

Performance Programming: Theory, Practice and Case Studies

Module IV: Case Studies



Computational Chemistry

- The computational study of atoms and molecules
 - From physics - chemistry - biology
 - Includes derivative areas like material science

Critical Issues

- Quantum systems
 - The interactions of protons, neutrons and electrons are governed by quantum mechanics
- Dynamical systems
 - Movement is fundamental to chemistry, e.g. chemical reactions, temperature
- Size
 - A glass of water contains over 10^{23} water molecules



Major Domains

- Quantum Chemistry
 - Started by people trying to solve the Schrödinger equation for atomic and simple molecular systems, eg H₂, LiH
 - Now some methods are applicable to hundreds of atoms
 - Example programs; ADF, GAMESS, **Gaussian**, QChem etc.
- Molecular Dynamics (MD)
 - Started by people trying to describe the dynamic behavior of a few hundred atoms interacting via a classical potential
 - Now tens of thousands of atoms for 10⁻⁷ seconds real time
 - Example programs: **Amber**, Charmm, **Gromos**, Tinker etc.
- Statistical Mechanics
 - Take a statistical approach to treating very large system
 - More home grown codes tailored to particular problem
- Now domains overlap, e.g. quantum MD



The Gaussian Program

- Probably the most widely used computational chemistry package
 - Originator, John Pople, awarded 1998 Nobel Prize
 - See www.gaussian.com
- Primarily a quantum chemistry code, solving the electronic Schrödinger equation for atoms, molecules and (development version) solids
 - Solves a set of partial differential equations
 - Spectral approach placing basis functions at the positions of each atomic nucleus
- Developed since 70's
 - Predominantly Fortran 77 with some C
 - 98 version contains approximately 600,000 lines



Benchmarking Issues

- Require benchmarks that
 - Span a range of different widely used functionality
 - Should complete in "reasonable" time
- Primary input options for Gaussian
 - Hartree Fock (HF), density functional (BLYP), mixed (eg B3LYP), perturbation (MP2), coupled cluster (eg QCISD), etc.
 - Energy, Gradient, Frequency
 - Gas phase, solvated, solid, electric/magnetic field
 - Basis set, e.g. 3-21g, cc-pvtz, 6-311++G(3df,3pd)
- Secondary input options
 - Memory
 - Input/Output
 - Sequential/parallel
- Requires knowledge of the application and code

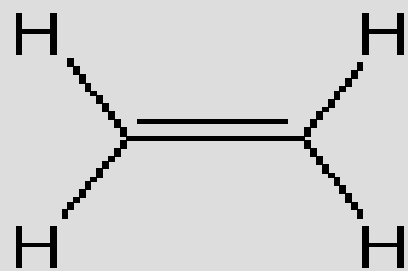


Some Gaussian Benchmarks

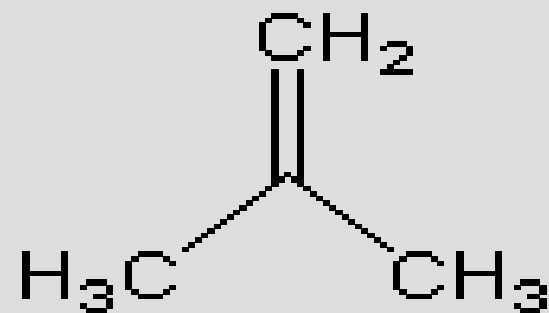
	Theory	Type	Basis	Functions
Ethylene				
1.1	MP2	Gradient	6-311++G(3df,3pd)	150
1.2	MP2	Frequency	6-311++G(3df,3pd)	150
Isobutene				
2.1	HF	Frequency	6-311++G**	144
2.2	BLYP	Frequency	6-311++G**	144
2.3	MP2	Energy	6-311++G(3df,3pd)	300
2.4	MP2	Gradient	6-311++G(3df,3pd)	300
2.5	QCISD(T)	Energy	6-311++G**	144
18-crown-6-ether				
3.1	HF	Gradient	6-31G**	390
3.2	BLYP	Gradient	6-31G**	390
3.3	MP2	Energy	6-31G**	390



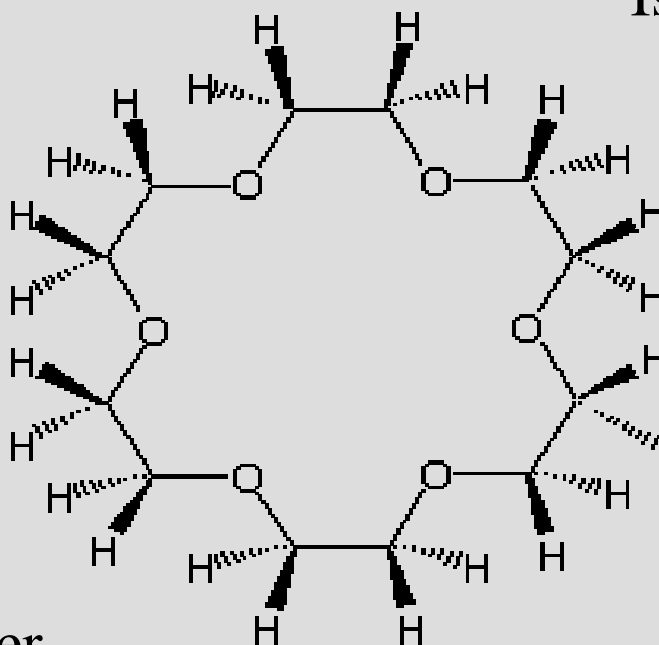
Our Systems



Ethylene



Isobutene



18-crown-6-ether



Machine Performance Ratios

- 3 machines (A, B and C) from 3 different vendors
 - Comparison of Gaussian performance ratios with Peak, SpecFP and Streams
- Looking for anything abnormal

