Best Practice mini-guide "Stokes"

SGI Altix ICE at ICHEC

Michael Lysaght, ICHEC

Niall Wilson, ICHEC

Eoin McHugh, ICHEC

Michael Browne, ICHEC

Gilles Civario, ICHEC

February 2013

Table of Contents

1. Introduction

This document is a Best Practice mini-guide for the PRACE Tier-1 SGI Altix at the Irish Centre for High End Computing (ICHEC). The guide should be seen as a quick reference guide. For more detailed information or assistance, please visit ICHEC support [http://www.ichec.ie/support/]

2. System architecture and configuration

This section contains information on Stokes's architecture and configuration including information on the Panasas I/O subsystem.

System configuration

Stokes (stokes.ichec.ie) is an SGI Altix ICE 8200EX cluster with 320 compute nodes. Each compute node has two Intel Westmere processors and 24GB of RAM. This results in a total of 3840 cores and 7680GB of RAM available for jobs. The nodes are interconnected via two planes of ConnectX Infiniband (DDR) providing high bandwidth and low latency for both computational communications and storage access. Storage is provided via a Panasas ActiveStor 5200 cluster with 143TB (formatted) of capacity accessible across all nodes. In addtion to the compute nodes a set of service and administrative nodes provide user login, batch scheduling, management, etc.

Stokes is intended to be used as a general purpose capacity oriented system to support a large and diverse user community. Two small partitions representing 5% of the compute nodes are reserved for interactive/development batch jobs and long-running/single-node jobs while the remainder is for multi-node production jobs of between 24 and 516 cores with a maximum runtime of 84 hours. As well as a standard HPC development environment, a significant number of applications are centrally installed and available via environment modules.

Processor architecture

All compute nodes (SGI ICE IP95 blades) contain two sockets, each with a 6-core, 2.6 GHz Intel Xeon X5650. Hyper-Threading is enabled on the processors giving 12 virtual cores per socket. Note that the login nodes have different processors (older Harpertown Xeon X5355) which should be taken into consideration for compiler optimisation.

Operating system

The operating system on all nodes is SUSE Linux Enterprise Server 11 Service Pack 1 (SLES 11 SP1). Additional applications and utilities are installed on the shared filesystem and available via environment modules - see the output of the 'module avail' command for a list of supported software.

Memory architecture

Each node contains a total of 24GB od DDR3 @1333 MHz. This is arranged as 3 x 4GB DIMMs per socket. As the X5650 processor has 3 memory channels, this configuration provides the maximum possible memory bandwidth of 32 GB/s.

Network infrastructure

Stokes has two separate Infiniband networks available on all nodes for application and storage traffic. By default, applications use one network for communications and the other network is used for storage traffic, although certain MPI implementations can be configured to run communications over both networks. The Infiniband switches operate at 4X DDR bandwidth (20 Gbit/s) and are interconnected in a Hypercube topology. The network interfaces on the nodes are ConnectX.

For the login nodes, Stokes has full 10 Gbit/s connectivity via the Irish NREN (HEAnet) to GEANT2.

I/O subsystem

Storage is provided via a Panasas ActiveStor 5200 cluster with 143TB (formatted) of capacity accessible to the compute and login nodes via a PanFS filesystem. The Panasas cluster uses a 10Gbit/s Ethernet internally and is connected to the Infiniband network via 8 Infiniband to Ethernet bridge servers. Storage traffic is routed across one of the Infiniband networks via IPoIB. The peak I/O bandwidth is approximately 4 GB/s.

Filesystems

The PanFS distributed filesystem provides a single logical namespace across all nodes. Compute nodes are diskless and so users should only use directories under the logical /ichec mountpoint. In particular, home directories are created under /ichec/home/users and project work directories under /ichec/work. User home directories have a 10GB quota but are backed up to tape whereas project work directories can have a much larger quota but are NOT backed up.

Hardware accelerators

Stokes does not contain any coprocessors or other hardware accelerators.

3. System Access

The following section describes access policies for Stokes.

Application for an account

Users based at Irish research institutions can directly apply for projects and accounts online at the ICHEC website [www.ichec.ie]. PRACE users apply via the DECI Tier-1 process with all project members being assigned a system account.

How to reach the system

There are two possible ways of logging into Stokes which are described below.

- Login with username and password/key

  Users with a username and password (or public key) can login to Stokes using SSH with the command *ssh account@stokes.ichec.ie*. SSH access to Stokes is restricted to the IP address ranges of ICHEC's member institutions and collaborators. PRACE users from outside Ireland will need to request additional firewall rules to allow access from their site.

- Login with grid certificate

  Users with a valid grid certificate may use GSI-SSH to login to Stokes instead of username and password with the following command *gsissh account@stokes.ichec.ie*. PRACE users will first need to GSI-SSH to a PRACE door node prior to connecting to Stokes. In both cases the default GSI-SSH port (2222) is used.

4. User environment and programming

The following section describes the Stokes production environment, programming environment and the steps for basic porting to the Stokes machine.

4.1. Generic x86 environment

Despite its much larger scale, the Stokes system is very similar to what one might find on a desktop Linux system. This can greatly simplify the development environment for users familiar with desktop Linux. The current version of the operating system is SUSE Linux Enterprise Server (SLES) 11 service pack 1.

Modules

The large array of software packages installed means that incompatibilities are inevitable. To minimize the problems this can cause, the module system is used. In order to use a software package that is not part of the base operating system one must load the appropriate module(s). Loading a module generally sets environment variables such as your PATH.

To see what modules are available type:

module avail

You can then load the appropriate modules as follows:

module load intel-cc

module load intel-fc

As well as loading the necessary modules at compile time it is also required that they be present at runtime on the compute nodes. If these modules are not loaded the program is likely to crash due to not being able to find the required libaries, etc. They can be loaded in two ways; you can use the PBS directive:

```
#PBS -V
```

to import your current environment settings at submission time. Or you can add **module load `package_name'** commands to the submission script itself.

