Theor Chim Acta (1993) 84: 457–473

Theoretica Chimica Acta

Investigating the performance of parallel eigensolvers for large processor counts

Richard J. Littlefield[1,*] **and Kristyn J. Maschhoff**[2]

[1] Pacific Northwest Laboratory, Richland, WA 99352, USA

[2] University of Washington, Seattle, WA, USA

Received October 1, 1991/Accepted May 12, 1992

Summary. Eigensolving (diagonalizing) small dense matrices threatens to become a bottleneck in the application of massively parallel computers to electronic structure methods. Because the computational cost of electronic structure methods typically scales as $O(N^3)$ or worse, even teraflop computer systems with thousands of processors will often confront problems with $N \ll 10{,}000$. At present, diagonalizing an $N \times N$ matrix on P processors is not efficient when P is large compared to N. The loss of efficiency can make diagonalization a bottleneck on a massively parallel computer, even though it is typically a minor operation on conventional serial machines. This situation motivates a search for both improved methods and identification of the computer characteristics that would be most productive to improve.

In this paper, we compare the performance of several parallel and serial methods for solving dense real symmetric eigensystems on a distributed memory message passing parallel computer. We focus on matrices of size $N = 200$ and processor counts $P = 1$ to $P = 512$, with execution on the Intel Touchstone DELTA computer. The best eigensolver method is found to depend on the number of available processors. Of the methods tested, a recently developed Blocked Factored Jacobi (BFJ) method is the slowest for small P, but the fastest for large P. Its speed is a complicated non-monotonic function of the number of processors used. A detailed performance analysis of the BFJ method shows that: (1) the factor most responsible for limited speedup is communication startup cost; (2) with current communication costs, the maximum achievable parallel speedup is modest (one order of magnitude) compared to the best serial method; and (3) the fastest solution is often achieved by using less than the maximum number of available processors.

Key words: Eigensolving – Massively parallel computers – Small dense matrices

1 Introduction

Many electronic structure methods require obtaining the eigenvalues and eigenvectors of a dense real symmetric $N \times N$ matrix, a process called *eigensolving* or

* Pacific Northwest Laboratory is operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under contract DE-AC06-76RLO 1830

diagonalization. N, the number of basis functions, is typically a few hundred in current applications.

On conventional computers, diagonalizing these dense matrices usually is not a bottleneck. On such machines, the time required for eigensolving is $O(N^3)$.[1] Other parts of the calculation, such as constructing the matrix, can vary from $O(N^2)$ to $O(N^4)$, depending on the chemical system and electronic structure method. However, the coefficient for those parts is large enough that eigensolving typically comprises a small fraction of the total computational cost.

Massively parallel computing threatens to change this situation. It is easy to see how to apply large numbers of processors to such work as constructing the matrix, which consists of many independent calculations. Indeed, it seems plausible that the time-to-completion for those tasks could be held virtually constant by simply increasing P in proportion to the amount of work. Unfortunately, it is not so easy to apply large numbers of processors to eigensolving small dense matrices. Most parallel eigensolver methods are limited to $P \leqslant N$ and, as shown later, can lose efficiency quickly even for much smaller P. Thus, if P is large compared to N, eigensolving can become a bottleneck for a computation done in parallel, even though it would not be for the same computation done serially.

This would be acceptable if $P \ll N$ were expected to be the normal situation, but this is not the case. Because total computational cost increases quickly, N will be limited even for teraflop machines (10^{12} floating point operations per second). At present, problems with N of a few hundred are often reserved for computers rated at perhaps 10^8 operations per second. Assuming $O(N^3)$ scaling, this suggests that teraflop machines will often confront problems with $N < 10{,}000$ (i.e., a factor of $10^{4/3}$ larger). Much smaller N may occur for some classes of problems. For example, hybrid molecular dynamics methods have been proposed in which quantum methods would be used only for small critical portions of the chemical system. Such methods might compute a long series of time steps, limiting N to a few hundred in order to make the calculation feasible even on a teraflop machine. There are supercomputer-class parallel systems with more than 500 processors today, and teraflop computers with more than 10,000 processors surely will be available in a few years. Thus, it seems likely that $P > N$ will be a common case with massively parallel computers.

This situation motivates a search for parallel eigensolvers that can exploit large numbers of processors to reduce time-to-completion. Although linear speedup (proportional to P) would be desirable, the eigensolving bottleneck sometimes can be avoided by more modest improvements. Suppose, for example, that one attempts to apply 500 processors to a problem that is 99.5% perfectly parallelizable work, plus 0.5% eigensolving. Then Amdahl's law:

$$Speedup = \frac{T_{serial} + T_{parallel}}{T_{serial} + \dfrac{T_{parallel}}{P}}$$

implies that the speedup will be only 143 if the eigensolving is done serially. However, if the eigensolving were parallelized so as to run 10 times faster than the

[1] We use "Big-O" order notation in the formal sense: a computation has cost $O(f(N))$ if and only if there exists some constant c and some minimum problem size N_0 such that for all $N > N_0$, $\text{cost}(N) \leqslant c*f(N)$. This notation gives some indication of how the cost varies with problem size, but says nothing about absolute cost. The coefficients may very well be such that an $O(N^2)$ algorithm is faster than some $O(N \log N)$ algorithm for all N of practical interest

serial version, then the overall speedup would increase to over 400. If more processors were available, correspondingly higher speedup would be required from the eigensolver to achieve the same gain.

