From Cluster to Grid: A Case Study in Scaling-Up a Molecular Electronics Simulation Code

Calvin Ribbens* Prachi Bora* Massimiliano Di Ventra† Joshua Hauck*
Sandeep Prabhakar* Christopher Taylor*

Abstract

This paper describes an ongoing project whose goal is to significantly improve the performance and applicability of a molecular electronics simulation code. The specific goals are to (1) increase computational performance on the simulation problems currently being solved by our physics collaborators; (2) allow much larger problems to be solved in reasonable time; and (3) expand the set of resources available to the code, from a single homogeneous cluster to a campus-wide computational grid, while maintaining acceptable performance across this larger set of resources. We describe the sequential performance of the code, the performance of two parallel versions, and the benefits of problem-specific load balancing strategies. The grid context motivates the need for runtime algorithm selection; we present a component-based software framework that makes this possible.

1 Introduction

Scalability is a central concern of high performance computing (HPC). Without scalable hardware, software systems, algorithms, and implementations, the potential of HPC to help solve large-scale computational science problems is not fulfilled. While many technical definitions and measures of scalability are possible, our applications collaborators simply want an answer to the question "Can I effectively use more resources to solve proportionally larger problems?". The goal of computer scientists working in HPC is to answer that question in the affirmative, and to make it happen in a way that is as transparent as possible to the scientist.

Our goal in this paper is to describe several stages in a two-year effort to significantly scale-up the performance and applicability of a large-scale computational science code. It is well-known in the HPC community that achieving good scalable performance on hundreds or thousands of processors is rarely a trivial process. It is made more difficult in our context because our ultimate goal is to run the application across a computational grid [11, 22] consisting of multiple clusters, shared-memory parallel machines (SMPs), and high-end workstations. Hence, after describing in Section 4 two parallel versions of the code designed for homogeneous clusters, we concentrate in Sections 5 and 6 on issues motivated primarily by heterogeneous platforms, namely application-aware load balancing and run-time algorithm selection. We turn first, however, to an introduction of the application code and its computational requirements in Section 2, and then describe its sequential performance in Section 3.

2 The application

The `transport` code is used by faculty and students of the Physics Department at Virginia Tech (VT) to study the physical properties of nanoscale electronics devices. Originally written by N. D. Lang (IBM, T.J. Watson), `transport` has been modified and extended by Massimiliano Di Ventra (VT, Physics) to model several new physical phenomena. The code uses first-principles approaches to calculate non-linear transport properties of molecular structures [6, 9, 8, 5, 7]. Molecular electronics has received tremendous attention recently as an approach to further miniaturization in device design [20]. Di Ventra's research seeks to understand and predict phenomena that are observed by experimentalists in the field. For example, Di Ventra was the first to account for the nonlinear current-voltage characteristics of phenyl molecules observed experimentally by Reed et al. [21]. He also elucidated the importance of contact geometry and chemistry in modulating the current in these systems and explained that the unusual switching effects ob-

*Department of Computer Science, Virginia Tech, Blacksburg, VA 24061. Corresponding author: ribbens@vt.edu.

†Department of Physics, Virginia Tech, Blacksburg, VA 24061.

Table 1: Problem parameters and memory requirements (in Gb).

N_x	N_y	N_z	N_{mat}	M_{est}
4	4	8	1377	0.17
5	5	10	2541	0.58
6	6	12	4225	1.60
7	7	14	6525	3.81
8	8	16	9537	8.13
9	9	18	13357	15.95
10	10	20	18081	29.23

served in certain organic molecules [3] might be due to the electronic coupling to low-energy bosons.

A typical numerical experiment enabled by `transport` is the computation of the 'I-V characteristic' curve shown in Figure 1. Note that `transport` must be run 100's or 1000's of times for one of these experiments. The cost of each run is dominated by the solution of many large nonsymmetric systems of linear algebraic equations—one or two for each energy level (loop 6 in Figure 1). Each discretized Lippman-Schwinger equation is actually a matrix equation, i.e., there are multiple right hand sides for each matrix. Typical values are 32 to 128 energy levels and 64 to 256 right hand sides per energy level. In Figure 2 we give a more detailed description of loop 6, the loop over energy levels. This pseudocode omits many details, but it is based directly on the code and helps to identify the computational bottlenecks.