Other useful module commands are:

module unload intel-cc (removes that module)

module list (lists the modules you are using at the moment)

Using Modules

For more information on modules see: Using Modules [http://www.ichec.ie/support/tutorials/modules.pdf] or type *man module* Note: the default module for a given package points to the most recent version of that package. To use an older version specify the name explicitly.

Unified PRACE Environment

Stokes provides the PRACE Common Production Environment (CPE) as a 'prace' environment module. This is loaded with *module load prace*. When loaded this module file ensures that all of the following are in the user's environment:

- Bash shell (bash)
- TC Shell (tcsh)
- OpenSSH
- Emacs
- nedit
- C Compiler
- C++ Compiler
- Fortran Compiler

- Java Compiler
- Perl Interpreter
- Python Interpreter
- Tool Command Language (TCL)
- TCL GUI Toolkit (TK)
- GNU Make (gmake)
- MPI Library
- BLACS Library
- BLAS Library
- LAPACK
- ScaLAPACK
- FFTW Library (versions 2 and 3)
- HDF5 Library
- NetCDF Library

4.2. Batch System

Stokes uses the Torque (PBS) resource manager along with the Moab scheduler. Jobs are submitted by creating a PBS script file then submitting it using the *qsub* command. The following sample script would be submitted by typing *qsub script.pbs* on the command line:

```
#!/bin/bash
#PBS -l nodes=4:ppn=12
#PBS -l walltime=1:00:00
#PBS -N my_job_name
#PBS -A prace_project_name
#PBS -r n
#PBS -j oe
#PBS -m bea
#PBS -M me@my_email.ie
#PBS -V
cd $PBS_O_WORKDIR
mpiexec ./my_prog my_args
```

A number of different queue types are configured on Stokes but users are advised to submit without specifying a queue and the scheduler will make a suitable selection. Job sizes of up to 516 cores can be run for up to 84 hours. Larger than 1032 core jobs can be run by special request. Debug jobs of up to 168 cores can be run for 30 minutes on dedicated development nodes. Single node (12 core) jobs with walltimes of more than 30 minutes are not advised as they are restricted to a limited number of nodes on Stokes resulting in long queue times.

The jobs in the queue can be listed with the *showq* command. It shows the jobs's position in the queue and also whether a job is running, idle or blocked.

4.3. Login and Compute Nodes

When you connect to the Stokes system your connection will automatically be routed to one of two login nodes: `stokes1' or `stokes2'. These nodes, sometimes called frontend nodes, are used by users for interactive tasks like compiling code, editing files and managing files. They are shared by users and should not be used for intensive computation.

In order to connect to Stokes it is necessary to connect from a machine with an IP address which belongs to one of ICHEC's participant institutions. Specific arrangements are put in place for PRACE users depending on their home country. These arrangements will be discussed with PRACE users at account creation time. Thus if you which to connect from home or while travelling you must first connect to a machine in your home institution and then connect to Stokes.

The vast majority of the Stokes system is made up of compute nodes. These nodes are used for running jobs that are submitted to system. They are sometimes referred to as backend nodes. They are dedicated to a single user at a given time and can be used for intensive long term work loads.

4.4. Compilers

Both the GNU and Intel compiler suites are available on Stokes. The GNU suite is available by default in the form of the OS distribution version with more up to date, and thus recommended versions, available via modules. To use the Intel compilers one must load the relevant modules (intel-cc, intel-fc). In general the Intel compilers should give better performance and are recommended. Several versions are provided allowing for backward compatibility and more recent versions. Note that the default is not always the most recent version as new versions are evaluated before being made the default option.

The following tables list both the commands to invoke the Intel and GNU compilers and the corresponding MPI wrapper commands. See the later section on MPI for an explaination of these wrappers.

Table 1. Intel Compilers

Language	**Intel Compilers**	**MPI Wrappers around Intel Compilers**
C	icc	mpicc
C++	icpc	mpicxx
Fortran 77	ifort	mpif77
Fortran 90	ifort	mpif90
OpenMP	yes	-

Table 2. GNU Compilers

Language	**GNU Compilers**	**MPI Wrappers around GNU Compilers**
C	gcc	mpicc
C++	g++	mpicxx
Fortran 77	gfortran	mpif77
Fortran 90	gfortran	mpif90
OpenMP	yes (via modules)	-

4.5. OpenMP

When using OpenMP on Stokes you need to be aware that Hyperthreading is enabled by default. This means that each physical core can appear as two logical cores. Thus by default an OpenMP program will typically try to use 24 threads rather than 12 as one might expect. Typical HPC workloads will not benefit from over subscribing the physical cores unless the code is constrained by I/O. The environment variable *OMP_NUM_THREADS* is normally used to control how many threads an OpenMP program will use. It can be set in the PBS job script prior to launching the program as follows:

export OMP_NUM_THREADS=12

4.6. MPI

There are a number MPI libraires available and sometimes it is perferable to use one rather than another however unless you have a specific reason to do so it is recommended to use the default libraires. For the Stokes there are two primary MPI modules to choose between:

module load mvapich2-gnu

For use when using the GNU compiler suite.

module load mvapich2-intel

For use when using the Intel compiler suite.

These modules provide support for MPI2 and the Infiniband based networking used in the machine. They also provide the compiler wrapper scripts listed in the previous tables which greatly simplify compiling and linking MPI based codes. To run a MPI job, the mpiexec command is used in a job submission script e.g. *mpiexec ./ my_prog my_args*.

Again there is a man page for mpiexec for more details. The Intel (intel-mpi) MPI libraries are also available, however the mvapich2 libraries are recommended. On Stokes the SGI (sgi-mpt) MPI libraries are available too. These can be useful if working wiht SGI specific performance analysis tools or if dual rail Infiniband support is desired.

