Optimizing AMBER for the CRAY T3E

Robert S. Sinkovits and Jerry P. Greenberg, San Diego Supercomputer Center, San Diego, CA USA

ABSTRACT: *AMBER is a widely used suite of programs employed to study the dynamics of biomolecules and is the single most used application on the SDSC CRAY T3E. In this paper, we describe the cache, intrinsic function, and other optimizations that were taken to tune the molecular dynamics module of AMBER, resulting in single processor speedups of more than 70% for pairlist and 25% for particle mesh Ewald calculations, respectively.*

An introduction to AMBER

AMBER [1] is the collective name given to a suite of programs that are used to carry out molecular dynamics simulations, primarily on biomolecules. Unlike chemistry packages that involve the application of quantum mechanics, AMBER employs purely classical techniques. Of the programs that comprise AMBER, the greatest interest and efforts are focused on the SANDER module which is responsible for carrying out the actual molecular dynamics simulations and accounts for the vast majority of the CPU usage attributed to AMBER.

The basic concept of molecular dynamics is straightforward. At each timestep of the simulation, the forces acting on each atom due to all other atoms are calculated and used to update the atoms' positions and velocities. In a naïve approach to molecular dynamics, the number of computations would scale as $O(N^2)$ since each atom would have to interact with N-1 other atoms. By taking advantage of the fact that most interatomic potentials drop off rapidly as the distance between the atoms is increased, a cutoff distance can be introduced beyond which all interactions are neglected reducing the problem from $O(N^2)$ to O(N).

Two types of molecular dynamics simulations can be performed using SANDER. In the traditional approach, pairlists (or neighbor lists) are created specifying which pairs of atoms interact. Over the course of the simulation, the positions of the atoms change and the neighbor lists are periodically reconstructed. While this approach is conceptually simple and has been used successfully in many simulations, it is hampered by the fact that it may not properly handle the effect of long-range electrostatic interactions. The other type of molecular dynamics simulation performed by SANDER involves the use of the particle mesh Ewald (PME) technique. In this approach, a charge grid is constructed based on the locations and charges of all atoms. The advantage of the PME based calculations is that they accurately account for all electrostatic interactions out to infinity essentially summing over all, periodic images of the system. This is in contrast to the pairlist-based calculations that take into account interactions with only the nearest periodic image of the system. There is concern though among some researchers that the PME calculations may introduce an artificial periodicity into the system and therefore lead to an incorrect treatment of the electrostatic forces. Throughout the remainder of this paper, we will refer to these two types of calculations simply as pairlist or PME.

Motivation for optimizing AMBER

In the spring of 1998, NPACI introduced the Strategic Application Collaborations (SAC) program. The goal of this program is to enhance the effectiveness of computational science and engineering research conducted by NPACI's leading users. In these collaborations, NPACI staff are paired with academic researchers to undertake activities ranging from code optimization to the parallelization of existing vector or serial codes. Whenever possible, the goal is to help not just the individual researchers, but to develop general solutions that could be of benefit to the entire computational science community.

To be selected for the SAC program, the researchers must be actively involved in cutting edge computational science or engineering, have large computational requirements, and be willing to actively collaborate with NPACI staff. The AMBER project meets all of these criteria. It is used by researchers in the biochemistry and pharmaceutical communities around the world and has accounted for more CPU usage than any other application on SDSC's CRAY T3E for the last several years. The current

collaboration has been most successful in that the work that we have done will be incorporated into future releases of the source code and will benefit all users of AMBER.

Hand tuning code for optimization of pairlist calculations

Identifying the computational hotspots

Before serious efforts were made to optimize the SANDER module of AMBER, the code was first profiled against the standard benchmarks using Cray's performance analysis tool (PAT). While PAT does not provide nearly the level of detail given by tools such as apprentice, it was chosen since it introduces very little overhead. We have found that simply being able to identify the percentage of time spent in each routine greatly reduced the complexity of hotspot identification. When necessary, timers were hand-coded into the most computationally intensive routines, but typically it was obvious which loops should be the focus of attention. An initial profile of the code revealed that the square root intrinsic function and the subroutines responsible for calculating the forces between non-bonded pairs of atoms (NONBON and QIKTIP), applying periodic boundary conditions (BOUND1, BOUND2, BOUND6, and BOUND7), and constructing pairlists (PSOL) accounted for the majority of the CPU time.

Optimizing the square root

A common feature of many molecular dynamics codes is that they require the calculation of the distances and inverse distances between pairs of atoms. This, in turn, requires invoking the square root intrinsic function. In many cases, the evaluation of the square root and other intrinsic functions (such as the exponential in Morse potentials) can account for a substantial fraction of the CPU usage. Two steps can be taken to optimize the square root operations in molecular dynamics codes. First, the separate inverse and square root operations can be combined so as to take advantage of the inverse square root intrinsic function. On the CRAY T3E, the inverse square root takes the same amount of time as the square root operation, essentially giving the inverse operation for free. Second, the code can be restructured to allow the compiler to replace the loop over repeated calls to the scalar version of the intrinsic with a call to the vector intrinsic function.

