Performance Optimization and Tuning for NEC TX7

including
MOLPRO, MOLCAS, AMBER, GAMESS-US

Jan Fredin, Ph.D.
NEC Solutions America
March 5, 2004

Talk Overview

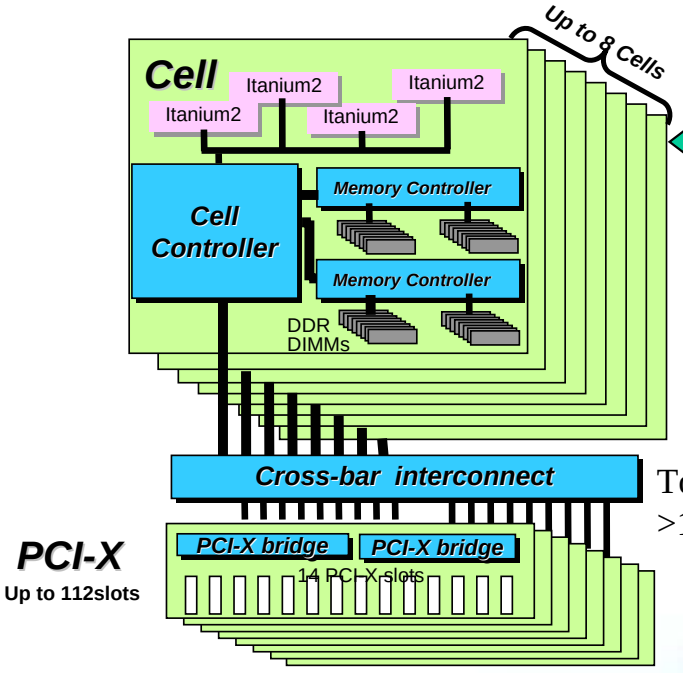
- NEC TX7 hardware
- General TX7 performance tuning techniques
- Case Studies: examples of tuning efforts on Chemistry applications:
  - *MOLPRO*
  - *MOLCAS*
  - *AMBER*
  - *GAMESS-US*
  - *User Molecular Dynamics Code*

NEC TX7 Servers

- Second generation technical computing server with Intel's latest 64-bit CPU "Itanium® 2 processor".
- Configured with up to 32 CPUs per node. NEC was the first company to release Itanium® 2 systems with 32 processors.
- Designed and built with supercomputer technology from NEC.
- Ideal systems for scientific and engineering applications.

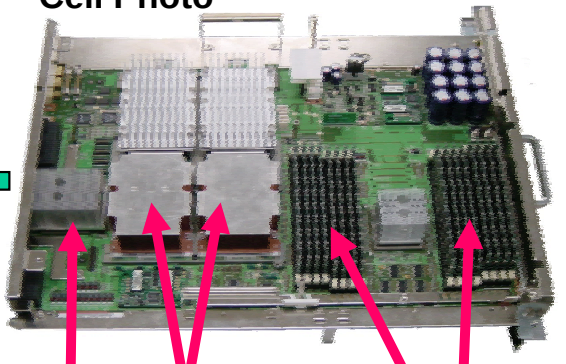
TX7 Internal Structure

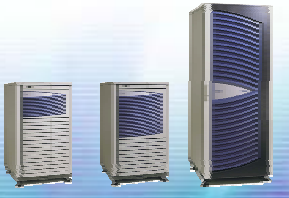
- ccNUMA architecture
- Near-flat 32-way memory access
- Flexible partitioning