4.7. System specific environment

Stokes provides a large number of additional or alternate applications, libraries and tools. These are made available via environment modules. Type *module avail* to see a full list.

4.8. Backup Policy

As stated in ICHEC's Acceptable Usage Policy backups are only made of user's home directories. Project directories under */ichec/work/projectname* are NOT backed up. Furthermore, backups are only carried out as part of ICHEC's system failure recovery plan; the restoration of user files deleted accidentally is not provided as a service.

5. Tuning applications and Performance analysis

Stokes is, from a user's perspective, a very standard x86-64 / InfiniBand Linux cluster. Therefore, standard x86-64 optimisation strategies and techniques can be applied here whithout difficulties. The very first techniques are to simply select the right compiler and libraries and to use the most effective compiler switches.

5.1. General optimization for x86 architecture

Since Stokes is fitted with Intel Xeon processors, the most effective compilers to use are usually the Intel ones. Many different versions of those are installed on Stokes and they can all be selected individually using the module command:

module avail intel-cc or **module avail intel-fc**

gives you the list of the C/C++ and Fortran Intel compilers.

Then

module load intel-cc or

module load intel-fc

will select the default version of these libraries. Should the default version not correspond to the one you need, you can select the exact version you need by using, for example

module load intel-fc/11.0.072.

In addition, various versions of the GNU compilers are also available on Stokes. As for the Intel compilers, these compilers are made accessible through the module command, using the module name `gcc` from both C, C++ and Fortran.

Intel compiler switches

Since only the Intel compilers are to be considered for performance purposes on Stokes, we will present their most useful performance switches:

`-O0`: No optimisation. Used during the early stages of application development and debugging. Use a higher setting when the application is working correctly. Using this option will result in quicker compile times which can be useful when doing intensive development. If stepping through code in a debugger such as DDT use this option to avoid the "erratic jumping" effect of code reordering.

`-O1`: Moderated optimisation. Optimise the code while keeping the generated binary relatively small.

`-O2`: Maximise speed. Default setting. Enables many optimisations, including vectorization. Creates faster code than `-O1` in most cases.

`-O3`: Enables `-O2` optimisations plus more aggressive loop and memory access optimisations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache and additional data prefetching. The `-O3` option is particularly recommended for applications that have loops that do many floating-point calculations or process large data sets. These aggressive optimisations may occasionally slow down other types of applications compared to `-O2`.

Production codes running on Stokes should be compiled with `-O3` unless there is reason not too. However it is not guaranteed to result in significant improvements over `-O2`.

`-xSSE4.2`: The processors on Stokes's compute nodes are Intel Xeon Westmere, and therefore support the Intel SSE extensions up to version 4.2. Usually, enabling this support at the compiler level with the `-xSSE4.2` switch is advisable. However, since the processors on the front-end nodes are of different architecture and only support SSE extensions up to version 3 extended, one might consider using the switch `-xSSSE3` or `-xhost` when compiling a code that needs to run on the front-end as well. The difference in performance between the two is in most cases negligible.

`-fnoalias`: Assumes no aliasing in the program. Off by default. This option might have significant effects but needs to be validated since the consumption it makes than no aliasing between variables in hidden inside the code might prove false, and thereafter generate wrong results.

`-fno-fnalias`: Assumes no aliasing within functions. Off by default. This option is slightly less aggressive than `-fno-alias` and might come handy when this one leads to wrong code.

`-unroll[n]`: Enable unrolling, with the optional n indicating the number of times to unroll. The two most useful versions of this option are `-unroll` to simply let the compiler decide the unrolling factor, and `-unroll0` to disable unrolling altogether.

`-openmp`: Instructs the compiler to generate multi-threaded code when OpenMP directives are present. Note a carefully written OpenMP code can support being compiled without enabling OpenMP and should produce a working serial code.

5.2. System specific optimisation

Following the Stokes upgrade (Aug. 2010) the compute nodes have significantly different processors from those found in the login nodes, which were not upgraded. If you are compiling code it is important to take full advantage of the capabilities of the newer compute node processors. They have faster cores and support additional instructions. Normally compilers will optimise for the processor they are being run on. The login nodes, where compilation normally takes place, have Harpertown processors and the compute nodes now have Westmere processors. Adding the following flag to your Intel compiler compilation commands will instruct the compiler to compile for the Westmere generation processor.

```
-axsse4.2
```

Using the above flag means that the resultant executable will most likely not run on the login nodes. If for some reason you need to use run the code on the login node as well the following flags will result in an alternate code path being produced which will be invoked automatically should the code be run on the older processors. In this case we are most concerned with performance on the compute nodes and merely compatability on the login nodes, which should not be used for intensive computation.

