Parallel Computational Chemistry

Alistair Rendell
Department of Computer Science
Australian National University
(Alistair.Rendell@anu.edu.au)

International Conference on Computational Science

June 2-4, 2003

Co-located in Melbourne and St Petersburg

- Various workshops (Those in Melbourne)
- Computational Chemistry in 21st century
Outline

- Computational Chemistry
  - All you ever wanted to know in 2 slides!
- Parallel Preface
  - Understanding your code, computer, compiler
- Parallel Computational Chemistry
  - OpenMP (Gaussian)
  - Global Arrays (NWChem)
  - Charm++ (NAMD)
- Aim
  - To highlight some things that may be of interest to people outside the computational chemistry community

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 $\text{H}_2$, 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**

---

**Parallel Preface**

- **Performance benchmarking and tuning**
  - Understanding your code and its performance
  - Fundamental to both single and parallel processing
- **The balance between single processor performance tuning and parallelization**
  - Should you even begin to parallelize a code that runs at 1% of peak on a single CPU?
Your Objective

a. To run “that one remaining grand challenge calculation for the Nature paper”
   or
b. To improve general performance for many future calculations

We will initially consider the Gaussian computational chemistry code from the perspective of b and on a single CPU

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
**Issues**

- Require benchmark calculations 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 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**

<table>
<thead>
<tr>
<th></th>
<th>Theory</th>
<th>Type</th>
<th>Basis</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ethylene</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1 MP2</td>
<td>Gradient</td>
<td>6-311++G(3df,3pd)</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>1.2 MP2</td>
<td>Frequency</td>
<td>6-311++G(3df,3pd)</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td><strong>Isobutene</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 HF</td>
<td>Frequency</td>
<td>6-311++G**</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>2.2 BLYP</td>
<td>Frequency</td>
<td>6-311++G**</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>2.3 MP2</td>
<td>Energy</td>
<td>6-311++G**(3df,3pd)</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>2.4 MP2</td>
<td>Gradient</td>
<td>6-311++G**(3df,3pd)</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>2.5 QCISD(T)</td>
<td>Energy</td>
<td>6-311++G**</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td><strong>18-crown-6-ether</strong></td>
<td>3.1 HF</td>
<td>Gradient</td>
<td>6-31G**</td>
<td>390</td>
</tr>
<tr>
<td>3.2 BLYP</td>
<td>Gradient</td>
<td>6-31G**</td>
<td>390</td>
<td></td>
</tr>
<tr>
<td>3.3 MP2</td>
<td>Energy</td>
<td>6-31G**</td>
<td>390</td>
<td></td>
</tr>
</tbody>
</table>
The Molecules!

- Ethylene
- Isobutene
- 18-crown-6-ether

#1: Comparative Performance

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

<table>
<thead>
<tr>
<th></th>
<th>Performance Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A:B</td>
</tr>
<tr>
<td>Peak (theoretical)</td>
<td>1.5</td>
</tr>
<tr>
<td>SpecFP2000</td>
<td>0.9</td>
</tr>
<tr>
<td>Streams (Triad)</td>
<td>1.2</td>
</tr>
<tr>
<td>Ethylene</td>
<td>1.1</td>
</tr>
<tr>
<td>Isobutene</td>
<td>1.2</td>
</tr>
<tr>
<td>16-Crown-6-Ether</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
</tr>
</tbody>
</table>

- Spec and Streams see: http://www.spec.org
  http://www.cs.virginia.edu/stream
  (Talk at SC about combined)

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

- Similar comparison for parallel performance
  - Parallel may consider SpecOMP instead of SpecFP
#2: Timing and Profiling

- Different profilers give different information
- Gprof (all unix systems)
  - Must recompile with -pg
  - Resolution of timer can be poor (0.01s)
- Collector/Analyzer (Sun)
  - Does not require recompilation
  - Run program under control of collector
  - Does not provide calling statistics (number of times routine initiated)
- Profile plus information from calling structure can provide useful information for parallelization of an unknown code

### Sample Link Profiles

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

- Two different profilers
  - Same “link” and benchmark
- Two different machines
- Beware of library functions
  - May or may not be in profile
Deciding where to Start!

