NEMO on HECToR
A dCSE Project

Fiona J. L. Reid
EPCC, The University of Edinburgh

May 4, 2009

Abstract

In this report we present the findings of the NEMO dCSE project which investigated the performance of a popular ocean modelling code, NEMO, on the Cray XT4 HECToR system.

Two different versions of NEMO (2.3 and 3.0) have been compiled and tested on HECToR. The performance of these versions has been evaluated and an optimal processor count suggested. The NEMO code is found to scale up to 1024 processors with the best performance in terms of AU usage being obtained between 128 and 256 processors. Square grids are found to give the best performance and where these cannot be used, choosing the grid dimensions such that `jpni < jpnj` is found to give the best performance. The removal of land only cells reduces the number of AU's by up to 25% and also gives a small reduction in the total runtime.

NetCDF 4.0, HDF5 1.8.1, zlib 1.2.3 and szip have been installed and tested. NetCDF 4.0 is found to give considerable reduction to both the amount of I/O and time taken in I/O when using the NOCSCOMBINE tool. The version of netCDF 4.0 installed under this dCSE project is 8-20% faster than the version installed centrally (via modules) on the system. NEMO has been adapted to use netCDF 4.0 for its main output files resulting in a reduction in file size of up to 3.55 times relative to the original code.

Nested models have also been investigated. The BASIC nested model has been compiled and tested and problems with the time step interval identified and rectified. The performance of the BASIC model has been investigated and an optimal processor count (in terms of AU usage) found to be 32.

Contents

1 Introduction **1**
1.1 The NEMO dCSE Project . . . 1
1.1.1 I/O performance . . . 1
1.1.2 Nested model performance . . . 1

2 What is NEMO? **1**

3 HECToR **2**
3.1 Introduction . . . 2
3.2 Architecture . . . 2
3.3 Modes of operation . . . 2
3.4 Communication and I/O networks . . . 3
3.5 Operating systems . . . 3

4 Compiling NEMO on HECToR **4**
4.1 PGI . . . 4
4.2 PathScale . . . 5

5 Running NEMO on HECToR **7**
5.1 Creating a new set of NEMO experiment data . . . 7
5.2 Combining the output data . . . 8
5.3 Using CDFTOOLS for verification of NEMO output . . . 8
5.4 Visualising the NEMO output . . . 10

6 NEMO performance **11**
6.1 Scaling for equal grid dimensions . . . 11
6.2 Single core versus dual core performance . . . 12
6.3 Performance for different grid dimensions . . . 13
6.4 Scaling plot . . . 13
6.5 Removing the land only grid cells . . . 13
6.6 Compiler optimisations . . . 17
6.7 Summary of benchmarking study . . . 18
6.8 Optimal processor count . . . 19
6.9 Time spent in file I/O and initialisation . . . 19

7 Aside on I/O strategies for parallel codes **23**
7.1 NEMO file I/O . . . 24

8 NetCDF performance and installation **24**
8.1 NetCDF 3.6.2 performance on HECToR . . . 24
8.2 Installing zlib 1.2.3 . . . 25
8.3 Installing HDF5 1.8.1 . . . 27
8.4 Installing netCDF 4.0 . . . 29
8.5 NOCSCOMBINE performance on HECToR . . . 31

9 NEMO V3.0 **33**
9.1 Compilation . . . 33
9.2 Performance for different compiler flags . . . 34
9.3 Performance of NEMO V3.0 . . . 35
9.4 Converting NEMO to use netCDF 4.0 . . . 35

10 NEMO AGRIF nested model running/debugging **40**
10.1 Introduction to the nested model problem . . . 40
10.2 BASIC nested model . . . 42
10.2.1 BASIC nested model performance . . . 46
10.3 MERGED nested model . . . 47

11 Conclusions and future work **51**
11.1 Summary of work and conclusions . . . 51
11.2 Other work . . . 52
11.3 Future work . . . 52

A NEMO output comparison scripts **53**

B Some notes on HDF5 datasets **57**
B.1 HDF5 Filters . . . 58

1 Introduction

This Distributed Computational Science and Engineering (dCSE) project is to investigate and where possible improve the I/O performance and nested model performance of the NEMO [1] ocean modelling code.