In the original version of the code, inverse squared particle distances are calculated in loops similar to the following:

```
do j=1,npr
   rw(j) = 1.0/(x(1,j)*x(1,j) + x(2,j)*x(2,j) + x(3,j)*x(3,j)
enddo
```

In the force vector calculations performed later in the code, the inverse particle distances are evaluated as needed. For example,

```
a = b*sqrt(rw(j))
```

Moving the square root into the loop where the inverse particle distances are calculated has the dual benefits of allowing the inverse square root function to be used and restructuring the code so that the compiler can recognize the opportunity to use the vector version of the intrinsic. The loop containing the square root in the original version of the code was sufficiently complicated that the compiler was not able to recognize the opportunity to take advantage of the vector square root function. Since both inverse and inverse squared particle distances are required in subsequent calculations, this optimization requires that an additional arithmetic operation be introduced. The savings from being able to utilize the vector inverse square root though far exceed the cost of this additional operation.

Optimization of periodic imaging

The length scales that are accessible in molecular dynamics simulations are typically many order of magnitude smaller than those found in macroscopic systems. One technique that is frequently used not only in molecular dynamics, but in many areas of computational science, is the application of periodic boundary conditions. This approach eliminates many of the problems associated with finite sized computational domains by replicating periodic images of the system. When periodic boundary conditions are used, an atom may interact with another atom that physically lies at the other end of the computational domain, but whose periodic image is very close.

The SANDER module in AMBER applies periodic imaging in the simulations, but in a less than optimal fashion. When both a finite distance cutoff for inter-atomic forces and periodic boundary conditions are applied, we can take advantage of the fact that periodic imaging is only required for those atoms that lie sufficiently close to the edge of the computational domain. This condition can be expressed as follows: if $|x\_atom - x\_boundary| > r\_cutoff + 2\ v\_max\ \delta t$, where δt is the time between restructuring of the neighbor list, and v_max is the maximum atom velocity; then periodic imaging is not required. By applying this test, a large number of arithmetic and logical operations can be avoided for atoms that lie near the center of the domain.

Neighbor list construction

One of the strong points of the AMBER suite of codes is that they are robust enough to handle a large number of different conditions involving details of the inter-atomic potentials, boundary conditions, integration schemes, statistical ensembles, etc. Unfortunately, this can result in sub-optimal code since the common cases may no longer be fast. The subroutine PSOL, which is responsible for constructing the neighbor lists, has been written in a general way so as to allow for both single and dual cutoff simulations. (In the dual cutoff simulations, the forces due to neighboring atoms that lie beyond the first cutoff, but within the second cutoff are updated less frequently.) This routine was rewritten so that separate blocks of code are executed depending on whether a single cutoff (common case) or dual cutoff simulation is being performed. By doing this, it was possible to rewrite the code for the single cutoff case so that the main computational loop could be exited early thereby saving many memory, logical, and floating-point operations.

Loop fusion

The routine QIKTIP is responsible for calculating the non-bonded interactions between pairs of water molecules. By restructuring the subroutine, it was possible to fuse the six loops over coordinates into two loops thereby maximizing data reuse.

Collection of multiple one-dimensional work arrays into single two-dimensional work array

In both routines NONBON and QIKTIP, multiple one-dimensional sections of a large real array are used for workspace. A typical declaration of these arrays is of the form REAL RW1(*), RW2(*), RW3(*). Since the space required by these work arrays is quite small and the corresponding elements of the arrays are always used together, it is convenient and makes better use of cache to declare a new multi-dimensional work array of the form RWX(3,1000). The second dimension of the array is chosen so as to avoid the possibility of cache conflicts between, for example, RWX(1,i), RWX(2,i), and RWX(3,i).

Declaration of common blocks to eliminate cache conflict possibilities

In certain routines, it is convenient to declare multiple two-dimensional work arrays. To avoid conflict between corresponding elements of the two arrays, they can be collected together into a common block so that the relative locations of the two arrays in cache can be controlled.

