Code Optimisation Check-list

by Ning Li
HECToR CSE Team

1. Prepare for performance studies

- ❑ Make sure your code is producing correct results before any optimisation.
- ❑ Set up a test case that represents typical workload of production runs.

  For example, for a time-dependent simulation, it is often sensible to benchmark only a few time steps (assuming algorithm complexity is not time-dependent and initial set-up cost is negligible). It is probably not sensible to benchmark a case with reduced mesh resolution, because the memory-access pattern and communication pattern would be very different from the real runs.

- ❑ The test case should complete within reasonable amount of time.

  Code optimisation is often an iterative process so be prepared to run the test case many times.

- ❑ If possible, disable the non-essential areas and concentrate on the key part of your application.

  For example, one may consider benchmarking the main loop of the application only so that the profiling is not distorted by an expensive one-time-only initialisation.

2. Use the right tools

- ❑ Understand the basic concepts of profiling, such as instrumentation, sampling, tracing, etc.
- ❑ On Cray XE HECToR, use the native profiling tool CrayPAT.

  A list of CrayPAT documentation:
  http://www.hector.ac.uk/support/documentation/userguide/tools.php#craypat
  http://www.hector.ac.uk/cse/documentation/Perf/

- ❑ Other tools may be helpful, in particular if you have previous experience with them.

  Other performance analysis tools (some available as HECToR modules) include: Scalasca, gprof, PAPI, VampirTrace and TAU. The HECToR CSE team offers a training course to cover some of these tools.

- ❑ Some applications have their own timing routines. You may perform your own timing, for example using MPI_WTIME or even SYSTEM_CLOCK.

  Do pay attention to the resolution and data range of these functions.

- ❑ Whatever tools you are using, profiling may have to be done multiple times for each configuration.

This is to take into account factors like variation of system load on a busy supercomputer, in particular for communication intensive and IO intensive code.

- ❑ Most compilers can generate code optimisation report, which can be highly useful, in particular for optimising serial code.

  For example, use '-Minfo -Mneginfo' flags with PGI to generate information about successful and failed optimisations. Read the Serial Code Optimisation Good Practice Guide for more details.

3. Load Balance Considerations

- ❑ If your code does not suffer from imbalanced workload, go to section 4.

  Always try to fix load balance issues first. Otherwise you are leaving valuable resources idle.

- ❑ For MPI applications, load-imbalance often appears in CrayPAT report as large portion of time in MPI_SYNC category, associated with collective operations.

- ❑ From profiling, determine what causes the imbalance (computation, communication or I/O) and target your problem.

- ❑ If imbalance is in communication, check if runtime tuning can help.

  For MPI applications, check the list of environment variables (on HECToR type 'man mpi' for more information). For example, variables controlling the behaviour of rank placement (such as MPICH_RANK_REORDER_METHOD) may affect the communication efficiency of applications.

- ❑ If imbalance is in I/O, is your I/O model to be blamed?

  Are there imbalanced data movement for implementing I/O? Are you handling I/O from the master rank only? You might want to adopt a genuinely 'parallel' I/O model. Read the IO Good Practice Guide by the CSE team.

- ❑ Is your decomposition strategy appropriate?

  For example, people studying particles may find it hard to achieve load balance if they evenly distribute their numerical meshes but have different numbers of particles in each sub-domain. If this is the case, fundamental rethink of the decomposition strategy may be required.

4. Find the bottleneck

- ❑ From profiling, find the bottleneck of your application. If the bottleneck is computation, go to section 5; communication, go to section 6; I/O, go to section 7.

  Always optimise the slowest part of your code first in order to achieve the largest possible benefit.

5. To optimise computations

- ❑ Does your code use libraries?

Common examples include linear algebra algorithms, ODE/PDE solvers and FFTs, where highly optimised libraries are always available, many specially tuned for supercomputers. Use these instead of writing your own.

- ❑ Are the big loops in your application vectorised?

  Turn on compilers' reporting functions to show this (for example '-ftree-vectorizer-verbose' for GNU compiler). CrayPAT also has a hardware counter group reporting vectorisation, see Performance Measurement Good Practice Guide for details.

- ❑ If a loop does not vectorise, does it contain subroutine calls?

  This often prevents vectorisation. Consider inlining the subroutines, in particular if they are small. Compilers may be able to help inlining (for example: '-Minline' flag of PGI). One such poor practice is to pass an element of an array to a subroutine for processing and call this subroutine from within a loop. Rewrite your subroutine to handle the whole array.

- ❑ If a loop does not vectorise, does it contain I/O?

  I/O operations are inherently serial. Printing information to standard output from within a loop will prevent vectorisation.

- ❑ If a loop does not vectorise, does it involve data dependency?