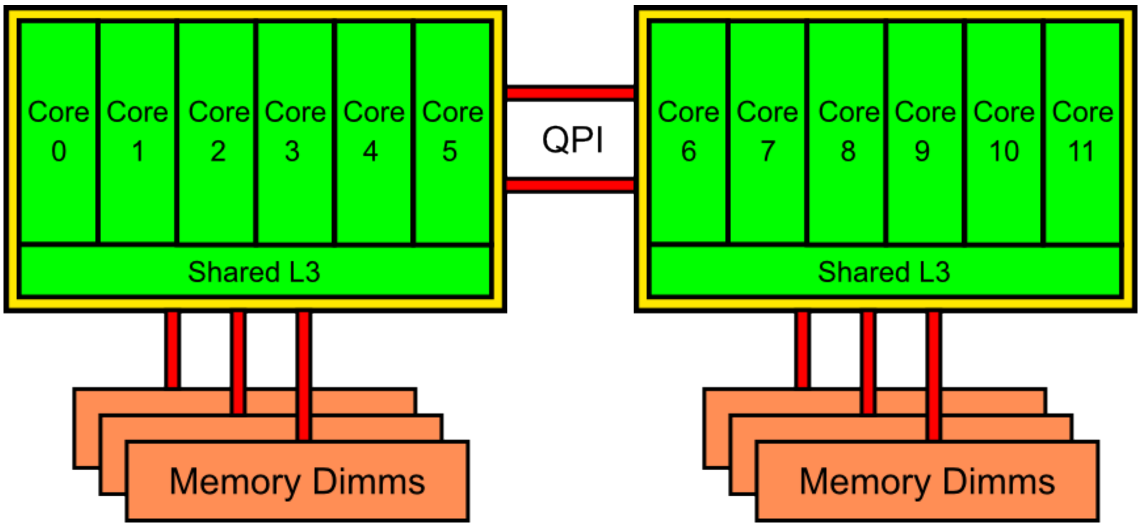
```
-axsse4.2 -msse3
```

5.3. Runtime choices and options

The main optimisation one can take advantage of is for mixed MPI + OpenMP codes. For such hybrid codes, finding the right balance and process placement for the MPI processes and OpenMP threads can lead to significant performance gains. However, for this case, one has to understand the exact architecture of the compute nodes and the best way of exploiting it:

Each compute node comprises of two distinct hexacore processors, directly attached to their own memory dimms. Even if the whole memory can be accessed by each of the cores within the node, "local" memory should be preferred to "distant" memory for performances reasons. Within a single compute node this creates two distinct "NUMA nodes" (NUMA standing for Non-Uniform Memory Access), where each core has symmetric access to the local memory. This architecture is outlined in the following schema:

Figure 1. Stokes system architecture



Thus, the best way of taking advantage of this particular node organisation for MPI + OpenMP codes is, whenever possible, to run one single process per CPU, with 6 OpenMP threads running on the 6 cores of the chip. This can be achieved for a code compiled with the mvapich2 library in the following way:

OMP_NUM_THREADS=6 and *MV2_CPU_MAPPING="0-5:6-11"*.

Of course, one can experiment to find the right balance between MPI processes and OpenMP threads. But the most likely effective strategies correspond to pinning the various OpenMP threads of each MPI process within a single NUMA node.

The other specificity one can take advantage of on Stokes is the fact that the processors support a 2-ways SMT mode, which means that each of the 6 cores a single chip counts, can accommodate up to two concurrent threads at a given time. By default, this feature won't be accessible since MPI tasks are attached to cores in an optimal way for single threaded MPI processes. But for mixed MPI + OpenMP codes, one can benefit from this feature in certain rare occasions. Knowing that this feature is exposed at the Operating System level by displaying a number of logical cores twice as large as the number of actual ones, and that those extra cores are numbered from 12 to 23 following the actual numbering display on the previous schema gives that:

Core number 0 hosts threads number 0 and 12

Core number 1 hosts threads number 1 and 13

And so on and so forth up to core number 11

CPU number 0 hosts threads [0-5,11-17]

CPU number 1 hosts threads [6-11,18-23]

Starting from this one can experiment with various MPI and OpenMP processes and threads numbers and attachment modes.

5.4. Memory optimizations

Linux is by default configured to minimise the memory footprint of each process. This means that by default all freed memory is given back to the kernel for later use. The drawback of this is that in some cases it can lead to a great many allocations and deallocations of memory while entering and leaving functions. This is especially true for some automatically allocated and deallocated Fortran local variables. Such memory allocation and deallocation can increase the runtime of jobs. For example, VASP can sometimes exhibit such behaviour. To identify such problems, you can add to your submission command line the /usr/bin/time command, as follows:

mpiexec /usr/bin/time usual_command_line

Be careful not to use the built-in 'time' command, as it would be of no use here. At the end of the execution this should give some timing information, including the number of minor page faults:

13.37user 0.01system 0:13.43elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k 0inputs+0outputs (4major+1740minor)pagefaults 0swaps If the minor page fault number is excessive, say more than a few hundred of them per second, you could greatly benefit from setting the following environment variables in your batch script:

export MALLOC_MMAP_MAX_=0

export MALLOC_TRIM_THRESHOLD_=-1

Furthermore, there are only a few cases where setting these variables could be counter-productive. You could therefore consider setting them globally in you .bashrc file. Should unexpected behaviour occur, you could unset them just for the problematic jobs, by adding the following lines in your script:

unset MALLOC_MMAP_MAX_ unset MALLOC_TRIM_THRESHOLD_

5.5. I/O optimizations

Stokes uses a high performance Panasas based IO subsystem. The Panasas filesystem, panfs, is a true parallel filesystem and so implements the semantics of file access slightly differently from other filesystems which you may be familiar with. A side effect of this that you may get unexpected performance results. In general for well written software using an appropriate approach to IO the results will be very good. However under certain circumstances a slowdown can be seen. One such case is VASP IO.

If VASP is writing small amounts of data sequentially it may do so very slowly. By default with the Intel Fortran compiler IO is assumed to be non buffered, as a result records are written to disk after each write. This results in a large number of small writes which must all be committed separately. There is a physical limit to how many operations a disk can handle at one time. While a file will be spread across a number of disks the limit can still be hit. Buffered IO will result in data being written in 4k blocks which can be handled much more efficiently. To enable this either:

export FORT_BUFFERED="true" before the job, or

compile with *-assume buffered_io*.

In addition, it is advisable not to do a *tail -f* on the file as it is being produced as this forces flushing of the file also.

