Performance and Programmability Comparison Between OpenMP and MPI Implementations of a Molecular Modeling Application *

Russell Brown[1] and Ilya Sharapov[2]

[1] Sun Microsystems, Inc., 15 Network Circle, Menlo Park, CA 94025, USA,
russ.brown@sun.com
[2] Sun Microsystems, Inc., 16 Network Circle, Menlo Park, CA 94025, USA,
ilya.sharapov@sun.com

Abstract. Important components of molecular modeling applications are estimation and minimization of the internal energy of a molecule. For macromolecules such as proteins and amino acids, energy estimation is performed using empirical equations known as force fields. Over the past several decades, much effort has been directed towards improving the accuracy of these equations, and the resulting increased accuracy has come at the expense of greater computational complexity. For example, the interactions between a protein and surrounding water molecules have been modeled with improved accuracy using the generalized Born solvation model, which increases the computational complexity to $O\left(n^3\right)$. Fortunately, many force-field calculations are amenable to parallel execution. This paper describes the steps that were required to transform the Born calculation from a serial program into a parallel program suitable for parallel execution in both the OpenMP and MPI environments. Measurements of the parallel performance on a symmetric multiprocessor reveal that the Born calculation scales well for up to 144 processors, and that programmability and performance are better for the OpenMP implementation than for the MPI implementation.

1 Introduction

Molecular modeling is one of the most demanding areas of scientific computing today. Although the high computational requirements of molecular simulation can produce long computation times, parallel execution may used to increase the size of molecules that can be analyzed in manageable time. Several software applications exist for molecular modeling and estimation of the internal energy of molecules using classical equations known as *force fields* [4]. An open-source application related to the well-known AMBER [18] software is Nucleic Acid Builder or NAB[3] [15], which we use as the basis for our analysis in this work.

* This material is based upon work supported by DARPA under Contract No. NBCH3039002.

[3] http://www.scripps.edu/case

A typical force field includes several energy terms, one of which models the interactions between a biomolecule and the solvent, or surrounding water molecules. This energy is known as the Born free energy of solvation. Using a method known as the generalized Born [3, 8, 17] approximation, the electrostatic contribution to this energy is computed as the sum of pairwise interactions:

$$E_{\texttt{Born}} = -\frac{1}{2}\sum_{i}\sum_{j>i} q_i q_j \left[1 - \frac{e^{-\kappa\sqrt{d_{ij}^2 + R_i R_j e^{\frac{-d_{ij}^2}{4R_i R_j}}}}}{\epsilon_w}\right] \tag{1}$$

In this equation, d_{ij} represents the distance between atoms i and j, ϵ_w represents the dielectric constant of water, and κ represents a Debye-Huckel screening constant [16]. R_i represents the *effective Born radius* that is a measure of the amount by which the atom i is screened from the solvent by all of the surrounding atoms k. The effective Born radius is calculated as:

$$R_i^{-1} = \frac{1}{\rho_i} + \sum_{k\neq i} f\left(d_{ik}, \rho_i, \rho_k\right) \tag{2}$$

In this equation, ρ_i and ρ_k represent the intrinsic radii of atoms i and k, and $f\left(\right)$ is a smooth function of the interatomic distance and the intrinsic radii [11].

Because of the presence of R_i and R_j in equation (1), computation of the Born free energy and its first and second derivatives can involve considerable complexity. However, it is possible to reduce this computational complexity via precomputation, the details of which are beyond the scope of this paper and are reported elsewhere [6]. This precomputation produces two vectors, R and A, each of length n (where n is the number of atoms). These vectors are used to produce four matrices: $\mathbf{N}$ (of size n by n) and $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{D}$ (each of size $3n$ by n). Products of these matrices are used to form the Hessian matrix $\mathbf{H}$ (of size $3n$ by $3n$) that contains the second derivatives. This precomputation reduces the computational complexity of the Born free energy and its first derivatives to $O\left(n^2\right)$, as well as reducing the complexity of the second derivatives to $O\left(n^3\right)$.