	Performance Ratio		
	A:B	A:C	B:C
Peak (theoretical)	1.5	1.7	1.1
SpecFP2000 (spec.org)	0.9	1.8	2.0
Streams (triad measured)	1.2	2.8	2.4
Ethylene			
	1.1	0.9	1.8
	1.2	1.0	2.1
Isobutene			
	2.1	0.8	1.7
	2.2	1.0	1.5
	2.3	0.9	1.7
	2.4	1.0	2.1
	2.5	1.3	2.5
18-Crown-6-Ether			
	3.1	1.0	1.8
	3.2	1.0	1.5
	3.3	1.1	2.1

A:B = 1.5 implies B is
1.5 times better than A



Other Times: Gaussian Links

- Many codes provide internal timing information
 - Is this useful!
- A Gaussian calculation requires the sequential execution of a series of "links"
 - Each link is a separate executable program
 - Subsequent links are executed via an `exec` system call
 - Links are often executed in a cyclic pattern
- Gaussian 98 has about 80 different links
 - Which are the time dominant ones?



Sample Gaussian Link Times

Link	Time(sec)	Link	Time(sec)	Link	Time(sec)
1	0.2	801	0.0	1102	0.4
101	0.2	906	1884.6	1110	42.5
103	0.0	1101	1.5	1112	391.4
202	0.1	1102	0.0	601	0.9
301	0.2	1110	41.8	701	3.0
302	1.1	1002	366.8	702	0.0
303	0.4	811	1362.2	703	135.3
401	1.1	804	62.7	716	0.0
502	63.5	1002	23.7	103	0.1

- MP2 frequency on ethylene
- Uses 23 different links
 - Some twice
 - Wide range of times
 - 9 unique links are important



Profiling

- Technical complication:
 - Profiling a job that sequentially uses many different executables can cause problems, e.g. the profile data file gets overwritten by the following link!
 - Solution - profile Gaussian in parts, running one link at a time
- gprof
 - Must recompile with -pg
 - Resolution of timer can be poor (0.01s)
- Collector/Analyzer (Sun) or similar
 - May not require recompilation
 - Requires running link under control of collector
 - Does not provide calling statistics (number of times routine initiated)



Sample Link Profiles

% time	cumulative seconds	self seconds	self calls	self ms/call	total ms/call	name
100.0	367.89					TOTAL
18.3	67.29	67.29	11277527	0.01	0.01	dgstr_ [8]
12.1	111.76	44.48	38175	1.17	1.17	dovr1_ [9]
11.3	153.51	41.75	163378	0.26	0.26	docont_ [10]
8.5	184.74	31.23	81689	0.38	0.38	scat20_ [11]
7.6	212.88	28.14	87028	0.32	0.32	dotran_ [12]
7.0	238.76	25.89	76350	0.34	0.34	dotrn_ [13]
5.9	260.44	21.67	81689	0.27	0.27	loadgo_ [14]
5.1	279.19	18.75	81689	0.23	0.23	calc0m_ [15]

- Two different profilers
- Same link and benchmark
- Two different machines
- Beware of library functions
 - May or may not be in profile

Excl.	Incl.	Name
User CPU	User CPU	
sec.	sec.	
1117.590	1117.590	<Total>
288.760	288.760	dgstr_
161.590	161.590	docont_
146.440	146.440	dovr1_
66.860	66.860	scat20_
66.350	66.350	dotran_
64.780	64.780	dotrn_
43.390	43.390	calc0m_
43.010	43.010	loadgo_



Where to Start

- From the 10 benchmarks we select 39 dominant links
- These links consume an aggregate time of 28637s
- Across all 39 profiles what are the dominant routines

Ordered by Total Time				Ordered by Average Time		
Routine	Frequency	Total Time		Routine	Frequency	Aver Time
dgstr_	17	3163.5		tri3cz_	1	1952.8
fqtril_	9	3050.5		matmp1_	1	511.4
dovr1_	36	2520.5		fqtril_	9	338.9
docont_	38	2203		splt01_	3	257.8
dotrn_	36	2094.4		dgst01_	3	200.1
tri3cz_	1	1952.8		dgstr_	17	186.1
dgemm_lib	21	1171.4		dg2d1_	1	149.7
splt01_	3	773.3		liacnt_	3	135.5
doshuf_	38	671.7		sprdnx_	4	125.3
dgst01_	3	600.3		gbtrn4_	2	98.5



Example Routine

```
Do I = 1, N
  Array1(I) = Array1(I) + Vector(I)*Constant1
  Array2(I) = Array2(I) + Vector(I)*Constant2
  Array3(I) = Array3(I) + Vector(I)*Constant3
enddo
```

- Dominated by 3 *daxpy* operations
 - Requires 4 loads and 3 store operations
 - Theoretical peak 6flops per 7 cycles (6/14 of 43% of peak)
- Does the compiled code permit this performance?
- What is the observed mflop rate?