In this paper, we explore some aspects of parallel eigensolvers in the regime where P is slightly larger than N. The algorithms and codes that we consider are all designed for a distributed memory MIMD (Multiple Instruction, Multiple Data) computer programmed with explicit message passing. Our goals are to determine how much the time-to-completion is reduced by parallel computation, what factors limit that reduction, and which method(s) perform best under various conditions. This study is not intended to be definitive, but rather to support our long-term goal of developing improved methods to avoid the eigensolver bottleneck.

The paper is organized as follows. Section 2 describes several approaches to parallel eigensolving, laying groundwork for understanding the behavior shown later. Section 3 discusses the results of empirical tests comparing the performance of five serial and four parallel methods, over a range of matrix types and processor counts. In these tests, a newly developed Blocked Factored Jacobi (BFJ) method was the fastest method for large P. In Sect. 4, the BFJ method is analyzed in detail using a theoretical performance model to determine what factors limit its performance. Conclusions and suggested directions for further work are found in Sect. 5. For completeness, the BFJ algorithm and performance model are detailed in the Appendix.

2 Parallel eigensolver methods

Many numerical methods for solving dense real symmetric eigensystems are in common use [5, 10], and most of them can be parallelized to some extent. We will outline only the methods and parallelization strategies considered in this paper.

Jacobi methods operate on the dense matrix using the Jacobi iteration:

$$A^{(k+1)} = J_k^T A^{(k)} J_k = V_k^T A^{(0)} V_k$$

where J_k is a plane rotation matrix chosen to annihilate one off-diagonal element of $A^{(k)}$. When performed in a series of sweeps addressing all $N(N-1)/2$ off-diagonal elements, this iteration converges with the eigenvalues appearing along the diagonal of A and the corresponding eigenvectors appearing as the columns of V.

Due to the special form of J_k, Jacobi methods can be distributed efficiently by factoring A^k into two matrices, one of which is grouped by columns and the other by rows:

$$A^{(k+1)} = V_k^T A^0 V_k = \underbrace{\begin{bmatrix} \cdots & r_1 & \cdots \\ \cdots & r_2 & \cdots \\ & \vdots & \\ \cdots & r_n & \cdots \end{bmatrix}}_{G^k \equiv V_k^T A^0} \underbrace{\begin{bmatrix} \vdots & \vdots & & \vdots \\ c_1 & c_2 & \cdots & c_n \\ \vdots & \vdots & & \vdots \end{bmatrix}}_{V_k}$$

Using this factored form, the rotation to annihilate a_{pq} works only with data contained in rows p and q of a matrix G^k (which is conceptually $V_k^T A^0$) and

columns p and q of V. To distribute the computation, disjoint groups of rows and columns are assigned to several processors. Each processor does all rotations for the data it owns, then the data are shuffled so as to bring new groups together, and so on, until all the rotations in a sweep have been completed. Essentially, a round-robin tournament is held, in which every unique (p, q) pair is formed exactly once per sweep. There are several simple and efficient shuffling schemes that generate all of the required pairs in the minimum number of rounds using only nearest neighbor communications on a ring topology [3, 4].

This basic algorithm, commonly called "one-sided Jacobi" [3], can be extended to use a blocked decomposition of the matrix across multiple rings, leading to the Blocked Factored Jacobi (BFJ) method outlined by Littlefield and Maschhoff [8] and detailed in the Appendix of this paper. Blocking is required to exploit $P > N/2$, but due to tradeoffs in load balance and communication costs, blocking often turns out to be superior even when P is substantially less than $N/2$ [8].

Compared to serial Jacobi methods, parallel Jacobi suffers mainly from communication costs and a reduced ability to exploit skipped rotations. In a serial Jacobi method, much work can be saved by skipping rotations for off-diagonal elements that are already near zero. This typically increases the number of iterations needed for convergence, but reduces the total computation cost. In parallel Jacobi methods, skipping rotations does not reduce the time-to-completion unless rotations can be skipped in all processors in the same step of the ring transfer. As the processor count increases, this becomes increasingly unlikely; in the limit of large P, the value of skipping rotations tends to zero.

Our tests included three Jacobi methods – two serial and one parallel. The two serial methods are quite similar except for their eagerness to skip rotations. The parallel method (BFJ) does not skip rotations at all, even with small P.

Most non-Jacobi methods start by reducing the dense matrix to tridiagonal form using a sequence of Householder transformations, each of which zeroes one column below the subdiagonal. This reduction does not parallelize perfectly. Although each transformation can be applied to the remainder of the matrix in parallel, the transformations must be determined in sequence and sent to all processors. With large P, the reduction step potentially suffers from load imbalance and high communication costs.

After tridiagonal form is obtained, there are several methods for extracting eigenvalues and eigenvectors. One method, used by the EISPACK RSP routine, is to use implicit-shift QL iteration to simultaneously find the eigenvalues and eigenvectors. A simple approach to parallelizing this method, used by the codes that we call PRS, is to duplicate the eigenvalue part of the computation on all processors, while simultaneously computing only a few of the components of each eigenvector on each processor. Because the eigenvalue part of the computation remains essentially serial, this approach results in only partial parallelization. This does not affect the computational complexity, since the eigenvalue computation is only $O(N^2)$, while the eigenvector computation is $O(N^3)$ serial but $O(N^2)$ parallel. However, it may increase the absolute time-to-completion.

Another general approach is to find all the eigenvalues first, then use those to compute the eigenvectors. In EISCUBE, eigenvalues are found by bisection using the Sturm sequence, while eigenvectors are determined by perfect-shift QL iteration; both steps are parallelized. Alternately, the eigenvalues may be found by implicit QL iteration (which does not parallelize), or the eigenvectors by inverse iteration (which does). The fastest serial solver that we tested (GIVEIS)

uses implicit QL followed by inverse iteration. We do not yet have a parallel method using inverse iteration. This is an obvious shortcoming of our current tests, since the combination of bisection and inverse iteration has been found to be the fastest method for solving symmetric tridiagonal matrices under some circumstances [6, 7].

