Improved Performance
Scaling on HPCx

Performance Analysis
Overview

• Introduction and motivation

• Timers

• Hardware event counters

• Profiling tools

• Communication tools
• Our aim
  - To make effective use of current HPC architectures (i.e. HPCx)

• Hence
  - Must achieve the best performance from our applications
    • Both serial and parallel performance

• Only possible through
  - Performance analysis and optimisation

• This tutorial
  - Particularly interested in parallel performance
Purpose of Performance Analysis

• Goals
  - To reduce the execution time of the code
  - To achieve good scaling to large numbers of processors
  - To reduce memory/disk usage?
• Requires identification of bottlenecks before optimisation
  - Components of the code which take a large amount of time to execute
  - Pieces of code that make inefficient use of underlying architecture
Typical Bottlenecks

• Your application
  - Synchronisation, load balance, communication, memory usage, IO usage
• The architecture
  - Memory hierarchy, network bandwidth and latency, the processor architecture, IO system setup
• Software
  - Compiler options, libraries, communication protocols, operating system
• To name but a few...
Performance Analysis Process

- This is an iterative process
- Involves the use of tools to
  - Instrument code, measure and analyse performance metrics
Tools on HPCx

Interval timers
- Elapsed time between two timer calls
  - gettimeofday, irtc, system_clock

Event counters
- Counts number of times hardware events occur
  - HPMcount

Profiling tools
- Periodically sampling the program counter
  - gprof and xprofiler

Event tracing
- Complete sequence of events
  - MPITrace, Vampir
Interval Timers

- Wide range of timers available on HPCx
  - Varying precision, portability, language, ease of use
- Useful for determining components of the code that take a large amount of time
- Problems
  - Time consuming to instrument code
  - Parallel applications can generate vast amounts of data
    - Timing results per processor, thousands of processors...
## Timers on HPCx

<table>
<thead>
<tr>
<th>Timer</th>
<th>Usage</th>
<th>Wallclock/CPU</th>
<th>Resolution</th>
<th>Language</th>
<th>Portable</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>shell</td>
<td>both</td>
<td>centisecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>timex</td>
<td>shell</td>
<td>both</td>
<td>centisecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>read_real_time</td>
<td>subroutine</td>
<td>wallclock</td>
<td>nanosecond</td>
<td>C/C++</td>
<td>no</td>
</tr>
<tr>
<td>rtc</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>irts</td>
<td>subroutine</td>
<td>wallclock</td>
<td>nanosecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>dtime_</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>etime_</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>mclock</td>
<td>subroutine</td>
<td>CPU</td>
<td>centisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>timef</td>
<td>subroutine</td>
<td>wallclock</td>
<td>millisecond</td>
<td>F</td>
<td>no</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td>subroutine</td>
<td>wallclock</td>
<td>microsecond</td>
<td>F, C/C++</td>
<td>yes</td>
</tr>
<tr>
<td>system_clock</td>
<td>subroutine</td>
<td>wallclock</td>
<td>centisecond</td>
<td>F</td>
<td>yes</td>
</tr>
</tbody>
</table>
Profiling Tools on HPCx

• prof and gprof
  - Standard unix commands provide simple profiling at the subroutine level

• xprofiler
  - Simple profiler for both serial and parallel applications, GUI alternative to gprof

• Both useful for determining components of the code that take a large amount of time
  - Less time consuming for the programmer

• Problems
  - Intrusive - may increase overall execution time
  - Only profiles CPU (busy) usage, lack of I/O and communication information
Profiling - XProfiler

HPCx
CAPABILITY COMPUTING

Function Menu

Statistics Report

Function Level Statistics Report

Source Code for forces

```
subroutine forces(npot, x, f, vir, eqot, sides, reoff)
  implicit double precision (a-h, o-z)
  dimension x(npot, 3), #npot, 3)  
  c
  compute forces and accumulate the virial and potential,

!OMP SINGLE
  vir = 0.040
  eqot = 0.040
!OMP END SINGLE
  sideh = 0.5*sideh
  reoffs = reoffs+reoffs

!OMP DO PRIVATE(j,i,yi,zi,fxi,fiy,exi,xx,yy)
  !OMP8 ss, rd, rrd, rrd2, rrd3, rrd4, rrd6, rrd7, rrd8, forcem, forceuy, forces)
  !OMP8 REDUCTION(+:vir), REDUCTION(+:eqot),
  !OMP8 SCHEDULE (STATIC, 16)
  do 170 i = 1, npart
  170  xi = x(i,1)
  180  yi = x(i,2)
```

Search Engine: (regular expressions supported)

Forces
Event Counters

- **Hardware Performance Monitor Toolkit (HPM)**
  - hpmcount, libhpm, hpmviz

- hpmcount provides performance details
  - Execution wall clock time
  - Hardware performance counter information (e.g. cache misses)
  - Derived hardware metrics (e.g. loads per cache miss)
  - Resource utilisation statistics (e.g. no of page faults)
Event Tracing

- Interval timers, event counters and profiling tools are all useful for serial and parallel performance analysis
  - Have reviewed previously on the performance optimisation course

- Event Tracing tools on HPCx
  - MPITrace
  - Vampir

- Particularly useful for analysing scaling
  - Detailed information about communication time
MPITrace

• Allows low-overhead MPI elapsed-time measurements
  - IBM specific
  - Text based
• Link special library in at compile time
  -L/usr/local/lib -lmpitrace
• Run the application as usual
  - Information written to different file for each rank: mpi_profile.X
• Simple and easy to use
• Awkward to produce collective processor information e.g. averages over processors
Event Tracing - Vampir