Examining Pipelining on the Sun

```
f95 -fast -xprefetch=no -xtypemap=real:64,double:64,integer:64  
-xtarget=native -xarch=v9 -xcache=generic -Qoption cg  
-Qms_pipe+info -S example.f
```

```
Doall Loop : YES  
Doacross Loop : NO  
TripCount : Unknown  
Instrn Count : 20  
Load operns : 4  
Store operns : 3  
Amortizable operns: 7  
Float add operns: 3  
Float mul operns: 3  
  
Recurrence II : 0  
Resource II : 7  
Result : Pipelined  
Achieved II : 7 cycles  
Load Latency : 2 cycles  
Kernel Unroll Factor: 2  
Stage Count : 2  
Kernel Label : .L900000106
```

- Pipelining correctly
- With prefetching
 - 1 result every 9 cycles

```
With Prefetch  
Recurrence II : 0  
Resource II : 4  
Result : Pipelined  
Achieved II : 4 cycles  
Load Latency : 2 cycles  
Kernel Unroll Factor: 7  
Stage Count : 9  
Kernel Label : .L900000106  
Prefetch Instrns: 8
```



Manual Timing and Counting

- Use fine grain timer and performance counters
 - We actually did this in the calling routine to reduce overhead
- Also explicitly counted operations
- Beware of overflows/counter wrapping

```
starthr = timingfunc()           ! Start time
call setcount()                  ! Start counters

n_flops=n_flops+N                ! Count operations
Do I = 1, N
  Array1(I) = Array1(I) + Vector(I)*Constant1
  Array2(I) = Array2(I) + Vector(I)*Constant2
  Array3(I) = Array3(I) + Vector(I)*Constant3
enddo

my_time=my_time+timingfunc()-starthr !end time
call getcount(fcnt1,fcnt2)         !end counters
my_count1=my_count1+fcnt1
my_count2=my_count2+fcnt2
```



Performance Data

Interpretation	Counter	Value	Value/750MHz
Cycles	Cycle_cnt	5237638074	6.98
E-Cache reference	EC_ref	1975775108	
E-Cache misses	EC_misses	4217797	
All D-Cache Misses	Re_DC_miss	888140038	1.18
Store queue full	Rstall_storeQ	2561770800	3.42
FP number not ready	Rstall_FP_use	13826845	0.02
Floating adds	FA_pipe_completion	847785919	
Floating multiplies	FM_pipe_completion	841439121	

- Measured time 7.12s,
 - cycle count \equiv 6.98s
- Measured flops 1,682,856,000 flops
 - FA+FM counters give 1,689,225,040
- Mflops = $1,689,225,040/6.98 = 241$ mflops
 - $241/1500 = 16\%$ of peak (cf 43% theoretical max)
- Store operations are the bottleneck



Effect of Blas Library

Benchmark	Fortran (cpu sec)	Sunperf (cpu sec)	Speedup (ratio)
Ethylene			
1.1	265	249	1.07
1.2	3883	3979	0.98
Isobutene			
2.1	1317	1281	1.03
2.2	1291	1115	1.16
2.3	1716	1680	1.02
2.4	4770	4615	1.03
2.5	1674	1449	1.16
18-crown-6-ether			
3.1	1257	1227	1.02
3.2	1581	1527	1.03
3.3	3807	3620	1.05

- Time comparison on a Sun system using a simple Fortran DGEMM and DGEMV code to replace the Sun performance library
- Relatively minor change
- But expected given profiles



Effect of Compiler Options

- Two vendors, both with O1-O5 optimization flags
- Recompile top 85 routines with different optimization
- Much bigger effect for vendor 2

Benchmark	Speedup Vendor 1		Speedup Vendor 2	
	O1/O3	O3/O5	O1/O3	O3/O5
Ethylene				
1.1	2.29	1.03	4.10	1.13
1.2	2.23	0.98	3.55	1.12
Isobutene				
2.1	2.26	1.01	4.10	1.06
2.2	1.64	1.12	3.62	1.32
2.3	2.35	0.99	4.19	1.17
2.4	2.57	0.95	4.11	1.26
2.5	1.81	1.04	3.29	1.11
18-crown-6-ether				
3.1	2.43	0.98	4.30	1.20
3.2	2.03	0.96	3.95	1.32
3.3	2.58	1.00	3.95	1.33



Effect of Cache Blocking

- The size of certain data structures are limited to ensure better cache usage
- Table gives speedup obtained when using cache blocking on two different machines, both with 8MB caches

Benchmark	Vendor1	Vendor2
Ethylene		
1.1	1.37	1.82
1.2	1.22	1.36
Isobutene		
2.1	1.48	2.46
2.2	1.13	1.70
2.3	1.63	2.58
2.4	1.68	3.12
2.5	1.06	1.15
18-crown-6-ether		
3.1	1.88	3.50
3.2	1.36	2.93
3.3	1.14	1.81



Effect of Loop Unrolling

- The following times were obtained with and without loop unrolling for ONE critical routine

Link	Before(sec)	After(sec)	Speedup
Isobutene			
1002	754	610	1.24
703	300	248	1.21
502	104	86	1.22
1110	204	160	1.28
TOTAL	1378	1115	1.24
18-crown-6-ether			
502	1279	1071	1.19
703	564	439	1.29
TOTAL	1864	1527	1.22



Exploiting Parallelism

- Shared memory
 - Vendor specific directives (particularly SGI, Cray) and UNIX utilities (fork/shmget) (e.g. Gaussian98)
 - OpenMP replacing above (e.g. Gaussian development code)
 - Explicitly threaded (e.g. pthreads), none known
- Distributed memory
 - Linda (e.g. Gaussian)
 - MPI (e.g. Amber)
 - Global Arrays/DDI (e.g. NWChem and GAMESS)
- Data parallel languages (HPF, C*)
 - Not aware of any widely used computational chemistry code
- Key issues
 - Fraction of sequential code
 - Overhead
 - Memory usage