- The 10 benchmarks we obtain 39 profiles
- These links consume an aggregate time of 28637s

<table>
<thead>
<tr>
<th>Ordered by Total Time</th>
<th>Ordered by Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routine</td>
<td>Frequency</td>
</tr>
<tr>
<td>dgstr_</td>
<td>17</td>
</tr>
<tr>
<td>fqtril_</td>
<td>9</td>
</tr>
<tr>
<td>dovr1_</td>
<td>36</td>
</tr>
<tr>
<td>docont_</td>
<td>38</td>
</tr>
<tr>
<td>dotn_</td>
<td>36</td>
</tr>
<tr>
<td>tri3cz_</td>
<td>1</td>
</tr>
<tr>
<td>dgemm_lib</td>
<td>21</td>
</tr>
<tr>
<td>split01_</td>
<td>3</td>
</tr>
<tr>
<td>dosht_</td>
<td>38</td>
</tr>
<tr>
<td>dst01_</td>
<td>3</td>
</tr>
</tbody>
</table>

#3: Theoretical Analysis

One routine is dominated by

\[
\text{Do } I = 1, N \\
\text{Array1} (I) = \text{Array1} (I) + \text{Vector} (I) \times \text{Constant1} \\
\text{Array2} (I) = \text{Array2} (I) + \text{Vector} (I) \times \text{Constant2} \\
\text{Array3} (I) = \text{Array3} (I) + \text{Vector} (I) \times \text{Constant3} \\
\text{enddo}
\]

- Dominated by 3 daxpy operations
- Requires 4 loads and 3 store operations
  - Theoretical peak 6flops per 7 cycles (6/14 of 43% of peak on machine with single load/store pipe)
- Does the compiled code permit this performance?
  - Via compiler option or looking at assembler!
- What is the observed mflop rate?
### Examining Pipelining (Sun)

```
./f95 -fast -x prefetch=now -xtypemap=real:64,double:64,integer:64
-x target=native -xarch=v9 -xcache=generic -Q option cg
-Qms_pipe+info -S example.f
```

- Pipelined correctly
  - with prefetching drops to 1 result every 9 cycles

### #4: Detailed Timing Measurements

- Use fine grain timer and performance counters
  - Actually did this in the calling routine
- 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
endo

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

- See PAPI (performance application programming interface)
  [http://icl.cs.utk.edu/papi](http://icl.cs.utk.edu/papi)
**Performance Data**

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

- Measured time 7.12s,
  - cycle count ≈ 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 mflips
  - 241/1500 = 16% of peak (cf 43% theoretical max)
- Store operations are the bottleneck

---

**#5: Compiler Options**

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

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

So you’ve tuned your code and it’s running at 90% of peak on a single CPU!

…..
Then you must using the Earth Simulator!
See: [http://www.es.jamstem.go.jp](http://www.es.jamstem.go.jp)
and Gordon Bell awards at SC2002
[http://access.ncsa.uiuc.edu/Releases/11.21.02_SC2002_Con.html](http://access.ncsa.uiuc.edu/Releases/11.21.02_SC2002_Con.html)

Parallel Paradigms

- **Shared memory**
- **Distributed memory** (message passing)

![Diagram showing shared and distributed memory paradigms](image-url)
Paradigms and Chemistry

- **Shared memory (within an SC node)**
  - 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 (between SC nodes)**
  - MPI (e.g. Amber)
  - Distributed shared memory (e.g. Linda in Gaussian)
  - Global Arrays/DDI (e.g. NWChem and GAMESS)
  - Remote Method Invocation (e.g. Charm++/NAMD)

- **Data parallel languages (HPF, C*)**
  - Not aware of any widely used comp chem code
  - Note that **HPF is actively used on earth simulator**
OpenMP

- Forms the basis for the next release of the Gaussian Package on shared memory architectures
- I have direct interest in this through an ARC Linkage Grant with Gaussian Inc and Sun Microsystems

“Programming Paradigms, Tools and Algorithms for the Spectral Solution of the Electronic Schrödinger Equation on Non-Uniform Memory Parallel Processors”

Processes and Threads

- Threads share access to memory (or variables) allocated on the heap
What’s OpenMP

- OpenMP is to shared memory what MPI is to distributed memory
  - See http://www.openmp.org
- It is thread programming for dummies
  - Shared heap, thread private stack and program counter
- Consists of compiler directives, a limited number of library calls and some environment variables
  - In contrast to MPI and pthreads it requires compiler support

OpenMP Assumptions

- Multiple processing elements
  - Targeting simultaneous use of multiple processing elements, i.e. parallel rather than concurrent processing
- Shared flat address space
  - Symmetric multiprocessors (SMP)
- Multiple lightweight processes managed by the O/S
OpenMP History