5.5.1. Specific optimized libraries

5.5.1.1. Intel MKL

The Intel Math Kernel Library (MKL) is a very useful package widely used on ICHEC. It provides optimised and documented versions of a large number of common mathematical routines. It supports both C and Fortran interfaces for most of these. It features the following routines:

- Basic Linear Algebra Subprograms (BLAS); vector, matrix-vector, matrix-matrix operations.
- Sparse BLAS Levels 1, 2, and 3.
- LAPACK routines for linear equations, least squares, eigenvalue, singular value problems and Sylvester's equations problems.
- ScaLAPACK Routines.
- PBLAS routines for distributed vector, matrix-vector and matrix-matrix operation.
- Direct and iterative sparse solver routines.
- Vector Mathematical Library (VML) for computing mathematical functions on vector arguments.
- Vector Statistical Library (VSL) for generating pseudorandom numbers and for performing convolution and correlation.
- General Fast Fourier Transform (FFT) functions for fast computation of Discrete FFTs.
- Cluster FFT fucntions.
- Basic Linear Algebra Communication Subprograms (BLACS).
- GNU multiple precision arithmetic library.

If your code depends on standard libraries such as BLAS or LAPACK, it is recommended that you link against the MKL versions for optimal performance. Parallelism in a program can be achieved at the process level as in most MPI development or at the thread level as in OpenMP development, or in some mix of these approaches, a so-called hybrid code. The most common mode of development on our systems is MPI based, as this allows you write programs which can run across many nodes. Often such codes will want to call routines provided by MKL. However many of these routines are themselves parallel so at the node level one is left with two levels of parallelism contenting with one and other.

To eliminate this the MKL module sets the environment variable:

MKL_NUM_THREADS=1.

If you are writing hybrid code or pure OpenMP code that uses MKL you may need to override this setting. Chapter 6 of the MKL userguide explains in detail how this and other related environment variables can be used. Note if you have used a version of MKL older than 10.0 you should be aware that MKL's method for controlling thread numbers has changed. Similarly version 10.3 and above dramatically changed the linking model. In general this has simplified build codes with MKL. Remember that when a code is linked against MKL it will be necessary for you to have the MKL module loaded via the submit script when running the code.

The following link provides a useful tool for advising on the appropriate linking settings.

Intel MKL Link Advisor [http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/]

5.6. Available generic x86 performance analysis tools

There are several performance analysis tools available for applications on Stokes, including the Intel Trace Collector and Analyzer, the Scalasca performance analysis tool and gprof. A brief overview of how to get started with these performance analysis tools on Stokes is provided here. This section will also demonstrate how the SGI ‘perfcatch’ tool can be used on Stokes. The information provided should be seen as a guide for the ’impatient’ user. In order to access the full functionality of the performance analysis tools mentioned, the reader is encouraged to read the official user documentation for the relevant tool.

For each of the performance analysis tools mentioned below, an example workflow is provided. In most cases, the classical Molecular Dynamics software package, DL_POLY, has been chosen as an example application. DL_POLY is available on Stokes in its "pure MPI" flavour.

In this section we will describe how to use the 'Intel Trace Collector and Analyzer' and 'Scalasca' tools on Stokes.

5.6.1. Intel Trace Collector and Analyzer

Intel Trace Collector for MPI applications produces tracefiles that can be analyzed with Intel Trace Analyzer performance analysis tool. In MPI, it records all calls to the MPI library and all transmitted messages, and allows arbitrary user defined events to be recorded. Instrumentation can be switched on or off at runtime, and a powerful filtering mechanism helps to limit the amount of the generated trace data.

The Intel Trace Analyzer is a graphical tool that displays and analyzes event trace data generated by the Intel Trace Collector. It helps in detecting performance problems, programming errors or in understanding the behavior of the application of interest.

The typical use of the Intel Trace Collector and Analyzer is as follows: (1) Let your application run together with the Intel Trace Collector to generate one (or more) trace file(s), (2) Start the Intel Trace Analyzer and to load the generated trace for analysis.

5.6.1.1. Setting up the environment

When logging into Stokes enable X11 forwarding:

ssh –Y stokes

Load the Intel Trace Collector and (Analyzer) module (The default version on Stokes is version 8.0):

module load intel-trace

5.6.1.2. Instrumenting the source code

Generating a trace file from an MPI application can be as simple as setting just one environment variable or adding an argument to mpiexec. Assume you start your application with the following command:

mpiexec -n 4 my_code

Then generating a trace can be accomplished (on Stokes) by adding:

LD_PRELOAD= $(VT_ROOT)/slib/libVT.so mpiexec -n 4 my_code

or even simpler (for the Intel MPI Library):

mpiexec -trace -n 4 my_code

This will create a set of trace files named my_code.stf* containing trace information for all MPI calls issued by the application. If your application is statically linked against the Intel MPI Library you have to re-link your binary like this:

mpif90 -trace [all object files] -o my_code *mpiexec -n 4 my_code*

will then create the trace files named my_code.stf*.

In the case of DL_POLY, a Makefile is used for compilation, the relevant section of which is shown below:

hpc:

*$(MAKE) LD="mpif90 -trace -o" LDFLAGS="-O3" *

*FC="mpif90 -trace -c" FCFLAGS="-O3" *

The *-trace* flag should be postfixed to compile and link commands. Once the *-trace* flag has been inserted into the Makefile, load the other modules relevant to compilation:

module load intel-fc

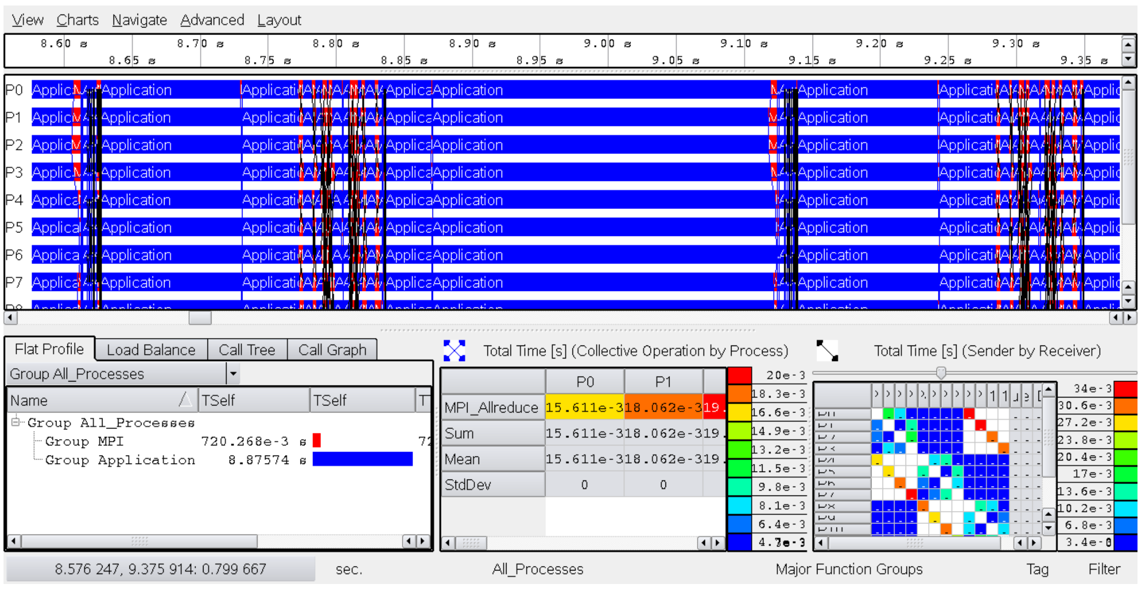
module load intel-cc

module load intel-mpi

and compile as usual:

make hpc

Figure 2. Intel Trace Analyzer output on Stokes



Name	TSelf	TSelf
Group All_Processes		
Group MPI	720.268e-3 s	🟥
Group Application	8.87574 s	🟦

	P0	P1
MPI_Allreduce	15.611e-3	18.062e-3
Sum	15.611e-3	18.062e-3
Mean	15.611e-3	18.062e-3
StdDev	0	0

8.576 247, 9.375 914: 0.799 667 sec.

The output of a trace analysis is shown in figure 1 where the chart layout of the Intel Trace Analyzer can be seen.

The Charts supported by Intel Trace Analyzer are divided into:

1. Timelines: Event Timeline, Qualitative Timeline, Quantitative Timeline and Counter Timeline.

2. Profiles: Function Profile, Message Profile and Collective Operations Profile.

Charts are grouped into Views. These Views provide ways to choose the time interval, the process grouping and optional filters that all Charts in the View use.

The output in figure 1 shows results for the 12 CPU core run of DL_POLY on Stokes described above (running on a single Stokes node). In the figure the Event Timeline, Message Profile and Collective Operations Profile charts can be seen. The Intel Trace Analyzer shows that for a 12 core run, most of the overall Wallclock time is spent in the application (as opposed to MPI calls). The charts also show that the dominant collective MPI operation in DL_POLY (for this test case) is MPI_ALLREDUCE. The Event Timeline shows that the application is reasonably well load-balanced for this particular run.

5.6.1.3. Further information:

The Intel Collector and Trace Analyzer has many more features that are beyond the scope of this mini-guide. To see the full functionality, visit intel.com [http://software.intel.com/en-us/articles/intel-trace-analyzer/]

5.6.2. Scalasca

Scalasca is a performance analysis toolset that has been specifically designed for use on large-scale systems but is also suitable for smaller HPC platforms using MPI and/or OpenMP. Scalasca integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of Scalasca is the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads.

5.6.2.1. Setting up the enviroment

When logging into Stokes enable X11 forwarding:

ssh –Y stokes

Load the Scalasca module (versions 1.3 and 1.4 available):

module load scalasca-intel (use 'module avail scalasca' for other options)

Run scalasca for brief usage information:

scalasca

5.6.2.2. Instrumenting the source code

By default, Scalasca uses the automatic compiler-based instrumentation feature. This is usually the best first approach, when you don’t have detailed knowledge about the application and need to identify the hotspots in your code. However, automatic function instrumentation may result in too many and/or too disruptive measurements, which can be addressed with selective instrumentation and measurement filtering (to see how this is done refer to the Scalasca user documentation at http://www.scalasca.org/software/documentation).

Typically, the *scalasca –instrument* (or *skin*) command is inserted before compile and link commands, e.g.:

scalasca -instrument mpicc -c my_code.c

scalasca -instrument mpicc my_code.o -o my_code.x

In the case of DL_POLY, a Makefile is used for compilation, the relevant section of which is shown below:

hpc:

*$(MAKE) LD="skin mpif90 -o" LDFLAGS="-O3" *

*FC="skin mpif90 -c" FCFLAGS="-O3" *

The *skin* command should be prefixed to compile and link commands. The instrumenter must be used with the link command. However, not all object files need to be instrumented, and it is often sufficient to only instrument source modules containing OpenMP and/or MPI references.

Once the scalasca instrument command has been inserted into the Makefile, load the other modules relevant to compilation:

module load intel-fc

module load intel-cc

module load mvapich2-intel/1.5.1p1

and compile as usual:

make hpc

5.6.2.3. Profiling the application

Ensure that the scalasca command is accessible when the batch script is executed by loading the Scalasca module. In the relevant section of your Stokes batch script, i.e., where the application execution command is found insert *scalasca –analyze* as follows:

scalasca –analyze mpiexec –np 12 ./DL_POLY.Z

and submit your job in the usual way.