Parallel Overhead

```
DO I=1,N
  SUM = SUM + X(I)*X(I)
ENDDO
```

- OpenMP
 - Cost of DO PARALLEL is 1000+ cycles (i.e. $\approx 10^{-6}$ s)
- MPI
 - Latency of best interconnects $\approx 10^{-6}$ s
 - Bandwidth on RISC based systems is typically few hundred MB/s
- Could complete ≈ 1000 iterations of the loop in the overhead associated with parallelizing it!
 - A good parallel BLAS library should consider if the dimensions warrant use of multiple threads
- Parallelism must be as coarse grain as possible



Memory Issues

- Provides better aggregate memory bandwidth
 - Multiple simultaneous paths to memory
- To replicate or distribute?
 - Both have merits regardless of shared or distributed memory architecture
- Replicated data structures
 - + Unimpeded access to replicated data
 - Cost of broadcasting and collecting data
 - Implications for overall memory usage
- Distributed data structures
 - + Scalable memory usage
 - Cost of locks to control access
 - Possible hidden bottlenecks (eg false cache line sharing, memory page placement and the underlying architecture)
 - Usually gives complicated communications on distributed memory



Simple SCF

$$F_{ij} = h_{ij} + \sum_k \sum_{l \in N_{\text{functions}}} D_{kl} (2[ij|kl] - [ik|jl])$$

- F is repeatedly
 - Formed: $O(N^4)$ - evaluation of "[ij/kl]" integrals
 - Transformed: $O(N^3)$ - matrix multiplications
 - Diagonalized $O(N^3)$ - matrix diagonalization
- Process dominated (>80%) by integral evaluation
 - Computed in batches according to function type
 - Batch time varies in unpredictable manner (load balancing)
- Strategy
 - Replicate $O(N^2)$ quantities
 - Different batches to different processors
 - Sum partial F matrices across processors
 - On shared memory use parallel libraries for $O(N^3)$ operations



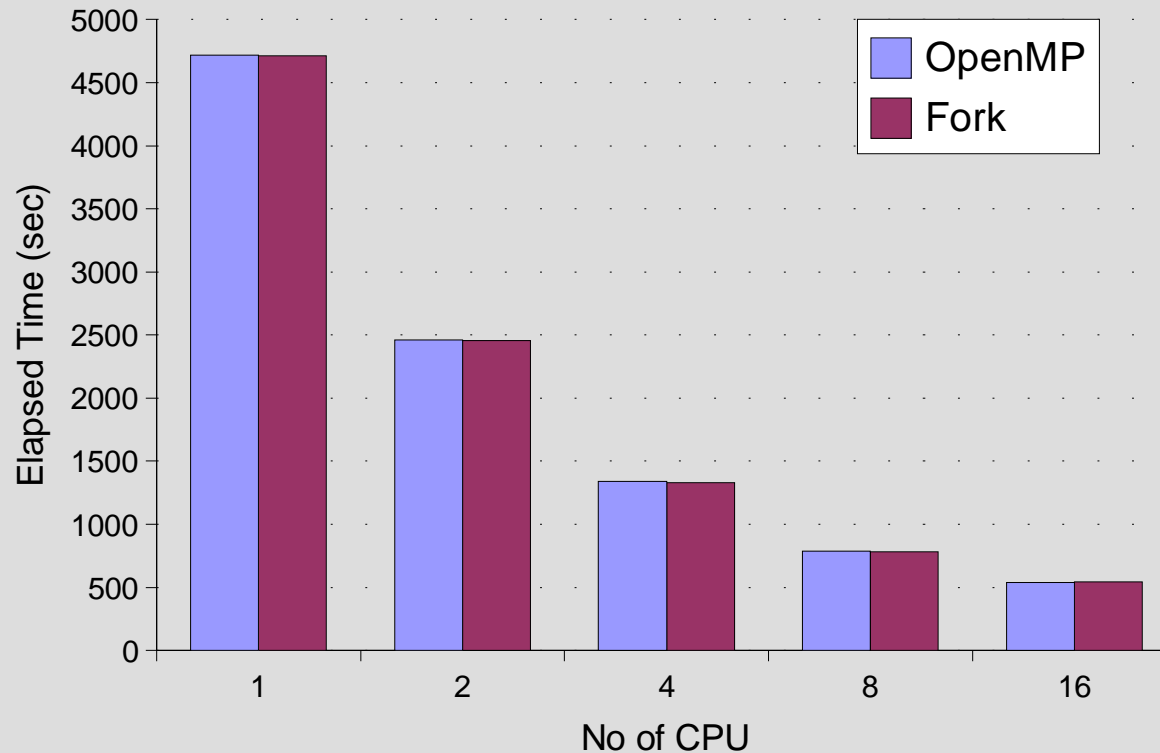
Implementations

- Simple UNIX utilities
 - Allocate multiple F matrices in a shared memory segment
 - Assign unique F to each process
 - Create multiple child processes via fork
 - Parent process sums partial F s when children terminate
 - Some use of parallel libraries
- OpenMP
 - Generate child processes via `PARALLEL DO` loop
 - Assign data structures as `PRIVATE` or `SHARED`
 - Master thread sums partial F s after parallel loop terminates
 - Some other loops via directives and use of parallel libraries
- Distributed memory
 - Linda `EVAL` function to generate processes
 - Partial F matrices communicated via Linda `IN/OUT` operations