1.1 The NEMO dCSE Project

The NEMO dCSE project commenced on 1st March 2008 and is scheduled to end on the 30th April 2009. The principal investigator on the grant was Dr Andrew Coward from the National Oceanographic Centre, Southampton (NOCS). Dr Chris Armstrong at NAG provided help and support from the CSE side.

The project comprised of two work packages (referred to as WP1 and WP2), one concerned with improving the I/O performance of NEMO and one which involved investigating and improving the performance of nested models within NEMO. The motivation for investigating these two subject areas are discussed below.

1.1.1 I/O performance

The way in which data is currently input/output to NEMO is not considered ideal for large numbers of processors. As researchers move to use increasingly more complex models at high spatial resolutions larger numbers of processors will be required and this potential I/O bottleneck will therefore need to be addressed. WP1 involves investigating the current scaling and I/O performance of NEMO and identifying methods to improve this via the use of lossless compression algorithms within the NEMO model.

1.1.2 Nested model performance

The NEMO code allows nested models to be used which enable different parts of the ocean to be modelled with different resolution within the same global model. E.g. an area of the Pacific Ocean could be model at 1 degree resolution with the remainder of the Earth's Oceans being modelled at 2 degree resolution. This type of modelling can help scientists to gain a better insight into particular ocean features whilst keeping the computational costs reasonable. In the past, setting up such models has been very time consuming and relatively few attempts have been made to run such configuration on high performance computing systems. In WP2 we aim to investigate the performance of such nested models and to improve the performance subject to our findings. Our main goal is to achieve a stable and optimised nested model with known scalability.

2 What is NEMO?

NEMO (Nucleus for European Modelling of the Ocean) is a modelling framework for oceanographic research and operational oceanography. The framework allows several ocean related components e.g. sea-ice, biochemistry, ocean dynamics, tracers etc to work either together or separately. Further information on NEMO and its varied capabilities can be found at, [1].

The NEMO framework currently consists of three components each of which (except for sea-ice) can be run in stand-alone mode. The three components are:

- OPA9 - New version of the OPA ocean model, FORTRAN90
- LIM2 - Louvain-la-Neuve sea-ice model, FORTRAN90
- TOP1 - Transport component based on the OPA9 advection-diffusion equation (TRP) and a biogeochemistry model which includes the two components LOBSTER and PISCES.

This report focuses primarily on the OPA9 component and uses a version of the NEMO code which has been modified by the National Oceanography Centre, Southampton (NOCS) researchers. The NOCS version of NEMO is essentially the release version with NOCS specific enhancements. Two different versions of NEMO are discussed, version 2.3 and version 3.0. The initial work on the project involved version 2.3 and when version 3.0 became available (autumn 2008) work continued using this version.

3 HECToR

3.1 Introduction

HECToR is the UK's new high-end computing resource available for researchers at UK universities. The HECToR Cray XT5 system began user service in October 2007 and consists of a scalar (XT4) and a vector (X2) component. This project uses only the scalar component.

3.2 Architecture

The scalar XT4 component comprises 1416 compute blades, each of which has 4 dual-core processor sockets amounting to a total of 11,328 cores which can be accessed independently. Each dual-core socket consisting of a single dual-core processor is referred to as a *node*. The processor used is an AMD 2.8 GHz Opteron. Each dual-core node shares 6 GB of memory. The theoretical peak performance of the system is 59 Tflops.

Each of the AMD Opteron cores has a floating point addition unit and a floating point multiplication unit. These units are independent of each other which means that an addition and a multiplication operation can take place simultaneously. The processor is capable of completing a single floating point operation from each of these units per cycle. Given the clock speed of 2.8 GHz this gives us a theoretical peak performance of 2 * 2.8 = 5.6 Gflops per core or 11.2 Gflops per dual core for double precision floating point operations.

