Performance of a Parallel Transport Code for Molecular Electronics Simulations

Brent Metz* Justin Wienckowski* Calvin Ribbens* Massimiliano Di Ventra†

February 4, 2002

Abstract

We describe the sequential and parallel performance of a nonlinear transport simulation code. This code is used by researchers at Virginia Tech to investigate phenomena underlying the emerging field of molecular electronics. The computational requirements of the code are summarized, and an initial distributed-memory parallel implementation of the code is evaluated. We conclude with several suggestions for improving the parallel performance and scalability of the code.

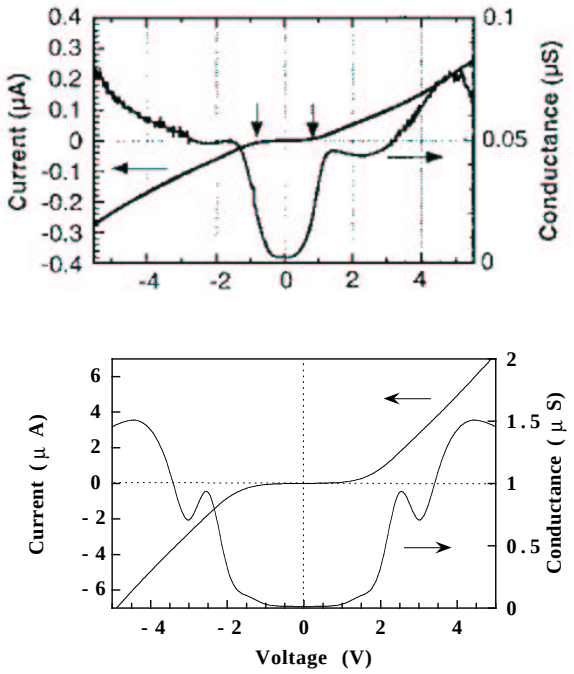
1 Introduction

The purpose of the experiments reported here was to evaluate and improve the parallel performance of a simulation code known as `transport`. This code is used by faculty and students in the Physics Department at Virginia Tech to do research in molecular electronics. Originally written by N. D. Lang (IBM, T.J. Watson), `transport` has been modified and extended by Massimiliano Di Ventra (VT, Physics) to include several new physical phenomena in the model. The goal of `transport` is to calculate non-linear transport properties of molecular structures using first-principles approaches [5, 8, 7, 4, 6]. Molecular electronics has received tremendous attention recently as an approach to further miniaturization in device design [14]. Experimental confirmations of Di Ventra's theoretical predictions have recently been reported by Schön et al. (Lucent Technologies) in *Nature* [16]. This combined theoretical and experimental work has led to the realization of a molecular transistor with promise for practical molecular electronics.

The `transport` code has been parallelized for distributed memory parallel machines, using the MPI [12] library for message passing. A typical numerical experiment enabled by `transport` is the computation of the 'IV-characteristic' curve shown in Figure 1 (bottom left). Note from the pseudocode in Figure 1 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 for `transport` are 32 to 128 energy levels and 100 right hand sides per energy level. In Figure 2 we give a more detailed description of the energy level loop. This version is abstracted directly from the code and will help us in identifying where the computational bottlenecks are; it leaves out many details.

The problem parameters that drive the computational cost of a run of `transport` are the number of energy levels N_e (`nevals` in Figure 2), and the number of plane waves in each direction

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



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. [15]. Bottom left: theoretical I-V curve [8]. 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 the `transport` code. Parallelism can be exploited at loops 1, 2, and 6 and within steps 7-8.

```
for ie = 1 to nevals            /* typical value of nevals 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 solves */
      endfor
    endfor
  endfor
endfor
```

Figure 2: Important steps in the main loop over energy levels.

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

(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 below 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_{est}, an estimate of the total amount of memory required. The memory estimate is based on the following:

$$M_{\text{est}} = (6 \text{ large arrays }) * (16 \text{ bytes per double complex }) * N_{\text{mat}}^2.$$

This estimate is a lower bound in that it only includes the six largest arrays, each of which is of dimension $N_{\text{mat}} \times N_{\text{mat}}$, where $N_{\text{mat}} = (2 * N_x + 1) * (2 * N_y + 1) * (2 * N_z + 1)$. Note 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 two sections.) All of our 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. With `transport`'s current parallelization scheme, the computation associated with a single energy level must fit on a single compute node. This means that problem sizes larger than $(5, 5, 10)$ will not fit in main memory. Furthermore, scalability is limited by the current parallelization strategy since there are no more than N_e tasks that can run in parallel. With $N_e = 32$ or at most 128, this is a significant limit on scalability. In Section 4 we discuss some possibilities for using more processors, motivated both by memory constraints and parallel scalability issues.

