. γ's and ξ's are sampled from exponential probability distributions with averages λ and μ, respectively. In cases without load balancing, 24 outer iterations and 1959 matrix-vector multiplications are always performed.

cycles is wasted into paging. Moreover, if the applications have non local memory access patterns, the vast majority of CPU cycles is wasted in thrashing. Although load balancing smoothes the differences between processors, it cannot reclaim cycles lost to swapping.

6.1 A strategy to minimize thrashing

To improve the performance of the load balanced JDcg in the presence of external, memory-intensive jobs, we have implemented a heuristic based on the following simple idea. If thrashing occurs, JDcg recedes on that node during the correction phase, performing no BCGSTAB iterations for a fixed period of time. The goal is to allow the other application to use 100% of the CPU/memory, hoping it finishes earlier and relinquishes the resources.

For this idea to be effective, the lifetime of the competing job should be less than our application. One could use various performance predictive models and tools [2, 30] for obtaining such estimations. In the absence of such predictors, an upper limit is set on how long JDcg stays backed off, before it resumes. However, it is important to note that unlike sequential or synchronous parallel jobs, our load balanced JDcg does converge, even if this anti-thrashing strategy "starves" the correction phase on one processor, because other processors assume the convergence task.

On entering the correction phase, our anti-thrashing algorithm checks to see if thrashing was occurring during the just completed projection phase. If so, the algorithm sleeps for T_{wait} seconds. The choice of T_{wait} is discussed later. After this period, the algorithm checks whether it is appropriate for it to resume with the correction equation or to continue sleeping. This can be repeated until the time elapsed since the start of the correction phase exceeds the maximum allotted time T, and all nodes must return to the projection phase. Once/if the BCGSTAB is started, the algorithm checks at every iteration whether thrashing is occurring and whether it needs to recede. The basic steps of this algorithm are given in the following pseudocode description.

Anti-thrashing modification of the correction phase

```
    Reset the clock T_elapsed = 0
L1: If ( [Test for receding] ) then
        repeat sleep (T_wait)
        until ( [Test for resuming] )
    endif
    while (T > T_elapsed + estimated time to complete a BCGSTAB iteration)
        perform one BCGSTAB iteration
        if ( [Test for receding] ) then goto L1
    end
```

The efficiency of the algorithm depends on being able to answer reliably the two tests: (a) test for receding if currently running, (b) test for resuming execution if in sleeping state. The tests amount to accurately identifying system thrashing or predicting it before JDcg resumes execution. Although, our ultimate goal is to interface with performance measurement and prediction middleware, in this paper we show that much can be achieved through simple system tools.

For the first test, JDcg recedes if both the page swap-out rate exceeds some certain threshold (*swap_threshold*), and the CPU idling time (which includes idle and system time) is more than a tolerable threshold (*idle_tolerable*). Neither of the two conditions alone are adequate. If the CPU idles in the absence of paging, the culprit may be communication. If there is some paging, but little CPU idling, there is not much to improve by backing our process off. Other possibilities exist (for example testing page fault rates), but we found this test to be a very reliable indicator. Later in this section, we discuss how to choose appropriate thresholds. A short algorithm follows:

Test for receding
Obtain the CPU idling percentage over the just completed computation phase: *idle*
Obtain the swap-out rate for the just completed computation phase: *swapout*
If (*idle* > *idle_tolerable*) AND (*swapout* > *swap_threshold*)
    then return TRUE

Once JDcg is backed off, the test of whether to begin/resume BCGSTAB execution amounts to predicting whether swapping JDcg back in memory would cause thrashing. The answer is easy if the measured free memory on the node is enough for JDcg to run. However, low priority jobs often occupy large chunks of memory, which they would release if asked by a higher priority job. A typical example is the X server running on Linux PCs. For this reason, we also test if the system's idling time is high enough (more than *idle_sufficient*). The algorithm for the resuming test follows:

Test for resuming
If ($T_{elapsed} + T_{wait} > T$) then
    return TRUE and exit the correction phase
else
    Obtain the CPU idling percentage over the waiting period: *idle*
    Obtain the current available free memory: *free_mem*
    Obtain the current resident memory for JDcg: *JD_res_mem*
    Estimate the memory JDcg requires to avoid thrashing: *JD_req_mem*
    If (*idle* > *idle_sufficient*) OR (*JD_req_mem* − *JD_res_mem* < *free_mem*)
        then return TRUE
endif

6.2 Choosing the parameters

The sleep time T_{wait} must be short enough to be able to quickly respond to the system becoming available when the contending job(s) finish, and thus minimize load imbalance. However, too short a T_{wait} may induce an unnecessary CPU load. We choose the longest T_{wait} that yields a tolerable load imbalance, when compared with the execution time for the correction phase T.

The *swap_threshold* is more sensitive to the time interval that *swapout* is sampled from. Swapping out of pages occurs in bursts, the frequency of which depends on the memory access pattern of the jobs running. Ideally, the computation interval between the two measurements for *swapout* should be larger than the burst interval. Alternatively, we can estimate the interval between bursts t_b, and accumulate the *swapout* rate over multiple measurements during the past t_b seconds. Notice that the presence of even low paging out, signifies that there is memory contention, and thus a small *swap_threshold* is sufficient.

The parameter *idle_tolerable* should be the smallest rate of wasted cycles that our anti-thrashing algorithm can reclaim efficiently. For example, an idling of 8% is not considered thrashing, and our scheme may not succeed in improving it. However, a CPU idling percentage of 30% leaves more room for our scheme to improve. The choice of *idle_sufficient* should be small enough that allows lower priority jobs to be ignored, but also large enough that does not mistake as free a processor idling because of thrashing. In our experience, a value between 80%-90% is sufficient. The required memory *JD_req_mem* can be measured at runtime on a node that is not loaded.

Finally, the idling percentage (*idle*) and the (*swapout*) rate over a specified interval, as well as the current amounts of free and resident JDcg memory on the code can be easily obtained through the system counters available in the `/proc` pseudo-filesystem.

7 Memory intensive experiments

On the SUN platforms, we observed that the Solaris scheduler on each processor prevents thrashing by arbitrarily "starving" one of the jobs. This policy is also documented in [22]. Although this is expected to increase node throughput, it has not been designed with parallel jobs in mind. In our application, the scheduler starves not only the correction but also the projection phase, thus impeding the use of the other free processors.

For this reason, we have conducted our memory intensive experiments on a system of four 1 GHz Pentium III machines with 128 MB RAM running RedHat Linux 6.2. The machines are interconnected via Fast Ethernet. Our test matrix is NASASRB for which JDcg requires about 67 MB of memory to run on an unloaded system. We pick *JD_req_mem* just below that value, equal to 66 MB. For this test case, T_{wait} is chosen a hefty 20 seconds, and the *swap_threshold* is set to 150 KB/second. Both values are relatively conservative and may result in JDcg receding fewer times than optimal.

In all our experiments JDcg solves the NASASRB on four nodes, from which node 0 is loaded with a dummy job that alternates between sleeping and performing matrix vector multiplications. We have found that a dummy job of size 80 MB is sufficient to consistently cause excessive swapping. Furthermore, increasing this size does not increase the amount of swapping observed (since the memory access pattern of the dummy job for matrix-vector multiplication remains the same), so we use that value in our experiments. The dummy job can be described as follows:

Code for dummy job:

Wait 60 seconds to allow JDcg to initialize.
Create a vector $\mathbf{v}$ and a random, dense matrix A of size 80 MB
For $i = 1$ to *lim* do
    For $j = 1$ to *nmv* do
        $\mathbf{b} = A\mathbf{v}$
    End for
    Sleep for λ seconds.
End for

