Performance Optimisation of an Environmental Modelling Code (POLCOMS)

Michael Holden
EPCC
The University of Edinburgh
Mayfield Road
Edinburgh
EH9 3JZ
United Kingdom

February 2005

Abstract

POLCOMS is a computationally intensive application that is used for modelling many aspects of coastal environments. The application is written in Fortran and has been modified to run as a parallel code; on HPCx this achieved by means of the MPI message passing library. The potentially lengthy run times for POLCOMS require that the code executes in an efficient manner to ensure as rapid a run time as possible. The POLCOMS code execution characteristics were analysed and some putative code optimisations were investigated. The results are presented here.

This is a Technical Report from the HPCx Consortium.

{Report available from http://www.hpcx.ac.uk/research/publications/HPCxTR0502.pdf}

© HPCx UoE Ltd 2005
Neither HPCx UoE Ltd nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.
Acknowledgements

Many thanks to Mike Ashworth of Daresbury Laboratory for his help during the familiarisation phase of this project and for his ongoing support and cooperation thereafter.

More locally, thanks to Lorna Smith and Mark Bull, both of EPCC, for their generous contributions of time, ideas and expertise which were most welcome.
1. Introduction

The Proudman Oceanographic Laboratory Coastal Ocean Modelling System (POLCOMS) is an application for modelling coastal shelf environments. It can simulate both fluid behaviour and a number of eco-system properties. For these benchmark tests POLCOMS was run with a 12km grid of size 200 x 200 x 34. The application software is written in Fortran and runs on the HPCx system. The code has been extended from a serial code to a parallel code through the use of the MPI library of message passing routines. There are a number of sources of technical information for HPCx and the POLCOMS application [1].

The objectives of the POLCOMS investigation were to gain insights into the following areas:

- **Scaling:** Investigate the performance of POLCOMS when it is run on a range of processor numbers (section 2).

- **Performance Analysis & Profiling:** Where the application uses significant resources and how effectively they are being used. How efficient is the implementation of the application code (section 3).

- **Optimisation options:** A discussion of possible methods of improving the performance of the POLCOMS application (section 4).

The deliverables from this exercise will be:

- This report: the report contents will outline the findings from the investigation areas listed above,

- Modified source code: source code modules containing code that has been shown to improve POLCOMS execution times.

The simulated time for which the POLCOMS program runs can be specified in two ways; by means of time steps or by means of hours. 24 simulated hours is equivalent to 216 time steps [2]; the baroclinic timestep is 400s. These options are listed in Table 1-1.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nsteps=n</td>
<td>Simulation steps e.g. 10</td>
</tr>
<tr>
<td>-tdur=h</td>
<td>Simulation time duration (hours) e.g. 24.0</td>
</tr>
</tbody>
</table>

Table 1-1: POLCOMS simulated time options
Currently the POLCOMS software is built using the compiler options listed in Table 1-2.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function [3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>-q64</td>
<td>64 bit compilation.</td>
</tr>
<tr>
<td>-O3</td>
<td>Optimisation level.</td>
</tr>
<tr>
<td></td>
<td>-O0 = No optimisation.</td>
</tr>
<tr>
<td></td>
<td>-O5 = Most aggressive optimisation.</td>
</tr>
<tr>
<td>-qarch=pwr4</td>
<td>Specifies what types of RISC systems the application program may be executed on. Produces an object that contains instructions that run on the POWER4 hardware platforms.</td>
</tr>
<tr>
<td>-qtune=pwr4</td>
<td>Specifies the architecture on which the executable program is optimized. Produces an object optimized for the POWER4 hardware platforms.</td>
</tr>
<tr>
<td>-qhot</td>
<td>Determines whether or not to perform high-order transformations on loops during optimization.</td>
</tr>
<tr>
<td>-qs suppress=1500-036</td>
<td>Suppresses generation in the output stream and in the listing file of the compiler [informational] message(s) indicated: 1500-036 (I) Optimization level 3 has the potential to alter the semantics of a program. Please refer to documentation on -O3 and the STRICT option for more information.</td>
</tr>
</tbody>
</table>

Table 1-2: POLCOMS current compilation options
2. Scaling

The elapsed times for a number of program runs were noted using POLCOMS’ own timing information. The program also collects timing information for individual functional components within the program. Timings were noted for the POLCOMS program run on a range of processor numbers for a simulated duration of 24 hours. The program was built using the delivered compilation options (see Table 1-2).