All of the non-Jacobi parallel methods that we tested are limited to $P \leqslant N$, giving a parallel cost of $O(N^2)$. In contrast, the BFJ method can exploit $P > N$, and has an asymptotic parallel cost of $O(N \log^2 N)$ using $cN^2/\log N$ processors. (The optimum coefficient c depends on the ratio of computation and communication startup speeds.)

In the limit of very large N, and given as many processors as each method could exploit, these cost orders indicate that the BFJ method would be faster. However, Jacobi methods, including the BFJ method, are typically several times slower than the competition on serial machines. To overcome this initial penalty, the BFJ method would have to scale much better than the other parallel methods, and it was not obvious a priori whether this would occur for problems of practical size.

3 Empirical results

To investigate some of the issues raised in the preceding discussion, we benchmarked several eigensolvers, using a variety of matrix types and processor counts. Table 1 outlines the solvers that we tested, and Figs. 1 and 2 show timings.

All of the solvers were coded in Fortran, except that PRS v.0 and PRS v.1 were coded in C. All used either naive Fortran BLAS[2] or the equivalent inline

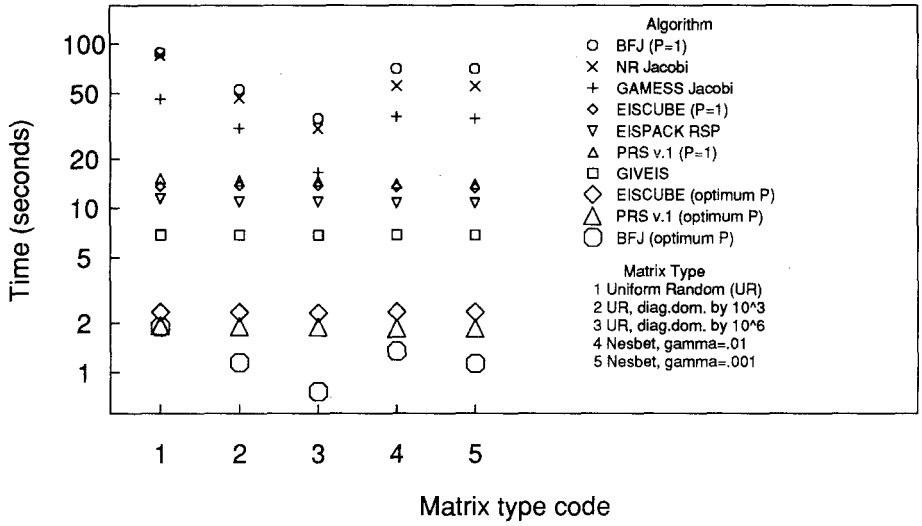

Fig. 1. Execution time for solving a 200 × 200 matrix on the Intel Touchstone DELTA computer, using either $P = 1$ or the P that produced the shortest time for a particular solver

[2] Basic Linear Algebra Subroutines, obtained from netlib@ornl.gov

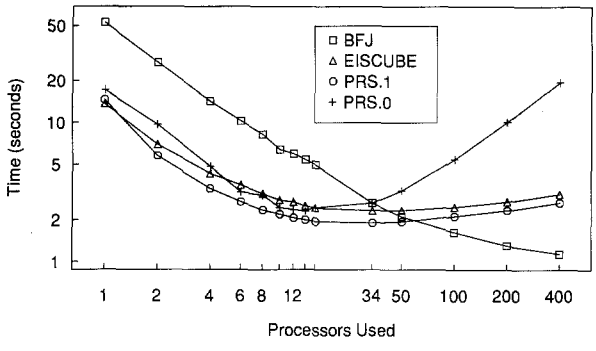

Fig. 2. Execution time for solving a 200×200 matrix, diagonally dominant by 10^3 ("Type 2"), as a function of the number of processors used

Table 1. Solvers tested in this study

Name	Parallel (*P*) or serial (*S*)	Source; Description
BFJ	*P*	Written at PNL; Blocked Factored Jacobi.
NR Jacobi	*S*	Numerical Recipes [10]; conventional Jacobi.
GAMESS Jacobi	*S*	Extracted from the GAMESS-UK program; conventional Jacobi with aggressive skipping of rotations.
EISCUBE	*P*	Supplied by Intel, modified for portability; reduction, bisection, perfect-shift QL, all parallelized.
RSP	*S*	EISPACK (netlib); reduction, implicit-shift QL for eigenvalues and vectors simultaneously.
PRS v.0	*P*	See acknowledgements; parallel reduction, serial implicit-shift QL for eigenvalues, with parallel construction of eigenvectors.
PRS v.1	*P*	See acknowledgements; same basic method as PRS v.0, see text for differences.
GIVEIS	*S*	Extracted from the TURBOMOLE program; reduction, implicit-shift QL for eigenvalues, inverse iteration for vectors.

code. All communications were done using synchronous message passing primitives (send/receive) provided by the operating system. Except for PRS v.0 (see below), all global communications were done using a binary tree strategy with cost $O(\log P)$. Convergence criteria were set so that all solvers produced similar accuracy. All of the parallel solvers terminate with their results distributed in some fashion, and the reported times do not include any reorganizing of the results. (Such reorganization would require only a small fraction of the eigensolving time. We omitted it to avoid dealing with application-specific details.) Because of these uniform conditions, we believe that the timing results are comparable between codes.