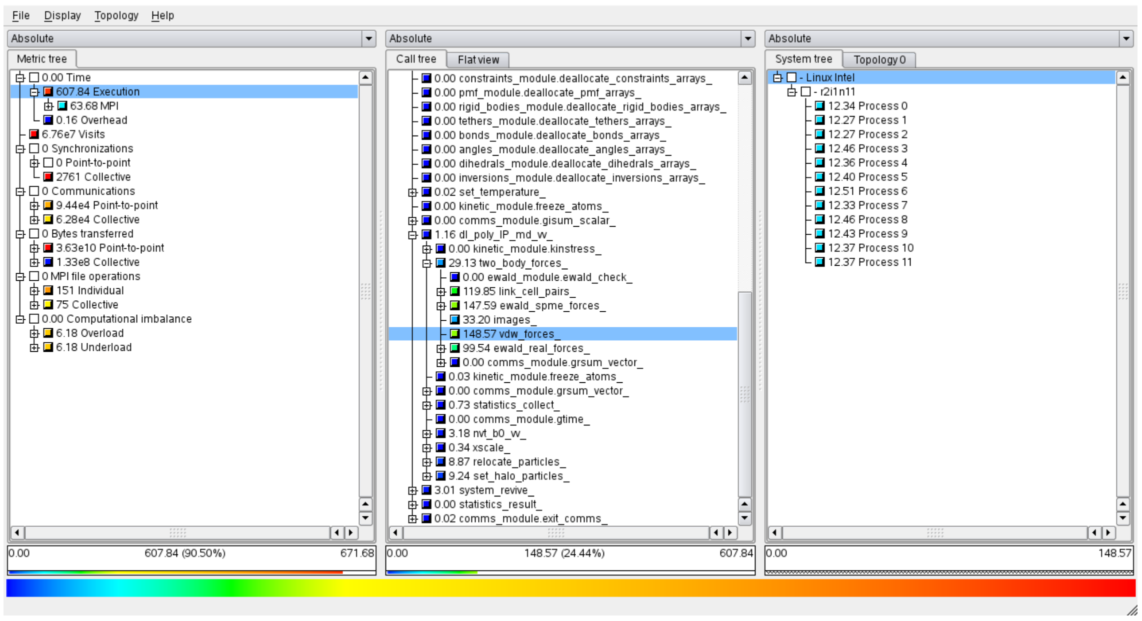
After successful execution of the job, a summary analysis report file is created within a new measurement directory. In this example, the automatically generated name of the measurement directory is epik_DLPOLY_12_sum.

The suffix _sum refers to a runtime summarization experiment. The summary analysis report can then be post-processed and examined with the Scalasca report browser as follows:

scalasca -examine epik_DLPOLY_12_sum

The output for the above run can be seen in figure 2 below:

Figure 3. Scalasca output on Stokes



```
0.00 Time
  607.84 Execution
    63.68 MPI
    0.16 Overhead
6.76e7 Visits
0 Synchronizations
  0 Point-to-point
  2761 Collective
0 Communications
  9.44e4 Point-to-point
  6.28e4 Collective
0 Bytes transferred
  3.63e10 Point-to-point
  1.33e8 Collective
0 MPI file operations
  151 Individual
  75 Collective
0.00 Computational imbalance
  6.18 Overload
  6.18 Underload

0.00          607.84 (90.50%)          671.68
```

```
0.00 constraints_module.deallocate_constraints_arrays_
0.00 pmf_module.deallocate_pmf_arrays_
0.00 rigid_bodies_module.deallocate_rigid_bodies_arrays_
0.00 tethers_module.deallocate_tethers_arrays_
0.00 bonds_module.deallocate_bonds_arrays_
0.00 angles_module.deallocate_angles_arrays_
0.00 dihedrals_module.deallocate_dihedrals_arrays_
0.00 inversions_module.deallocate_inversions_arrays_
0.02 set_temperature_
0.00 kinetic_module.freeze_atoms_
0.00 comms_module.gisum_scalar_
1.16 dl_poly_IP_md_w_
  0.00 kinetic_module.kinstress_
  29.13 two_body_forces_
    0.00 ewald_module.ewald_check_
    119.85 link_cell_pairs_
    147.59 ewald_spme_forces_
    33.20 images_
    148.57 vdw_forces_
    99.54 ewald_real_forces_
    0.00 comms_module.grsum_vector_
  0.03 kinetic_module.freeze_atoms_
  0.00 comms_module.grsum_vector_
  0.73 statistics_collect_
  0.00 comms_module.gtime_
  3.18 nvt_b0_w_
  0.34 xscale_
  8.87 relocate_particles_
  9.24 set_halo_particles_
3.01 system_revive_
0.00 statistics_result_
0.02 comms_module.exit_comms_

0.00          148.57 (24.44%)          607.84
```

```
- Linux Intel
  - r2i1n11
    12.34 Process 0
    12.27 Process 1
    12.27 Process 2
    12.46 Process 3
    12.36 Process 4
    12.40 Process 5
    12.51 Process 6
    12.33 Process 7
    12.46 Process 8
    12.43 Process 9
    12.37 Process 10
    12.37 Process 11

0.00                                   148.57
```

Results are displayed using three coupled tree browsers showing

– Metrics (i.e., performance properties/problems)

– Call-tree or flat region profile

– System location

When a node is selected from any tree, its severity value (and percentage) are shown in the panel below it, and that value distributed across the tree(s) to the right of it.

Selective expansion of critical nodes, guided by the colour scale, can be used to hone in on performance problems. The summary analysis above reveals that a significant percentage of overall wallclock time is spent in the 'vdw_forces' subroutine in DL_POLY, but also that the subroutine is well load balanced.

Each tree browser provides additional information via a context menu (on the right mouse button), such as the description of the selected metric or source code for the selected region (where available).

5.6.2.4. Further information:

Visit http://www.scalasca.org [http://www.scalasca.org] for more information.

5.6.3. gprof

gprof is well-known profiling tool which collects and arranges statistics on your application.