Total band width >100GB/s

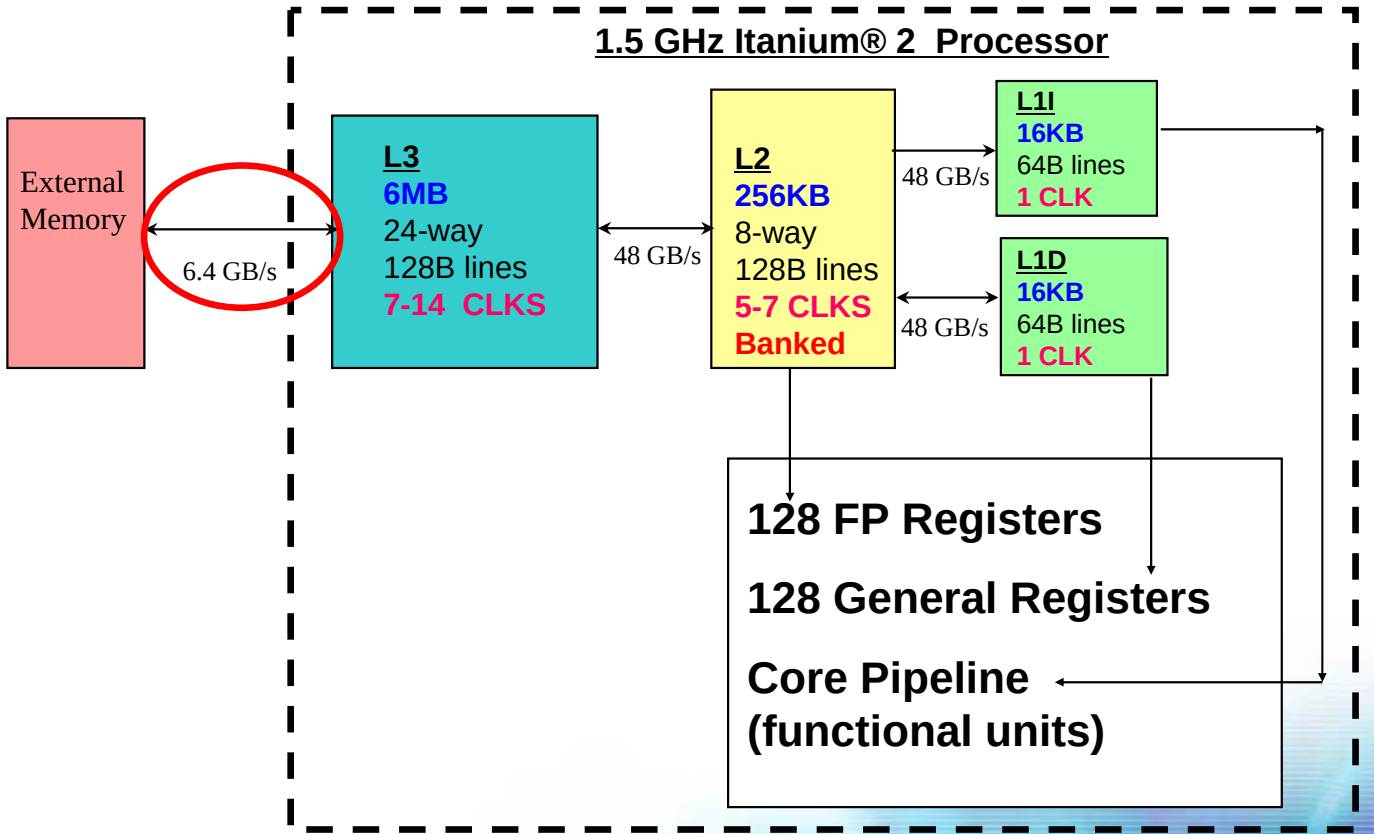
Cell Photo



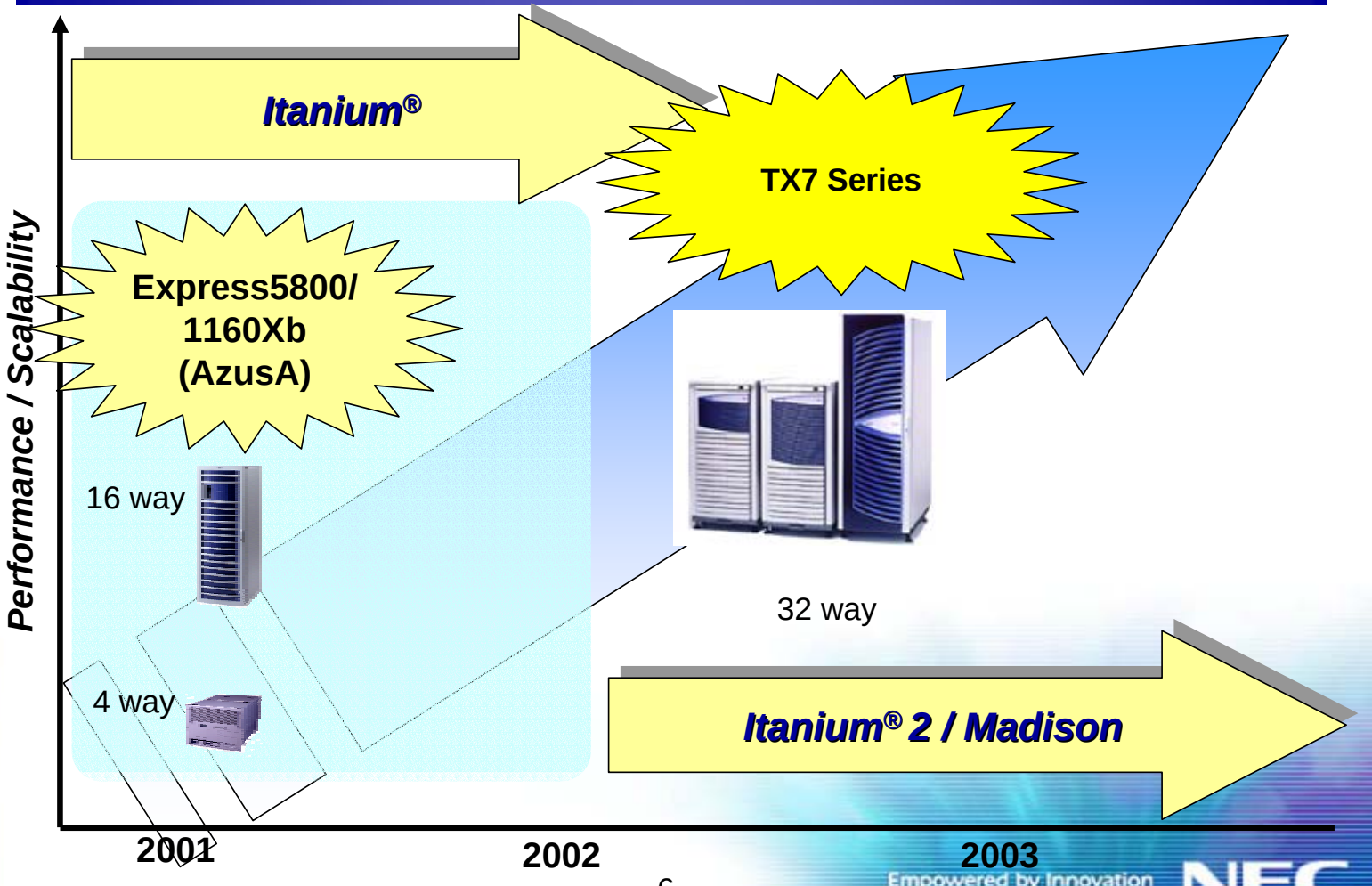


TX7 / i6010, i6510, i9510

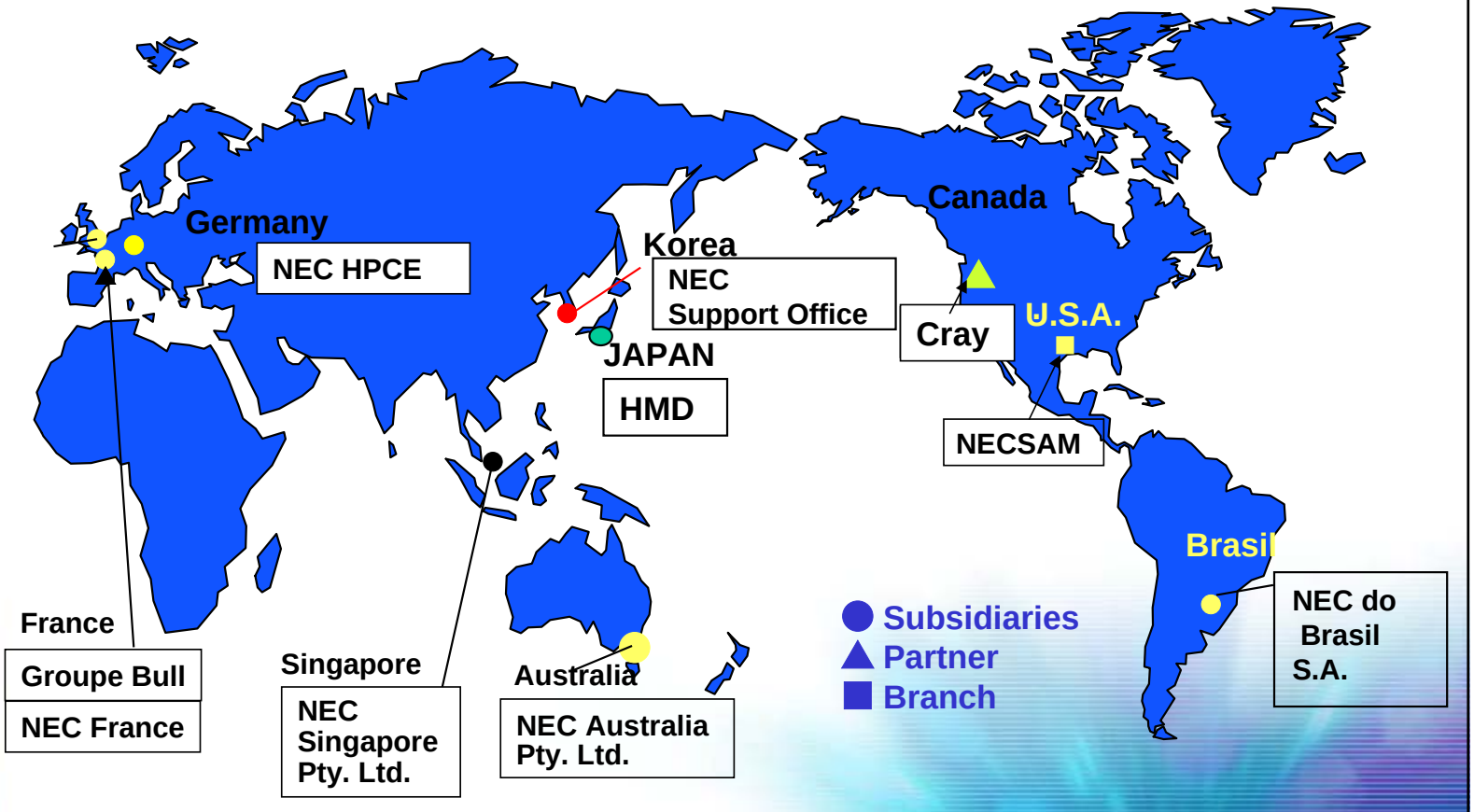
Itanium 2 Processor Memory Hierarchy



NEC IPF Server Roadmap



NEC Worldwide HPC Support and Development Organizations



Software Support Cycle

- Port - successfully run code for all QA available
  - typical compile efc -O2 -ftz
  - reduce optimization for problem areas
- Tune - improve performance
  - identify time consuming routines for typical workload
  - measure performance using UNIX gprof, NEC compiler add-of ftrace, Intel VTUNE or timing calls
  - test impact of changes in compile options, compile directives and source code restructure
  - can be limited by ISV code maintenance rules

Software Support Cycle

- Update, supporting software vendor and user sites
  - test and verify full QA correctness
  - provide changes through vendor distribution or release notes
  - update for:
    - new vendor software release
    - compiler update
    - user problems, feedback and benchmarks
    - hardware upgrades
- Repeat the tune and update steps regularly

Tuning Challenge

- keep fast functional units busy, providing all data required
  - 2 FMA / clock - 6GFLOPS / sec for Itanium2 1.5GHz
- maximize loop optimization
  - provide compiler with loops that can achieve high performance
  - software pipelining - similar to vectorization
  - instructional parallelism - 6 (2 bundles) / clock
  - prefetching - avoid waiting at the L3 cache bottleneck
- high performance is achieved by balance in the loops
  - computational complexity
  - data locality
  - number of variables

Software Tuning Techniques

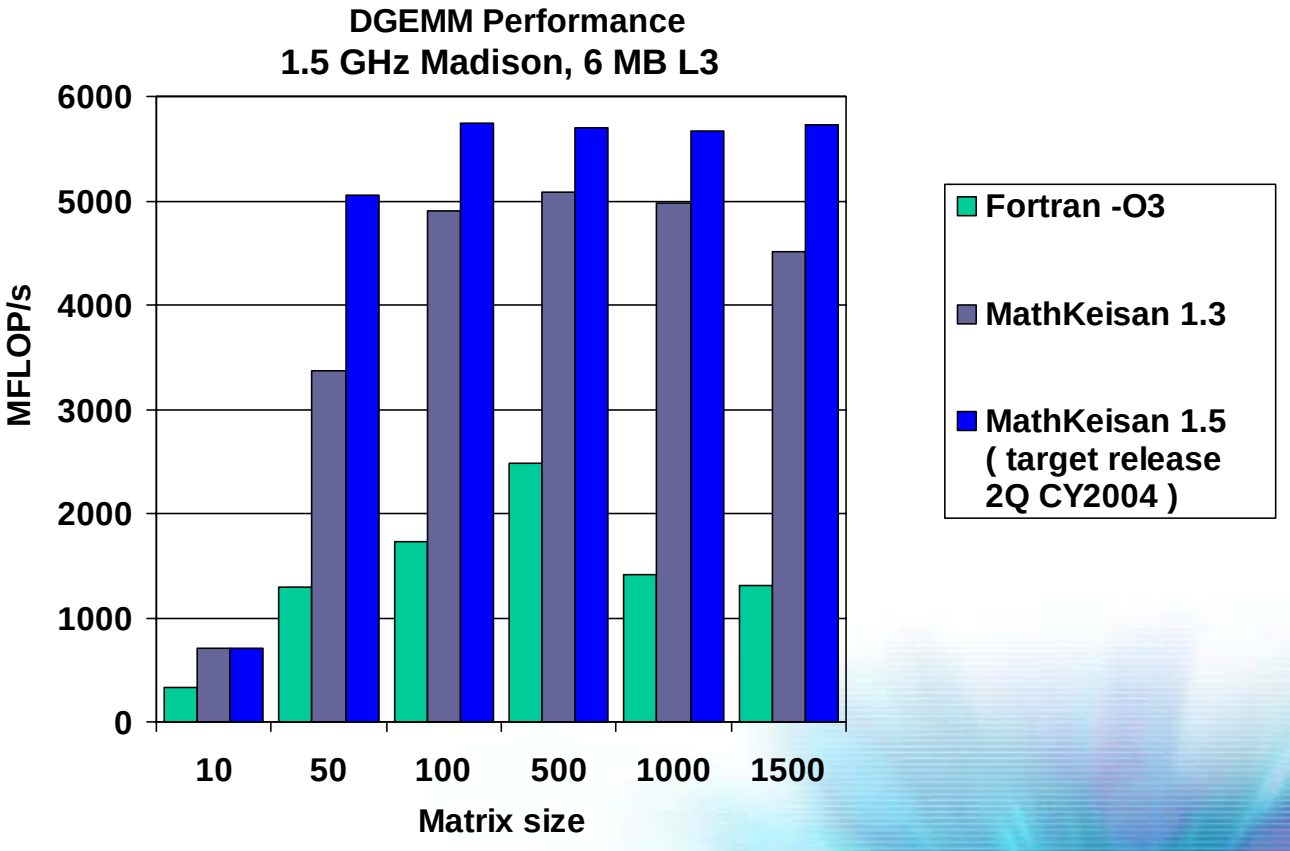
- know the good application input and algorithm choices
  - memory, processors, site defaults
- use tuned kernels like NEC MathKeisan
- choose effective compile options
- restructure loops
- help compiler optimize through source-code directives