Two fundamentally different types of matrices were used for testing. The first type was constructed of uniform (0, 1) random numbers, then made diagonally dominant by dividing the off-diagonal elements by an appropriate constant (1, 10^3, 10^6). The second type was constructed of full-bandwidth Nesbet

matrices,[3] with the γ parameter set to either 0.01 or 0.001. Both methods generate matrices whose eigenvalues are almost always well separated.

Testing was done on the Intel Touchstone DELTA[4] computer [2]. The DELTA computer consists of 520 nodes, each containing an i860 processor chip and 16 megabytes of memory. The processors are interconnected with cut-through routing on a 2-D mesh. The DELTA differs from an Intel iPSC/860™ [1] primarily in having more processors and higher node-node bandwidth. (Absolute performance numbers are discussed in Sect. 4.)

PRS v.0 and PRS v.1 require more explanation than appears in Table 1. These codes are based on a partial parallelization of the EISPACK RSP method, as outlined in the previous section: parallel reduction, serial solution for eigenvalues, and parallel accumulation of vectors. There are two important differences between these codes. First, v.0 implements global communications using a simple one-to-many serial scheme, while v.1 uses a more sophisticated binary tree approach. Second, v.1 uses external BLAS, while v.0 uses inline code.

Results are shown in Figs. 1 and 2. All of the results shown were done with $N = 200$. This value is typical of current problems, and is small enough to allow P modestly greater than N. Parallel codes were tested with a variety of processor counts, $P = 1$ to $P = 400$.

Figure 1 summarizes all the performance test results. It displays times for the serial solvers, plus times for parallel solvers at $P = 1$ and at whatever "optimum" P produced the minimum execution time. Several interesting features are apparent:

- BFJ is the fastest parallel solver for most matrix types, but is always the slowest serial one.
- All of the parallel solvers are faster than any serial solver, but none of the parallel solvers is more than 10 times faster than the best serial one (GIVEIS).
- All of the Jacobi methods improve in proportion to the degree of diagonal dominance. In the remainder of this paper, we focus on "type 2" matrices (diagonally dominant by a factor of 10^3), as representative of what might be found in practice in highly iterative applications, where good eigenvector approximations are available to use for preconditioning the matrix.

Figure 2 shows the speedup curves for the parallel solvers. Again, several interesting features are apparent:

- PRS v.0 scales well out to about 12 processors, then begins to suffer from its $O(P)$ serial broadcast scheme. Beyond 14 processors, PRS v.0 gets dramatically slower.
- PRS v.1 and EISCUBE, which use $O(\log P)$ global operations, do not suffer much beyond 50 processors. However, they too achieve their minima at around $P = 50$, and get slightly slower after that.
- BFJ improves out to $P = 400$, crossing under the EISCUBE and PRS v.1 curves at slightly over $P = 50$. (Beyond $P = 400$, BFJ would turn up also; see Sect. 4.)

---

[3] $M_{ij} = 1 + \gamma(2i - 1)\delta(i, j)$

[4] Intel Supercomputer Systems Division, Intel Corporation, Beaverton, Oregon. The Touchstone DELTA computer is a result of specially directed efforts in support of the Concurrent Supercomputing Consortium, and is not marketed by Intel. ™ Intel Corporation

It has long been conjectured that Jacobi methods might be faster than other methods for sufficiently large processor counts [3]. The data reported here provide the first empirical support for that position, suggesting that the BFJ method is in fact a competitive algorithm when, for example, $P > N/4$ and N is relatively small.

This conclusion must be tempered, however, by the observation that we have not yet attempted to adapt the non-Jacobi methods for the large-P regime. In the case of Jacobi methods, detailed performance modeling led directly to the creation of the BFJ method, making it possible to exploit $P > N/2$ processors. The same effort also provided a method for optimizing BFJ's use of available processors, producing an average 30% performance improvement in some useful regimes [8]. We are hopeful that further study of the non-Jacobi methods will yield similar benefits, and it would not be surprising for the BFJ method to be overtaken by an improved non-Jacobi method. At present, both EISCUBE and PRS v.1 take slightly more time to reduce the matrix to tridiagonal form than BFJ does to completely solve the system. However, there are several potential improvements in the reduction that we have yet to evaluate.

Several other points are more clear:

- As shown in Fig. 1, the parallel speedup of small dense eigensolvers will be modest for matrices of this size, unless there are changes in the methods and/or computer systems. This point is investigated further in the next section.
- As shown by PRS v.0 and PRS v.1 in Fig. 2, performance with large P cannot be predicted solely from performance with small P. The nature of the underlying algorithm must be considered, and an appropriate performance model used.
- As shown by the behavior of the BFJ method, the large-P and small-P regimes should be considered separately – different algorithms may be preferred in each regime.

4 Performance analysis of the BFJ method

In the previous section, we showed empirically that the BFJ method is competitive with other methods in the large-P regime. We now use a detailed performance model of BFJ to address several questions:

1. What floating point and communication speeds is the BJF method actually getting out of the current computer?

2. What limits the performance of the BFJ method?

3. How fast could the BFJ method run, given an unlimited number of processors with some specified characteristics?

Note that the answers we get, strictly speaking, will apply only to the particular implementation of the BFJ method that we analyze. There is no guarantee that the bottlenecks for the BFJ method are the same as those for PRS v.1. Nonetheless, we hope that a detailed analysis of one eigensolver will yield some valuable insight about parallel computing, as well as providing a model for analyzing other methods.