A performance metric for the whole POLCOMS program in terms of simulated days per day is shown in Figure 2-1.

![Figure 2-1: POLCOMS Simulated Days per Day](image)

Parallel efficiency provides an indication of the effectiveness with which an application is making use of extra processor resources. Ideal parallel efficiency is represented by a value of 100%. Parallel efficiency values for the POLCOMS program are shown in Figure 2-2.
Figure 2-2: POLCOMS Parallel Efficiency (tdur=24.0)
3. Performance Analysis & Profiling

Performance timings were collected for the POLCOMS program run on 32 processors over a range of time steps. The program was built using the delivered compilation options (see Table 1-2). In addition to overall program run times the POLCOMS code also produces timing information for its constituent functional components. A number of runs were executed for each simulated period to ensure that a single value would not give an unrepresentative timing. Unrepresentative timings might arise from causes such as variations in overall machine activity or background system activity.

Run times for POLCOMS’ functional components as a percentage of the overall program run time indicate which parts of the code are the most expensive time consumers. This also indicates which components of the application might be the most rewarding when investigating opportunities for optimizing the application’s performance. Run time percentages are shown in Figure 3-1.

For large numbers of time steps the portion of time required by each functional component becomes nearly constant. As the period of simulated time increases it becomes apparent that three functional components are the major time consumers namely “Vel. adv”, “Sca. adv” and BAROC (see [1] for functional details of POLCOMS).
Xprofiler was used to gather information on the time spent within each POLCOMS subroutine. For this, POLCOMS was run for a simulated duration of 120 time steps on 32 processors. The run time was approximately 30 seconds. The extract from the Xprofiler generated report shown in Figure 3-2 indicates which Fortran subroutines were responsible for most program execution time.

<table>
<thead>
<tr>
<th>Time</th>
<th>cumulative</th>
<th>self</th>
<th>calls</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.4</td>
<td>135.42</td>
<td>135.42</td>
<td>3840</td>
<td>30.41</td>
<td>35.67</td>
<td>_vrec_GP [?]</td>
</tr>
<tr>
<td>10.9</td>
<td>475.24</td>
<td>110.41</td>
<td>7680</td>
<td>14.38</td>
<td>14.49</td>
<td>adypsi [9]</td>
</tr>
<tr>
<td>10.4</td>
<td>580.09</td>
<td>104.85</td>
<td>7680</td>
<td>13.65</td>
<td>13.67</td>
<td>adypsh [10]</td>
</tr>
<tr>
<td>10.3</td>
<td>684.82</td>
<td>104.73</td>
<td>7680</td>
<td>13.64</td>
<td>13.66</td>
<td>adypshu [11]</td>
</tr>
<tr>
<td>3.8</td>
<td>723.65</td>
<td>38.83</td>
<td>3840</td>
<td>10.11</td>
<td>10.84</td>
<td>barot [13]</td>
</tr>
<tr>
<td>3.0</td>
<td>754.31</td>
<td>30.66</td>
<td></td>
<td></td>
<td></td>
<td>_lapi_shm_dispatcher [14]</td>
</tr>
<tr>
<td>2.7</td>
<td>782.06</td>
<td>27.75</td>
<td>23688000</td>
<td>0.00</td>
<td>0.00</td>
<td>_trigd  [15]</td>
</tr>
<tr>
<td>2.3</td>
<td>806.41</td>
<td>23.35</td>
<td>15456</td>
<td>1.81</td>
<td>1.81</td>
<td>_hsgl [16]</td>
</tr>
<tr>
<td>1.4</td>
<td>819.51</td>
<td>14.10</td>
<td>7680</td>
<td>1.84</td>
<td>3.28</td>
<td>diffusion [16]</td>
</tr>
<tr>
<td>1.2</td>
<td>831.94</td>
<td>12.43</td>
<td>7680</td>
<td>1.62</td>
<td>3.07</td>
<td>_diffuseb [17]</td>
</tr>
<tr>
<td>0.9</td>
<td>841.20</td>
<td>9.26</td>
<td></td>
<td></td>
<td></td>
<td>_barrier_shm_aig [22]</td>
</tr>
</tbody>
</table>