$O(N^3)$ computation (at least) \gg $O(N^2)$ communication



Typical Performance

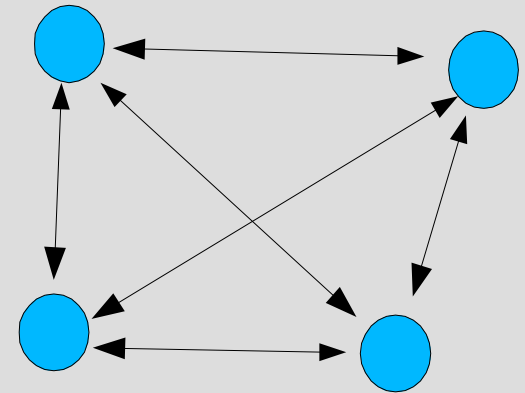


- Energy evaluation for α -pinene on SGI Origin (195MHz)
- Data from Sosa et al, Parallel Computing v28, p843
- Speedup 6 on 8cpus, 8.8 on 16



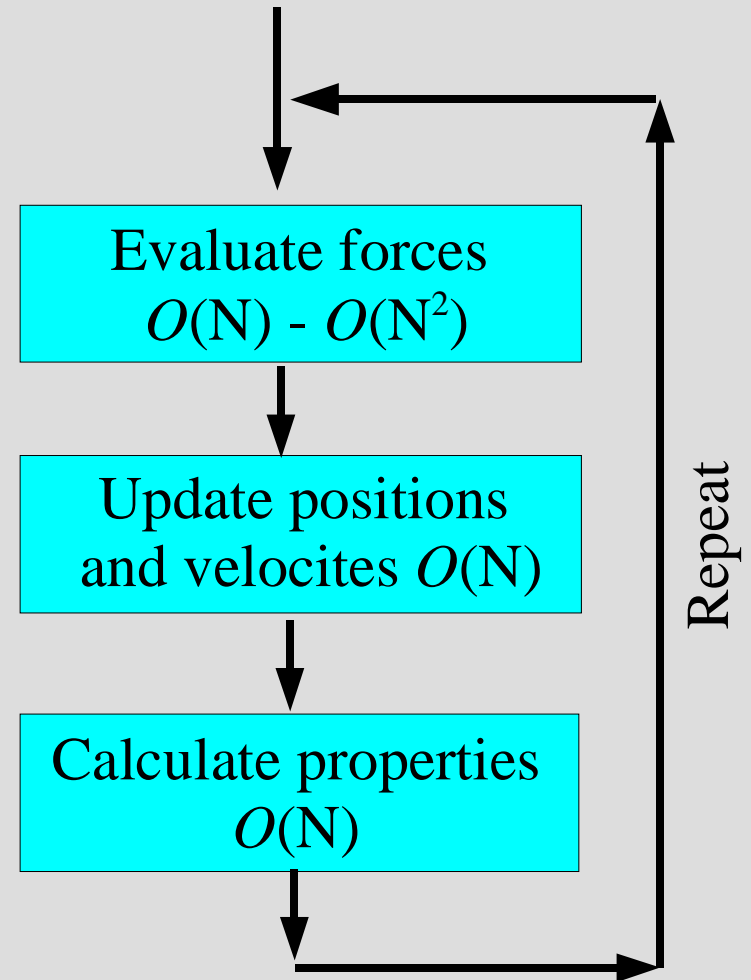
Molecular Dynamics

- Atoms interact via a long range pair potential
- Aim to follow motion of atoms over time
 - Integrating the equations of motion
- Biological simulations typically involve 20,000--100,000 atoms
 - Balance between wanting more atoms and wanting the simulation to correspond to a longer time period
- Integration timestep corresponds to $\approx 10^{-15}$ s
 - Longest current simulations $\approx 10^{-6}$ s or around 10^9 timesteps
 - Compare this to 1GHz clock speed = 10^9 cycles/sec \Rightarrow need to minimize cycles in each integration timestep!