For molecular modeling applications, it is desirable to estimate the internal energy of a molecule and to minimize that energy. Minimization may be performed using the iterative Newton-Raphson method [4] that is calculated as:

$$x_1 = x_0 - \mathbf{H^{-1}}\left(\mathbf{x_0}\right)\nabla\mathbf{E}\left(\mathbf{x_0}\right) \tag{3}$$

In the above equation, x_0 represents the initial Cartesian coordinates of the atomic nuclei prior to a step of Newton-Raphson iteration, x_1 represents the Cartesian coordinates after the step of Newton-Raphson iteration, $\nabla E\left(x_0\right)$ represents the gradient vector of first derivatives that are calculated from the initial Cartesian coordinates, and $\mathbf{H^{-1}}\left(\mathbf{x_0}\right)$ represents the inverse of the Hessian matrix of second derivatives that are calculated from the initial Cartesian coordinates. In practice, inversion of the Hessian matrix is avoided by solving the linear system via techniques such as Cholesky factorization.

2 Implementation

As shown in Figure 1, each iteration of Newton-Raphson minimization is subdivided into four phases of computation: (1) calculation of the non-Born energy terms and their derivatives; (2) calculation of the Born free energy, its first derivatives, and the matrices **N**, **F**, **G** and **D**; (3) matrix multiplication to produce the Hessian matrix **H** of second derivatives; and (4) solution of the linear system via Cholesky factorization.

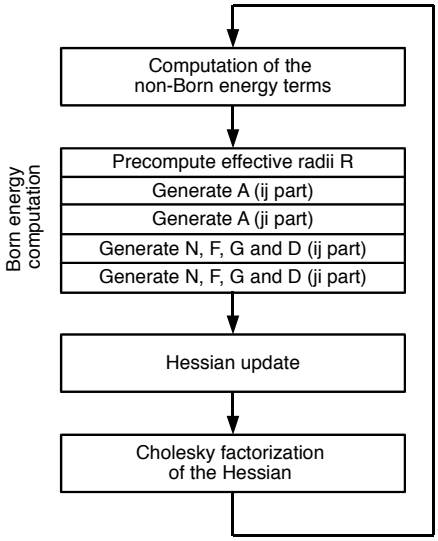

Fig. 1. Phases 1-4 of Newton-Raphson. The ij and ji parts are computed separately.

We can visualize the phases of an iteration by monitoring the changes in low-level activity in the system. Figure 2 gives an example of the low-level activity that illustrates the change in the *cycle per instruction* (CPI) measurements for different phases of execution of the MPI version of NAB. The CPI measures the efficiency of the CPU; low CPI values indicate good performance. Superscalar processors are capable of executing multiple instructions in one cycle and therefore CPI values can be less than one for carefully tuned sections of code. In our experiments we measured the CPI and other low-level statistics, such as cache miss rates, using UltraSPARC® processor on-chip hardware counters [9].

In Phase 1 we observe fairly poor processor efficiency, which can be explained by non-contiguous memory references that occur in the computation of the non-Born energy terms. Relatively little computation is performed during this phase, which doesn't significantly impact the overall performance. In Phase 2 we can see five distinct regions corresponding to the five Born energy computations

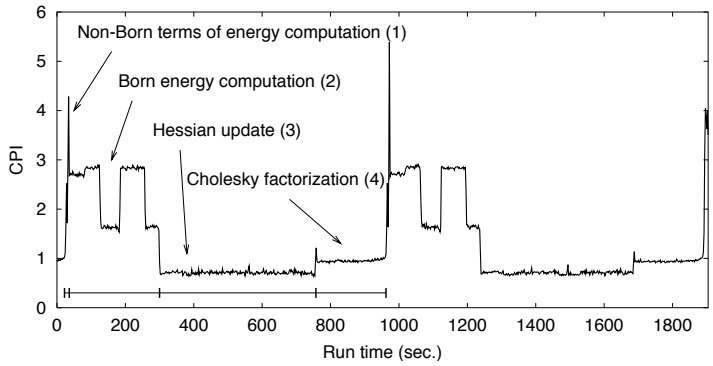

Fig. 2. CPI for Phases 1-4 of Newton-Raphson minimization, using 5-way parallelism for Phase 2 and 25-way parallelism for Phases 3 and 4.

outlined in Figure 1. Phase 3 updates the Hessian matrix via three matrix-matrix multiplications. Phase 4 performs Cholesky factorization. Phases 3 and 4 are executed using subroutines from the ScaLAPACK[4] scientific library [5].