**Figure 3-2: POLCOMS CPU usage from Xprofiler (extract)**

Xprofiler also indicates where these ‘busy’ subroutines fit into the POLCOMS’ program procedure hierarchy. The call totals are summed across all 32 processors. A call total of 3840 represents one call per timestep per processor (120x32); similarly a call total of 7680 represents two calls per timestep per processor. The named procedure _vrec_GP is not application software; it originates from compiler generated calls to a vector reciprocal procedure from the MASS library. The _qhot compiler option allows the compiler to transform loops so that expensive divides can be calculated as multiply by reciprocals. This process makes use of software that can be pipelined rather than non-pipelined hardware operations. The transformations that might be implemented are illustrated in Figure 3-3; the transformed code example is representative only and not exactly how the compiler might function.

```
... 
! Pre-compute the denominator.
  do i = 1,n
    tmp(i) = y(i)+z(i)
  enddo

! Calculate reciprocals.
  call vrec_GP (tmp, rtmp, n)

! Expensive non-pipelined ! divisions to calculate x.
  do i = 1,n
    x(i) = b(i)/(y(i)+z(i))
  enddo

... 

! Multiply by reciprocal which ! can be pipelined.
  do i = 1,n
    x(i) = b(i)*rtmp(i)
  enddo

... 
```

**Figure 3-3: Vector reciprocal transformation**
The call to \texttt{vrec\_GP} will have a start-up overhead but for a sufficiently large number of reciprocal calculations it should perform more efficiently than the equivalent divides executed within a loop. As will be discussed in section 4, the loop length in the \texttt{adv*} routines (and many other routines) may not be long enough for this to be a suitable approach.

It can be seen that the functional components that have the longer run times from Figure 3-1 i.e. “Vel. Adv”, “Sca. Adv” and “BAROC” contain the subroutines responsible for most elapsed time i.e. \texttt{pgrad\_spline, advpuv, advpbv, advpbu} and \texttt{advpuu}. This is shown in Figure 3-4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{polcoms_structure.png}
\caption{POLCOMS structure from Xprofiler (extract)}
\end{figure}

The Xprofiler program hierarchy output for the BAROC functional component is shown in Figure 3-5. This also shows the \texttt{tridag} procedure which is one of several that are called many millions of times; \texttt{tridag} is called once for each modelling point by the \texttt{diffuse*} routines.
Xprofiler output also indicated that some routines were called millions of times and in some cases tens of millions of times. The most frequently called routines are listed in Table 3-1. For some small size functions the overhead for frequent invocation can sometimes be removed by in-lining the separate procedure; that is the code is merged into the calling function removing the overhead associated with calling the function. The effect of forcing the in-lining of small size but frequently called procedures is discussed in section 4.

Good load balance is essential for good parallel performance. Good load balance will occur when all processors complete the work assigned to them in equal times. An unbalanced work load will result in processors standing idle and this processor time being wasted. An estimate of the load imbalance for each functional component is calculated by POLCOMS as a function of average and maximum time. The estimates include MPI wait times so will contain some inaccuracy. The formula which is taken from subroutine tstats.F is shown in Equation 3-1.

\[
\text{Percent imbalance} = \left(1.0 - \frac{\text{AverageTime}}{\text{MaximumTime}}\right) \times 100.0
\]

Equation 3-1: POLCOMS processor time imbalance calculation
Ideal load balance occurs when the average time equals the maximum time and hence the percentage imbalance is zero. The calculated load imbalance is never very high for the significant components of the timed problem. For example, “Sca. adv”, “Vel. adv” and BAROC typically have calculated load imbalances in the range of zero to three per-cent. The functional components “Turbo. Mo” and BAROT have calculated load imbalances of up to thirteen per-cent, however, their time consumption is far less significant and hence so is the impact of the greater load imbalance. The Initialisation and “End Step” functional components have a negligible calculated load imbalance. The Boundary functional component has a much higher calculated load imbalance, sometimes as high as 25%, but uses such an insignificant part of the execution time that this is highly unlikely to provide any opportunity for performance optimisation.

To summarise, the functional components that use most time seem to be well load balanced. Functional components with poor load balancing are not responsible for significant amount of execution time.

VAMPIR is a third party profiling tool that allows inter-processor communications to be analysed and presented visually [4]; for the POLCOMS application this will permit the usage of MPI library routines to be analysed. The output from VAMPIR can indicate the fraction of POLCOMS’ execution time consumed by communications; high parallel efficiency requires a high ratio of computation to communication. Another factor that will help, but not guarantee, good parallel performance is for the computation and communication to be evenly spread across all processors; a single processor with significantly higher communication time might indicate long waiting times for communications to complete resulting in poorer performance.