NEC MathKeisan

MathKeisan is a set of NEC highly tuned math libraries.

Name	Description
BLAS	Basic Linear Algebra Subprograms
LAPACK	Linear Algebra PACKage for high performance computers.
ScaLAPACK	Scalable Linear Algebra PACKage (contains PBLAS)
BLACS	Basic Linear Algebra Communication Subprograms. Uses MPI.
CBLAS	C interface to BLAS
SBLAS	Sparse BLAS (from ACM Algorithm 692)
FFT	Fast Fourier Transforms
METIS	Matrix/Graph ordering and partitioning library
ParMETIS	Parallel Matrix/Graph ordering and partition library. Uses MPI.
SOLVER	Direct solver for sparse symmetric systems
ARPACK	Solution of large scale eigenvalue problems

MathKeisan 1.5 Improvements



Compile Choices

- NEC compiler is a superset of the Intel compiler
  - NEC compiler provides ftrace function for code section performance tracing
- Compiler optimization level
  - -O3 highest level of optimization
    - software pipelining on
    - activates prefetching
    - activates other loop optimizations
  - -O2 safe optimization level
    - software pipelining on
  - -O0 no optimization for debugging

Other Compile Options

- -ftz flush-to-zero denormalized values
  - should always use
  - default at compiler level -O3
- -opt_report compiler optimization report
  - messages about pipelining, dependencies, unrolling, other loop restructuring
- -ip interprocedural optimization within object files
- -prof_gen and -prof_use profile generated optimization
- -restrict / -ansi_alias enable strict pointer aliasing
- -i8/-r8 integer and real default word size

Loop Restructuring

- code developers know overall structure
- provide compiler with loops that can achieve maximum performance
- multi-purpose scientific and engineering codes often have heavy usage of long complex loops with conditionals and indirect indexing
- goal of hand recoding is to isolate sections that are hard for the compiler to optimization
  - too complex to analyze
  - external calls
  - data dependencies
  - conditionals

Restructuring Example - Loop Blocking

- breaks up multi-level loop into blocks that fit into cache
- best blocking sizes determined experimentally
- blocked sections are prime targets for parallelism

Original Code

```
do j = 1, n
 do i = 1, p
  do k = 1, m
   c(k,j)=c(k,j)-a(k,i)*b(i,j)
  enddo
 enddo
enddo
```

Modified Code

```
parameter (kblock = 1024)
do ks = 1, m, kblock
 do j = 1, n
  do i = 1, p
   do k = ks, min(ks+kblock-1,m)
    c(k,j)=c(k,j)-a(k,i)*b(i,j)
   enddo
  enddo
 enddo
enddo
```

Compile Automated Loop Restructuring

- Loop Unrolling
- Unroll and Jam
- Reduction Optimization
- Loop Interchange
- Loop Fusion
- Loop Fission

Loop Unrolling

- perform n iterations of the inner loop on each pass
- actives at compile level -O3, compiler sets unroll depth n
  - !DIR$ unroll (n) or !DIR$ nounroll
  - directive controls unroll depth n or switchs off

Original Code

```
      subroutine vadd(n, a, b)
       integer n, i
       double precision a(*),b(*)
!DIR$ UNROLL(4)
       do i = 1,n
         a(i) = a(i) + b(i)
       enddo
       end
```

Output from -O3 -opt_report

Block, Unroll, Jam Report:

(loop line numbers, unroll factors and type of transformation)

Loop at line 6 unrolled with remainder by 4

Modified Code

```
      subroutine vadd(n, a, b)
       integer n, i
       double precision a(*), b(*)
c  explicitely unrolled by 4
       do i = 1,n,4
         a(i)   = a(i)   + b(i)
         a(i+1) = a(i+1) + b(i+1)
         a(i+2) = a(i+2) + b(i+2)
         a(i+3) = a(i+3) + b(i+3)
       enddo
       do i=i,n
         a(i)   = a(i)   + b(i)
       enddo
       end
```

Unroll and Jam

- "jam" n iterations outer loop into each pass of inner loop
- actives at compile level -O3, no user control

Original Code

```
do j = 1, n
 do k = 1, p
  do i = 1, m
   c(i,j)=c(i,j)+a(i,k)*b(k,j)
  enddo
 enddo
enddo
```

Output from -opt_report at -O3:
Block, Unroll, Jam Report:
(loop line numbers, unroll factors and type of transformation)
Loop at line 1 unrolled and jammed by 4
Loop at line 2 unrolled and jammed by 4