The computation in Phase 1 has $O(n)$ complexity and contributes only minimally to the total computation. The computational in Phases 3 and 4 has $O(n^3)$ complexity and is performed by parallelized subroutines from a scientific library. The computation in Phase 2 exhibits $O(n^2)$ complexity and must be parallelized in order that the total computation achieve reasonable scalability [1]. We have parallelized the Phase 2 computation via an OpenMP[5] [7] implementation, as well as via an MPI[6] [10] implementation. Prior studies have reported comparisons between OpenMP and MPI versions of applications software [2, 12–14].

2.1 OpenMP

The computation of Phase 2 of Newton-Raphson minimization involves several summations such as $\sum_i \sum_{j>i}$ or $\sum_i \sum_{j \neq i}$, each of which implies a loop nest having i as the outer loop index, and j as the inner loop index. Each loop nest updates a vector and a matrix, as shown for the shared vector A and the shared matrix $\mathbf{N}$ in C code fragment (4) that is parallelized via OpenMP by adding a `#pragma omp parallel for` directive to the serial code.

Code fragment (4) exhibits a race condition for the update of $A[j]$ and $\mathbf{N}[j][j]$. Because the j loop index is not partitioned amongst the OpenMP threads, all of the threads can potentially update $A[j]$ and $\mathbf{N}[j][j]$. There is no guarantee that these updates will occur atomically, and therefore the threads may overwrite one another's updates. This race condition is removed by splitting the loop nest to

[4] http://www.netlib.org/scalapack
[5] http://www.openmp.org
[6] http://www.mpi-forum.org

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++) {
  for (j = i+1; j < n; j++) {
     A[i] += f1(i, j);
     A[j] += f2(i, j); // Incorrect!
     N[i][i] += f3(i, j);
     N[i][j] += f4(i, j);
     N[j][j] += f5(i, j); // Incorrect!
     N[j][i] += f6(i, j);
  }
}
```
(4)

create two loop nests. The first loop nest uses i as the outer index, and j as the inner index. It updates $A[i]$, $\mathbf{N}[i][i]$ and $\mathbf{N}[i][j]$ as shown in code fragment (5).

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++) {
  for (j = i+1; j < n; j++) {
     A[i] += f1(i, j);
     N[i][i] += f3(i, j);
     N[i][j] += f4(i, j);
  }
}
```
(5)

The second loop nest uses j as the outer index, and i as the inner index. It updates $A[j]$, $\mathbf{N}[j][j]$ and $\mathbf{N}[j][i]$ as shown in code fragment (6).

```
#pragma omp parallel for private(i)
for (j = 0; j < n; j++) {
  for (i = 0; i < j; i++) {
     A[j] += f2(i, j);
     N[j][j] += f5(i, j);
     N[j][i] += f6(i, j);
 }
}
```
(6)

For either loop nest, a given value of i is combined with the same values of j. However, a race condition will exist unless $A[i]$ and $\mathbf{N}[i][i]$ are updated in the first loop nest, and unless $A[j]$ and $\mathbf{N}[j][j]$ are updated in the second loop nest. In contrast, $\mathbf{N}[i][j]$ and $\mathbf{N}[j][i]$ may be updated in either loop nest. No race condition can exist for these updates because each update involves both i and j indices that guarantee partitioning of the matrix elements amongst the threads.

However, by updating $\mathbf{N}[i][i]$ and $\mathbf{N}[i][j]$ in the first loop nest, and by updating $\mathbf{N}[j][j]$ and $\mathbf{N}[j][i]$ in the second loop nest, the matrix elements are partitioned amongst the OpenMP threads by matrix row. Groups of r contiguous rows may be partitioned amongst the threads by adding a `schedule(static, r)` clause to the `#pragma omp parallel for` directive. This partitioning is known as *row cyclic* partitioning, and promotes locality of memory access by each OpenMP thread for a matrix that is allocated in row-major order.

Instead of using the `schedule(static, r)` clause, we map the OpenMP threads to matrix rows explicitly, as shown for MPI in code fragment (7), in order to maintain compatibility between the OpenMP and MPI implementations of NAB. This approach is an example of the SPMD OpenMP programming style that can improve the performance of OpenMP applications when used in conjunction with code tuning techniques [13]. A performance analysis tool such as the Sun™ Performance Analyzer[7] is essential to the code tuning process.