The problem parameters that drive the computational cost of a run of `transport` are the number of energy levels N_e (`n_e` in Figure 2) and the number of plane waves in each direction (N_x, N_y, N_z). Typical values for N_e are between 32 and 128. Typical values of (N_x, N_y, N_z) are shown in Table 1. For all experiments reported in this paper, N_e is fixed at 32. Thus, problem size for these results is completely determined by the number of plane waves; we use the simple notation (N_x, N_y, N_z) to indicate 'problem size' throughout. Table 1 also shows N_{mat}, the (one-dimensional) size of the linear systems that are solved for a given problem instance; and $M_{\text{est}} = 6 * 16 * N_{\text{mat}}^2$, an estimate of the total amount of memory required. The memory estimate is based on the need for six $N_{\text{mat}} \times N_{\text{mat}}$ double complex arrays, a requirement that dominates the overall memory requirements of the code. Note that the matrix size is given by $N_{\text{mat}} = (2{*}N_x+1){*}(2{*}N_y+1){*}(2{*}N_z+1)$, and that the matrices in `transport` are essentially dense and so are stored as two-dimensional arrays. Clearly, the memory and computational requirements of `transport` are substantial. We profile the performance of the code more carefully in the next section.

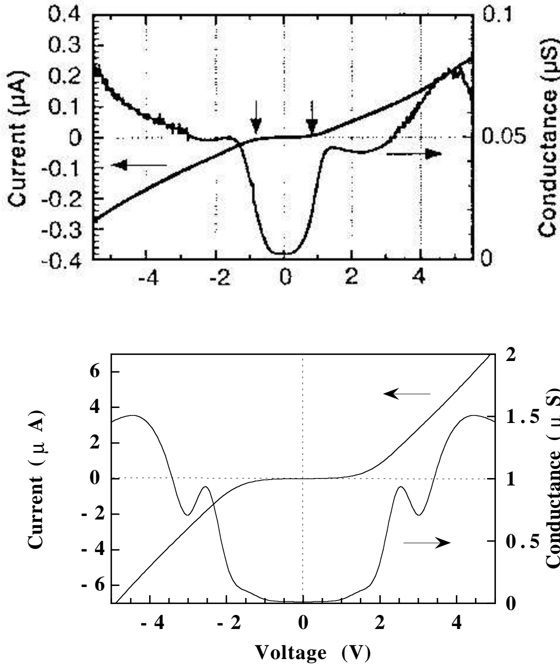
All experimental results were run on the *Anantham* cluster located in Virginia Tech's Laboratory for Advanced Scientific Computing and Applications (LASCA). Each of the 200 compute nodes of *Anantham* is a 1.0 GHz AMD Athlon running Linux, with 1 GB of memory. The nodes of the cluster are interconnected by a 2.56 Gb/s Myrinet network.

We began this project with a version of `transport` that had been parallelized in a straightforward way for execution in a distributed memory, message passing environment. This version of the code parallelizes only the energy loop (line 6 in Figure 1). The computation associated with a single energy level must fit on a single compute node. On *Anantham*, this means that problem sizes larger than $(5, 5, 10)$ will not fit in main memory (see Table 1). Furthermore, scalability is limited by this strategy since there are no more than N_e tasks that can run in parallel. With $32 \leq N_e \leq 128$, this is a significant limit on scalability. In Section 4.1 we describe a second parallel version of `transport` which can use more processors, motivated both by memory constraints and parallel scalability issues.

3 Sequential performance

Importance of fast BLAS

Like many scientific applications, `transport`'s run time is dominated by a few simple linear algebra operations. For many years the scientific computing community has relied on efficient implementations of the Basic Linear Algebra Subprograms (BLAS) [14] to achieve high performance on a variety of architectures. Since `transport` uses BLAS functions, an efficient implementation of these kernels makes a substantial difference in performance. We used the ATLAS [23] implementation of the BLAS for our experiments (ATLAS version 3.2.1). The improvement over the reference (FORTRAN) implementation of the BLAS is significant. For example, for problem size $(3, 3, 6)$ the time for a 2-iteration test case is reduced from 3507 seconds to 1618 seconds when ATLAS is used instead of the reference implementation, an improvement of a factor of 2.17. For a $(4, 4, 8)$ test case the improvement is even greater: from 28468 seconds to 8764 seconds, a factor of 3.25. The benefit of fast BLAS is greater for larger problem sizes because



