Optimizing AMBER for the CRAY T3E

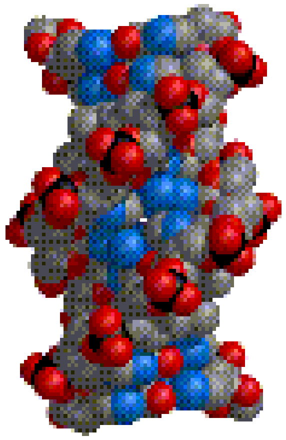
Bob Sinkovits and Jerry Greenberg
Scientific Computing Group
San Diego Supercomputer Center

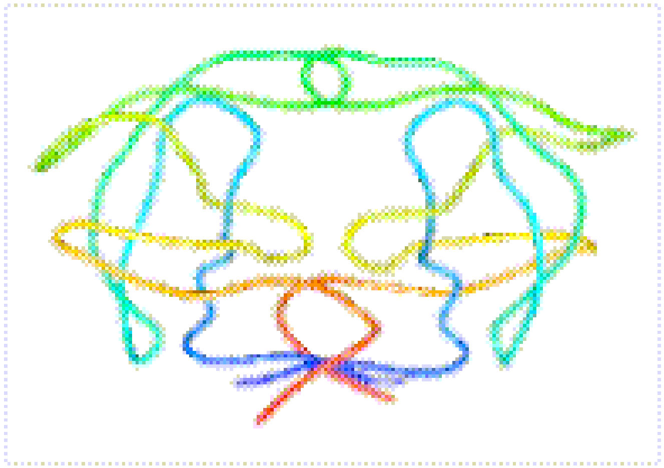
What is this presentation all about?

- What is AMBER?
- Why are we interested in AMBER?
- Brief(!) introduction to molecular dynamics
- Optimizing AMBER
    - Neighbor list based calculations
    - Particle mesh Ewald calculations
- Lessons learned / applicability to other MD calculations

What is AMBER?

"AMBER is the collective name for a suite of programs that allow users to carry out molecular dynamics simulations, particularly on biomolecules." *AMBER 5 users guide*





Why are we interested in AMBER?

- Largest user of CPU time on SDSC's CRAY T3E in both both 1997 and 1998.

- State of the art program used by researchers worldwide.
  Developed at UCSF, TSRI, Penn State, U. Minnesota, NIEHS, and Vertex Pharmaceuticals

- Recognized as an application that could benefit from serious performance tuning

Getting started

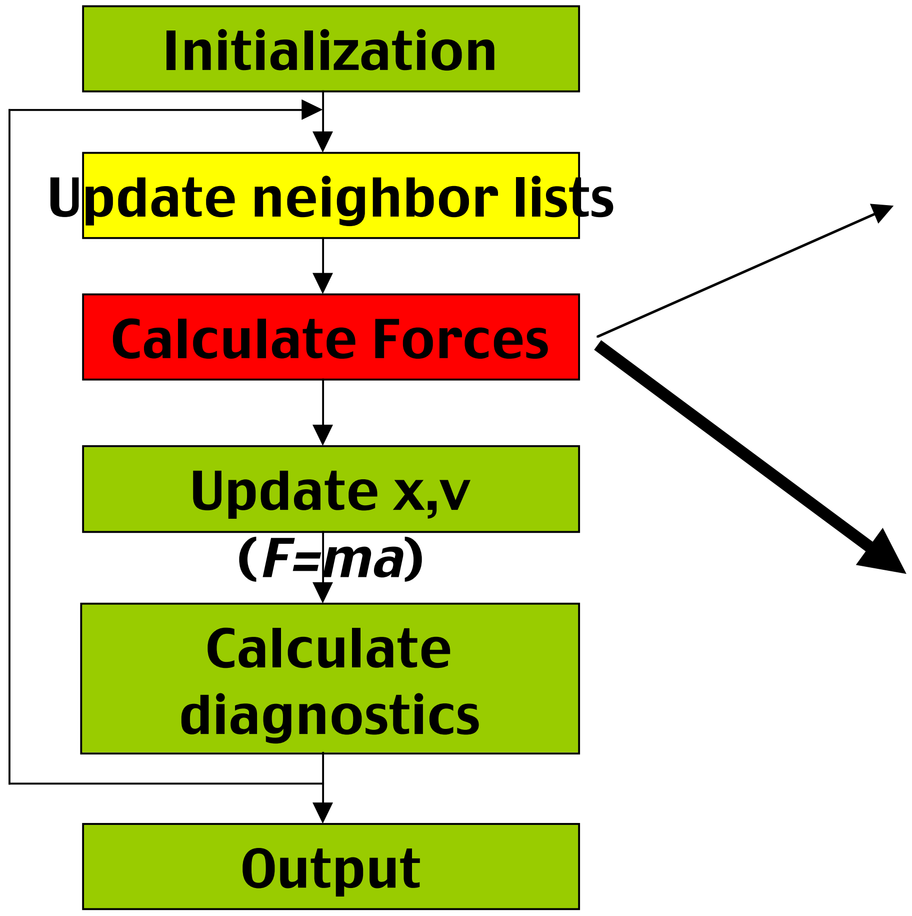
Although AMBER is a large suite of programs, the majority of the CPU usage is accounted for in the SANDER module