Modified Code

```
do j = 1, n-3, 4
 do k = 1, p
  do i = 1, m
   c(i,j+0)=c(i,j+0)+a(i,k)*b(k,j+0)
   c(i,j+1)=c(i,j+1)+a(i,k)*b(k,j+1)
   c(i,j+2)=c(i,j+2)+a(i,k)*b(k,j+2)
   c(i,j+3)=c(i,j+3)+a(i,k)*b(k,j+3)
  enddo
 enddo
enddo
do j = j, n
 do k = 1, p
  do i = 1, m
   c(i,j)=c(i,j)+a(i,k)*b(k,j)
  enddo
 enddo
enddo
```

Reduction Optimization

- split sum into n partial sums then collect partial sums
- actives at compiler level -O3, no user control

Original Code

```
s = 0.0d0
do i = 1, n
  s = s + a(i)
enddo
```

Output from -opt_report at -O3

Block, Unroll, Jam Report:

(loop line numbers, unroll factors and type of transformation)

Loop at line 2 unrolled with remainder by 8 (reduction)

Modified Code

```
s1 = 0.0d0
s2 = 0.0d0
s3 = 0.0d0
s4 = 0.0d0
do i = 1, n-3, 4
  s1 = s1 + a(i)
  s2 = s2 + a(i+1)
  s3 = s3 + a(i+2)
  s4 = s4 + a(i+3)
enddo
do i = i, n
  s1 = s1 + a(i)
enddo
s = s1 + s2 + s3 + s4
```

Loop Interchange

- change in nesting order of 2 or more loops
- unit stride on loads to reuse data in cache
- activates at compile level -O3, no user control

Original Code

```
do i = 1, m
 do j = 1, n
  do k = 1, p
   c(i,j)=c(i,j)-a(i,k)*b(k,j)
  enddo
 enddo
enddo
```

Modified Code

```
do k = 1, p
 do j = 1, n
  do i = 1, m
   c(i,j)=c(i,j)-a(i,k)*b(k,j)
  enddo
 enddo
enddo
```

Output from -opt_report using -O3

```
LOOP INTERCHANGE in test_ at line 1
LOOP INTERCHANGE in test_ at line 2
LOOP INTERCHANGE in test_ at line 3
```

Loop Fusion

- combine 2 or more loops with same iteration count into 1
- reduces overhead, can promote data sharing and computational complexity
- can block pipelining if too many registers get used
- activates at compile level -O3, no user control

Original Code

```
subroutine vmul(n,alpha,x,y)
integer n, i
double precision x(*), y(*)
do i = 1, n
  x(i) = x(i) * alpha
enddo
do i = 1, n
  y(i) = y(i) * alpha
enddo
end
```

Output from -opt_report at -O3

Fused Loops: (5 8)

Modified Code

```
subroutine vmul(n,alpha,x,y)
integer n, i
double precision x(*), y(*)
do i = 1, n
  x(i) = x(i) * alpha
  y(i) = y(i) * alpha
enddo
end
```

Loop Fission

- also called loop distribution or loop splitting
- smaller loops for optimization like prefetching and pipelining
- activates at compile level -O3
  - !DIR DISTRIBUTE POINT focuses compiler on trying fusion
    -outside loop, compiler sets location
    -inside loop, programmer sets location

Original Code

```
      do i = lft, llt
         fail(i)=1.0
         hgener(i)=0.0
         diagm(i)=1.e+31
         sieu(i)=ies(nnm1+i)
!dir$ distribute point
         x1(i)=x(1,ix1(i))
         y1(i)=x(2,ix1(i))
         z1(i)=x(3,ix1(i))
       enddo
```

Output of -opt_report for -O3

LOOP DISTRIBUTION in test_ at line 1

Distributed for large ii at __ performed in test_ at line 7

Modified Code

```
do i = lft, llt
   fail(i)=1.0
   hgener(i)=0.0
   diagm(i)=1.e+31
   sieu(i)=ies(nnm1+i)
enddo
do i = lft, llt
   x1(i)=x(1,ix1(i))
   y1(i)=x(2,ix1(i))
   z1(i)=x(3,ix1(i))
enddo
```

Compile Directive - Variable Dependencies

- compiler cannot determine array references overlap so pipelining is blocked
- !dir$ IVDEP - fortran loop level
- #pragma IVDEP - C loop level
- C restrict pointer - routine call level

```
      subroutine test(nu1,nu2, tx, sxx, dd1, dd2, np)
       integer nu1, nu2, i3, k, np(*)
       double precision tx(3,*), sxx(9,*), dd1, dd2

!dir$ ivdep
       do k = nu1, nu2
          i3 = np(k)
          tx(1,i3) = tx(1,i3) + sxx(1,k)*dd1 + sxx(2,k)*dd2
          tx(2,i3) = tx(2,i3) + sxx(4,k)*dd1 + sxx(5,k)*dd2
          tx(3,i3) = tx(3,i3) + sxx(7,k)*dd1 + sxx(8,k)*dd2
       enddo
       end
```

Other Compile Directives

- prefetch
  - data values are loaded into cache ahead of access to reduce latency in loading data
  - too much prefetching can cause cache misses
  - !dir$ prefetch a
  - !dir$ noprefetch b
- software pipelining
  - default in compile level -O2 and -O3
  - directive to switch on/off software pipelining
  - !dir$ swp
  - !dir$ noswp
- loopcount
  - hint for compiler to optimize loop
  - !dir$ maxloopcount_value

Case Studies

- Examples of tuning efforts on Chemistry applications:
  - *MOLPRO*
  - *MOLCAS*
  - *AMBER*
  - *GAMESS*
  - *User Molecular Dynamics (MD) Code*

MOLPRO 2002.6

- MOLPRO is designed and maintained by H.-J. Werner (University of Stuttgart) and P. J. Knowles (University of Birmingham)

  http://www.molpro.net/

- MOLPRO is a complete system of *ab initio* programs for molecular electronic structure calculations with an emphasis on highly accurate computations and extensive treatment of the electron correlation.