The BFJ algorithm and a performance model for it are described in detail in the Appendix. Briefly, the BJF method distributes rows and columns of two

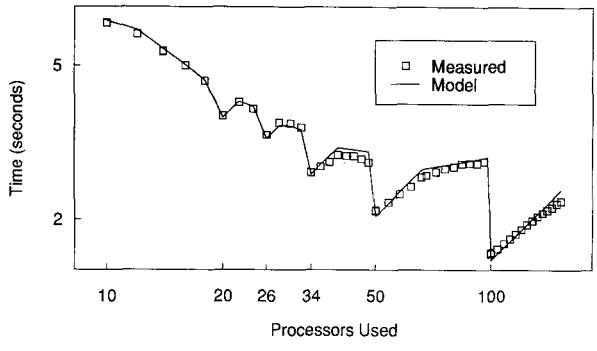

Fig. 3. Predicted and observed execution time for solving a 200×200 "Type 2" matrix using the BFJ method, as a function of the number of processors used

matrices across a ring of processors (multiple rings, in general), then alternately operates on local data and scrolls the data around the ring. This algorithm exhibits "stairstep" speedup (see Fig. 3) because its execution time is largely proportional to both the ring length and the maximum number of matrix rows and column assigned to any processor. Adding processors (increasing the ring length) actually makes the algorithm run slower, until enough processors have been added to reduce the amount of data in each one. Thus the BJF method should be run only with processor counts at the bottom of a step. (The processor counts in Fig. 1 were chosen by this rule to avoid stairstep artifacts.)

The performance of the BFJ method can be modeled well by assuming that computation and communication happen in lockstep, and then calculating the time consumed by the busiest processor at each step. This process yields a closed-form analytic expression for the execution time in terms of six parameters that are closely tied to basic machine speeds, application code design, and compiler quality:

1. $F_{overhead}$ is the fixed computational cost for a single annihilation, expressed as an equivalent number of floating point operations. This cost includes subroutine entry/exit, computing the rotation angle, and so on.

2. t_{flop} is the floating point operation time (as measured for the per-element operations of the BLAS functions).

3. $t_{xstartup}$ is the transfer startup time.

4. $t_{xperelem}$ is the transfer time per element.

5. $t_{cstartup}$ is the combine startup time (to "combine" means to sum values across several processors).

6. $t_{cperelem}$ is the combine time per element.

These parameters can be measured fairly easily with testjig programs, and their accuracy can be checked by comparing predicted performance with that measured for the eigensolver.[5] For the DELTA, we measured the following

[5] Significant discrepancies indicate that either the model is incorrect or that the testjig and actual codes are somehow behaving differently. In early testing, we found factor of 5 discrepancy in one parameter ($t_{xstartup}$). This discrepancy was traced to an anomaly in the message-passing primitives. A workaround was developed, and the computer development team was notified.

values:

t_{flop}	0.122 μsec (8.2 MFLOPS)
$t_{xstartup}$	153 μsec
$t_{xperelem}$	1.66 μsec (4.8 MB/sec/link = 0.6 MB/sec/node)
$t_{cstartup}$	210 μsec
$t_{cperelem}$	2.78 μsec
$f_{overhead}$	265

Figure 3 shows that using these parameter values in the performance model predicts the behavior of the BFJ method quite closely. Since this accuracy results from analyzing the algorithm, rather than from coincidentally fitting some standard function, we can be fairly confident about interpreting the numbers.

Each of the parameter values implies something about how well the BFJ method is using the machine. First, 8.2 MFLOPS for the BLAS calculations suggests that the Fortran compiler has done a fairly good job in this case – the BFJ method can use only Level 1 BLAS (vector-vector) and tends to overflow cache memory. For the computations done by the BFJ method, this is a situation in which the i860 would be hard-pressed to exceed 18 MFLOPS because of memory bottlenecks [9]. Second, a transfer startup time of 153 μsec suggests that only small improvements in startup time could be achieved by recoding the application – the best time to date by an optimized testjig code for a similar type of transfer is around 110 μsec. Third, the 9.6 MB/sec/node indicates that we are using the communication links reasonably efficiently – at the time these benchmarks were run, optimized testjig codes could achieve only slightly over 12 MB/sec/node. Fourth, the computational overhead cost of 265 indicates that the i860 handles straight-line code, subroutine calls, divide, and/or sqrt functions relatively less efficiently than other machines that we have tested. For example, the equivalent computational overhead cost for an NCUBE/ten™ [6] computer was only 48. This is not surprising since (1) the i860 architecture and compilers tend to reward loops that can be pipelined, and (2) the i860 computes sqrt in software, while the NCUBE does it with hardware. However, it is important to be aware of these differences – at one point, an inappropriate choice of compiler switches[7] caused the computational overhead cost to increase to 797 and significantly changed the relative performance of EISCUBE, PRS, and EISPACK RSP.

There are two ways to address the question of which characteristic limits the performance of the BFJ method. Viewed in isolation, the performance model says that execution time is a linear function of each parameter. Thus, it is tempting to look at the marginal effects, that is, $\partial \text{Time}/\partial \text{Parameter}_i$.

In the broader view, however, the performance of the BFJ method is actually a nonlinear function of the parameters because the algorithm allows more processors to be exploited, or the same number of processors to be used in

[6] ™ NCUBE/ten is a trademark of NCUBE, Beaverton, Oregon

[7] We omitted the -Knoieee flag, an oversight that caused the i860 to work very hard preserving the last two bits in a 64-bit number for divide and sqrt

Table 2. Predicted effect of improving each performance parameter

Improvement				
$t_{xstartup}$	$t_{xperelem}$	$F_{overhead}$	Optimum P	Relative time
			400	1.00
		$2X$	400	0.98
	$2X$		400	0.89
	$2X$	$2X$	400	0.87
$2X$			800	0.65
$2X$		$2X$	800	0.63
$2X$	$2X$		800	0.58
$2X$	$2X$	$2X$	800	0.56
$10X$	$2X$	$2X$	2900	0.20

different ways, depending on the relative values of various parameters. For example, as startup cost drops, it may be productive to add more rings, or to shift from one long ring to several short ones. Because of these nonlinear effects, it is more meaningful to hypothesize substantial changes in the parameters and look at the performance that could be achieved after re-optimizing the number and usage of processors.