***S*imulated *A*nnealing with *N*MR-*D*erived *E*nergy *R*estraints**

Most simulations though have nothing to do with NMR refinement. Primarily used for energy minimization and molecular dynamics.

All optimization efforts focused on SANDER

Classical molecular dynamics in a nutshell



<u>Bonded interactions</u>

stretching: $V_{st} = SV(r_i - r_j)$

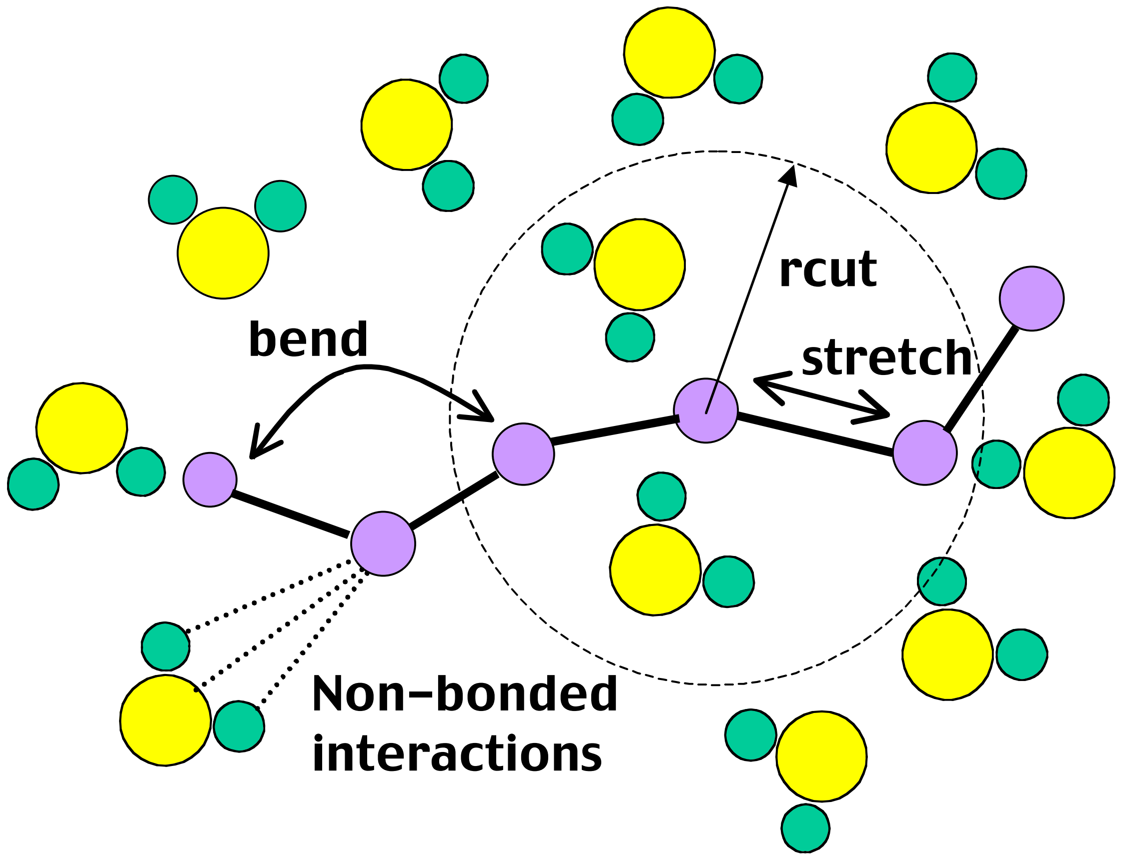
bending: $V_{bend} = SV(r_i, r_j, r_k)$

dihedral: $V_{dihed} = SV(r_i, r_j, r_k, r_l)$

<u>Non-bonded interactions</u>

van der Waals
hydrogen bonds
electrostatic

Molecular dynamics schematic



Employing a finite cutoff reduces problem from $O(N^2)$ to $O(N)$

Initial protein kinase benchmark on CRAY T3E (4 CPUs)