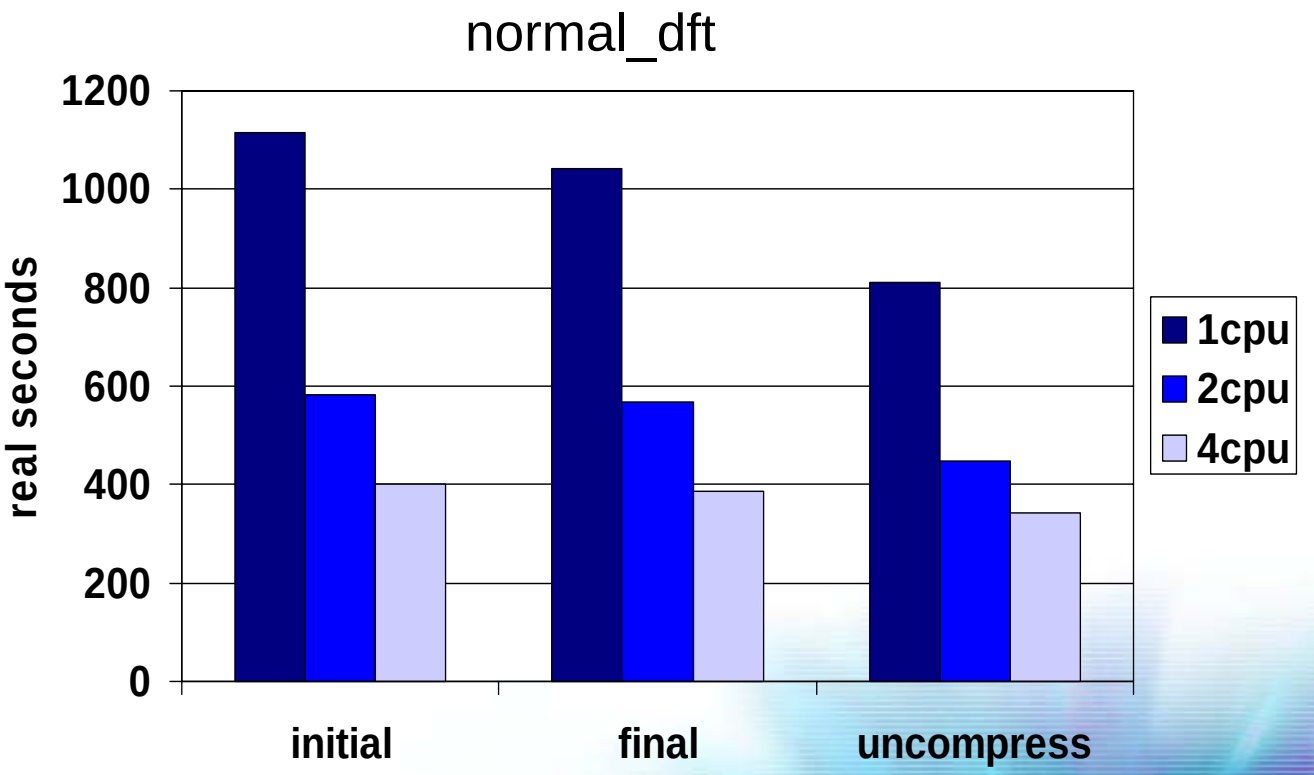
- Mixed Fortran and C using PNNL Global Arrays (GA) for shared memory and parallel MPI tasks. Driver program checks license and exec's computational executable. 64 bit real, integer, pointer

- normal_dft dataset
  - adrenaline SVWN, BLYP from MOLPRO bench directory

MOLPRO 2002.6

- upgrade source from MolPro2002.5 to MolPro2002.6
- compiler initial optimization -O2 and < 20 routines –O0
- upgrade MathKeisan for blas and lapack
- profile top routines using MolPro development environment to run computational unit stand-alone: normal_dft
  - 21.8% uncompress_double
  - 9.0% dfti_block
  - 7.8% dft_rho
  - 5.2% dform
  - 5.0% dgenrl
- aioint, compress=0 turns off compression
  - at end of molpro2002.6/bin/molproi.rc add:
  - nocompress
- dft/dfti.f -O3 gives correct numerical results for all QA
- another customer test profile showed top routines addmx, mxmas, fzero tuned with -O3 (3 files) gave 22% improvement

MOLPRO 2002.6



MOLCAS 5.4

- MOLCAS is developed and distributed from the Department of Theoretical Chemistry at Lund University
  http://www.teokem.lu.se/molcas/

- MOLCAS is a quantum chemistry software package with a focus on methods that will allow an accurate *ab initio* treatment of very general electronic structure problems for molecular systems in both ground and excited states.
- MOLCAS is a research product developed by scientists to be used by scientists.