VAMPIR output can also display the interleaving, if any, of communication and computation; overlapping of communication with computation is another factor which helps to boost parallel performance.

POLCOMS was executed on eight processors for a simulated duration of 24 hours with VAMPIR being used to collect data on MPI communication activity. Analysis of VAMPIR output for the whole POLCOMS program indicates that the application spends far more time in computational code than it does in communication code. Of a total run time of over four minutes, less than ten seconds was spent executing MPI communications routines. These ten seconds are less than four percent of the total POLCOMS run time. This can be seen from the VAMPIR visual display shown in Figure 3-6.

If the proportions of time spent in computation and communication for each processor are examined individually it can be seen that each processor spends a similar amount of time performing communication activities. There is no major imbalance in the amount of time spent performing communication or computation. This is illustrated in the VAMPIR output shown in Figure 3-7.
Figure 3-6: POLCOMS computational and communication time

Figure 3-7: POLCOMS' processors computational and communication time
hpmcount is an IBM software profiling tool that collects run time statistics when an executable program is run. Statistics collected for the POLCOMS application indicated that each CPU was performing between 660Mflops and 670 Mflops; this equates to a little under 10 per-cent of the IBM Power4+ theoretical peak performance of 6.8Gflops (based on two CPUs each performing a Floatingpoint Multiply Add with a clock speed of 1.7Ghz). hpmcount has previously been used to gather performance statistics and the calculated percentage of peak performance agrees with previously calculated [5 section 5].
4. Optimisation Options

The performance of the POLCOMS application and the efficiency of its constituent code has already been the subject of some investigation [5]. The partition of the problem space across the available processors with a view to achieving good load balance has already been considered in some depth. Another potential area of optimisation that has received a thorough inspection is minimising the exchange of data in boundary swapping operations. Some work has already been performed in optimising the serial code; expensive divide operations have already been replaced by multiplication in many places. There are a number of other possible optimisation options that can be considered, some of which have a significant impact on POLCOMS run times. Verification of results from any modified code should not be hampered by rounding errors introduced by any potential optimisations. Calculated real values are multiplied by 1000 and then converted to integers before being written to output files; thus any changes in to the last digit of real values is unlikely to effect the final calculated integer value.

Simple Compiler Optimisation: The compiler’s own optimisation level was increased from \texttt{O3} through \texttt{O4} to \texttt{O5} which is the most aggressive level of optimisation. The compiler options “\texttt{-qipa=level=2}” and “\texttt{-Q+<procedures>}” were also tried and found to bring little or no benefit.

The \texttt{-qipa} option causes more inter-procedural analysis to be performed that would otherwise be done and together with the \texttt{-q} option results in increased in-lining of procedures. The procedures selected for forced in-lining were those listed in Table \ref{tab:opt}, these being smaller size routines where the overhead of multiple calls might have been eliminated by in-lining. Increasingly aggressive optimisation produced no consistent or significant run time reduction.

The effects of the compiler flag \texttt{qhot=[novector]} were quite difficult to interpret. The use of \texttt{O3} with \texttt{qhot=novector} was found to improve the performance of many routines. For some (advection) subroutines the best performance was gained from using \texttt{O4} which includes \texttt{qhot}. The innermost loop in many subroutines has a loop count equal to the number of vertical layers in the POLCOMS model; in this case a value of 34. The benefits of vector reciprocal calculation seem largely to arise for vectors (i.e. loop length) greater than this. One contributing factor to the differences in performance for the different compiler option is the variation in the cost of a division operation. Using \texttt{O3} with \texttt{qhot=novector} forces division to be done using hardware.
Figure 4-1: Cost of Division in Cycles
Using \texttt{O3} with \texttt{qhot} or using \texttt{O4} which includes \texttt{qhot} causes the compiler to insert calls to MASS library routines \texttt{__vrec} and \texttt{__vrecGP}. These routines cause division to be executed within software using numerical means that are optimised for the IBM Power series processors [6]. The differing costs of each method can be seen in Figure 4-1 which shows the cost of division in cycles for increasing vector lengths [7]. As the vector length increases, the hardware division performed using \texttt{O3} –\texttt{qhot=novector} is clearly the most expensive i.e. slowest and the software division performed using \texttt{O4} is the fastest. Between the two lie software divides performed when using \texttt{O3} –\texttt{qhot}. For short vector lengths the best compiler option to use is not obvious. By gathering all reciprocal calculations into one vector in each advection routine, the vector length was increased significantly to well over 100; this resulted in superior program performance when using \texttt{O4}.