2 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) [13, 10, 9] to achieve high performance on a variety of architectures. Since the `transport` code uses BLAS functions, an efficient implementation of these kernels makes a substantial difference in performance. We used the ATLAS [17] 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

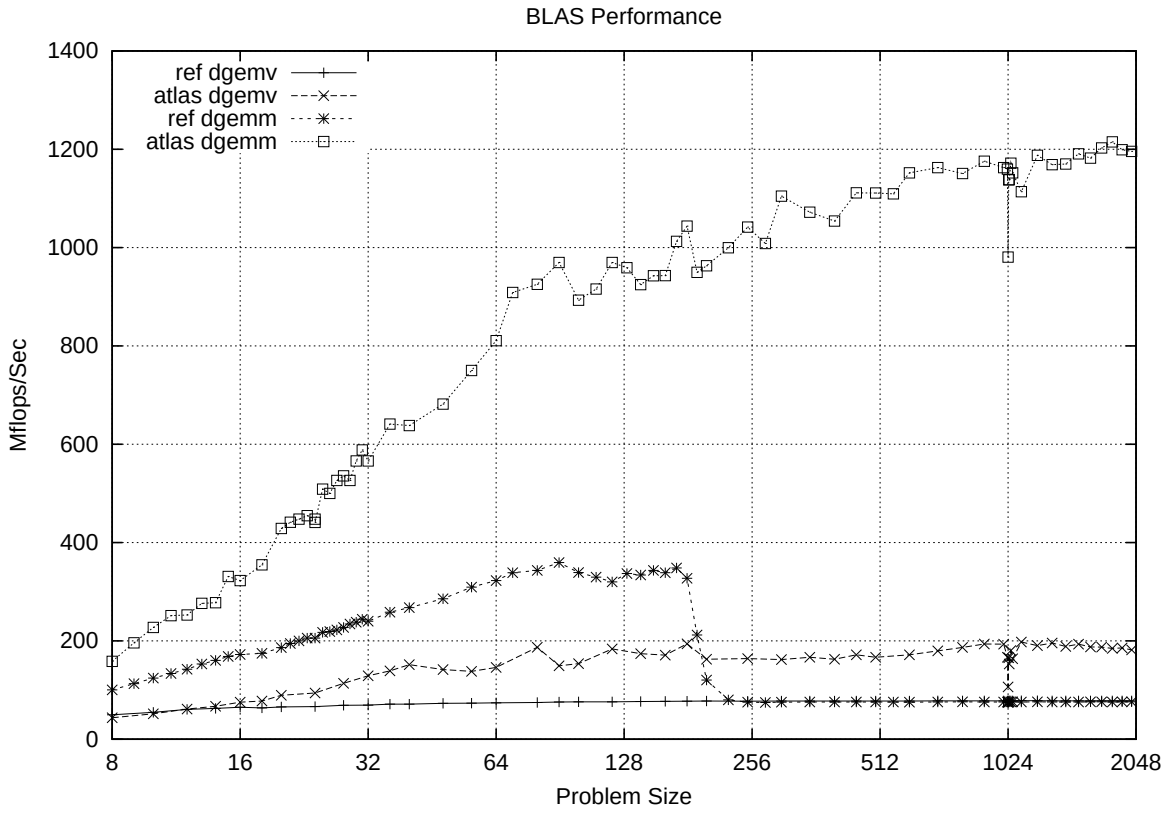


Figure 3: Performance of ATLAS and reference implementation of two BLAS kernels.

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

Before turning to profiling the sequential performance of `transport`, we summarize the performance of ATLAS BLAS compared to the reference implementation in Figure 3. Performance for two standard BLAS kernels is shown—matrix-vector multiplication (`dgemv`) and matrix-matrix multiplication (`dgemm`). The x-axis in Figure 3 gives the one-dimensional size of the square matrices used in each test. The results are consistent with well-understood properties of hierarchical memories. The advantages of block-oriented level-3 BLAS operations, which benefit from favorable 'computation-to-communication' ratio, is obvious when one compares `dgemm` performance to that of `dgemv`. We observe that both ATLAS implementations achieve relatively stable performance as the problem size grows, with `dgemm` reaching an asymptotic limit of over 1 Gflop. The only exception is a small performance problem at $n = 1024$; cache mapping conflicts are the likely cause of this glitch. We also observe that the reference implementation of `dgemm` performs reasonably well until about $n = 175$, at which point the performance degrades substantially, to the point where it is no better than `dgemv`. Again, cache management is the obvious culprit.