			Without anti-thrashing			With anti-thrashing		
λ	*nmv*	*lim*	Its	Mvecs	Time	Its	Mvecs	Time
n/a	46	1	19.4 (0.9)	8681 (714)	1125 (175)	19.6 (0.5)	9625 (287)	913 (27)
n/a	93	1	19.6 (0.5)	8202 (319)	1256 (92)	19.2 (0.8)	9317 (516)	928 (39)
30	3	4	19.6 (0.9)	9011 (654)	1004 (89)	19.0 (0.0)	8820 (108)	961 (46)
60	7	3	20.4 (0.9)	9088 (649)	1136 (98)	19.6 (0.9)	9088 (222)	943 (15)
60	15	2	20.6 (1.3)	9136 (812)	1157 (131)	19.6 (0.5)	9199 (416)	946 (49)

Table 5: Performance of the anti-thrashing scheme for NASASRB with come-and-go memory intensive dummy job on one node. The tests for receding/resuming in the anti-thrashing algorithm are performed through system measurements. The dummy job performs *lim* repetitions of *nmv* matrix vector multiplications with an 80MB matrix, sleeping for exactly λ seconds between each pass.

The *lim* passes through the outer loop can be viewed as the number of jobs to execute in a come-and-go fashion. The inner loop is the actual computational job that completes *nmv* matrix vector multiplications. The deterministic parameter λ specifies the amount of time in seconds to wait before repeating the inner loop. Unlike in the experiments described in previous sections, it is necessary to use a dummy job that does a specified amount of work and then stops, because the idea of the anti-thrashing scheme is to speed up JDcg by speeding up the completion of memory intensive jobs that compete with it. Thus, we choose values of *nmv* and *lim* so that the dummy job completes its work within the lifetime of the JDcg without the anti-thrashing scheme. Table 5 summarizes our results. We report averages over five runs and their standard deviations in parentheses.

Our experiments show that the anti-thrashing scheme gives an appreciable performance gain, often reducing total execution times by about 20% over the original load balanced code. To put these improvements into perspective, we should mention that the original load balanced JDcg often mimics the anti-thrashing scheme, by iterating only once in the correction phase, which makes the 20% improvements even more commendable. Obviously, the non load balanced code is simply not viable in such situations, with some of our experiments showing about 2 orders of magnitude performance degradation.

The performance of our scheme depends on its ability to decide when to recede or to resume. To evaluate how accurately our tests are answered and to obtain an upper bound on what this scheme can do, we test against an "ideal" version of our application. Rather than using system parameters to determine if the system is thrashing, we make the dummy job create a specific file when it starts computing and delete it after it finishes. Thus, the tests for receding and resuming execution in the JDcg reduce to simply checking if a file exists, providing for our test problem the most accurate system information. Our results are summarized in Table 6. Comparing these "ideal" results to those in Table 5, we see that the performance is essentially the same.

An important characteristic of the anti-thrashing scheme is that, besides improving the JDcg execution time, it also increases system throughput. The reduction in execution time for the dummy job can be impressive, as shown in Table 7.

One might be concerned with how our scheme performs when the competing memory-intensive application outlives JDcg. In this case, the scheme cannot reduce the runtime of JDcg, and could conceivably even degrade performance, because a node that is thrashing will never do any BCGSTAB

			With anti-thrashing		
λ	*nmv*	*lim*	Its	Mvecs	Time
n/a	46	1	20.2 (1.1)	9936 (651)	942 (71)
n/a	93	1	19.4 (0.5)	9233 (313)	920 (39)
30	3	4	19.5 (0.6)	9346 (323)	970 (48)
60	7	3	19.6 (0.9)	9414 (461)	917 (42)
60	15	2	20.0 (0.7)	9774 (440)	961 (45)

Table 6: Performance of the "ideal" anti-thrashing scheme for NASASRB with come-and-go memory intensive job on one node. The JDcg knows exactly when the system is thrashing by communicating directly with the dummy job.

			without anti-thrashing	with anti-thrashing	with ideal testing
λ	*nmv*	*lim*	Time	Time	Time
n/a	46	1	736 (409)	113 (17)	140 (15)
n/a	93	1	1055 (129)	187 (22)	202 (26)
30	3	4	554 (66)	223 (73)	171 (40)
60	7	3	808 (97)	152 (50)	141 (19)
60	15	2	864 (150)	158 (20)	141 (22)