**Code optimisation of adv routines:** POLCOMS spends significant amounts of time in subroutines \texttt{pgrad_spline}, \texttt{advpuv}, \texttt{advpbv}, \texttt{advpbu} and \texttt{advpuu}. If these routines could be made to execute more efficiently then overall run times could be reduced. The existing code had correct loop ordering for fast accessing of array values (spatial locality). There are some conditional or branching statements within longish loops that could be moved outside of the loop, however, timings on amended code indicated that there was no reduction in execution time. It could be the case that the compiler can recognise these potential optimisations and perform them itself. An example of this type of manual optimisation, taken from subroutine \texttt{pgrad_spline} is shown in Figure 4-2.

The subroutines \texttt{adv*} perform advection; in very simple terms they move things about within the model such as liquid, salinity. This modelling action is performed by looping over each point of the model and performing calculations based on the cell values of the cell under consideration and any neighbouring cells appropriate to the calculation. The routines each contain approximately 500 lines of executable code (excluding debugging and timing functionality).

```fortran
... do j=1,JESUB
    do i=1,IESUB
      if (.not.compress) then
        b(n,i,j) = b(n-1,i,j)
        b(1,i,j) = b(2,i,j)
      endif
      di(1,i,j) = 0.0d0
      di(n,i,j) = 0.0d0
    enddo
  enddo
...
```

The value of \texttt{compress} does not change within the loop. The \texttt{if} block is always executed or always not executed, however, the condition could be evaluated unnecessarily on every loop iteration.

```fortran
... if (.not.compress) then
    do j=1,JESUB
      do i=1,IESUB
        b(n,i,j) = b(n-1,i,j)
        b(1,i,j) = b(2,i,j)
      enddo
      di(1,i,j) = 0.0d0
      di(n,i,j) = 0.0d0
    enddo
  enddo
else
    do j=1,JESUB
      do i=1,IESUB
        di(1,i,j) = 0.0d0
        di(n,i,j) = 0.0d0
      enddo
    endif
  endif
 ...
```

**Figure 4-2:** Original code and manually "optimised" code
Pre-Computation: Inspecting the code indicated that there were some inefficiencies within these routines. The subroutines contained some division operations within loops. These would interrupt pipelined program execution since division is not pipelined. Where possible, the divisor was calculated in advance in the form of a reciprocal so the division could be replaced by a pipelined multiplication operation. This is shown in Figure 4-3.

Once all the divisions have been pulled out of the computation loops the values can be packed into a single array, the reciprocal of this array calculated and then the values unpacked into temporary arrays for each loop. Placing a greater number of values in a vector array increases the chance of the MASS routine `vrec_GP` outperforming the programmed loop when calls to it are generated by the compiler. Timed runs indicated that the fastest times were gained from compiling all code using O3 `qhot=novector` except for the advection routines which were compiled with O4 `qhot` to gain the best performance during division operations. Inserting an explicit call to `vrec` within the advection routines was found to bring no performance benefits. Examples of the changes made can be found in the advection routines.

![Figure 4-3: Pre-computing reciprocals.](image)

The cumulative effect of pre-computing values that could replace division operations was a noticeable reduction in the program execution time. The reduction in run time over a range of simulation times and numbers of processors varied between ten and twenty per-cent. For example when running on eight processors for 24 simulated hours the run time was reduced from 238.59 seconds to 198.17 seconds; the optimised version of the code was also run with the additional Load Leveller options previously described. This is a raw percentage reduction of nearly 17 per-cent.
Replace subroutine **tridag**: This subroutine is a Numerical Recipes routine [8]. It is called frequently and is in the top ten of all the application routines in terms of elapsed time although its time consumption is considerably less than the top five application routines. It has been shown that in some circumstances the equivalent IBM ESSL library routine **DGTNP** [9] can be thirty to fifty per-cent faster than the **tridag** routine [10]. The ESSL library is currently used in the make file options for HPCx. The effect of replacing **tridag** with **DGTNP** was a small decrease in the program run time with no change in the output file **zet.Run001**. For a simulated time of 24 hours on eight processors the run times were:

<table>
<thead>
<tr>
<th></th>
<th>POLCOMS with tridag</th>
<th>198.17 secs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POLCOMS with DGTNP</td>
<td>195.18 secs</td>
</tr>
</tbody>
</table>

This represents a further 1.25\% raw reduction from the programs original execution time.