1. **foreach** bias level
2. **foreach** molecule configuration
3. initialize potential
4. **while** force not converged
5. **while** density not converged
6. **foreach** energy level
7. setup Lippman-Schwinger equation
8. solve for new wave functions
9. **endfor**
10. compute new density
11. compute new potential
12. check for convergence in density
13. **endwhile**
14. compute new force
15. check for convergence in force
16. **endwhile**
17. **endfor**
18. **endfor**

Figure 1: Top left: experimental I-V characteristic of benzene-1,4-dithiolate molecules between bulk electrodes measured by Reed et al. [21]. Bottom left: theoretical I-V curve [9]. Right: algorithm used to compute the theoretical curve. Ten or more bias levels are required (loop 1); for each bias 100 or more molecular configurations may be needed (loop 2). Steps 3-16 correspond to one run of `transport`. Parallelism can be exploited trivially at loops 1 and 2, and with more effort at loop 6 and within steps 7-8.

```
for ie = 1 to n_e             /* typical value of n_e is 32 or 128 */
  for idirec = 1 to id        /* id is 1 or 2 */
    call store_ggz            /* compute/restore Green's function matrices */
    call zgemm                /* matrix-matrix multiply */
    call zgetrf               /* LU factorization */
    for ikapax = 1 to nkapax  /* nkapax varies from 8 to 32 */
      for iphi = 1 to nphi    /* nphi is always 8 */
        call zgetrs           /* triangular solve */
      endfor
    endfor
  endfor
endfor
```

Figure 2: Important steps in the main loop over energy levels (steps 6-9 in Figure 1).

a greater percentage of the overall computing time is spent in BLAS calls as problem size grows. For the remainder of this paper, all results use the ATLAS BLAS.

Profiling the code

Recall from the pseudocode in Figures 1 and 2 that the algorithm implemented in `transport` is basically an outer iteration toward convergence in force and density, with each iteration dominated by a loop over N_e energy levels. In the initial parallel implementation of `transport`, only the energy loop is parallelized, i.e., loop 6 in Figure 1, expanded in Figure 2. To evaluate parallel performance, it is useful to determine which sections of the code dominate the sequential performance. Table 2 gives timing results for three test problems, each run for two iterations of the outer loop (loop 5 in Figure 1). The table gives time and percent of the total time for various phases of the computation.

Note first that the initialization time is negligible compared to the time spent in the main iteration; this is especially true as problem size grows and when we recall that a typical computation may run for 10 or 20 iterations rather than just two. Within the iteration, the time spent in the energy loop is increasingly dominant as problem size grows. (The time reported for the 'Energy Loop' is a subset of the 'Iteration' time; and the time reported as 'Linear Algebra' is in turn a subset of the 'Energy Loop' time.) However, we already see a hint of scalability limitations in the current implementation. For example, for problem size (5,5,10), 3.7% of the iteration time is *not* in the parallelized energy loop; so by Amdahl's Law, we know we will never see parallel speedup of more than about 25 on this problem with this parallel implementation. Finally, the time reported as 'Linear Algebra' includes only the calls to the BLAS (`zgemm`) and LAPACK [1] (`zgetrf` and `zgetrs`) routines (see Figure 2). We total these separately because there are parallel implementations of these codes which can be used to increase the scalability of `transport`. Notice that the Linear Algebra time is substantial, and growing with problem size; but there is considerable work being done outside these calls as well.

4 Parallel performance

We now turn to the performance of the initial parallel version of `transport`. Figure 3 shows fixed problem-size parallel speedup for problem sizes $(3, 3, 6)$ and $(4, 4, 8)$. We show results for both a statically scheduled and a dynamically scheduled version of the code. The scheduling strategy refers to how the energy levels are distributed to processors. There are 32 energy levels for all of the test problems used in this paper. In the statically scheduled version, each processor gets approximately the same number of energy levels. The dynamically scheduled version uses a 'master/worker' paradigm, where energy levels are assigned to processors one at a time. In the current implementation, one processor serves as the master and does none of the work corresponding to an energy level. The 'Number of Processors' shown in Figure 3 actually corresponds to the number of 'workers'; one additional processor was used in the role of 'master.'