Elimination of array references in NVT ensemble calculations

AMBER is capable of performing calculations using both the NPT (constant number of particles, pressure and temperature) and NVT (constant number of particles, volume, and temperature) statistical ensembles. The NPT ensemble is typically used only during the equilibration stage of the simulation, while the NVT ensemble is employed in most long simulations. The NPT calculations require that a quantity known as the collision virial be calculated, which in turn require that the intermediate force vectors be saved. Since the force vectors are not necessary for the NVT calculations, they can be replaced by scalar temporary variables, thereby significantly reducing the amount of data that needs to be moved between memory and cache.

```
if (NPT_ensemble) then   ! fwx(1,j) needed in later calculations
   do j=1,npr
        . . .
         fwx(1,j) = xij(1,j)*df
         f(1,j) = f(1,j) + fwx(1,jn)
        . . .
   enddo
else                     ! fwx1 not needed for later calculations
   do j=1,npr
        . . .
       fwx1 = xij(1,j)*df
       f(1,j) = f(1,j) + fwx1
        . . .
   enddo
endif
```

Hand tuning code for optimization of PME calculations

Identifying the computational hotspots

PAT was again used to profile the PME version of the SANDER module. As expected, the inverse square root intrinsic function and the routines directly responsible for the PME calculations (SHORT_ENE, PACK_NB_LIST, FILL_CHARGE_GRID, and GRAD_SUM) accounted for the majority of the CPU time. In comparison to the pairlist calculations,

the PME routines were much more difficult to optimize since the most computationally intensive loops contain multiple levels of indirect addressing.

Optimization of bitwise operations

The current version of the PME routines makes extensive use of bitwise operations to both minimize memory requirements and reduce the amount of data that must be moved between main memory and cache. Multiple integer variables that are accessed together are packed into the low- and high-order bits of integer words, but the algorithms used to perform this packing and extraction of data are less than optimal. In the original version of the code, the variables *n* and *itran* are extracted as follows

```
itran = ishft(n,-27)
n = n - ishft(itran,27)
```

The variable *n* can be extracted much more efficiently by replacing the combination of arithmetic and bitwise operations with a single bitwise mask operation as follows

```
n = iand(n,2**27-1)
```

In a similar fashion, the inefficient statement used to pack the variable *jtran* into the high-order bits of variable *iwa*

```
iwa(lpr) = n + ishft(jtran,27)
```

can be rewritten using only bitwise operations

```
iwa(lpr) = ior(n,ishft(jtran,27))
```

Common sub-expression elimination

The Cray Fortran compiler is typically very good at performing common sub-expression elimination. In certain cases though where the complexity of the statements is too great, the compiler is unable to perform this optimization. Performing this optimization by hand as shown below resulted in faster code.

```
c = dx * (arr(3,I) + dx*arr(4,I)*third)
erfcc = arr(1,I) + dx*(arr(2,I) + c*0.5)
derfc = -(arr(2,I) + c)
```

Loop invariant optimization

This is another transformation that the compiler can perform, but only if the complexity of the code is not too great. In the following fragment, the compiler was not able to recognize the opportunity to pre-calculate the product of invariant terms (shown in bold) before entering the inner loop. Performing this by hand resulted in better performance.

```
do i3 = i3l,i3u
  . . .
  do i2 = i2l,i2u
    . . .
    do i1 = i1l,i1u
      . . .
      f1 = f1 - nfft1*term*dth(i1,ig)*th2(i2,ig)*th3(i3,ig)
      f2 = f2 - nfft2*term*th(i1,ig)*dth2(i2,ig)*th3(i3,ig)
      f1 = f1 - nfft3*term*th(i1,ig)*th2(i2,ig)*dth3(i3,ig)
      . . .
    enddo
  enddo
enddo
```

Elimination of unnecessary operations in NVT ensemble calculations

As was the case for the pairlist calculations, the PME routines are written in such a way so as to be completely general for both the NPT and NVT statistical ensembles. By writing separate blocks of code for the two cases, the common case (NVT ensemble) can be made much more efficient. The code was restructured as shown below.

```
if (NPT_ensemble) then
  - common set of calculations -
  vxx = vxx - dfx*delx
  . . .
  vzz = vzz - dfz*delz
  virial(1) = virial(1) + vxx
  . . .
  virial(6) = virial(6) + vzz
else
  - common set of calculations -
endif
```

Comparison of original and modified codes

Pairlist calculations

The following table provides timings of the ***prowat*** (plastocyanine in water) benchmark on the SDSC CRAY T3E600. Only the non-startup contribution to the time is listed since in typical long runs, the amount of time spent on initialization and finalization of the code is negligible.

Table 1: Results of prowat benchmark on SDSC CRAY T3E600. Speedups are defined as the ratio of times required for the original and hand-tuned versions of the code on the same number of processors.

Processors	time (original)	time (hand-tuned)	speedup
2	316	184	1.72
4	162	96.2	1.69
8	85.0	51.8	1.64
16	46.2	29.6	1.56
32	25.6	17.6	1.45
64	16.4	12.3	1.33

PME calculations

The following table provides timings for the standard ***dhfr*** benchmark obtained on the SDSC CRAY T3E600. As was the case for the pairlist timing results, only the non-startup contribution to the total time is listed.