The reduction is achieved by more efficient use of memory resulting from overwriting an input array that is not is needed with calculated results when they are returned from the subroutine. This eliminates the need for an entire array. This change is shown in Figure 4-4. Examples within the POLCOMS code can be found in source file **diffuse.F** (subroutines **diffuseb** and **diffuseu**) and in source file **turbulence.F** (subroutine **update_TKE**).

![Figure 4-4: Replacing tridag with DGTNP](image)

**Figure 4-4: Replacing tridag with DGTNP**

**Modifications to pgrad_spline**: Some small changes were made to this routine giving improved performance. One loop was split into two allowing the compiler to identify additional optimisations. The ‘strength’ of some calculations was reduced by replacing non pipelined exponentiation with simpler multiplication. For example, \( r**4 \) would be replaced by \( (r2 \times r2) \) where \( r2 \) has been previously been calculated as \( (r \times r) \). These changes reduced the POLCOMS execution time by a further two per-cent when running for 24 simulated hours on eight processors.
5. Performance Improvement

Most of the proposed modifications that were discussed in section 4 were found to have a beneficial effect on the performance of the POLCOMS code. Timed runs of POLCOMS with all successful optimisations in place were performed to evaluate the overall performance improvement when compared with pre-optimisation timings. Raw percentage reduction figures from two sets of timings from runs using 32 processors are shown in Figure 5-1 and for 128 processors in Figure 5-2.

The POLCOMS run times were significantly reduced for all simulation times and processor counts.

<table>
<thead>
<tr>
<th>np</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>48</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Time (s)</td>
<td>924.16</td>
<td>461.94</td>
<td>238.59</td>
<td>127.60</td>
<td>75.0</td>
<td>48.50</td>
<td>41.06</td>
<td>28.48</td>
</tr>
<tr>
<td>Optimised Time (s)</td>
<td>795.50</td>
<td>387.88</td>
<td>191.14</td>
<td>105.24</td>
<td>58.5</td>
<td>40.99</td>
<td>32.76</td>
<td>23.59</td>
</tr>
<tr>
<td>Raw % reduction</td>
<td>13.92</td>
<td>16.03</td>
<td>19.89</td>
<td>17.52</td>
<td>21.99</td>
<td>15.48</td>
<td>20.21</td>
<td>17.17</td>
</tr>
</tbody>
</table>

Table 5-1: Run time reduction (tdur=24)

<table>
<thead>
<tr>
<th>Simulated hours</th>
<th>24</th>
<th>48</th>
<th>72</th>
<th>96</th>
<th>120</th>
<th>144</th>
<th>288</th>
<th>576</th>
<th>1152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Time (s)</td>
<td>28.48</td>
<td>49.98</td>
<td>75.80</td>
<td>97.86</td>
<td>123.90</td>
<td>148.80</td>
<td>292.48</td>
<td>582.75</td>
<td>1152.44</td>
</tr>
<tr>
<td>Optimised Time (s)</td>
<td>23.59</td>
<td>45.32</td>
<td>67.52</td>
<td>88.56</td>
<td>108.87</td>
<td>127.05</td>
<td>253.25</td>
<td>501.71</td>
<td>987.00</td>
</tr>
</tbody>
</table>

Using Xprofiler on the optimised code gives an indication of where the time savings might be occurring. An extract of the Xprofiler report is shown in Figure 5-3. Comparing this report with the earlier version for the un-optimised code (Figure 3-2), a number of differences are apparent. Usage of the compiler inserted vrec_GP has decreased because of the usage of the qhot=nocvector compiler option on most subroutines. The time for each call of pgrad_spline and the advection (adv*) routines is greatly reduced. The total time for calls to DGTNP is less than for calls to tridag although the times for diffuseu and diffuseb which call DGTNP/tridag have increased slightly.
Figure 5-1: Pre & Post Optimisation Timings (np=32)

Figure 5-2: Pre & Post Optimisation Timings (np=128)
Re-running `hpmcount` using the optimised version of the POLCOMS code indicated that each CPU was performing between 690Mflop/s and 700 Mflop/s; the performance has improved and is now a little over 10 per-cent of the IBM Power 4+ theoretical peak performance.
6. Conclusions

POLCOMS as a whole as well as the major time consuming components exhibit good scaling characteristics when run across larger numbers of processors as shown by the parallel efficiency graph. The procedures called within these functional components are already well tuned and contain code that has already undergone some optimising modifications.