As can be seen from Figure 3, dynamic scheduling is clearly better, both in terms of speedup and in terms of efficiently using any number processors. This result is not surprising since different energy levels require different amounts of computation. For example, in Figure 4 we see that the relative cost of a single energy level computation can vary by more than a factor of 3. Since there are only 32 energy levels, we can also see that using more than 16 worker processes is not likely to be helpful; as long as at least one worker is assigned 2 energy levels, that worker is likely to be the bottleneck. With 32 workers we can assign only one energy level to each worker; but Figure 4 shows that this case corresponds to considerable processor idle time as well. Although not shown in Figure 3, we did run problem size $(4, 4, 8)$ with 32 worker processors, achieving a speedup of only 8.8; this is very small improvement over the speedup of 7.6 achieved with 16 workers.

With 16 or fewer workers the dynamic scheduling strategy yields a fairly balanced workload. For example, with problem size $(4, 4, 8)$ and 8 workers, the *load balance factor* (lbf) is only 1.03, where

$$lbf = \frac{\text{time for most loaded processor}}{\text{average time for all processors}}.$$

With 12 and 16 workers, the corresponding lbf values are 1.11 and 1.07, respectively—quite acceptable considering the large granularity of the tasks being assigned. Finally, we note that any load balancing problems are relative to the number of energy levels and the number of worker nodes, i.e., as the number of energy levels grows with respect to the number of workers, the load balancing problems are reduced.

In terms of parallel speedup, the peak performance shown in Figure 3 is only a speedup of 7.6 on 16 processors. Even with a larger problem size $((5, 5, 10)$,

Table 2: Sequential performance of `transport`: time in seconds for three test problems.

	(3,3,6)		(4,4,8)		(5,5,10)	
Phase	Time	Percent	Time	Percent	Time	Percent
Initialization	38	2.4	84	1.0	157	0.4
Iteration (2 iters)	1547	97.6	8576	99.0	37361	99.6
Energy Loop	1305	82.3	8072	93.2	36143	96.3
Linear Algebra	704	44.4	5359	61.9	26839	71.5
Total	1585	100.0	8660	100.0	37518	100.0

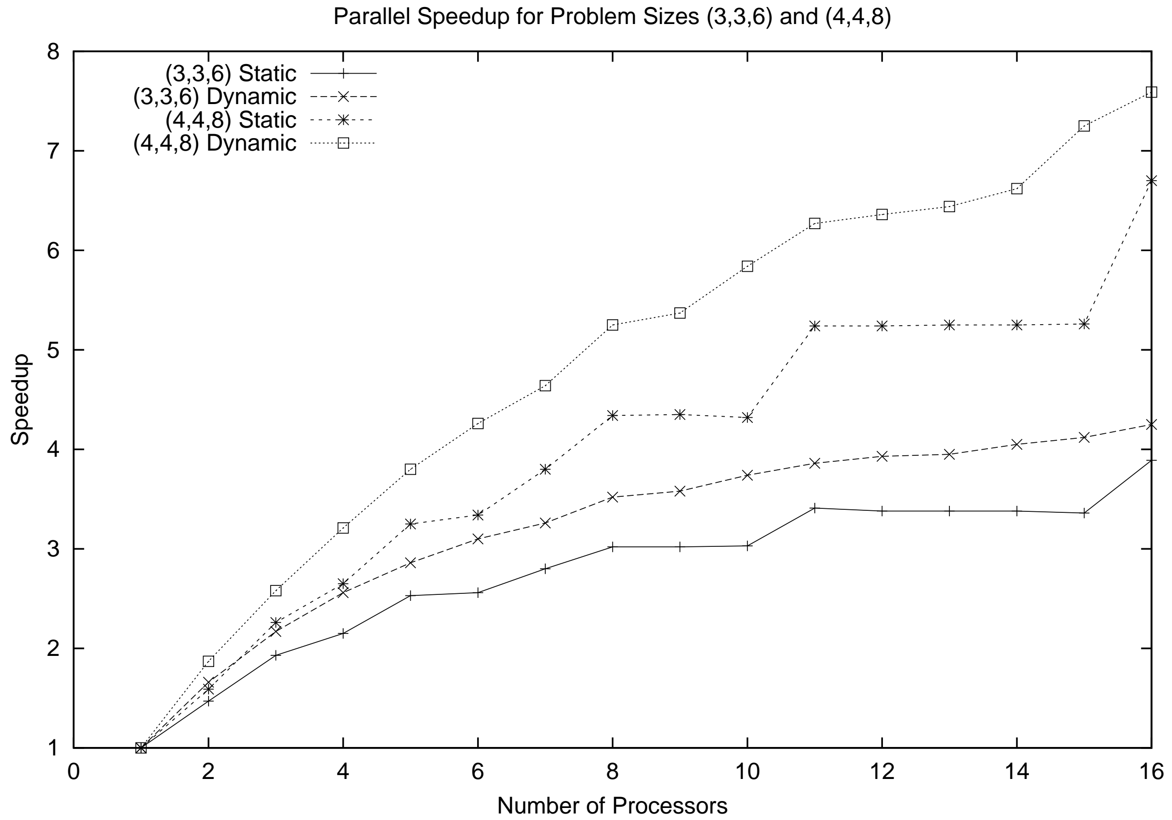


Figure 3: Fixed problem-size parallel speedup.