Table 2 shows the predicted effect of improving three of the performance parameters and assuming that an unlimited number of processors is available. (We have left t_{flop} fixed, since it establishes the single-processor speed against which comparisons should be done.) The information in this Table reveals that:

- Communication startup time is the most important parameter.
- Reducing communication startup time would allow exploiting more processors.
- Given the current communication startup time, the optimum number of processors is only $P = 400$ (for a 200×200 matrix). Even if more processors were available, attempting to use larger P would make BFJ run slower.

From one perspective, these are disappointing results, since little improvement can be made in the startup time by modifying the BFJ method at the application program level. However, the observed startup time is two orders of magnitude larger than the hardware latency, indicating that most of the time is due to software processing. We are hopeful that significant improvements might be accomplished by changes in the computer operating system, and we are pursuing this possibility with the vendor.

5 Conclusions

The most important points from this work are that:

- $P > N$ is expected to be a common case for electronic structure calculations on massively parallel computers.
- Eigensolving is more likely to be a bottleneck with massively parallel computers than with serial or modestly parallel computers, where $P \ll N$.

- The maximum speedup for solving small dense eigensystems is modest at present (about 10 times for $N = 200$ in our tests). Maximum speedup depends on the relative speeds of communication and computation.
- Every method in our tests ran the fastest using less than the maximum available number of processors.
- Different algorithms may be required for large-scale parallelism (e.g., $P \sim N$ or larger) than with modest parallelism ($P \ll N$) – the fastest parallel solver in our tests (BFJ) is also the slowest serial method.
- Communication startup is the single most important limiting factor for the BFJ method.

The current results are not sufficient to conclude which methods will ultimately prove best. Further study is required to determine why the non-Jacobi methods do not scale well into the massively parallel regime, and what can be done to improve their performance. More practical experience will also be required to determine the performance impacts of such issues as solving for only some of the eigenvectors and maintaining high accuracy and orthogonality in the presence of degenerate eigenvalues.

Acknowledgements. This research was supported by the U.S. Department of Energy (DOE). Access to the Intel Touchstone DELTA System of the Concurrent Supercomputing Consortium was provided by the DOE. The PRS v.0 and v.1 codes were provided by Shirish Chinchalkar at Cornell, who developed them for applications in which $P \ll N$, but graciously allowed us to test them in a much different regime. We thank our colleagues, especially Dr. M. W. Feyereisen, Dr. R. A. Kendall, and Dr. M. A. Thompson, and an anonymous referee, for valuable discussions and reviews.

References

1. Intel Corp (1990) iPSC/2 and iPSC/860 User's Guide. Intel, Beaverton, Oregon
2. Intel Corp (1991) A Touchstone DELTA System Description. Intel Supercomputer Systems Division, Beaverton, Oregon
3. Eberlein PJ (1987) On using the Jacobi method on the hypercube. In: Proc Second Conf on Hypercube Multiprocessors, p 605–611
4. Eggers WS (1988) On parallel computation using one-sided jacobi method. MS Thesis, SUNY/Buffalo, June 1988
5. Golub G, Van Loan C (1989) Matrix computations, 2nd ed. Johns Hopkins Press, Baltimore, Maryland
6. Ipsen ICF, Jessup ER (1990) Solving the symmetric tridiagonal eigenvalue problem on the hypercube. SIAM J Sci Stat Comput 11(2):203–229, March 1990
7. Jessup ER (1989) Parallel solution of the symmetric tridiagonal eigenproblem. Technical Report YALEU/DCS/RR-728, Yale Univ, October 1989
8. Littlefield RJ, Maschhoff KJ (1991) Choosing processor array configuration by performance modeling for a highly parallel linear algebra algorithm. In: Proc Sixth Distributed Memory Computing Conference, p 600–605
9. Moyer SA (1991) Performance of the iPSC/860 node architecture. Technical Report IPC-TR-91-007, Univ of Virginia, May 1991
10. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1986) Numerical recipes – The art of scientific computing. Cambridge Univ Press

Appendix: BFJ algorithm and timing model

A1 BFJ algorithm

The BFJ method, like other "one-sided" Jacobi eigensolvers, is based on the same mathematics as the classic Jacobi method, namely:

$$A^{(k+1)} = J_k^T A^{(k)} J_k = V_k^T A^{(0)} V_k$$

The classic method stores and updates the A matrix to obtain eigenvalues. The V matrix is stored and updated only if eigenvectors are required. In contrast, the BFJ method stores and updates a new matrix G defined as $G^k \equiv V_k^T A^{(0)}$, so that $A^k = G^k V_k$. The V matrix is always stored and updated. When they are needed to compute rotations, elements of A are reconstructed via dot-products of individual rows of G and columns of V.