5.6.3.1. Compiling and linking codes for gprof

For enabling the generation at run-time of a file containing the profiling data gprof expects, the code has to be linked using the *-p* switch. This is the bare minimum. It will only give you access to function-level calling and timing information, but won't usually have a significant impact on the code's performance.

For collecting more accurate data, even to the level of per-line profiling, one can add, both at compilation and linking time, the *-p -g* compiler switches. Simply remember that on all compilers available on Stokes (as well as on most compilers available on the market), the *-g* switch also implies by default *-O0* which means that if you don't specify an optimisation level in addition to *-g*, gprof will silently turn off all compiler optimisations previously on. Indeed, to recover level O2 while profiling, one has to use *-p -g -O2*. Obviously, if you are using a different optimisation level in your code, like *-O3*, you can keep this level when profiling.

5.6.3.2. Collecting sampling data for gprof

gprof is a sample-based profiler, which means that at run-time, data regarding the code's instantaneous state (essentially the exact state of the call stack) will be collected and stored in a profiling trace every 0.01s. The profiling file is flushed to disk at the code's termination. This sampling frequency means that events with a typical duration less than 0.01s cannot be accurately profiled. However, when their number becomes large enough and the global run duration is several orders of magnitude larger than the sampling interval, gprof gives reasonably accurate information. To collect accurate profiling data, and provided the typical run duration is greater than a few seconds, one has simply to run one's code as usual. The profiling trace generated at the end of the run is called *gmon.out*.

One very important remark is that for MPI codes, without any further action, all the various MPI processes launched by *mpiexec* will generate their own *gmon.out* file, creating a race condition in which the final file is globally unexploitable. To avoid this, one has to set the environment variable *GMON_OUT_PREFIX* to a non-void character string. This string will be used by the system at trace file generation time as a prefix to use instead of gmon.out, and to which the traced process id will be appended. In the case of a ten process MPI run where one has use *export GMON_OUT_PREFIX="foo.bar"*, one can expect to get 10 profiling files at the end of the run, called *foo.bar.<pid>*, with 10 different values of pid. This solves the immediate MPI issue, but this does give the correspondance between pid and MPI rank. Be aware that on a compute node, the process ids and the MPI ranking are coherent, which means that if pids for 3 MPI processes are 10213, 10214 and 10215, then MPI rank 0 was pid 10213, MPI rank 1 was 10214 and MPI rank 2 was 10215. But this doesn't give you a ranking between compute nodes.

5.6.3.3. Generating profiling reports with gprof

Once the sampling traces generated, the typical command to generate the profiling report is:

gprof [option] <binary_name> <trace_name> [<other_trace_name> [...]]

gprof will generate a single profiling report aggregating all the data found in the aforementionned profiling traces. However, one will usually prefer to pass only one single profiling trace to get a clean pre-process profiling report.

gprof can generate 2 main types of profiling reports:

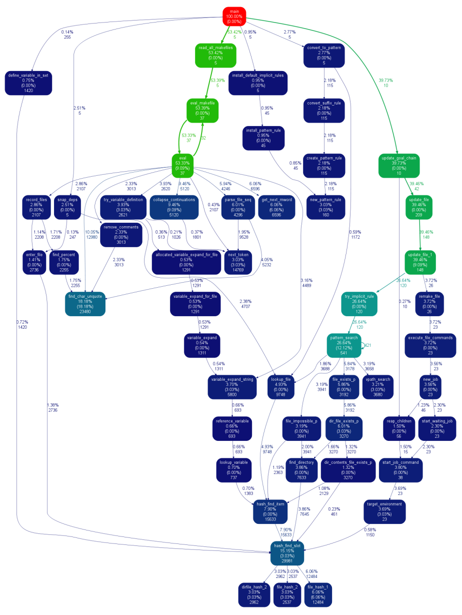
Flat profile: this is the most simple and common profiling report. It corresponds to simply display per-function or per-line accumulated times and occurence counting. The profile is given sorted as to print the most time consuming functions first. The per-function profile is triggered by the *-p* gprof option, and the per-line one by the *-l* gprof option. It has to be notice that the per-line profile might take a very long time to generate and that its accuracy very much depend on the optimisations performed by the compiler. In general, one will prefer starting with the per-function profiling and only go to the per-line profiling if the per-function doesn't give enough informations to figure out where to put optimisation efforts in the code.

For the functions which have been compiled with the *-p -g* switches, extra information regarding the number of calls and per-call time may be indicated. The call count can be a valuable piece of information for functions with very short execution time, but with a very large number of calls, for which an inlining might prove effective.

Call graph: this profile displays for each function, which function calls it and which function it calls, along with the number of calls. It also displays an estimate of the time spent in each function. This type of profile is triggered by the *-q* gprof switch.

This type of profile might be very useful to estimate where to put your parallelisation effort, e.g., by selecting the outermost function where parallelisation might be the most effective. The printed graph is hard to read, but one can benefit from post-processing tools such as *gprof2dot* to generate illustrative charts such as the one below:

Figure 4. gprof output on Stokes



5.7. System specific performance analysis tools

The only system specific performanace analysis tool available on Stokes is the SGI `perfcatch' tool.

5.7.1. SGI perfcatch

The simple-to-use Perfcatcher tool uses a wrapper library to return MPI and SHMEM function profiling information. Some analysis is done, and information like percent CPU time, total time spent per function, message size, and load imbalance are reported

To use perfcatch with an SGI Message Passing Toolkit MPI program, insert the perfcatch command in front of the executable name. Here are some examples:

mpirun -np 12 perfcatch ./DL_POLY.Z

To use perfcatch with Intel MPI, add the *-i* option:

```
mpiexec -np 12 perfcatch -i ./DL_POLY.Z
```

The rank-by-rank profiling information section lists every profiled MPI function called by a particular MPI process. The number of calls and the total time consumed by these calls is reported. Some functions report additional information such as average data counts and communication peer lists.