2.2 MPI

The parallelization of Phases 2 to 4 of Newton-Raphson minimization is more complex for MPI than for OpenMP. The increased complexity is due principally to the ScaLAPACK library that is used with the MPI implementation of NAB. The ScaLAPACK library does not support global, shared vectors and matrices; instead, it supports vectors and matrices that are distributed across all of the MPI processes. Under this distributed paradigm, each process has exclusive access to a unique subset of the global vector or matrix elements, which we will call the *sub-vector* or *sub-matrix*. Each process initializes its sub-vector or sub-matrix, and then the ScaLAPACK subroutines distribute computation such as matrix multiplication and Cholesky factorization across all of the processes.

Before a vector or matrix can be processed by a ScaLAPACK subroutine, it must be distributed onto a *process grid*. The process grid is a group of MPI processes that are placed on a rectangular grid of `nprow` rows by `npcol` columns. Each process has unique row and column coordinates `myrow` and `mycol` that indicate the location of the process on the grid. The matrix elements are mapped onto the process grid in a *block cyclic* manner [5] wherein a matrix of m rows by n columns is subdivided into blocks of `mb` rows by `nb` columns of contiguous matrix elements. Block cyclic mapping is the two-dimensional analog of the one-dimensional row cyclic mapping that was described for OpenMP, and is used by ScaLAPACK to achieve reasonable load balancing across the MPI processes.

Each MPI process has exclusive access to a unique sub-vector or sub-matrix. Thus, the i and j loop indices of a loop nest must be restricted to only those values that are required to update the accessible sub-vector or sub-matrix elements. The loop nest must be split because a process that can access $\mathbf{N}[i][j]$ cannot necessarily access $\mathbf{N}[j][i]$. Code fragment (7) satisfies these constraints and accomplishes block cyclic partitioning of the matrix elements amongst the

[7] http://developers.sun.com/sunstudio

MPI processes. Each process asserts a unique subset of the matrix i and j indices, which is selected by the `myrow` and `mycol` coordinates of the process.

```
for (i = 0; i < n; i++) {
  if ( (i/mb)%nprow != myrow ) continue;
  for (j = i+1; j < n; j++) {
     if ( (j/nb)%npcol != mycol ) continue;
     A[i] += f1(i, j); // Incorrect!
     N[i][i] += f3(i, j); // Incorrect!
     N[i][j] += f4(i, j);
  }
}

for (j = 0; j < n; j++) {
  if ( (j/mb)%nprow != myrow ) continue;
  for (i = 0; i < j; i++) {
     if ( (i/nb)%npcol != mycol ) continue;
     A[j] += f2(i, j); // Incorrect!
     N[j][j] += f5(i, j); // Incorrect!
     N[j][i] += f6(i, j);
  }
}
```

(7)

Code fragment (7) is incorrect for several reasons. First, although the first loop nest can access $\mathbf{N}[i][j]$, it cannot necessarily access $\mathbf{N}[i][i]$. Similarly, although the second loop nest can access $\mathbf{N}[j][i]$, it cannot necessarily access $\mathbf{N}[j][j]$. These access restrictions arise because the diagonal elements of the matrix $\mathbf{N}$ are not guaranteed to belong to the process that calculates updates to those elements. A solution to this problem is for each process to maintain a private copy of the diagonal elements. When all of the processes have finished updating their private copies of the diagonal elements, these private copies are combined and the result is rebroadcast to each process. The `MPI_Allreduce` function is used for this combine and rebroadcast operation.

The second problem with code fragment (7) arises due to a ScaLAPACK convention that requires that a distributed vector such as the vector A exist only in column zero of the process grid. Hence, only a process that exists in column zero of the grid possesses a sub-vector of the vector A. A particular process that calculates updates to the vector A may not lie in column zero of the grid and therefore may not be able to access the elements of the sub-vector that it needs to update. This problem is very similar to the first problem discussed above, and it has a similar solution. Each process must maintain a private copy of the entire vector A. When all of the processes have finished updating their private copies of the vector A, these private copies are combined and the result is rebroadcast to each process via the `MPI_Allreduce` function.