Ignoring parallelism and distribution of data, the BFJ algorithm for a real symmetric matrix can be outlined as follows:

Algorithm 1 (Non-distributed BFJ)

▷ Initialize $G = A$ and $V = I$ ($n \times n$ identity matrix)
▷ Loop until convergence
  ▷ For each off-diagonal element pq, annihilate A_{pq} as follows:
    /* *Reconstruct elements of A needed to determine the rotation* */
    $a_{pq} = G_{p*} \cdot V_{*q} \equiv \sum_{i=1}^{n} G_{pi} V_{iq}$
    $a_{pp} = G_{p*} \cdot V_{*p}$
    $a_{qq} = G_{q*} \cdot V_{*q}$
    /* *Compute rotation angle* */
    $\theta = (a_{qq} - a_{pp})/(2a_{pq})$
    $t = \text{sgn}(\theta)/(|\theta| + sqrt(\theta^2 + 1))$
    $c = 1/\text{sqrt}(t^2 + 1)$
    $s = tc$
    /* *Rotate rows of G and columns of V* */
    $T = G_{p*}$
    $G_{p*} = cT - sG_{q*}$
    $G_{q*} = sT + cG_{q*}$
    $T = V_{*p}$
    $V_{*p} = cT - sV_{*q}$
    $V_{*q} = sT + cV_{*q}$
  – End "for each off-diagonal element pq"
– End "loop until convergence"

The algorithm terminates with V being a matrix whose columns are eigenvectors. The corresponding eigenvalues can be retrieved as $e_i = G_{i*} \cdot V_{*i}$.

Convergence is tested by examining the magnitude of the off-diagonal elements. For the work reported here, we used simply $\max(|a_{pq}|) < 10^{-14}$ (before rotation). The development of a convergence test that is both efficient and accurate under all circumstances is beyond the scope of this paper. However, it is important to note that one-sided eigensolvers, including BFJ, are qualitatively different from the classic Jacobi method with respect to convergence. In the classic Jacobi method, the A matrix is stored explicitly and the off-diagonals can be driven arbitrarily close to zero by continued iteration. In one-sided methods, the off-diagonals are computed as dot-products, so that floating point roundoff errors bound their magnitude away from zero.

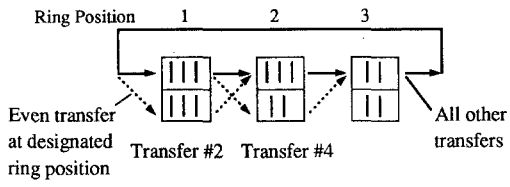

Fig. 4. Scrolling pattern for 3 processors organized in 1 ring. Each vertical line represents a *unit* consisting of a row of G and the corresponding column of V. Each processor holds 2 groups of units, called "Group I" and "Group II". In this example, $N = 15$; there are 3 groups containing 3 units and 3 groups containing 2 units

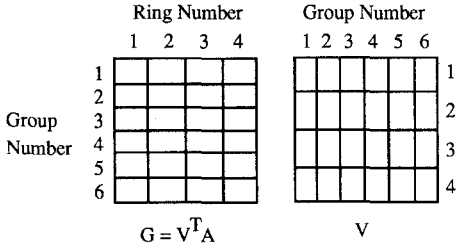

Fig. 5. Matrix distribution for 12 processors organized in 4 rings of 3 processors each. Each unit is spread across 4 processors at whatever ring position holds its group

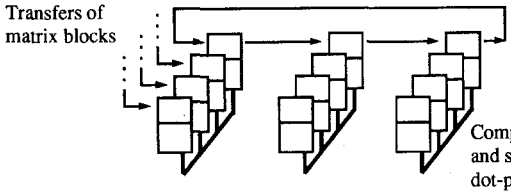

Fig. 6. Communication pattern for 12 processors organized in 4 rings

Many refinements are possible in this algorithm, such as avoiding the dot-products for a_{pp} and a_{qq} by explicitly storing and updating the diagonals, or reducing the operation count by using "fast rotations" to update G and V. However, the code that was tested for this paper, and the performance model described below, did not incorporate such refinements.

To parallelize the BFJ computation, the G and V matrices are distributed such that each processor holds G_{pi} and V_{ip} for at least two different p's and a range of i's. Inner loops are then added to process all off-diagonal elements by scrolling the data around a ring of P_R processors. The data distribution and scrolling are illustrated in Figs. 4, 5, and 6. Then each processor executes a program outlined as follows:

Algorithm 2 (Distributed BFJ)

- ▷ Initialize G_{pi} and V_{ip} for all i and p assigned to this processor
- ▷ Split the set of p's owned by this processor into "Group I" and "Group II". (The split is arbitrary except that all groups should be as close as possible to the same size.)
- ▷ Loop to convergence
  - ▷ For all pairs of p and q selected from Group I in this processor, annihilate A_{pq} as follows:

/* *Reconstruct elements of A* */
$a_{pq} = G_{p*} \cdot V_{*q}$
$a_{pp} = G_{p*} \cdot V_{*p}$
$a_{qq} = G_{q*} \cdot V_{*q}$
/* *Note that each processor can compute only part of each dot product or* a_{pp}, a_{qq}, *and* a_{pq}. *Communication across all processors at each ring position is required to combine the parts and share the complete dot products across those processors. For example,*

$$a_{pp} = \sum_{\text{processors owning } p} \left(\sum_{i \text{ owned by each processor}} G_{pi} V_{ip} \right)$$

*/
/* *Compute rotation angle (redundantly on each processor)* */
$\theta = (a_{qq} - a_{pp})/(2a_{pq})$
$t = \text{sgn}(\theta)/(|\theta| + sqrt(\theta^2 + 1))$
$c = 1/\text{sqrt}(t^2 + 1)$
$s = tc$
/* *Rotate those portions of G and V residing in this processor* */
$T = G_{p*}$
$G_{p*} = cT - sG_{q*}$
$G_{q*} = sT + cG_{q*}$
$T = V_{*p}$
$V_{*p} = cT - sV_{*q}$
$V_{*q} = sT + cV_{*q}$