- **VAMPIR (Visualization and Analysis of MPI Programs)**
  - Displays timeline of code execution and timing statistics
  - Allows communications in an MPI code to be visualised
  - Third party application - PALLAS (http://www.pallas.com/)
  - Portable

- **Vampirtrace (VT) library**
  - A library which produces a tracefile containing information about MPI communications and timings

- **Vampir**
  - A graphical user interface for visualisation of the tracefile generated by Vampirtrace
Vampir on HPCx

- **Setup environment**
  
  ```
  export PAL_ROOT=/usr/local/packages/vampir/
  export VT_ROOT=/usr/local/packages/vampir/
  export PAL_LICENSEFILE=$PAL_ROOT/etc/license.dat
  export PATH=$PATH:$PAL_ROOT/bin
  ```

  - This must be done in the Loadleveler script as well as in the interactive shell!

- **Compilation**
  
  ```
  mpxlf90_r -O3 -qsuffix=f=f90 -o example example.f90
  -L$PAL_ROOT/lib -lVT -lld
  ```

- **Execution**
  
  ```
  poe ./example
  ```

  - creates file: example.stf
  - View this using: vampir
Case Study – H2MOL

• The Code
  - From the Multiphoton and Electron Collision Consortium (CCP2)
  - Written by Ken Taylor & Daniel Dundas, Queens University Belfast
  - Solves the time-dependent Schrodinger equation

• Parallelisation
  - The Z domain of the grid is distributed amongst an array of processors
  - The code specifies a constant number of grid points per processor in the Z-direction
  - Hence perfect scaling would be represented by a flat timing profile across the different processor counts
• Interesting communication pattern due to:
  • A series of Bsend and receive operations
    - Overlapped with some computation
  • Due to symmetry we need only store the lower half of the grid: needs less processors

```
6 7 8
3 4 5
0 1 2
```

→
```
2 4 5
1 3 4
0 1 2
```

• For example – send to the right
  - no periodic boundary conditions
  - 0 → 1, 1 → 2, 3 → 4 and 1 → 3, 2 → 4, 4 → 5
Scaling

Number of Processors

Time (seconds)

Phase 1
Phase 2 - Large Pages
Phase 2 - Small Pages
Phase 2 - Small Pages, new switch
**Communication Time - MPITrace**

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Bsend</td>
<td>1400</td>
<td>2376000.0</td>
<td>1.249</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>840</td>
<td>2376000.0</td>
<td>7.996</td>
</tr>
<tr>
<td>MPI_Buffer_attach</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>8</td>
<td>3088.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>437</td>
<td>0.0</td>
<td>2.247</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>415</td>
<td>15.5</td>
<td>0.027</td>
</tr>
</tbody>
</table>

**total communication time** = 11.520 seconds.
**total elapsed time** = 112.998 seconds.
**user cpu time** = 108.970 seconds.
**system time** = 1.050 seconds.
**maximum memory size** = 803344 KBytes.
### Communication Time - MPITrace

**Message size distributions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bsend</td>
<td>1400</td>
<td>2376000.0</td>
<td>1.249</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>840</td>
<td>2376000.0</td>
<td>7.996</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>1</td>
<td>88.0</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>240.0</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>360.0</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1936.0</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7200.0</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>14400.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>24</td>
<td>8.0</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>391</td>
<td>16.0</td>
<td>0.026</td>
</tr>
</tbody>
</table>
Communication Time – Load Imbalance
Communication Time - MPI_Bsend

Excessive time spent in MPI_Recv
Communication Time - MPI_Bsend,
MP_CSS_INTERRUPT
Excessive time spent in MPI_Recv
Communication Time - MPI_Isend
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proc = 6</td>
</tr>
<tr>
<td>MPI_Bsend, MPI_Recv</td>
<td>144</td>
</tr>
<tr>
<td>MPI_Bsend, MPI_Recv, MP_CSS_INTERRUPT</td>
<td>135</td>
</tr>
<tr>
<td>MPI_Send, MPI_Recv</td>
<td>135</td>
</tr>
<tr>
<td>MPI_Send, MPI_Recv, MP_CSS_INTERRUPT</td>
<td>134</td>
</tr>
<tr>
<td>MPI_Isend, MPI_Irecv</td>
<td>131</td>
</tr>
<tr>
<td>MPI_Isend, MPI_Irecv, MP_CSS_INTERRUPT</td>
<td>133</td>
</tr>
</tbody>
</table>
H2MOL Conclusions

• Load imbalance due to processor allocation on MCMs

• MPI_BSend results in excessive amounts of time spent in MPI_Recv
  - MP_CSS_INTERRUPT fixes this problem on small processor numbers

• Using MPI_ISend and MPI_IRecv allows overlapping of communications
  - Without excessive time spent in the receive call
  - Shows slight performance improvement
Conclusions

- **MPITrace**
  - Demonstrates scaling issues are due to communication
  - Highlights `MPIRecv` and `MPIBarrier` as most expensive routines
- **VAMPIR**
  - Provides detailed view of individual communications
  - Shows complexity of communication patterns in a highly visual way
Summary

• Performance analysis tools
  - Help identify bottlenecks for optimisation

• HPCx has a range of tools to help you analyse the performance of your code
  - Timers, xprofiler, hpmcount
  - Vampir, MPITrace

• For constellation systems
  - Vampir and MPITrace help identify communication issues
  - E.g. slower MPI operations between nodes