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

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 some number of energy levels. In the current 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 only two iterations. The table gives time (in seconds) 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), almost 4% 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 we plan to use in an attempt 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.

3 Parallel Performance

We now turn to the parallel performance of `transport`. Figure 4 shows fixed problem-size parallel speedup for problem sizes $(3, 3, 6)$ and $(4, 4, 8)$ considered. 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 report. In the statically scheduled version, each processor simply gets the same number of energy levels (or close to the same number, depending on how evenly the number of processors divides the 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 4 actually corresponds to the number of 'workers'; one additional processor was used in the role of 'master.'

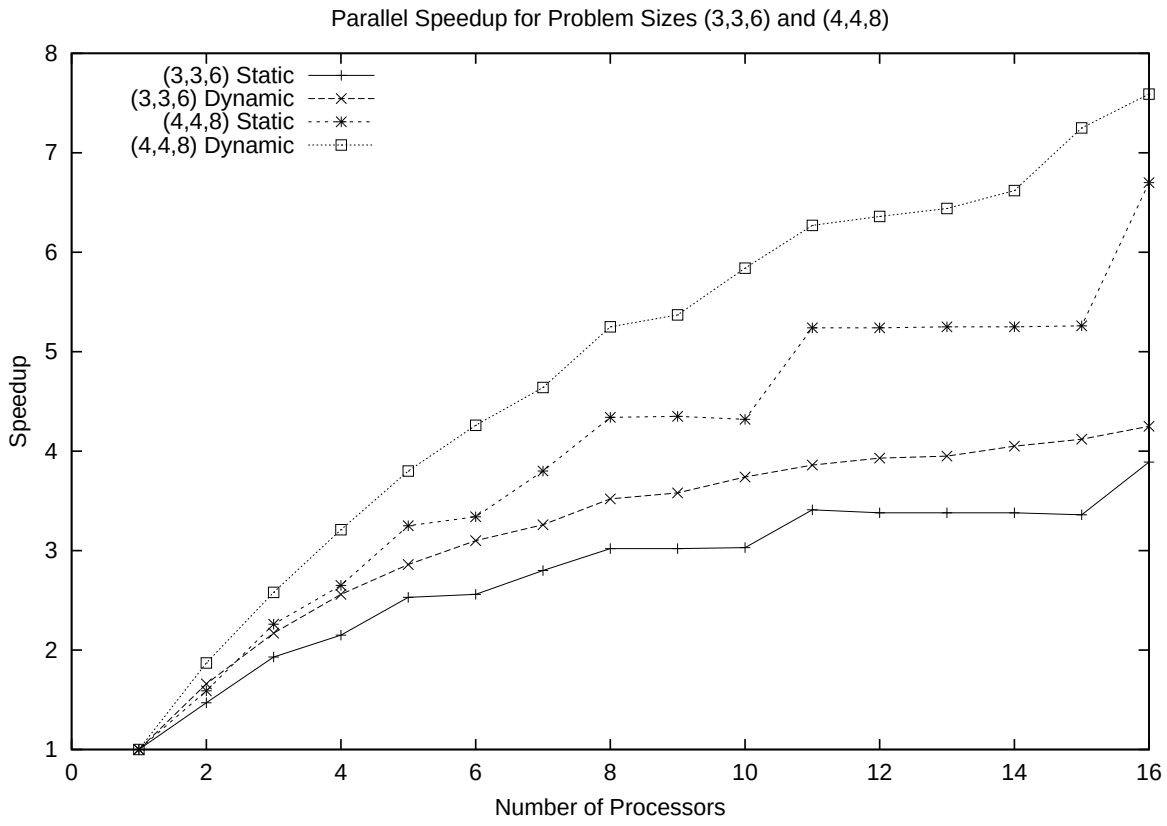


Figure 4: Parallel speedup.