The third problem with code fragment (7) is that the matrix elements $\mathbf{N}[i][j]$ and $\mathbf{N}[j][i]$ are not accessed as elements of the global matrix $\mathbf{N}$, but rather as elements of a sub-matrix that is owned by a particular process. The sub-matrix is not addressed using the global $[i][j]$ or $[j][i]$ address directly. Instead, the global address is converted to an offset into the sub-matrix. This global-to-local address mapping involves four integer divisions and four modulus operations for access to *each* element of the matrix. Fortunately, the divisors are `mb`, `nb`, `nprow` and `npcol`, which are constants for a given matrix, and the dividends are i and j whose values lie in the range $0 \leq i < n$. These features of the divisors and dividends permit precomputation of all of the division and modulus operations. The results of the precomputation are stored in eight lookup tables, each of length n. Because a matrix requires $O\left(n^2\right)$ memory, these tables represent only a small fraction of the size of a typical matrix, and hence offer the possibility of accelerated computation at the expense of minimal additional storage.

3 Programmability

Creating OpenMP and MPI versions of the same application using different approaches to parallelization allowed us to compare the effort required to implement both versions. Parallelization of Phase 2 required the splitting of nested loops for both versions. Once this splitting was completed, the OpenMP version was straightforward. However, creating the MPI version required substantial additional effort (required by ScaLAPACK) to map global matrices onto a two-dimensional process grid and to modify one-dimensional row cyclic partitioning to obtain two-dimensional block cyclic partitioning of the matrix elements.

We have estimated the relative complexity of the two versions of NAB by counting the non-comment source code lines that are related to the Newton-Raphson minimization and to the calculation of the Born energy and its derivatives. Three categories of source code lines were counted: (1) source code lines that are required for serial execution, (2) source code lines that are required to modify the serial code for parallel execution by OpenMP, and (3) source code lines that are required to modify the serial code for parallel execution by MPI. The serial line count is 1643. The OpenMP line count is 180. The MPI line count is 962. These line counts reveal that adaptation of the serial code for MPI produced significantly (*i.e.*, a factor of five) more source code than adaptation of the serial code for OpenMP. This finding suggests that the programmer's productivity may be higher when an application that relies on linear algebra is parallelized using OpenMP instead of MPI.

4 Performance and Scalability

In order to obtain an accurate comparison between the OpenMP and MPI implementations of the code, we performed all of the measurements using the same server: a Sun Fire™ E25K server with 72 dual-core UltraSPARC IV processors. We have compared the performance of the two implementations for up to 144

OpenMP threads and MPI processes using the 1AKD, 1AFS and 1AMO molecular models from the RCSB Protein Data Bank[8] that comprise 6,370, 10,350 and 19,030 atoms (including hydrogen atoms), respectively.

We start by comparing different versions of the MPI implementation of NAB. Figure 2 (see the Implementation section) shows the profile of our baseline MPI implementation which does not use the table lookup optimization for address mapping. The division and modulus operations that are used in address mapping between the global matrix and sub-matrices are relatively inefficient and inhibit pipelined execution, thereby prolonging Phase 2. Moreover, for this version the parallelization of Phase 2 is implemented in a row cyclic manner similar to the approach that was discussed for OpenMP. For a given row of the process grid, all of the columns perform redundant computation. Therefore, when this baseline implementation executes using 25 processes on a 5 by 5 process grid, the degree of parallelization is limited to 5. The Phase 2 execution time is 269 seconds.

Figure 3 shows the execution profile for the improved version of the code in which the address mapping is implemented with lookup tables. Because these tables are small they are stored entirely within the processor caches, which facilitate table lookup and reduce the Phase 2 execution time to 105 seconds.

Figure 4 shows the execution profile for full parallelization of Phase 2. Full parallelization is achieved by having all process columns of a particular process row perform unique (instead of redundant) computation, thus increasing the degree of parallelization from 5 to 25 for a 5 by 5 process grid. The execution time for Phase 2 is 20 seconds, and thus this phase is no longer a barrier to the scalabiliity of the overall computation, as required by Amdahl's law [1].

Figure 5 shows the execution profile for the OpenMP implementation of NAB that fully parallelizes Phase 2 of the Newton-Raphson minimization. The execution time for Phase 2 is 10 seconds, compared to 20 seconds for the fastest MPI implementation. This improvement can be attributed to the direct access to global matrix elements by OpenMP, which avoids the address mapping required by ScaLAPACK for distributed sub-matrix access in the MPI implementation.