– End "for all pairs of p and q selected from Group I . . ."
▷ For all pairs of p and q selected from Group II in this processor, annihilate A_{pq} as above.
/* *The following section annihilates* A_{pq} *for all p and q selected from all different groups on all processors.* */
▷ For round = 1 . . $2P_R - 1$
/* *Annihilate* A_{pq} *for all p and q in this processor.* */
▷ If Group I is larger than Group II then
▷ M = number of p's in Group I
▷ For step = 0 . . M − 1 (sequentially)
▷ For all q's in Group II (in parallel)
▷ Select a p from Group I so that all pq pairs are unique, e.g., for q_i, select $p_{(i+step) \bmod M}$
▷ Annihilate A_{pq} as above.
/* *For high efficiency, care should be taken to complete the dot products for all current pq pairs in a single communication phase.* */
– End "for all q's in Group II (in parallel)"
– End "for step = 1 . . M − 1 (sequentially)"
– else *Group II is larger or equal*
▷ Perform above loops, switching Groups I and II.
– endif "Group I is larger than Group II"
/* *Scroll data around the ring of processors.* */
▷ If round is even and processor position within ring = round/2 then send/receive Group II (*I.e., send Group II to successor and receive data from predecessor into Group II.*)

- else
  send/receive Group I
- endif
- End "for round = 1 . . $2P_R - 1$ . . ."

▷ Combine convergence data across all processors.

- End "loop to convergence"

A2 BFJ timing model

A2.1 Preliminaries. A performance model for the BFJ algorithm can be developed as follows. In general, we want to predict execution time as a function of matrix size, number and length of rings, and a small set of machine parameters that are easy to measure. Our basic strategy is to count operations and add up their times.

We assume the following:

1. Execution is loosely synchronous – all processors compute and exchange data at the same time.

2. There is no significant overlap between computation and communication.

3. Matrix rows and columns have been distributed to minimize the load imbalance, i.e., all groups are of size M or $M - 1$.

4. All communications can progress simultaneously. This corresponds to assuming that either (a) the computation can be mapped to the physical machine so as to avoid conflicts, or (b) the machine's physical interconnection scheme is fast enough that conflicts can be ignored.

A2.2 Definitions

- Problem variables and algorithmic parameters:

  N matrix dimension
  P_R number of processors per ring
  R number of rings
  M maximum columns per group: $M = \lceil N/(2P_R) \rceil$
  L maximum length of vector segment: $L = \lceil N/R \rceil$
  $F_{overhead}$ fixed computational cost for a single Jacobi rotation, expressed as an equivalent number of floating point operations. This cost includes subroutine entry/exit, computing the rotation angle, and so on.

- Basic machine and operating system speeds.

  t_{flop} floating point operation time (as measured for the vector operations of a rotation)
  $t_{xstartup}$ transfer startup time
  $t_{xperelem}$ transfer time per element
  $t_{cstartup}$ combine startup time (to complete and share dot products)
  $t_{cperelem}$ combine time per element

Combine costs will depend on the number of processors involved. We model these as

$$t_{cstartup}(p) = \log_2 p * t_{cstartup}(2)$$

$$t_{cperelem}(p) = \log_2 p * t_{cperelem}(2)$$

where p is the number of processors involved and the times on the right are constants that can be measured easily.

A2.3 Equations. At the highest level, BFJ is modeled as:

$$t_{sweep} = t_{int} + (2P_R - 1)t_{scroll} + t_{collect}$$

where

t_{sweep} time per sweep
t_{int} time required to perform all Jacobi rotations.
t_{scroll} time required to transfer row and column groups from one processor to another.
$t_{collect}$ time required to collect convergence status.

Two of these components can be modeled quite simply.

$$t_{scroll} = \begin{cases} t_{xstartup} + t_{xperelem} * (2ML + 5), & P_R > 1 \\ 0 & P_R = 1 \end{cases}$$

$$t_{collect} = t_{cstartup}(P) + t_{cperelem}(P) * 1$$

The number of 5 in t_{scroll} accounts for a small amount of auxiliary data that our code carries along with the matrix rows and columns.

To model t_{int}, we rely on the assumption that execution is loosely synchronous, so that the total time for each round is that of the busiest processor. In some rounds, the busiest processor will have two groups containing M units (a "big-big" pairing). In other rounds, big groups will be paired only with "little" groups containing $M - 1$ units.

Define the following:

B the number of rounds containing big-big pairings.
$C(m)$ the average cost of a Jacobi rotation in the busiest processor, whose smaller group is size m:

$$C(m) = ((18L + F_{overhead}) * t_{flop} + t_{comb}(3, m, R))$$

$t_{comb}(n, m, r)$ the average time needed to combine n numbers across r processors, presuming that m sets of r are done at once.

$$t_{comb}(n, m, r) = \frac{t_{cstartup}(r) + t_{cperelem}(r) * n * m}{m}$$

Then:

$$\begin{aligned} t_{int} = {} & 1 * (M * (M - 1) * C(M - 1)) + \\ & B * (M * M * C(M)) + \\ & (2P_R - 1 - B) * (M * (M - 1) * C(M - 1)) \end{aligned}$$

The three lines of this formula account for the rounds that are determined by intra-group, big-big, and big-little pairings, respectively. This expression simplifies to:

$$\begin{aligned} t_{int} = {} & B * (M * M * C(M)) + \\ & (2P_R - B) * (M * (M - 1) * C(M - 1)) \end{aligned}$$

Finally, for the scrolling pattern that we use, the number of big-big rounds is simply

$$B = \begin{cases} 0 & \text{if } K \text{ is } 1 \\ 2K - 3 & \text{if } 1 < K \leqslant P_R \\ 2P_R - 1 & \text{otherwise } (K > P_R) \end{cases}$$