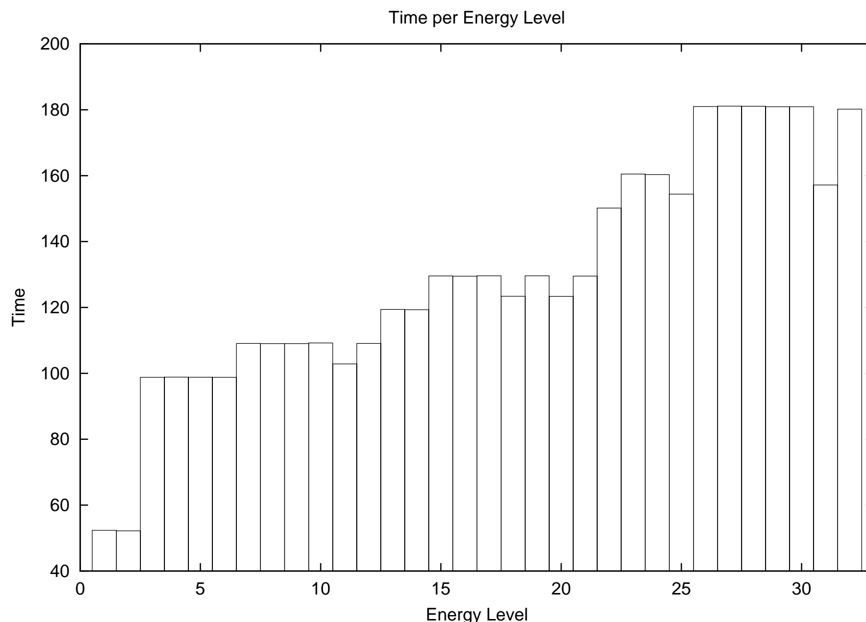


Figure 4: Time (in seconds) for various energy levels, for problem size (4,4,8), first iteration.

not shown in Figure 3), the best observed speedup was 11.3 with 32 worker processors. These less-than-ideal results are not surprising considering the size of the remaining sequential portion of the code and the load imbalance issues just discussed. The remaining sequential portion of the code is the primary culprit. For example, on problem size (4,4,8) with 16 worker processes, of the 1150 seconds total wallclock time required to complete the computation, 591 seconds are spent in purely sequential (redundant) work— 86 seconds in the Initialization phase and 505 in the sequential portions of the main iteration. In fact, the parallel speedup of the energy loop is relatively good— from 8072 seconds with 1 worker down to 559 with 16 workers, a speedup of 14.4. Only some relatively minor load imbalance causes this speedup in the energy loop to be less than a perfect 16.0. This load imbalance cannot be avoided unless the parallel algorithm is changed. We have not given much attention to the remaining sequential portion of the mani iteration because its relative importance shrinks as problem size grows; fortunately, it will parallelize straightforwardly.

4.1 A more scalable implementation

In order to scale up in both problem size and number of processors, we must exploit parallelism at more than one level in `transport`. This is also necessary in order to investigate new scientific questions, which require larger problem sizes, e.g., (10,10,20) plane waves. In either case—using more processors efficiently on current problem sizes, or handling larger problem sizes—it is necessary to exploit parallelism *within* an energy level computation as well as between energy levels. This involves distributing the largest data structures and the work associated with them across multiple processors. We focus on the 'Linear Algebra' portion of the energy loop because it increasingly dominates as problem size grows. This modification uses distributed memory parallel implementations of BLAS and LAPACK, namely PBLAS [4] and ScaLAPACK [2], respectively. We note that in setting up the matrix problems to reflect ScaLAPACK's preferred data decomposition, the overhead of calls to utility functions at inner loops can be a serious bottleneck; careful by-hand optimization was needed to avoid this overhead.