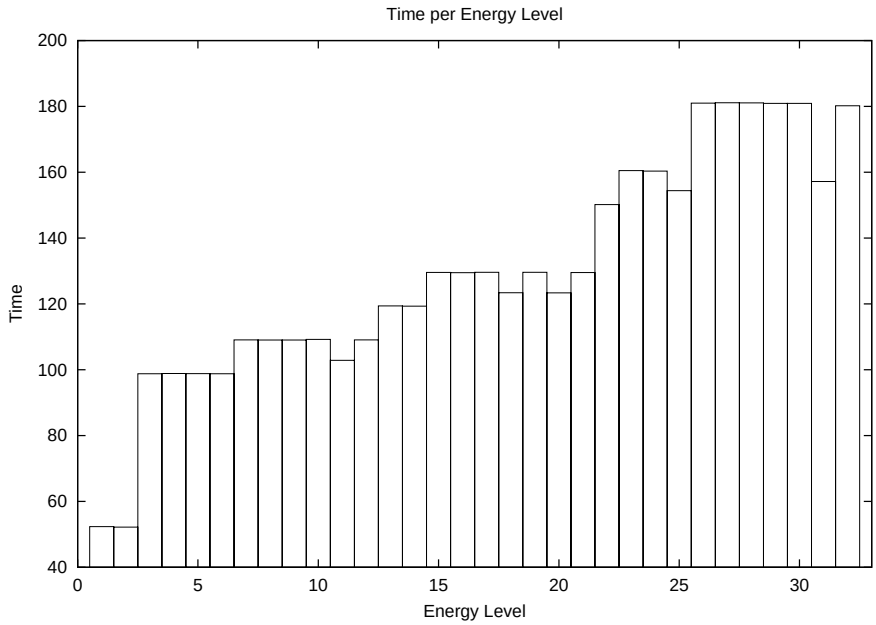


Figure 5: Time (in seconds) for various energy levels, for problem size (4,4,8), first iteration.

As can be seen from Figure 4, 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 5 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 5 shows that this case corresponds to considerable processor idle time as well. (Although not shown in Figure 4, we did run problem size $(4, 4, 8)$ with 32 worker processors, achieving a speedup of only 8.8—very little 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, Figure 6 illustrates the load balance for 8 and 12 worker processes. Clearly the 8-worker case is better; but even with 12 workers the load imbalance is less than 20%—quite acceptable considering the large granularity of the tasks being assigned. Finally, we make the obvious point 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 4 is only a speedup of 7.6 on 16 processors. Even with a larger problem size ($(5, 5, 10)$, not shown in Figure 4), 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; and this load imbalance cannot be avoided unless the parallel algorithm is changed.