Table 2: Results of dhfr benchmark on SDSC CRAY T3E600. Speedups are defined as the ratio of times required for the original and hand-tuned versions of the code on the same number of processors.

Processors	time (original)	time (hand-tuned)	speedup
2	423	332	1.27
4	223	176	1.26
8	120	96.5	1.24
16	69.5	56.0	1.24

A note on the importance of compiler options

The standard release of AMBER comes with a hierarchy of makefiles designed to allow ease of compilation across a large number of computational platforms. A general makefile located in the source code directory invokes a machine specific makefile containing macro definitions of compilers and compiler options, loaders, libraries, and preprocessor definitions used in conditional compilation. The original version of the machine specific file for the CRAY T3E specifies only the -Oscalar3 optimization flag. Adding the full complement of flags typically used to obtain highly optimal code (-Oscalar3 -Ovector3 -Ounroll2 -Opipeline2) made virtually no difference in run time for the original version of the code. In some cases, such as the protein kinase simulation

benchmarked below, the inclusion of these additional flags actually resulted in a slight degradation of performance. The effect of the compiler flags on the hand-tuned code though was quite profound. The largest gain was probably due to the fact that the higher level of vector optimization allows the compiler to substitute the vector version of the inverse square root intrinsic for the loop over calls to the scalar function. The table below illustrates the effect of the compiler options on performance for the original and hand-tuned codes.

Table 3: Effect of compiler options on hand-tuned and original code. Protein kinase simulation run on four processors of the SDSC CRAY T3E600. Original flags = -Oscalar3, new flags = -Oscalar3 -Ovector3 -Ounroll2 -Opipeline2

Case	speedup
original code / original flags	-
original code / new flags	0.97
hand-tuned code / original flags	1.48
had-tuned code / new flags	1.76

The fast vector library was linked (-lmfastv) in an attempt to further reduce run time, but had virtually no effect. A series of benchmarks run to measure the performance of the standard and fast versions of the vector intrinsics shows that there is a negligible difference between the speed of the standard and fast inverse square root function. Nonetheless, since the fast versions of certain vector intrinsics can be significantly faster than their standard counterparts, it has been suggested to the maintainers of AMBER that the fast vector library be linked so as to get best performance if additional code modifications are made in the future.

Summary

This paper has described the steps that were taken in order to optimize the SANDER module of AMBER for the CRAY T3E. Speedups of up to 72% have been realized for pairlist-based calculations and up to 27% for PME calculations. The impact of this work is significant since AMBER is not only the most heavily used application on SDSC's CRAY T3E600, but receives use worldwide in the biochemistry and pharmaceutical communities. The modifications made to the routines used in the pairlist calculations have been incorporated into a version of AMBER5 being made available to existing AMBER sites and the PME optimizations will be included in the next major AMBER release.

While many different techniques were used to improve the performance of AMBER, the single most beneficial optimization involved restructuring the code so as to take advantage of the vector inverse square root function. Unfortunately, most users are unaware of the inverse square root function, vector intrinsic routines, or the fast vector library. In an effort to remedy this situation, we have written new documentation (see http://www.npaci.edu/online/v3.9/SCAN1.html) instructing NPACI users how to:

- restructure their codes to take advantage of the vector intrinsic functions
- link the fast vector library
- specify compiler flags that allow the compiler to substitute vector intrinsics for the scalar routines
- invoke the inverse square root function
- determine from listing and assembly code files whether or not vector intrinsics have been used

Another important observation made during this project was the role of compiler options. In the case of the original code, using the entire complement of standard optimization flags either had a negligible effect or resulted in slightly poorer performance. By contrast, the choice of compiler options (in particular -Ovector3) had a tremendous effect on the performance of the hand-tuned code. NPACI documentation is being updated to convey this important information to our users.

Acknowledgements

We are grateful to Drs. Peter Kollman and Yong Duan of the University of California San Francisco for taking the time to collaborate on this project and provide us with their unreleased version of the SANDER source code. We would also like to thank Drs. Dave Case and Mike Crowley of The Scripps Research Institute both for their help with the particle mesh Ewald calculations and for taking the effort to ensure that the code modifications find their way into the next source code release.

References

[1] D.A. Case, D.A. Pearlman, J.W. Caldwell, T.E. Cheatham III, W.S. Ross, C.L. Simmerling, T.A. Darden, K.M. Merz, R.V. Stanton, A.L. Cheng, J.J. Vincent, M. Crowley, D.M. Ferguson, R.J. Radmer, G.L. Seibel, U.C. Singh, P.K. Weiner, and P.A. Kollman. 1997. AMBER 5, University of California, San Francisco. See also, http://www.amber.ucsf.edu/amber.