It is not the best strategy to spread a single energy level computation across all P processors, for large P. The dominant computations for a given energy level are linear algebra operations on matrices on the order of $10^4 \times 10^4$; this is not a large problem if distributed across 100's of processors. Instead, we use two levels of parallelism, assigning each energy level to a small 'team' of processors, with each team working in parallel on the computations for that energy level. (The question of team size is addressed in Sec-

Table 3: Performance of a single ScaLAPACK LU factorization arising in `transport`, for various processor grids. Matrix size is $N_{\mathrm{mat}} = 2541$. 'Spdup' is speedup relative to the one processor LAPACK case, 'Rate' is in millions of double complex floating point operations per second.

Proc. Grid	Time	Spdup	Rate	$\frac{\text{Rate}}{\text{Proc}}$
LAPACK	44.8	—	244.1	244.1
1×1	44.9	1.00	243.6	243.6
1×2	29.6	1.51	369.5	184.8
1×3	24.8	1.81	441.0	147.0
2×2	14.5	3.09	754.3	188.6
1×5	17.0	2.64	643.4	128.7
2×3	11.6	3.86	942.9	157.1
1×7	14.4	3.11	759.6	108.5
2×4	10.0	4.48	1093.8	136.7
3×3	8.7	5.15	1257.2	139.7

tion 5.) By keeping the team sizes relatively small, we are able to achieve near peak speed for the linear algebra portion of `transport`. For example, Table 3 shows the computational rate for a single LU factorization step for problem size (5,5,10). Note that the rate per processor degrades relatively slowly as the number of processors increases, despite the relatively small fixed problem size. Beyond nine processors, performance tails of sharply.

5 Load balancing strategies

As noted in Section 4, our first parallel version of `transport` achieves a reasonably balanced work decomposition as long as the number of worker processes is small relative to the number of energy levels N_e. However, load balancing is not so straightforward for the second, more scalable, version of the algorithm. The main question is how to choose the number of teams N_t, given the total number of processors P. (For now, we assume P is given and that each team will be of the same size.) At one extreme, setting $N_t = 1$ would achieve good load balance, but often with a poor computational rate—especially for large P—because each linear algebra problem would be distributed across too many processors. In general, computational rates for the dominant linear algebra steps improve as N_t grows, i.e., as team size shrinks; but there are two constraints which must be enforced: (1) there must be enough aggregate memory in a team to handle a single energy level, and (2) N_t must be relatively small compared to N_e to avoid serious load imbalance. Heuristically, we set $N_t = \min\{P/k_{\min}, N_e/2\}$, where $k_{\min}$ is the minimum number of processors required to handle a single energy level. The value of $k_{\min}$ can be computed *a priori*, given the problem size (N_x, N_y, N_z) and the amount of memory per processor.

Notice that even this simple load-balancing scheme requires information from the application (N_e and (N_x, N_y, N_z)) and the machine (memory size). In fact, with a little more information from the application we can improve the load balance further in many cases, by replacing the dynamic master/worker scheduling by a near-optimal static schedule. The key point is that for this application and algorithm, the amount of work needed to solve for a given energy level can be estimated very accurately, given the problem parameters shown in Figure 2. Given the cost of each energy level, an optimal assignment of the N_e energy levels to N_t teams can easily be derived. When $N_t \approx N_e/2$, we realize an improvement in running time of at least 10% on typical problems with optimal static scheduling rather than simple master/worker scheduling.

The assumption so far in this discussion of load balancing has been that the number of processors P is given. A further assumption has been that we are using a homogeneous computing platform, i.e., we have assumed that workload should be proportional to team size. Relaxing these assumptions is desirable because it will allow users or schedulers to choose different values of P depending on various criteria; relaxing these assumptions is necessary if we are going to scale up to a heterogeneous grid environment.

The key to scheduling and load-balancing across a grid is performance prediction. We are designing a *performance prediction service* (PPS) for `transport`. The PPS will take as input problem parameters describing a particular energy level and a computing resource proposed for that energy level, including number of processors and machine and interconnection characteristics (taken from a list of known resources on our campus grid). The PPS will return an estimate of the running time of that energy level on that resource. Our scheduling strategy [17] requires that a single energy level be assigned to a single tightly-coupled homogeneous resource—this simplifies the performance prediction enormously, and reflects the fact that there is considerably more communication *within* a team than *between* teams. The PPS derives performance estimates using a *recom-*

mender system which uses a data-mining approach to infer patterns and relationships from a database of previous performance results [12, 19]. In this way, load-balancing and scheduling is application-aware and resource-aware in a very broad sense: we leverage not only information from a single instance of an application on a single resource, but potentially from all past runs of that application on all the resources of our grid.