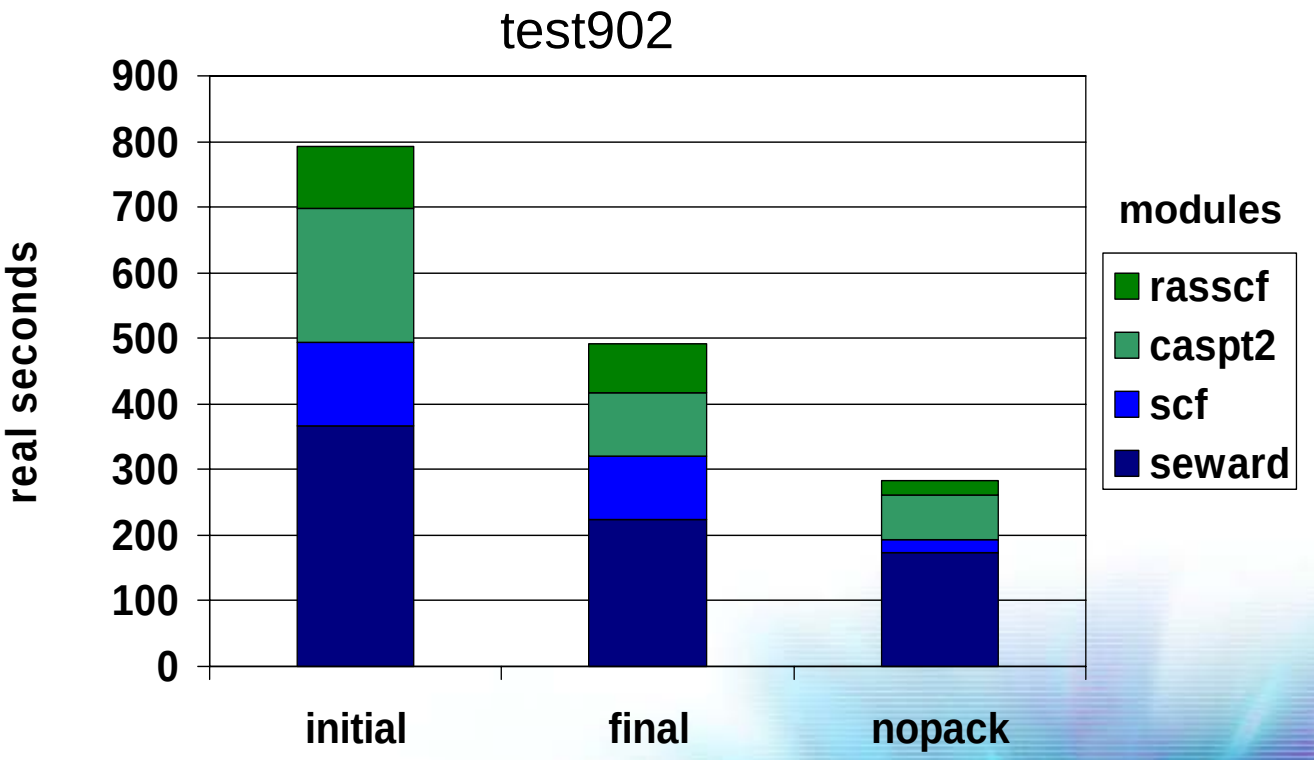
- Fortran with some C systems control routines, MPI parallel and shared memory through PNNL GA. 64 bit real, integer, pointer. Independent modules are called in sequence from MOLCAS-generated scripts.

- test902 performance test provided in the MOLCAS distribution. Disk-based Hartree-Fock using modules seward and scf

MOLCAS 5.4

- initial compile optimization -O2 with 4 routines at -O0, static link
- serial focus due to limited parallel implementation in MOLCAS 5.4
  - parallel support targeted for next release.
- 4 main modules:
  - 46% seward
  - 26% caspt2
  - 16% scf
  - 12% rasscf
- add MathKeisan blas – 38% improvement
- profile showed pack / nopack are in top routines
- added nopack keyword for Seward module input

MOLCAS 5.4



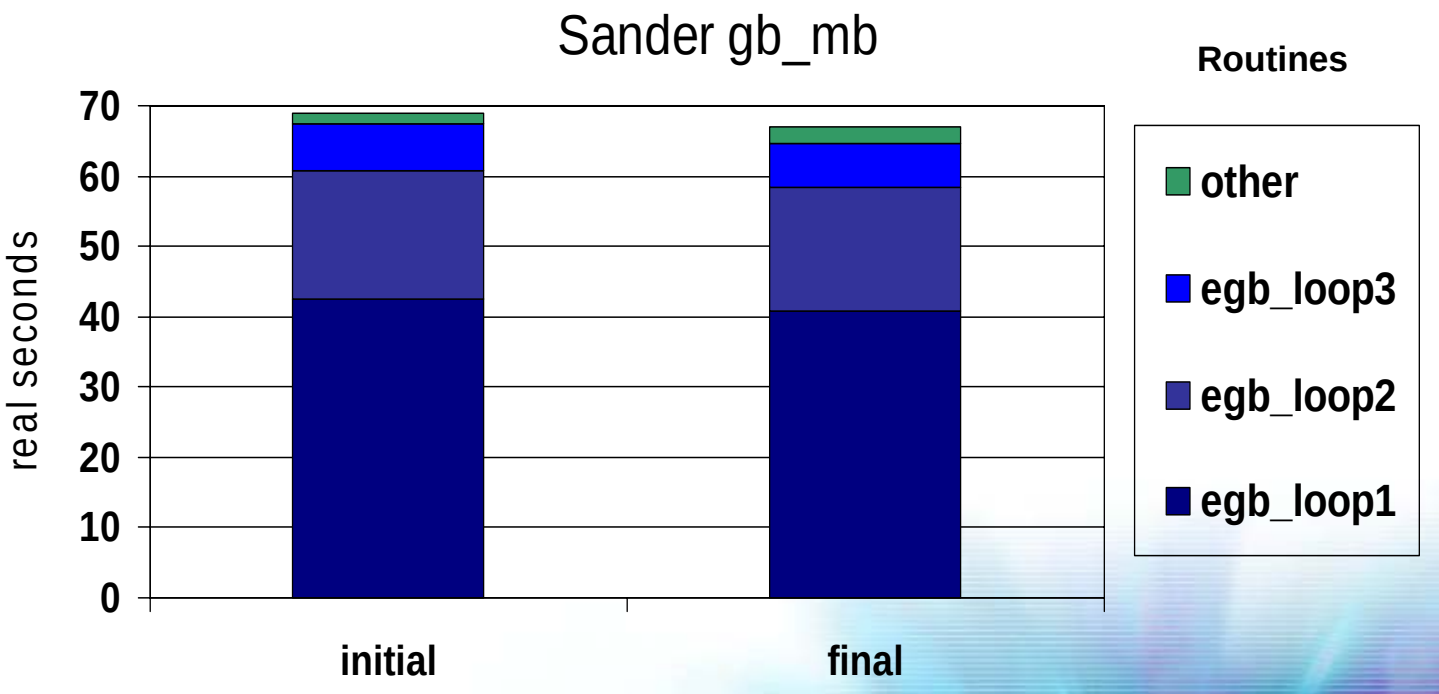
AMBER 7.0

- Distributed by University of California, San Francisco
  http://amber.scripps.edu/
- About 60 programs that work reasonably well together solving molecular modeling and dynamics problems.
  - sander
    - Simulated annealing with NMR-derived energy restraints.
    - main program used for molecular dynamics simulations.
- Mostly Fortran with C for dynamic memory and file handling and MPI parallel. Each module is a separate executable. Sander is built from 98 source files consisting of 408 subroutines.
- sander gb_mb dataset provided in AMBER benchmarks directory
  - Generalized Born myoglobin simulation, 2492 atoms in 100,000 waters
  - 20 Ang. cutoff, salt 0.2Molar, long-range forces every 4 cycles