ROUTINE	%CPU	comments
NONBON	26	*non-bonded interactions*
QIKTIP	20	*water-water interactions*
_SQRT_CODE	13	*square root operations*
BOUND2	11	*periodic bc*
PSOL	7	*pairlist construction*
BOUND7	5	*periodic bc*
_sma_deadlock_wait	4	*parallel overhead*
barrier	2	*parallel overhead*
FASTWT	1	*startup*
RESNBA	1	*startup*

As expected, majority of time spent in routines responsible for force calculations

Optimizing square root operations

Inverse and inverse-squared interatomic distances are required in force/energy calculations. In original version of code, $1/r^2$ is calculated first and then $1/r$ is computed as needed. Doing this saves an FPM operation

Original code

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

. . .

df2 = -cgoh*sqrt(rw(jn))
r6 = rw(jn)**3
```

Optimizing square root operations (continued)

Unfortunately, original coding is slow. Computation of the inverse square root does not take any longer than simple square root – get the inverse operation for free at cost of added FPM

Original code

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

. . .

df2 = -cgoh*sqrt(rw(jn))
r6 = rw(jn)**3
```

Modified code

```
do jn=1,npr
 rw(jn) = 1.0/sqrt
            (xwij(1,jn)**2
       +    xwij(2,jn)**2
       +    xwij(3,jn)**2)
enddo

. . .

df2 = -cgoh*rw(jn)
rw(jn) = rw(jn)*rw(jn)
r6 = rw(jn)**3
```

Optimizing square root operations (continued)

By isolating inverse square root operation, get the added bonus of being able to use highly efficient vector version of function.

```
do jn=1,npr
 rw(jn) = 1.0/sqrt
            (xwij(1,jn)**2
      +     xwij(2,jn)**2
      +     xwij(3,jn)**2)
enddo
```

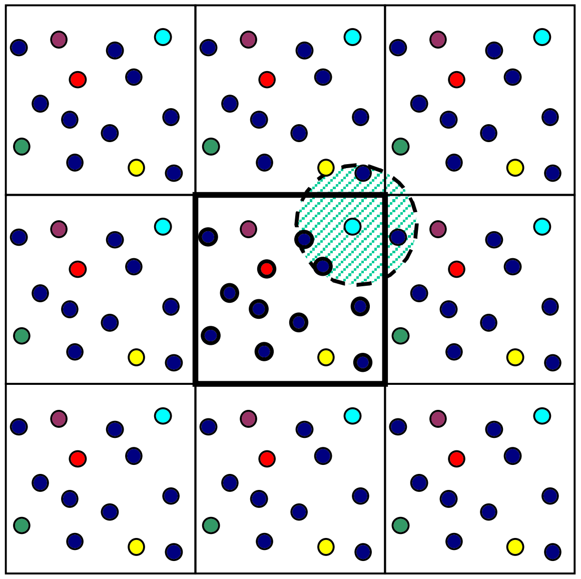
The CRAY f90 compiler automatically replaces this with a call to the vector inverse sqrt function

$\downarrow$

```
do jn=1,npr
 rw(jn) =  xwij(1,jn)**2
      + xwij(2,jn)**2
      + xwij(3,jn)**2
enddo
call vrsqrt(rw,rw,npr)
```

IBM xlf90 compiler cannot do this automatically, requires user to insert call to vrsqrt by hand