6 Run-time algorithm selection

A second major issue we confront as we scale up to grid-based computations is the need for run-time algorithm selection. To motivate this requirement, consider the example of linear solvers. It is well-known that different algorithms and implementations work best on different platforms, e.g., ScaLAPACK for dense systems on distributed memory machines, LAPACK on a sequential machine, perhaps SuperLU on a shared-memory machine, and dozens of preconditioned iterative solvers to choose from for sparse systems on various architectures. Even if the application knew the best solver to use on a given platform (a research topic in its own right), there is a serious software engineering obstacle to deploying the right solver at the right time. It is unreasonable to expect the application writer to know about all possible solvers at development time. Maintaining different versions of the application for each platform is also tedious, at best, and rules out the possibility of choosing a solver based on run-time information, e.g., matrix characteristics, team size, interconnection network, etc. A better solution is to allow selection of algorithms at run-time.

One approach to run-time algorithm selection is to use shared libraries. When shared libraries are used, the application must contain enough information to find the library containing a given solver, and the library is loaded when the executable is invoked. Using shared libraries we could load the solver we want at runtime. However, this is not transparent to the application since it needs to make explicit system calls, like `dlopen`, to load appropriate libraries.

6.1 Babel: component technology for scientific computing

A better approach to supporting run-time algorithm selection takes advantage of the rapidly maturing world of component frameworks. Code reuse using component technologies has gained popularity in the software industry. Technologies like CORBA and COM [18] are being used on a large scale to develop components for business applications. However, these technologies do not address many of the specific needs of scientific applications. For example, they do not support complex numbers, do not have FORTRAN style dynamic multi-dimensional arrays, do not have a way to describe massively parallel distributed objects and are not optimized for function calls within the same address space.

Babel is a component tool developed at the Lawrence Livermore National Laboratory [10, 15] which addresses these issues pertaining to scientific applications. Babel provides a way to define and support interoperability between codes written in multiple languages. The functionality provided by a component is expressed using Babel's Scientific Interface Definition Language (SIDL). SIDL is similar to Interface Definition Language in COM and CORBA, but is targeted at scientific applications. SIDL only defines the interfaces and not their implementation. The Babel compiler reads the SIDL definition and generates glue code—stubs, skeletons, internal object representation and implementation prototypes—for each component. It is the responsibility of library writers to fill in library specific code in the implementation prototypes. Application (client) programs can then access components through standard interfaces.

6.2 The SolverBuilder abstraction

Babel supplies a useful tool for handling language interoperability problems, and it gives us convenient 'plug-and-play' capabilities in that we can try different solvers on different resources and problem instances. The straightforward approach is to define a 'Solver' interface and provide multiple implementations of that interface, one for each particular solver. However, the application writer must still make decisions at compile or link time about which particular solver to instantiate, and where to find that library. We want to make algorithm and software selection much more dynamic and seamless from the point of view of the application developer. A more powerful abstraction is required. Following Kohn et al. [13], we define an interface called 'SolverBuilder' to make this possible (see Figure 5). A class implementing the SolverBuilder interface is similar to a factory class in CORBA [16]. A SolverBuilder class provides the capability for creating families of related objects. They are used when the decision of which class to instantiate must be made at run time and cannot be determined during development. The SolverBuilder en-

capsulates the logic needed to decide which subclass to instantiate. This is transparent to the application because object selection is delegated to the SolverBuilder.

The SolverBuilder abstraction can be used in `transport` as follows (see Figure 5). First, we provide various implementations of the Solver interface, e.g., one for LAPACK, one for ScaLAPACK, etc. When a solver needs to be instantiated, the application calls an implementation of the SolverBuilder interface and sets a parameter indicating resource constraints. Our simplest implementation of SolverBuilder just chooses between LAPACK and ScaLAPACK depending on whether team size is greater than one or not. The method GetSolver of SolverBuilder is called to obtain a reference to the appropriate solver. This method in turn looks at the parameter set (in this case, 'team-size') and instantiates the corresponding solver. The application then calls SetMatrix to initialize the matrix, and subsequently calls Solve on the Solver object thus obtained. More powerful SolverBuilders can be used to choose algorithms in more complicated situations. For example, if the choice of the solver is to be made based on the matrix (linear operator) characteristics, then the Setup method of SolverBuilder would be called before obtaining a reference to the Solver. The solver chosen by SolverBuilder then would be a function of the matrix characteristics.