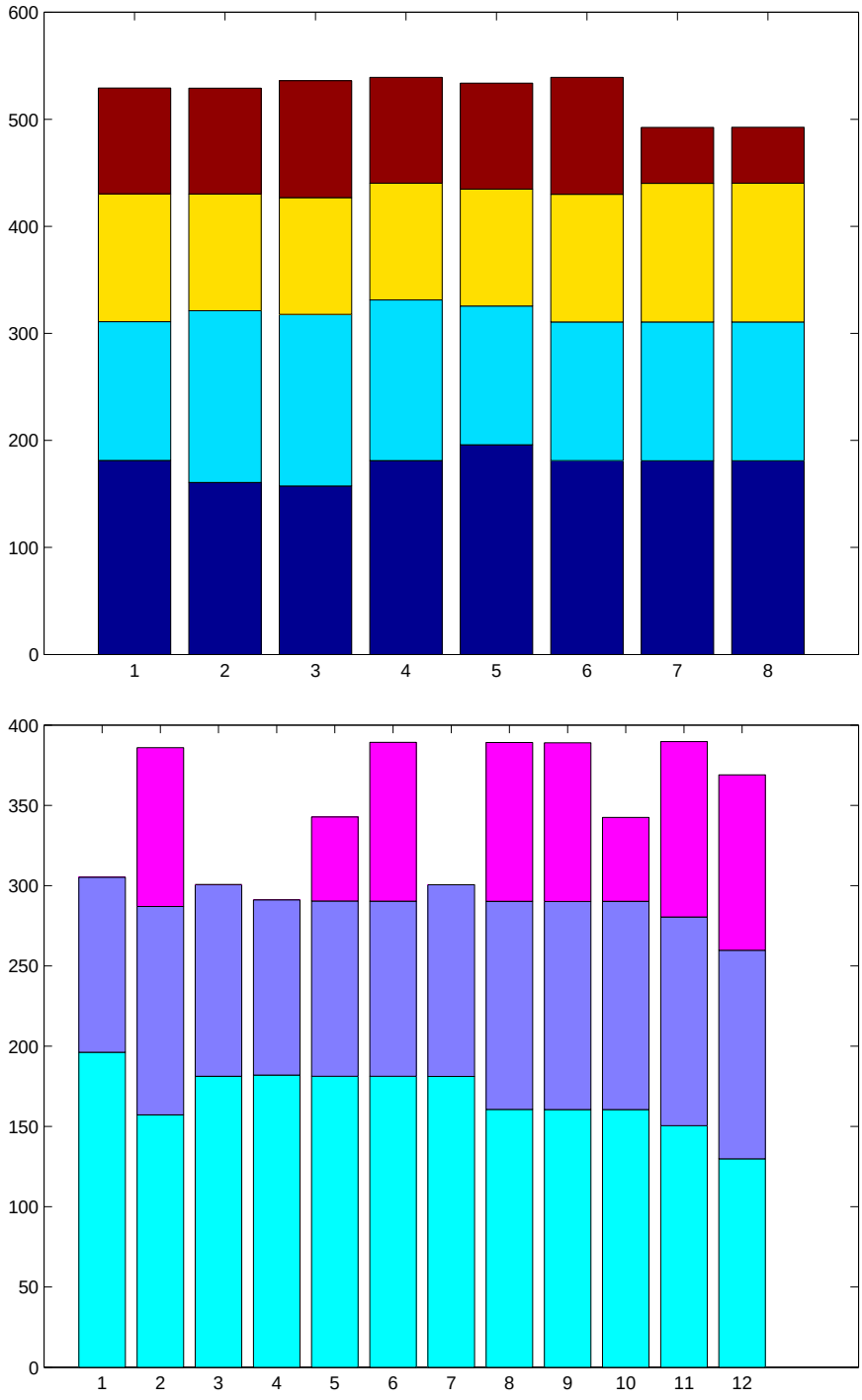


Figure 6: Total work (time in seconds) assigned to each worker node during first iteration of problem (4,4,8): 8 nodes (top), 12 nodes (bottom). Different shading represents different energy levels.

4 Future Directions

For current problem sizes the obvious next step is to attempt to parallelize the portion of the main iteration that lies outside the energy loop. We have not seriously investigated this issue yet. The only other thing that may be worth doing with respect to the current code is to investigate whether a more sophisticated dynamic scheduling strategy would be useful. Since the approximate cost of the computation associated with each energy level is known (either *a priori* or from previous iterations), one could schedule the most expensive energy levels first and the least expensive ones last, yielding a simple but nearly optimal dynamic load balancing scheme for the energy loop.

In order to achieve more parallel scalability with `transport` our next step will be to exploit parallelism at more levels. This will also be necessary in order to investigate new scientific questions, since the physicists who use `transport` need to solve larger problems, e.g., with (10,10,20) plane waves. In either case—using more processors efficiently on current problem sizes, or handling larger problem sizes—it will be necessary to exploit parallelism *within* an instance of the energy loop. This will involve distributing the largest data structures and the work associated with them across a number of processors. The obvious starting point for the 'Linear Algebra' portion of the energy loop is the distributed memory implementations of BLAS and LAPACK—PBLAS [3] and ScaLAPACK [2], respectively. However, as can be seen from Table 2, there is considerable additional work within the energy loop that will need to be parallelized. Furthermore, it is probably 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 large when spread across 100 or more processors. Instead, we will explore using 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 team sizes should probably only be big enough so that a single energy level fits in main memory when distributed across the team.

As problem sizes grow, we anticipate `transport` being an excellent motivating application for research in grid computing [11]. The master/worker paradigm is a natural one in the computational grid context, but challenging new issues will arise as we move from the comfort of a homogeneous cluster like *Anantham* to the dynamic and heterogeneous world of a grid, e.g., cluster-to-cluster MPI, fault tolerance, and code migration.

Finally, there are several algorithmic ideas that we hope to pursue. These ideas include adaptive selection of energy levels (rather than the equal-spaced energy levels currently used), initial iterations using fewer plane waves, and iterative linear solvers (with all the consequent opportunities for preconditioning strategies).

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. 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[10] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

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

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, Cambridge, MA, second edition, 1999.

[13] 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.

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

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

[16] J.H. Schön, H. Meng, and Z. Bao. Self-assembled monolayer organic field-effect transistors. *Nature*, 413:713–715, 2001.

[17] 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.