The use of recommended Load Leveller environment variables will give a small but noticeable improvement in performance for next to no effort and with minimal risk; this should be implemented immediately.

The pre-computation of reciprocals to enable the replacement of division with multiplication can also be implemented. This modification carries the potential risk of introducing computational errors, as do all software modifications, since it requires software changes to a number of modules. However, the number of modified code modules is small and so long as rigorous testing of the modified code is performed the risk can be minimised.

A small number of other minor changes such as loop splitting and strength reduction were also found to bring improved performance. As with the pre-computation of reciprocals, implementing these changes carries some risk but again this can be minimised by thorough testing of modified software.

The techniques used to reduced execution time do not work in all subroutines; timing of program runs is essential to determine if a change has a beneficial effect. The use of strength reduction helped reduce the time spent in subroutine \texttt{pgrad\_spline}; however, it had no measurable effect when the technique was used in source file \texttt{bcalc}. Similarly, the pre-computation of reciprocals that was successful in the advection routines had no measurable effect when used in source file \texttt{barot.F} (pre-computing reciprocals of \texttt{rac} and \texttt{rbc}). It may well be the case that there was some increased pipelining of instructions but that any performance improvement was too small to be apparent with the current timing methods. It is also possible that making the changes had no effect on code execution and that the compiler was already optimising to the best of its ability.

Replacing the Numerical Recipes subroutine \texttt{tridag} with the ESSL library routine \texttt{DGTNP} gives a small reduction without impacting the program results.

Tailoring compilation options to suit individual modules also resulted in improved performance. The fastest execution times were recorded when all modules were compiled with \texttt{-O3 -qhot=novector} except for \texttt{advpuu.F} and \texttt{advpuv.F} which were compiled with \texttt{-O4}.

At the time of writing there are some outstanding tasks. The modifications have been implemented on certain paths through the code only; other paths that would be utilised depending on the setting of pre-compiler \texttt{cpp} directives still require modification.
use of different compiler options for different modules would require modifications to the Make files used to build the application.

The reduction in run times that has been achieved varies with the number of processors being used and is significant. When running with 128 processors the run time reduction is between 10 and 20 per-cent using the preferred HPCx performance improvement metric.

[End of Report]
7. Appendix A: References and Sources

1. HPCx/POLCOMS [http://www.hpcx.ac.uk/research/environment/polcoms.html](http://www.hpcx.ac.uk/research/environment/polcoms.html)
   POLCOMS home page [http://www.pol.ac.uk/home/research/polcoms/](http://www.pol.ac.uk/home/research/polcoms/)

2. Personal email from M. Ashworth 14th October 2004

3. Man pages for xlf90 on HPCx.


7. Data collected by M. Bull, EPCC.


8. Appendix B: Changed Software Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Subroutine</th>
<th>Change</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>advpbu.F</td>
<td>advpbu</td>
<td>Pre-computation of reciprocals.</td>
<td>-O4</td>
</tr>
<tr>
<td>advpbv.F</td>
<td>advpbv</td>
<td>ditto</td>
<td>ditto</td>
</tr>
<tr>
<td>advpuu.F</td>
<td>advpuu</td>
<td>ditto</td>
<td>ditto</td>
</tr>
<tr>
<td>advpuv.F</td>
<td>advpuv</td>
<td>ditto</td>
<td>ditto</td>
</tr>
<tr>
<td>diffuse.F</td>
<td>diffuseb</td>
<td>Replace tridag with DGTNP.</td>
<td>-O3 -qhot=novector</td>
</tr>
<tr>
<td></td>
<td>diffuseu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgrad.F</td>
<td>pgrad_spline</td>
<td>Pre-computation of reciprocals. Loop splitting.</td>
<td>ditto</td>
</tr>
<tr>
<td>turbulence.F</td>
<td>update_TKE</td>
<td>Replace tridag with DGTNP.</td>
<td>ditto</td>
</tr>
<tr>
<td>All others</td>
<td></td>
<td>Unchanged</td>
<td>-O3 -qhot=novector</td>
</tr>
</tbody>
</table>

[End of Document]