- 97 – version 1 Fortran standard
- 98 – version 1 C and C++ standard
- 99 – version 1.1 fixes problems in Fortran version 1
- 00 – version 2.0 Fortran
  - Support for F90
  - New directives, COPYPRIVATE
  - New behavior for old directives, REDUCTION for arrays
- 02 – version 2.0 C and C++
- To date there has been a catch up between Fortran and C versions, in future this will be streamed

OpenMP Directives

```plaintext
#pragma omp directive-name [clause]
```

- **Parallel Region**
  - #pragma omp parallel [clause]
- **Work Sharing**
  - #pragma omp for [clause]
  - #pragma omp sections [clause]
- **Synchronization**
  - #pragma omp barrier
  - #pragma omp master
- **Data Environment**
  - #pragma omp threadprivate (list)

- Each directive refers to a structured block
- Parallel and some worksharing constructs can be combined
- Similar directives for Fortran (!$OMP)
OpenMP Directives Clauses

- shared(list)
- private(list)
- firstprivate(list) / lastprivate(list)
- default(private | shared | none)
- reduction (operator | intrinsic : list)
- copyin(list)
- if (expr)
- ordered
- schedule(type[, chunk])
- nowait

- Not applicable to every directive
- Some subtle differences between Fortran and C/C++

Runtime Library Routines

- void omp_set_num_threads (int)
- int omp_get_num_threads (void)
- int omp_get_max_threads (void)
- int omp_get_thread_num (void)
- int omp_get_num_procs (void)
- int omp_in_parallel (void)
- void omp_init_lock (omp_lock_t*)
- void omp_destroy_lock (omp_lock_t*)
- void omp_set_lock (omp_lock_t*)
- void omp_unset_lock (omp_lock_t*)
- void omp_test_lock (omp_lock_t*)

- Also environment variables
  - OMP_SCHEDULE
  - OMP_NUM_THREADS
  - OMP_DYNAMIC
  - OMP_NESTED
Example: Summing Integers

\[ SUM = \sum_{i=0}^{10000} i \]

//SEQUENTIAL CODE
#include <stdio.h>
define TOTALSUM 10000
int main()
  int i, n, sum;
  n=TOTALSUM;
  sum=0;
  for (i=0; i< n+1; i++) sum+=i;
  printf("Sum of %d integers = %d \n", n, sum);
}

MPI Code
#include <stdio.h>
#include <mpi.h>
define TOTALSUM 10000
#define NUMTHREADS 4 // Not determined by MPI code
int main( argc, argv )
  int argc;
  char **argv;
  {
    int rank, size, i, n, lsun=0, sum;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (!rank) n = TOTALSUM;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    for (i=rank; i< n+1; i+=size)lsun+=i;
    MPI_Allreduce(&sun, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    if (!rank) printf("Sum of %d integers = %d \n", n,sum);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize( );
    return 0;
  }
#include <stdio.h>
#include <pthread.h>
#define TOTALSUM 10000
#define NUM_THREADS 4
typedef struct { int id, nproc, maxn, lsum; } parm;

void *work(void *arg)
{   parm *p = (parm *) arg;
    int iam = p->id;
    int nproc = p->nproc;
    int n = p->maxn;
    int mysum=0,i;
    for (i=iam; i<n+1; i+=nproc) mysum+=i;
    p->lsum=mymum;
    return (0);
}

int main()
{   int i, n, sum=0;
    parm *arg;
    pthread_t *threads;
    n=TOTALSUM;
    threads = (pthread_t *) malloc(NUM_THREADS * sizeof(*threads));
    arg = (parm *) malloc(NUM_THREADS * sizeof(parm));
    /* Create threads */
    for (i = 0; i < NUM_THREADS; i++) {
      arg[i].id = i;
      arg[i].nproc = NUM_THREADS;
      arg[i].maxn = n;
      pthread_create(threads[i], NULL, work, (void *)(arg+i));
    }
    /* Join threads and sum totals */
    for (i = 0; i < NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
      sum+=arg[i].lsum;
    }
    printf("Sum of %d integers = %d \n", n,sum);
    return 0;
}