  If there is data dependency between loop iterations, such loop naturally cannot be vectorised. Consider adjusting your algorithm to remove data dependencies. Another possible solution is to completely unroll the inner-most loop (in particular if it is small), making the nesting loop a candidate for vectorisation.

- ❑ If a loop does not vectorise, does it contain conditional tests?

  Instead of having if/else statements in the loop body, is that possible to do the conditional test outside and have separate loops for different branches?

- ❑ Even if your loops are already vectorised, check if there is room for improvement.

  For nested loops, use the longest dimension as the inner-most loop (larger vector length) for more efficient vectorisation. However, only do this without sacrificing the cache efficiency (see the next two bullets). One example is to swap the dimensions of your multi-dimensional arrays, as done in many CFD applications.

- ❑ Does your application make use of cache and TLB (Translation Lookaside Buffer) efficiently?

  CrayPAT has hardware counter groups reporting cache and TLB usage. see Performance Measurement Good Practice Guide for details.

- ❑ Does your application use stride-1 memory-access wherever possible?

  For nested loops, the inner-most loop should have stride of one or small values to help cache efficiency. Use the inner-most loop to traverse the first dimension of Fortran arrays and last dimension of C/C++.

- ❑ Have you tried to combine adjacent small loops into large ones?

This may or may not be beneficial. By making the loop body larger (this is called *fusion*), the compiler may have better chances to optimise the code (for example by using the hardware registers more smartly, therefore increase the computational intensity – number of flops per memory access). For the same reason, unrolling small inner loops may help too.

- ❑ Try loop tiling (or blocking) if you have very large nested loops and your cache usage figure is poor.

  This is the technique that partitions loop iteration space into smaller blocks so that array elements being accessed consecutively fit in system cache. This has good potential to improve application performance, although a portable implementation can be tricky.

6. To optimise communications:

- ❑ For MPI point-to-point communication, always try to post receives before sends.

  This can help reduce the use of the internal MPI buffers therefore improving performance.

- ❑ Pay attention to your message sizes.

  CrayPAT can generate such a report.

- ❑ Instead of sending many small messages, is that possible to combine them into one large message?

  Although the Gemini interconnect on HECToR is very powerful, it can be beneficial to send a small amount of large messages to minimise the communication latency.

- ❑ If your application is communication intensive and you work with large data set, can you identify opportunities in your algorithm to overlap communications and computations?

  For example, is that possible to divide your data into smaller chunks, perform computations on one chunk and communications on the other?

- ❑ Use collectives properly.

  Only use it when necessary. Trust the MPI library and do not attempt to implement collectives using point-to-point communications by yourself.

- ❑ Only use barriers when it is really necessary.

- ❑ Try runtime tuning.

  There are many environmental variables that modify the runtime behaviour of the MPI library. Try 'man mpi' to see the full list.

- ❑ Consider hybrid or mixed-mode implementations.

  HECToR phase-3 node has 32 cores, having non-uniform access (NUMA) to their shared local memory. This architecture is really suitable for mixed-mode applications using OpenMP within nodes.

- ❑ Consider underpopulating the nodes.

  Underpopulating the nodes may reduce contention on system resources, such as memory bandwidth for computation intensive code and network bandwidth for communication intensive code. This does not necessarily mean computational resources being wasted – you may be able to use the threaded libraries (such as Cray's libsci, which can make use of the idle cores) even if your code itself does not directly use OpenMP. Do remember that you will be charged as if you are using the whole node. Read HECToR Phase 3 Good Practice Guide for examples using underpopulated nodes.

- ❑ For certain applications, shared-memory optimisation may help.

  The CSE team has a very successful record using System V IPC to optimise the communication of global data for applications using ALLTOALL type of communication.

7. To optimise I/O:

- ❑ Are you using the correct file types for I/O?

  Human-readable ASCII files are only suitable for small amount of data, such as program parameters and event logs. Binary files can be read or written more efficiently by computers and should be used to handle the large scientific data sets.

- ❑ Be aware of the cost of doing runtime data conversion.

  Compilers are able to convert the endianness of binary data to facilitate data sharing between HECToR (a little endian system) and other big-endian systems (for PGI, use -Mbyteswapio flag). Be aware of the extra I/O cost of such conversion. It may be better to use a portable data format, such as netCDF or HDF.

- ❑ For parallel I/O, are you using a suitable I/O model?

  The IO Good Practice Guide covers most popular parallel I/O models. Note the some models may introduce imbalanced workload. To achieve the best I/O performance, it may be possible to create dedicated I/O processes that receive data from the computation processes and perform IO separately when computations are in progress.

- ❑ Consider advanced I/O techniques, such as controlling file striping.

  This requires understanding of the LUSTRE File System and its configurations.