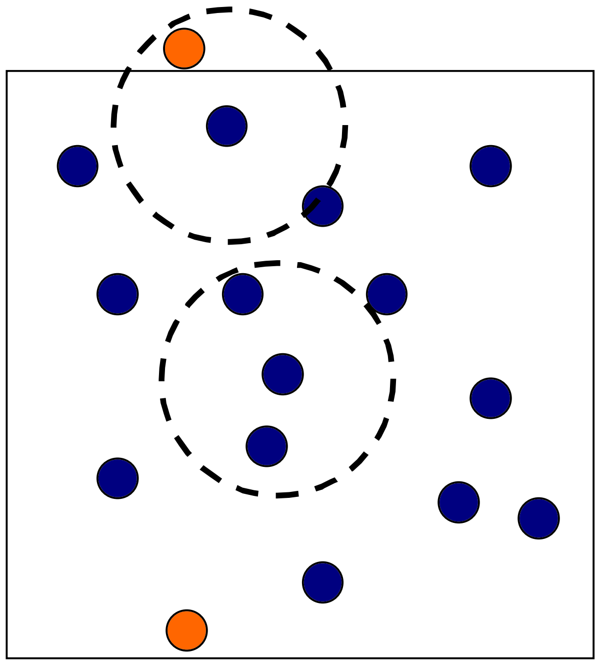
Periodic imaging



Periodic boundary conditions are commonly employed in molecular dynamics simulations to avoid problems associated with finite sized domains.

In the figure, the central square is the real system and the surrounding squares are the replicated periodic images of the system.

Optimization of periodic imaging



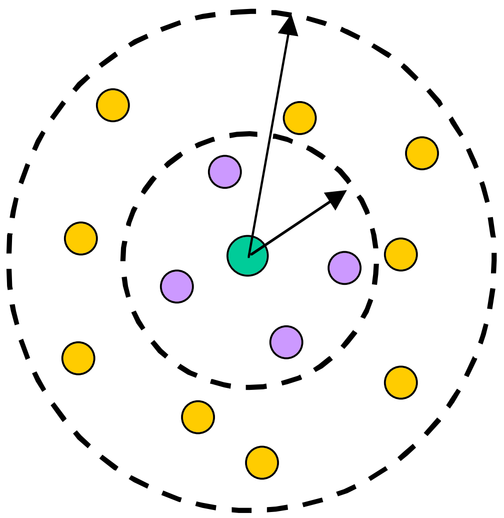
Can drastically reduce time by applying periodic imaging only to atoms that are within a distance r^{cut} of the edge of the box.

```
ibctype=0
if(abs(x(1,i)−boxh(1)).gt.boxh(1)−cutoff)
  ibctype = ibctype + 4
if(abs(y(2,i)−boxh(2)).gt.boxh(2)−cutoff)
  ibctype = ibctype + 2
if(abs(x(3,i)−boxh(3)).gt.boxh(3)−cutoff)
  ibctype = ibctype + 1

call periodic_imaging_routine(..,ibctype)
```

Optimization of neighbor list construction

The code used for neighbor list construction is written in such a way that it can handle the general case of single and dual cufoffs.



Forces between green and lavender recalculated each timestep

Forces between green and yellow recalculated every *n* timesteps

Rewriting the code so that different code blocks are called for the two cases, the common case (single cutoff) can be made very fast.

Cache optimizations in force vector evaluation

- **Loop fusion:**

  Six loops used to calculate water-water interactions in QIKTIP fused into two loops. Maximizes reuse of data.

- **Collection of 1d work arrays into single 2d-arrays:**

  RW1(*), RW2(*), RW3(*) fi **RWX(3,1000)**

- **Declaration of force vector work array:**

  FW(3,*) fi **FWX(9,1000)**

- **Creation of common block to eliminate cache conflict possibility**

  common /local_qiktip/RWX(3,1000),FWX(9,1000)

Making the common case fast – NVT ensemble

AMBER allows for calculations using a number of different statistical ensembles, including NVT and NPT. For NVT calculations, don't need to save intermediate force vector results.

NPT ensemble

```
do jn=1,npr
 . . .
 fwx(1,jn)=xwij(1,jn)*dfa
 f(1,j)=f(1,j)+fwx(1,jn)
 . . . = . . . +fwx(1,jn)
 . . .
Enddo

– fwx(1,jn) used later –
```

NVT ensemble

```
do jn=1,npr
 . . .
 fwx1=xwij(1,jn)*dfa
 f(1,j)=f(1,j)+fwx1
 . . . = . . . +fwx1
 . . .
Enddo

– fwx1 not needed later –
```

Importance of compiler options

Performance of the optimized code on the CRAY T3E was very sensitive to the choice of compiler options. Very little dependence on IBM SP options.