7 Conclusions and future work

The paper illustrates the challenge inherent in achieving good performance with large-scale applications on the grid. A wide variety of contributions were used, including optimized low-level kernels, parallel algorithms and data decomposition strategies, parallel mathematical software, application-aware and resource-aware load-balancing schemes, and component frameworks for runtime software selection. Several of these contributions are quite problem-specific. There is no substitute for knowing the application and working with the physicist. Yet there are good possibilities for general strategies and tools as well.

We are investigating a more general PPS tool which uses a data-centric approach to make performance predictions for a wide class of problems, based on results from many previous runs. The SolverBuilder abstraction can also be generalized; we are investigating its use in other applications where algorithms and software for key problem-solving steps should be selected at run-time, based on problem and resource characteristics. *Context-aware* approaches such as these will be required to achieve consistent, scalable performance for large-scale grid-based scientific applications.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] L. S. Blackford, J. Choi, A Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[3] J. Chen, M.A. Reed, A.M. Rawlett, and J.M. Tour. Large on-off ratios and negative differential resistance in a molecular electronic device. *Science*, 286:1550, 1999.

[4] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, University of Tennessee, 1995.

[5] M. Di Ventra, S-G. Kim, S.T. Pantelides, and N.D. Lang. Temperature effects on the transport properties of molecules. *Phys. Rev. Lett.*, 86:288, 2001.

[6] M. Di Ventra and N.D. Lang. Transport in nanoscale conductors from first principles. *Phys. Rev. B*, 65:045402, 2002.

[7] M. Di Ventra and S.T. Pantelides. Hellmann-feynman theorem and the definition of forces in quantum time-dependent and transport problems. *Phys. Rev B*, 61:16207, 2000.

[8] M. Di Ventra, S.T. Pantelides, and N.D. Lang. The benzene molecule as a resonant-tunneling transistor. *Appl. Phys. Lett.*, 76:3448, 2000.

[9] M. Di Ventra, S.T. Pantelides, and N.D. Lang. First-principles calculation of transport properties of a molecular device. *Phys. Rev. Lett.*, 84:979, 2000.

```
Interface Solver
{
   int SetMatrix(in Matrix A);
   int Solve(in Vector b, inout Vector x);
}

Interface SolverBuilder
{
   int SetParameterDouble(in string name, in double value);
   int SetParameterInt(in string name, in int value);
   int Setup(in Matrix A);
   int GetSolver(out Solver S);
}
```

Figure 5: SIDL fragment representing SolverBuilder abstraction.

[10] T. Epperly, S. Kohn, and G. Kumfert. Component technology for high-performance scientific simulation software. In *Proceedings of the Working Conference on Software Architectures for Scientific Computing Applications*, Ottowa, Ontario, Canada, October 2000. International Federation for Information Processing.

[11] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Orlando, FL, 1999.

[12] E.N. Houstis, A.C. Catlin, J.R. Rice, V.S. Verykios, N. Ramakrishnan, and C.E. Houstis. PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Transactions on Mathematical Software*, Vol. 26(2):pages 227–253, June 2000.

[13] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In C. Koelbel and J. Meza, editors, *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, PA, 2001. SIAM.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[15] LLNL components research group. See www.llnl.gov/CASC/components.

[16] Object management group homepage. http://www.omg.org/.

[17] S. Prabhakar, C. Ribbens, and P. Bora. Multifaceted web services: An approach to secure and scalable grid scheduling. Technical Report 02–26, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, 2002.

[18] J. Pritchard. *COM and CORBA Side by Side*. Addison Wesley, Reading, MA, 1999.

[19] N. Ramakrishnan and C. J. Ribbens. Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes. *ACM Trans. Math. Softw.*, 26:254–273, 2000.

[20] M.A. Reed and J.M. Tour. Computing with molecules. *Scientific American*, 282(86):86–93, June 2000.

[21] M.A. Reed, C. Zhou, C.J. Muller, T.P. Burgin, and J.M. Tour. Conductance of a molecular junction. *Science*, 278:252, 1997.

[22] VT grid research group. See research.cs.vt.edu/lasca/grid.

[23] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. Technical Report CS-00-448, University of Tennessee, Knoxville, TN, 2000.