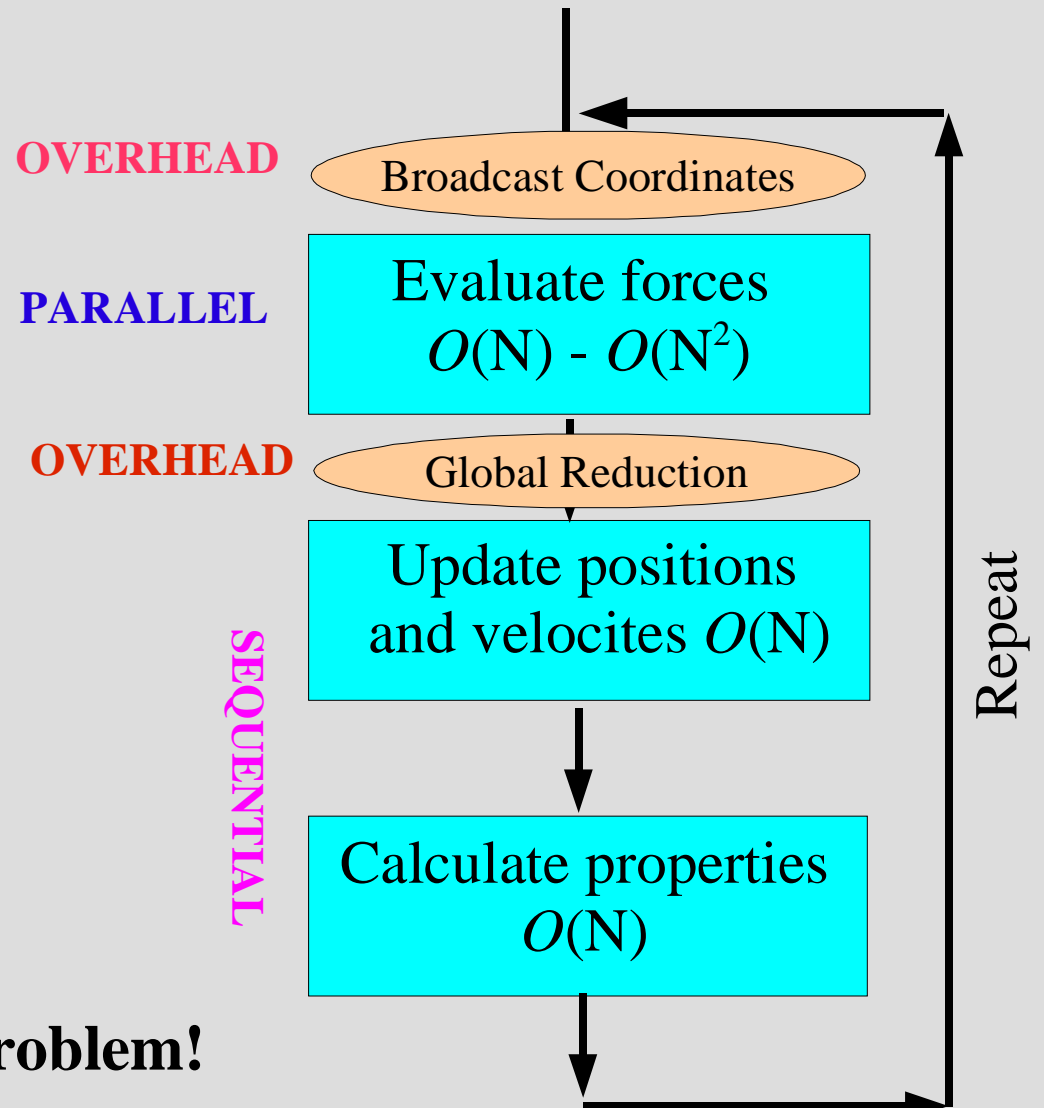
Typical Algorithm

- Each timestep involves force evaluation and updating of positions and velocities
- Force evaluation dominates
 - Potentially $O(N^2)$
 - Cutoffs using pairlists or other linear scaling techniques (Ewald summation or fast multipole methods) reduce this
- Enforced synchronization at each timestep
 - Computation time for each timestep must be short to perform many timesteps



Typical Parallel Molecular Dynamics

- Replicate $O(N)$ arrays
 - i.e. force, coordinates etc.
- Divide up force computation
 - Parallelizing integration step usually not worth it, especially on distributed memory
- Requires $O(N)$ communication