In addition to measuring the relative speeds of the OpenMP and MPI implementations of NAB, we have examined the relative scalabilities of these implementations. Figure 6 shows that Phase 2 scales better for MPI than for OpenMP. This observation has been reported for other parallel applications [12]. In an attempt to understand the disparity between the scalabilities of the OpenMP and MPI implementations of NAB, we used the Sun Performance Analyzer to measure the level 2 cache miss rates during Phase 2 of the computation. We obtained these measurements for the 1AMO model and 4, 9, 16, 25 and 36 OpenMP threads or MPI processes executed on a Sun Fire E6900 server with 24 dual-core UltraSPARC IV processors. The cache miss rates per instruction were 0.0034 for OpenMP and 0.0012 for MPI, measured for between 4 and 36 OpenMP threads or MPI processes. Hence, relative to MPI we observed a nearly three-fold increase in the cache miss rate for OpenMP.

[8] `http://www.rcsb.org/pdb`

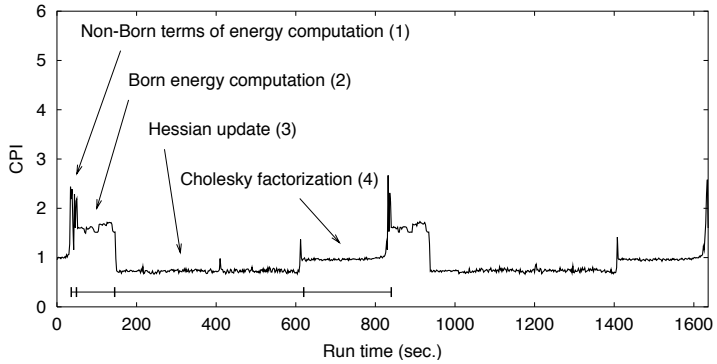

Fig. 3. CPI for MPI implementation with table lookup optimization, and using 5-way parallelism for Phase 2 and 25-way parallelization for Phases 3-4.

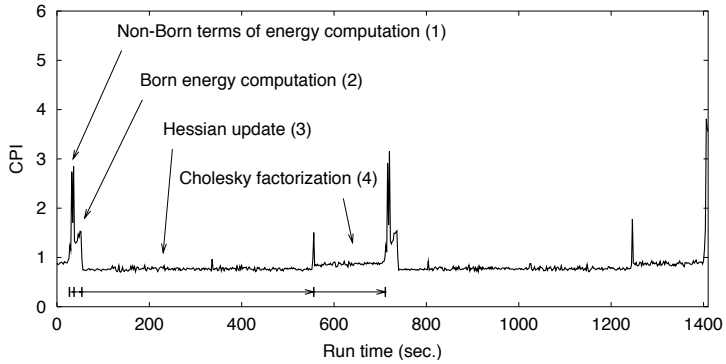

Fig. 4. CPI for the MPI implementation using 25-way parallelism for Phases 2-4.

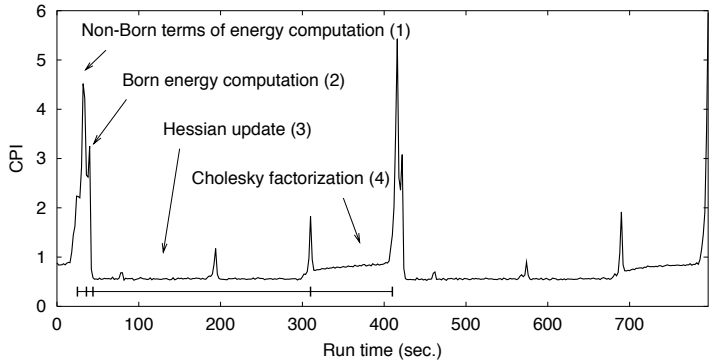

Fig. 5. CPI for the OpenMP implementation using 25-way parallelism for Phases 2-4.

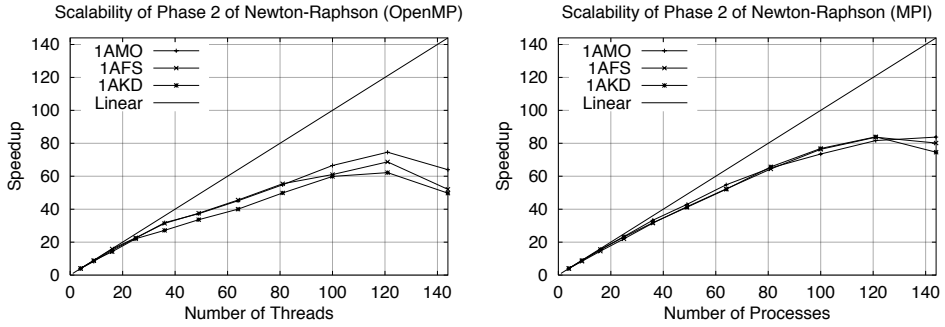

Fig. 6. Scalability of Phase 2 of Newton-Raphson for the 1AKD, 1AFS and 1AMO models, using a 144-core Sun Fire E25K server. For both plots, the scalability at 144 OpenMP threads or MPI processes increases in the order 1AKD, 1AFS and 1AMO. The "Linear" plot represents perfect scalability.