OpenMP Code

#include <stdio.h>
#include <omp.h>
#define TOTALSUM 10000
#define NUM_THREADS 4
int main()
{   int i, n, sum;
    n=TOTALSUM;
    omp_set_num_threads(NUM_THREADS);
    sum=0;
    #pragma omp parallel for shared(n) reduction(+:sum)
    for (i=0; i<n+1; i++) sum+=i;
    printf("Sum of %d integers = %d \n", n,sum);
}

- Just 3 new lines
OpenMP Pros/Cons

- Very easy to use!
- Minimal modifications from sequential code
  - Major software engineering advantage
- Compiler based
  - Emergence of hyperthreaded architectures promotes threading
- Scalability
  - Depends how you write your program (HPF + Earth Simulator)
  - Also note use of hybrid OpenMP/MPI
- Portability (just shared memory?)
  - SCORE has OpenMP for distributed memory
    See PC Cluster Consortium: http://www.pccluster.org
- Flat memory model
  - Various NUMA extensions have been proposed (SGI, Compaq, omni compiler)

Simple Example (SCF)

\[ F_{ij} = h_j + \sum_{kl}^{N_{functions}} D_{ij}(2[ij | kl] - [ik | jl]) \]

- \( F \) 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 \( O(N^2) \) 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 $Fs$ 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 $Fs$ 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) $>> O(N^2)$ communication

**Typical Performance**

![Graph showing typical performance](image)

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

• Concept that was driven by the computational chemistry group at Pacific Northwest National Laboratory
• Global Arrays or similar are at the heart of the NWChem, Molcas, Molpro and GAMESS-US chemistry codes
• I have developed coupled-cluster and perturbation theory algorithms using Global Arrays

Programming Trade-offs
Portability v Efficiency v Ease of Coding

• Shared memory
  – Greatly simplifies coding but portability and scalability an issue
  – Provides little control over data transfer costs
• MPI
  – Portable
  – Some applications are too complex to code while maintaining computational load balance and avoiding redundant computation
• Global Arrays (GA) and the closely related Distributed Data Interface (DDI) attempt to address these issues
  – For DDI see M.W. Schmidt et al, Comp Phys Com 128, 190 (2000)
Global Arrays

A portable “shared memory” programming model for distributed memory computers

- Combines the better features of message passing and shared memory
- Provides a portable interface through which each process in an MIMD parallel program can independently, asynchronously and efficiently access logical blocks of physically distributed matrices, with no need for the explicit cooperation of other processes
- Similar to shared memory, but like message passing it acknowledges that remote data transfers take longer than local ones

---

GA Schematic

- Assign some of each processor’s local memory to be use by GA to store physically distributed array quantities
- Each process can independently and asynchronously access patches of any “global array” without cooperation between processors (effectively GA provides remote memory read and write operations)
GA Programming Model

- MIMD parallelism using multiprocessor approach
  - All non-GA data, file descriptors etc are replicated or unique to each process
- Communication achieved by creating and accessing GA matrices augmented (if required) by message passing
- Access to GAs is via get, put, accumulate and get-and-increment operations
- Programmer assumes fast access to parts of a GA that are stored locally, and slower access to the remainder
- Target applications
  - Task parallel MIMD
  - Large distributed matrices
  - Wide variation in task execution time (load balancing)
  - Large ratio of compute to communicate

GA Functions #1

- Create (and destroy) an array controlling alignment and distribution
- Synchronize all processors
- Identify number of processors and my process ID
- Fetch, store and accumulate into a patch of an array
- Gather/scatter elements of a GA
- Atomic read and increment of an array element (shared counter)
- Inquire about location and distribution of an array
- Vector and matrix operations on entire GA (eg Array multiplication)
GA Functions #2

```c
ga_create(data_type, n, m, 'label' 10, 5, g_a)

ga_zero(g_a)

g_a_put(g_a, ilo, ihi, jlo, jhi, local, ldim)

g_a_acc(g_a, ilo, ihi, jlo, jhi, local, ldim, alpha)

g_a_add(alpha, g_a, beta, g_b, g_c)

g_a_distribute(g_a, iproc, ilo, ihi, jlo, jhi)
```

GA Implementation

- One-sided communication is the key requirement
- Four strategies used
  - Interrupt messages/active message (eg LAPI on SP2)
  - Shared memory primitives (eg SGI Origin)
  - Underlying hardware support (eg Shmem Compaq SC)
  - Data server model (eg network workstations)
- With some difficulty GA can be layered on top of the one-sided communication protocol in MPI-2
  - Although the one-sided communication model in MPI-2 is not to be recommended!!
Example use of GA