AMBER 7.0

- initial distribution compile optimization
  - compile level2=-O2
  - compile level3=-O2 used for 36 out of 96 files
- profile shows 98% of time spent in egb
  - subroutine egb is composed of 3 large loops
- upgrade MathKeisan blas and lapack
- compile tuning
  - level3=-O3, QA shows egb has numerical errors
  - level3=-O3 used for all other level3 files, used -O2 for egb
- code restructuring
  - tried splitting egb_loop1 and egb_loop2 - same performance
  - tested impact of !dir$ IVDEP in many locations; no improvement

AMBER 7.0



GAMESS-US

- GAMESS is maintained by the members of the Gordon research group at Iowa State University

  http://www.msg.ameslab.gov/GAMESS/GAMESS.html

- The General Atomic and Molecular Electronic Structure System (GAMESS) is a program for *ab initio* quantum chemistry.

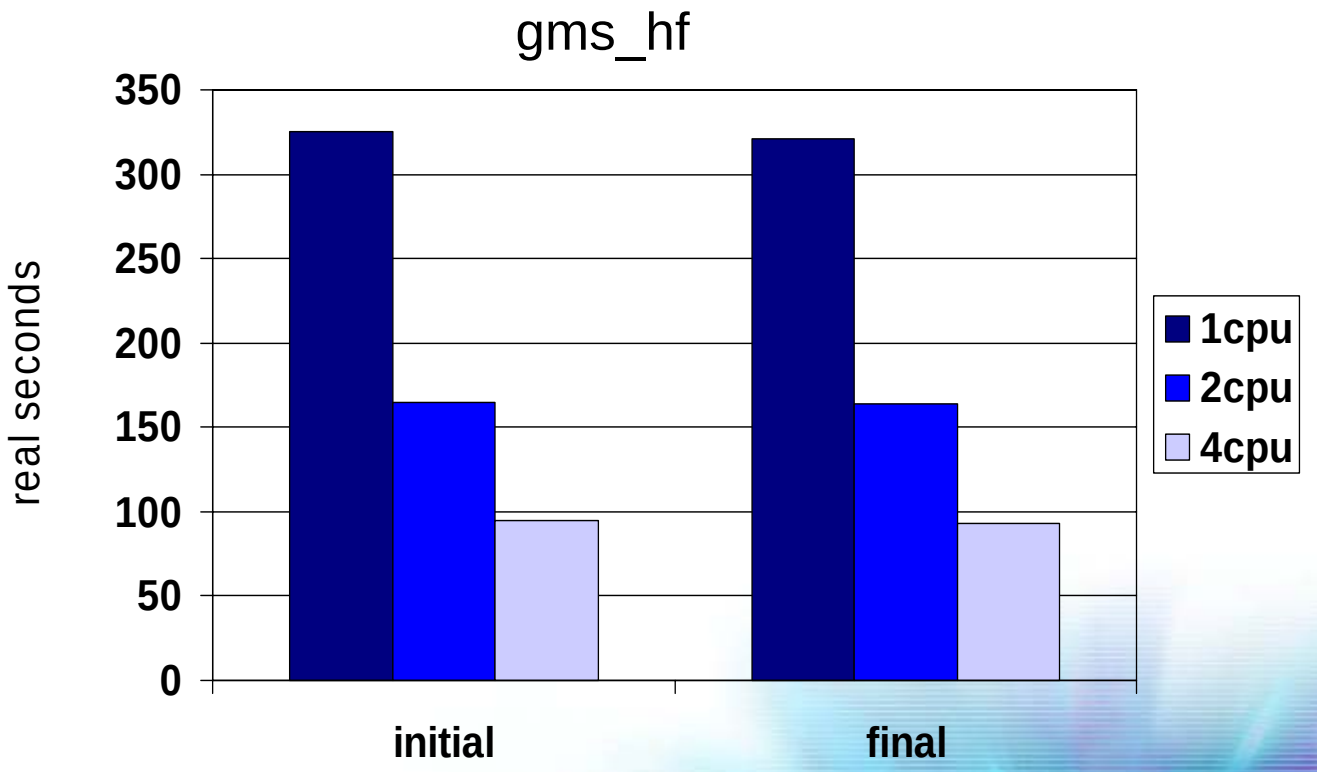
- Fortran single executable using MPI over sockets for parallel. 64bit real, integer, pointer

- gms_hf dataset
  - nicotine Hartree Fock SCF single point calculation
  - one of GAMESS benchmarks distributed in early 1990's

GAMESS-US

- profile gms_hf:
  - 42.0% hstar
  - 9.5% dspdfs
  - 3.7% forms
- initial compiler optimization -O2
- -O3 top 5 routines: slower
- -O1 hstar: slower
- hstar: ivdep on nonpipelining loops: slower
- turn off packing: slower
- use vector code for dspdfc & forms: slower
- update source, compiler, MathKeisan
- saw some tests with 12% time in __libc_read: comes from compile libraries on cross-mounted disk: use static link and no shared object (lib*.so)

GAMESS-US



User MD Tuning - compile options

- customer provided molecular dynamics application, written in about 12,000 lines of C with customer test input
- original compile flags
  - ineffective with -mp (maintain precision) severly limits performance
  - ecc -O3 -tpp2 -Zp16 -static -mp -IPF_fma -IPF_fltacc
- tuned compile flags
  - ecc -O3 -tpp2 -Zp16 -static -restrict -ftz -ip
  - -restrict (or -ansi_alias) tells compiler that pointers labeled restrict in routine call have independent locations
    - restrict pointers provides information about potential dependencies that can enable pipelining
    - restrict pointers declared on most of MD's malloc'ed locations
  - -ip turns on inter-procedural optimizing within the file that is being compiled

User MD Tuning - code modifications

- use tuned library call
  - replace original code with MathKeisan function vdSinCos
  - computes (sin(a[i],cos(a[i]) for i=0…n
  - pipelines multiple sin and cos calculations
- programmer level loop recoding
  - split main loop of important routine into 3
    - non-bonded pair list generation
    - calculation of force potentials across all pairs
    - application of forces across all pairs
  - enables pipelining

User MD