Protein kinase test case on four CRAY T3E PEs

Case	speedup
original code/original flags	–
original code/new flags	0.97
tuned code/original flags	1.48
tuned code/new flags	1.76

Original flags: –dp –Oscalar3
New flags: –dp –Oscalar3 –Ounroll2 –Opipeline2 –Ovector3

Comparison of original and hand tuned codes

Plastocyanine in water benchmark

	IBM SP			CRAY T3E		
PEs	Orig	Tuned	speedup	Orig	Tuned	speedup
1	315	220	1.43	–	–	–
2	160	113	1.41	316	184	1.72
4	84.4	60.8	1.39	162	96.2	1.69
8	44.8	33.6	1.33	85.0	51.8	1.64
16	25.9	20.3	1.27	46.2	29.6	1.56
32	18.3	15.2	1.20	25.6	17.6	1.45
64	17.1	15.3	1.12	16.4	12.3	1.33

- Excellent speedup relative to original code on small numbers of processors
- T3E wins at larger number of PEs due to better inter-processor network

Particle mesh Ewald (PME) calculations

- **PME is the “correct” way to handle long ranged electrostatic forces. Effectively sums effects of all periodic images out to infinity**
- **Due to the structure of the PME routines – most loops contain multiple levels of indirect addressing – difficult to optimize**
- **Optimization efforts focused on the statement and basic block levels**
- **PME routines provided less “low hanging fruit” – already attacked by Mike Crowley (TSRI)**

Particle mesh Ewald (PME) – bitwise optimizations

The PME routines make use of bitwise operations to pack multiple integer variables into a single integer word. Example: extracting integer data

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

Extracting high-order bits from variable n

Same as extracting low-order bits from original variable n

Optimization

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

More efficient way to extract low-order bits using mask

Particle mesh Ewald (PME) – bitwise optimizations

The PME routines make use of bitwise operations to pack multiple integer variables into a single integer word. Example: <u>packing</u> integer data

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

High-order bits of n empty, packing jtran into high-order bits using ishft and add

Optimization

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

Achieves same result as above code, but avoids arithmetic operation

Common subexpression elimination

Most Fortran compilers are pretty good at performing common subexpression elimination, but only if expressions are simple enough.

Following optimization missed by compiler

```
common_expr = dx*(erf_arr(3,ind)+dx*erf_arr(4,ind)*third)

erfcc = erf_arr(1,ind)+dx*(erf)arr(2,ind)+common_expr*0.5)

derfc = -(erf_arr(2,ind)+common_expr)
```

Pulling loop invariants outside of inner loops

This is another optimization that most compilers can do, but only if the expressions are not too complex.

Following optimization opportunity missed by compilers

```
– several layers of nested loops –
term = . . .
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)
f3 = f3 – nfft2*term*th(I1,ig)*th2(i2,ig)*dth3(i3,ig)
. . .
```

Bold terms are loop invariants (w/ regards to innermost loop). Product of terms pre-calculated in next level up of loop nesting

Manual prefetching of cache lines

Technique most useful for accessing randomly accessed data, but can give performance benefits for hardware that does not support hardware streams

```
do m=1,numvdw
  n = ipairs(n)
  . . .
enddo
```

```
nprefetch = ipairs(1)
do m=1,numvdw
  n = nprefetch
  nprefetch = ipairs(m+1)
  . . .
enddo
```

Guarantees that n will be in cache at start of each iteration, minimizes effects of cache misses

Making the common case fast – NVT ensemble

As was done earlier, took advantage of opportunity to eliminate operations that are not need for NVT ensemble calculations

```
vxx = vxx – dfx*delx
. . .
vzz = vzz – dfz*delz

virial(1) = virial(1)+vxx
. . .
virial(6) = virial(6)+vzz
```

$\longrightarrow$

```
if(NPT_calc) then
 vxx = vxx – dfx*delx
 . . .
 vzz = vzz – dfz*delz

 virial(1) = virial(1)+vxx
 . . .
 virial(6) = virial(6)+vzz
endif
```

Comparison of original and hand tuned PME codes

CRAY T3E benchmarks

	Water			dhfr		
PEs	Orig	Tuned	speedup	Orig	Tuned	speedup
2	222.5	172.5	1.29	422.7	331.6	1.27
4	118.4	91.9	1.29	223.2	176.5	1.26
8	64.5	51.0	1.26	119.8	96.5	1.24
16	38.2	30.83	1.24	69.5	56.0	1.24

Lessons learned / applicability to other MD calculations

- Take advantage of vector inverse square root intrinsics
- Compilers are limited in their ability to identify loop invariants and common subexpressions. Do these optimizations by hand
- Optimize for the common case, create multiple versions of code blocks or subroutines if necessary
- Experiment with compiler options – don't assume that highest level of optimization will work best
- Keep in mind physics of the code
  - What quantities required for NVT, NPT ensembles?
  - When is periodic imaging required?