Table 7: Improvements on execution time of the dummy job because of increased processor throughput due to our anti-thrashing scheme.

iterations. We performed a few experiments with dummy jobs that outlive JDcg, and did not notice any measurable performance degradation. In fact, the execution time with the anti-thrashing scheme was similar to the time obtained without the anti-thrashing scheme. This is because when the load balanced JDcg runs on a node that is thrashing, typically the node is so slow that it only completes one BCGSTAB iteration per correction phase. Reducing this number to zero with the anti-thrashing scheme does not have a noticeable effect on the JDcg convergence. However, this scheme significantly benefits the overall system throughput.

8 Conclusions and future directions

We have provided application-level CPU and memory load balancing techniques that utilize runtime system information to dynamically modify and tune the correction phase of the Jacobi-Davidson eigensolver. Our load balancing experiments show that our method significantly reduces execution time and minimizes load imbalance on time-shared COWs with external CPU load. The method is robust under a variety of static and dynamic workloads, without compromising numerical convergence. Also, the overhead for obtaining the system information is negligible. Our anti-thrashing scheme uses similar system information to accurately determine the thrashing status on a node, and recede its execution appropriately. In our experiments, this scheme consistently reduces the running time for the JDcg code by 10-20%, and it dramatically improves processor throughput. It has also demonstrated robustness regardless of the length or size of the external memory intensive jobs, suggesting wide applicability.

Using system information to dynamically modify and optimize the runtime execution of an algorithm is a relatively new approach, and many issues remain unanswered. The use of state-of-the-art performance measuring libraries (such as PAPI or NWS) may be useful in improving the accuracy of our system predictions, which would allow algorithmic modifications for longer scales. Other shared resources may also be of interest, such as network load. An obvious extension of the above techniques is to the FGMRES method for solving linear systems of equations. The issues involved are surprisingly similar to the JDcg. Finally, we would like to implement the resource balancing techniques to more traditional local preconditioners such as domain decomposition. Despite the computational similarity, maintaining the numerical convergence of the method is more involved.

Acknowledgement

We would like to thank James R. McCombs for helping in the earlier stages of this work and for providing the code and continuing support of the coarse grain implementation of the Jacobi-Davidson method.

References

[1] F. Berman and R. Wolski. Scheduling from the perspective of the application networks. In *5th IEEE Inlt. Symp. on High Performance Distributed Computing*, 1996.

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

[3] 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.

[4] 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.

[5] P. Buchholz, G. Ciardo, P. Kemper, and S. Donatelli. Complexity of memory-efficient kronecker operations with applications to the solution markov models. *INFORMS Journal on Computing*, 13(3):203–202, 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] 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.

[9] D. G. Feitelson and L. Rudolph, editors. *1998 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1457. LNCS, 1998.

[10] D. G. Feitelson and L. Rudolph, editors. *1999 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659. LNCS, 1999.

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

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

[18] 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.

[19] George Karypis and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47:109–124, 1997.

[20] 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.

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

[22] Jim Mauro and Richard McDougall. *SOLARIS Internals, Core Kernel Architecture.* Prentice Hall PTR, 2001.

[23] 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.

[24] Yousef Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994. Technical Report 92-38, Minnesota Supercomputer Institute, University of Minnesota, 1992.

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

[26] Barry Smith, P. Bjorstad, and W. Gropp. *Demain Decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996.

[27] 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.

[28] A. Stathopoulos and J. R. McCombs. Parallel, multi-grain eigensolvers with applications to materials science. In *First SIAM Conference on Computational Science and Engineering*, 2000.

[29] 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.

[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.

[31] R. Wolski, N. Spring, and J. Hayes. Predicting the cpu availability of time-shared unix systems. In *8th IEEE High Performance Distributed Computing Conference (HPDC8)*, number also UCSD TR: CS98-602, August 1999.