- Use in perturbation theory calculations
- Requires “integral transformation”

\[
[i | j] = \sum_{\alpha, \beta, \gamma, \delta}^{N_{\text{functions}}} C_{ia} C_{\alpha \beta} C_{j\gamma} C_{\beta \delta} [\alpha \beta | \gamma \delta] \quad \forall i, j, a, b
\]

\[
N_i = N_j \approx \frac{1}{10} \quad N_a = \frac{1}{10} \quad N_b \approx \frac{1}{10} \quad N_{\text{functions}}
\]

- Driven by computation of batches of \([\alpha \beta | \gamma \delta]\)
  - Parallel but requires load balancing
  - Computational cost scales as \(O(N^3)\)
- \(C_{ia}\) are replicated
- \([ia | jb]\) and intermediates are stored as distributed global arrays into which we accumulate

Morphine Integral Transformation

- Superlinear speedup!!
- Results obtained on Intel Delta
- Code now part of NWChem package
NAMD/Charm++

- NAMD is a parallel, object-oriented molecular dynamics program
  - [http://www.ks.uiuc.edu/Research/namd/namd.html](http://www.ks.uiuc.edu/Research/namd/namd.html)
- Employs the prioritized message-driven execution capabilities of the Charm++ parallel runtime system
  - [http://charm.cs.uiuc.edu](http://charm.cs.uiuc.edu)
- Won a Gordon Bell award at SC02
  - [http://access.ncsa.uiuc.edu/Releases/11.21.02_SC2002_Con.html](http://access.ncsa.uiuc.edu/Releases/11.21.02_SC2002_Con.html)

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 $\approx 10^{-15}$ sec
  - Longest current simulations $= 10^6$ sec or around 109 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

\[
O(N - N^2) \text{ computation} \approx O(N) \text{ communication}
\]
Parallel MD Results

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

The Charm++ Approach

- Decompose the problem into objects
- Objects can only access their own local memory
- Access to other data is only possible via asynchronous method invocation to other objects
- Data driven approach mean that while an object is waiting for incoming data other objects execute
- Objects can migrate from processor to processor at runtime under control of Charm++ loadbalancer
  - Ideal for iterative schemes where information from early iterations can adjust later iterations
NAMD

- Modern C++ code
- Decomposes problem domain into cubes based upon cutoff radius used in calculation
- Objects correspond to force calculations between or within cubes
- Interactions are further partition by type and to increase the number of objects they may be further split into groups
  - Aim to have many more objects than processors
- Load balancing performed after running few hundred initial timesteps
- Optimised for to use low level Elan communications on the Pittsburgh Compaq SC

NAMD Performance

<table>
<thead>
<tr>
<th>Processors</th>
<th>Total</th>
<th>Per Node</th>
<th>Time/sec</th>
<th>Speedup</th>
<th>Gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7080</td>
<td>1</td>
<td>0.497</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>4</td>
<td>62.5</td>
<td>113</td>
<td>56</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>4</td>
<td>37.4</td>
<td>189</td>
<td>94</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>4</td>
<td>21.0</td>
<td>337</td>
<td>167</td>
</tr>
<tr>
<td>510</td>
<td>3</td>
<td>3</td>
<td>20.5</td>
<td>345</td>
<td>171</td>
</tr>
<tr>
<td>1024</td>
<td>4</td>
<td>4</td>
<td>18.2</td>
<td>389</td>
<td>193</td>
</tr>
<tr>
<td>1023</td>
<td>3</td>
<td>3</td>
<td>13.8</td>
<td>513</td>
<td>254</td>
</tr>
<tr>
<td>1536</td>
<td>4</td>
<td>4</td>
<td>16.6</td>
<td>427</td>
<td>212</td>
</tr>
<tr>
<td>1536</td>
<td>3</td>
<td>3</td>
<td>11.2</td>
<td>632</td>
<td>312</td>
</tr>
<tr>
<td>2250</td>
<td>3</td>
<td>3</td>
<td>11.2</td>
<td>632</td>
<td>312</td>
</tr>
</tbody>
</table>

- Cutoff based simulation with 92K atoms
  - J.C. Phillips et al, SC02 paper
- Impressive performance, note
  - Measure Gflops as well as speedup
  - Use of 3 CPU per node best!
Conclusions

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

Questions

Email Alistair.Rendell@anu.edu.au