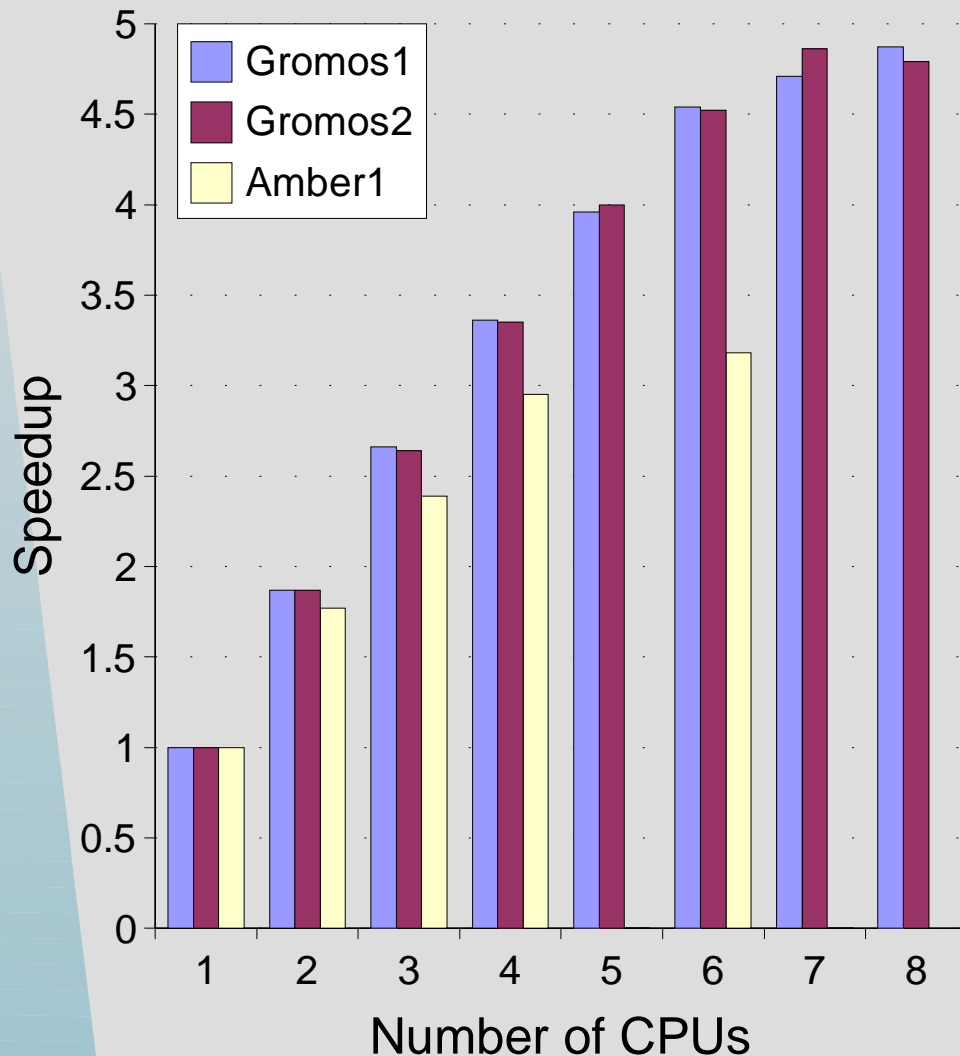


Much harder problem!

$O(N - N^2)$ computation $\approx O(N)$ communication



Parallel MD Results

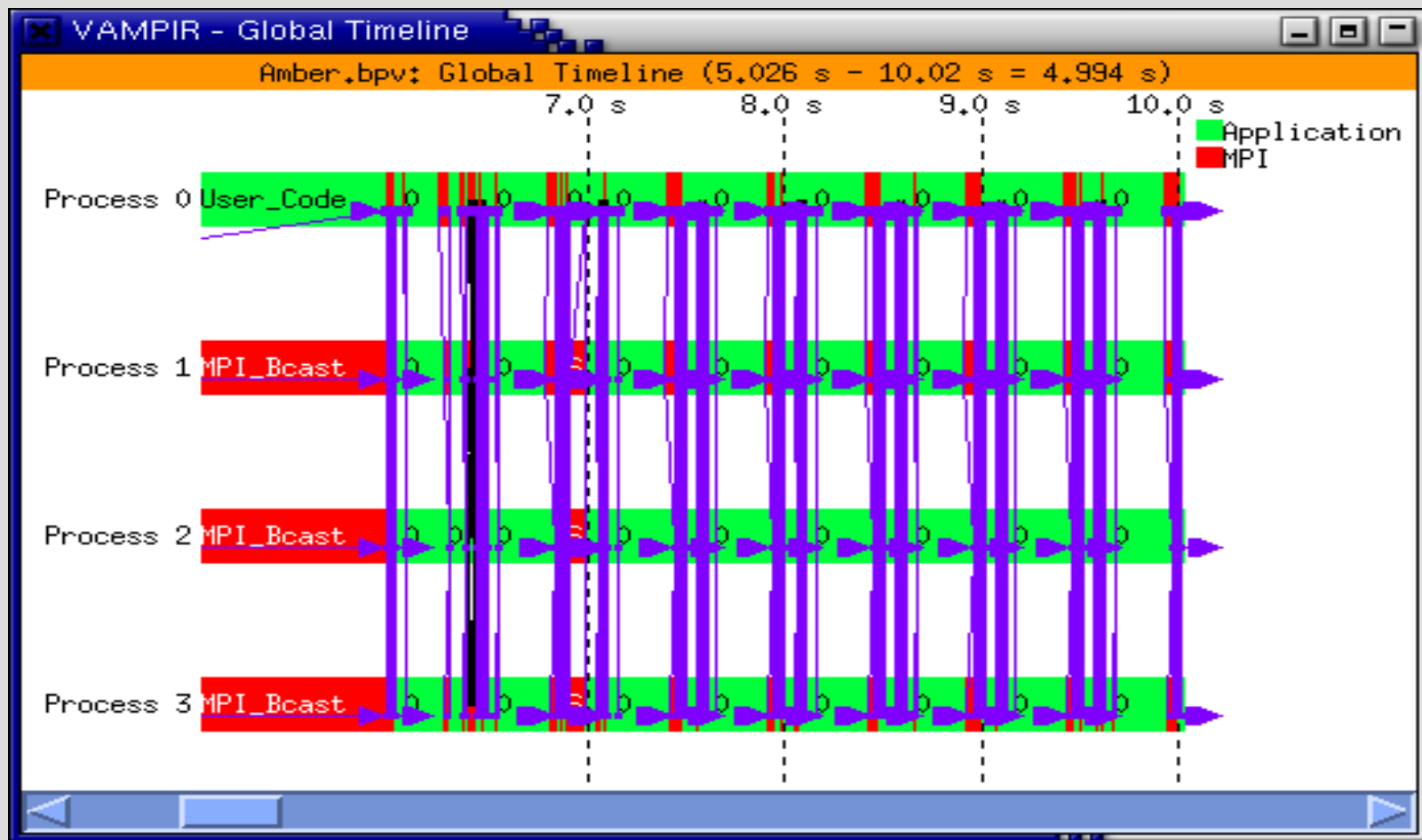


- GROMOS96, OpenMP implementation
 - (igc.ethz.ch/gromos)
- Amber6, MPI implementation
 - (sigyn.compchem.ucsf.edu/amber)
- GROMOS1:
 - ≈ 3000 atom protein with ≈ 5000 water molecules
 - ≈ 2 s/timestep on 400MHz Ultra II
- GROMOS2:
 - ≈ 14000 water molecules
 - ≈ 4 s/timestep on 400MHz Ultra II
- Amber1:
 - ≈ 12000 atom system
 - ≈ 0.1 s/timestep on Fujitsu VPP5K
- Work performed with T. Huber