5 Conclusions

This article is a case study of parallelizing a molecular modeling application. We have demonstrated that energy minimization computations can be implemented in a highly-scalable way and can utilize up to 144 processors efficiently. In our experiments, the programmability and performance of the OpenMP version were superior to those of the MPI version, but the scalability of the MPI version was superior to that of the OpenMP version. Two other studies [2, 14] have found the performance of OpenMP to be inferior to, or at best equal to that of MPI. One other study has found that the SPMD OpenMP programming style combined with careful code tuning can result in better performance for OpenMP [13]. In our experiments, it is likely that OpenMP performs better than MPI not only due to our use of the SPMD OpenMP programming style, but also due to the array indexing overhead imposed by ScaLAPACK in the MPI version.

We observe that the MPI version can execute on clusters of processors, in addition to symmetric multiprocessors. Clusters may have better nominal price/performance characteristics than large symmetric multiprocessors, although typically clusters have slower interconnects. A performance comparison between symmetric multiprocessors and clusters is beyond the scope of this work.

6 Acknowledgments

We thank David Case, Guy Delamarter, Gabriele Jost, Daryl Madura, Eugene Loh and Ruud van der Pas for helpful comments.

7 Trademark Legend

References

1. G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Reston, VA, 1967. AFIPS Press.
2. B. Armstrong, S. W. Kim, and R. Eigenmann. Quantifying differences between OpenMP and MPI using a large-scale application suite. In *LNCS 1940: ISHPC International Workshop on OpenMP: Experiences and Implementations (WOMPEI 2000)*, Berlin Heidelberg, 2000. Springer-Verlag.
3. D. Bashford and D. Case. Generalized born models of macromolecular solvation effects. *Ann. Rev. Phys. Chem*, 51:129, 2000.
4. U. Berkert and N. Allinger. Molecular mechanics. ACS Monograph 177, American Chemical Society, 1982.
5. L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongara, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C.Whaley. *Scalapack Users' Guide*. Society for Industrial and Applied Math, 1977.
6. R. Brown and D. Case. Second derivatives in generalized born theory. *J. Comp. Chem*, 2006. Accepted for publication.
7. R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.
8. M. Feig, W. Im, and C. Brooks. Implicit solvation based on generalized born theory in different dielectric environments. *J. Chem. Phys.*, 120:903–911, 2004.
9. R. P. Garg and I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall, PTR, 2001.
10. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed.* MIT Press, Cambridge, MA, 1999.
11. G. Hawkins, C. Cramer, and D. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *J. Phys. Chem*, 100:19824–19839, 1996.
12. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of the NAS parallel benchmarks and its performance. NASA Ames Research Center, editor, *Technical Report NAS-99-01*, 1999.
13. G. Krawezik and F. Cappello. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms*, pages 118–127, New York, NY, 2003. ACM.
14. G. R. Luecke and W. H. Lin. Scalability and performance of OpenMP and MPI on a 128-processor SGI Origin 2000. *Concurrency and Computation: Practice and Experience*, 13(10):905–928, 2001.
15. T. Macke. *NAB, a language for molecular manipulation*. PhD thesis, the Scripps Research Institute, 1996.
16. J. Srinivasan, M. Trevathan, P. Beroza, and D. Case. Application of a pairwise generalized born model to proteins and nucleic acids: inclusion of salt effects. *Theor. Chem. Acc.*, 101:426–434, 1999.
17. W. Still, A. Tempczyk, R. Hawley, and T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc*, 112:6127–6129, 1990.
18. P. Weiner and P. Kollman. AMBER: Assisted model building with energy refinement. a general program for modeling molecules and their interactions. *J. Comp. Chem*, 2:287–303, 1981.