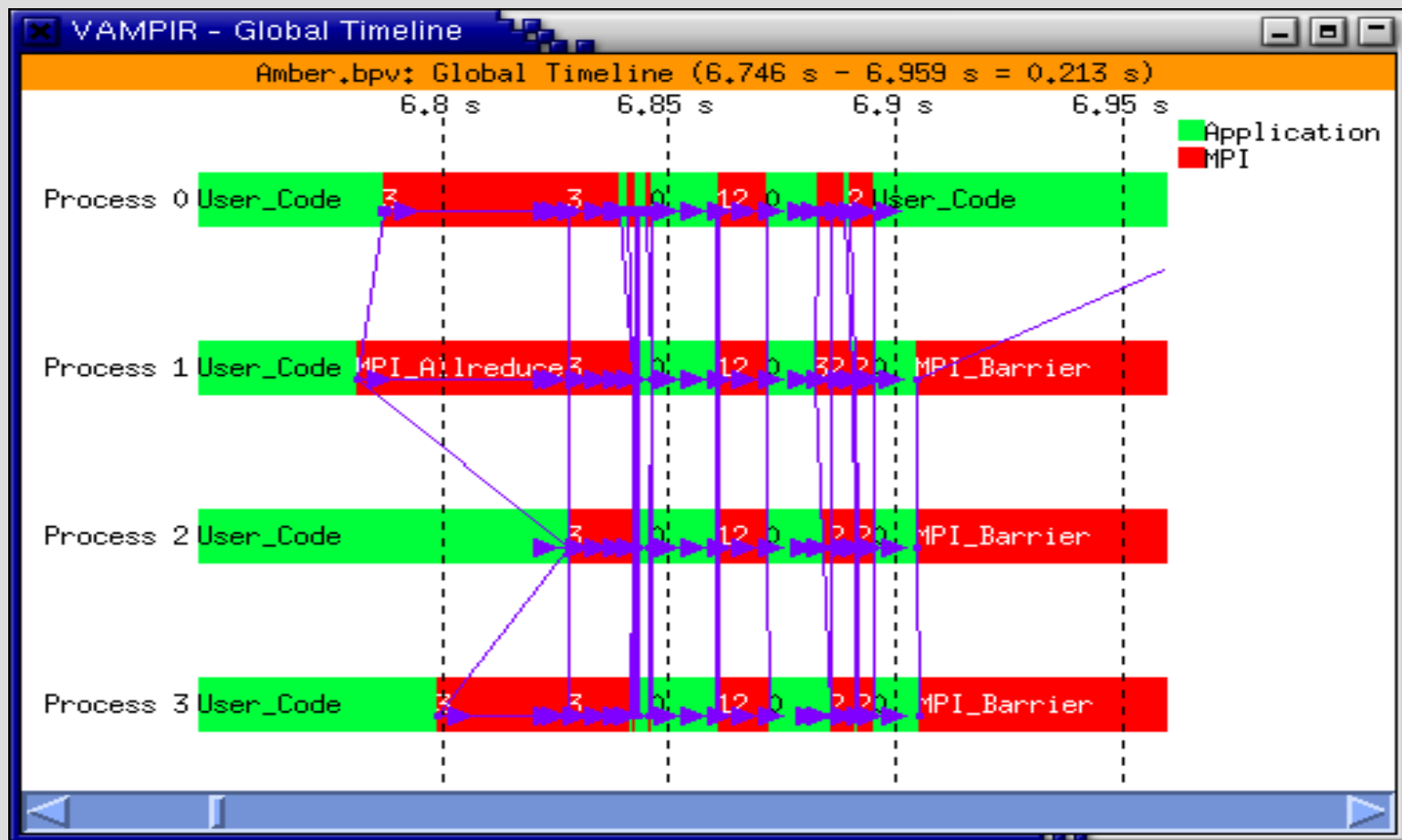
Amber Vampir Trace

- Illustrates initial broadcast of starting data
 - See www.pallas.com/e/products/vampir
- Each cluster of blue lines is one timestep



Amber Vampir Trace Closeup

- Shows load imbalance during timestep



Computational Chemistry Future

- Linear scaling quantum chemical algorithms for large systems!
- Algorithms are based on locality
 - Bonds, lone pairs etc.
- Why now
 - Increased computer speed enables access of the crossover point
- The problems
 - Data placement and retrieval
 - Load balancing
 - Computation and communication both $O(N)$
 - Software engineering issues



Conclusions (Single CPU)

- Analyse your code for a range of benchmarks
 - Usually requires knowledge of application domain
- Use profiling to identify what is important
 - Noting different profiles provide different information
- Use detailed timing and performance counters to assess performance relative to peak
- Compiler options can have a huge effect
 - Look at the flags the vendor uses for their SPEC benchmarks
- Cache blocking and loop unrolling for critical routines can be very important
- Tune for 1 CPU before parallelising your code!



Conclusions (Parallel)

- Exploiting multiple CPUs is hard
 - Must parallelize most of the code
 - Must minimize overheads
 - Must load balance your tasks
- Target large problems
- Consider which paradigm to use (merits to both)
 - Shared or distributed memory
 - OpenMP/MPI or other
- Code maintenance
 - Commercial applications need to minimize difference between multiple version of the same code, especially if code is actively being developed. This is much easier with OpenMP on shared memory, than MPI on distributed memory