The caches on each core are private. Unlike many systems there are no shared caches on HECToR. Each core has a separate 2-way set associative level 1 cache of 64 kB. The level 2 cache is a 16-way combined data and instruction cache totalling 1 MB. Both the level 1 and 2 caches use 64 byte cache lines, equating to eight double precision words. The level 2 cache acts as a victim cache for the level 1 cache which means that data evicted from the level 1 cache gets placed onto the level 2 cache.

3.3 Modes of operation

When running jobs on the system users can choose whether they wish to run in single node (SN) mode or virtual node (VN) mode. In SN mode only one of the dual core

processors will be utilised leaving the second core idle. In this case all 6 GB of memory will be available to the single core. In VN mode both of the cores are used and the memory gets shared between them, making 3 GB available to each core. At present the 6 GB memory is composed of one 2 GB and one 4 GB dimm which means round-robining or stripping of memory is not possible. For codes which are memory bound their performance may be adversely affected as a result. In future it is hoped that the system will be re-configured to provide a symmetric memory distribution, e.g. two 2 GB or two 4 GB dimms.

3.4 Communication and I/O networks

The nodes communicate via a high bandwidth interconnect which uses the Cray SeaStar2 communication chips. Each dual core processor has its own private SeaStar2 chip which is connected directly into its HyperTransport link. The SeaStar chips are arranged on a 3-dimensional torus with dimensions 20 x 12 x 24. Each SeaStar2 chip provides high speed links to its 6 neighbours (see figure 1). Each link is capable of delivering a peak bi-directional bandwidth of 7.6 GB/s.

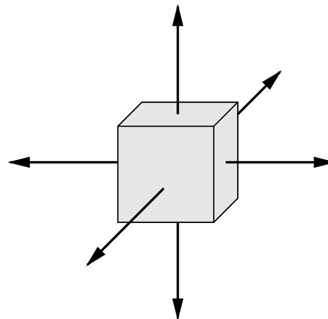


Figure 1: Diagram illustrating the directionality of the 6 links coming from each SeaStar2 chip.

In addition to the compute nodes there are also dedicated I/O nodes, login nodes and nodes set aside for serial compute jobs. The login nodes can be used for editing, compilation, profiling, de-bugging, job submission etc. When a user connects to HECToR via `ssh` the least loaded login node is selected to ensure that users are evenly loaded across the system and that no single login node ends up with an excessive load.

HECToR has 12 I/O nodes which are directly connected to the toroidal communications network described above. These I/O nodes are also connected to the data storage system (i.e. physical disks). The data storage on HECToR consists of 576 TB of high performance RAID disks. The service uses the Lustre distributed parallel file system to access these disks.

3.5 Operating systems

The service nodes (i.e. login and I/O) run SUSE Linux. The compute nodes run a lightweight Linux kernel called the Cray Linux Environment, or CLE. CLE was formally known as Compute Node Linux (CNL) and some documentation may still refer to this.

CLE is essentially a stripped down version of Linux (c.f. Blue Gene's compute node kernel, CNK). It is designed to be extremely lightweight so as to limit the number of interruptions from the operating system. The rationale is to keep the compute nodes as uninterrupted by the operating system as possible by outsourcing the usual operating system tasks to dedicated additional hardware. The benefits of this are potentially excellent scalability up to large task counts which should results in minimal variation of application run time. The downside of this reduced kernel is that some services which a small number of applications expect or rely on, e.g. access to the X11 libraries, are not available on the compute nodes.

4 Compiling NEMO on HECToR

In this section we describe how to compile NEMO on the HECToR system. The system has three different compiler suites available; Portland Group International (PGI), PathScale and GNU. Generally, PGI and PathScale are found to give the best performance for Fortran/Fortran 90 codes so we will only consider these two compiler suites. Further information on the compilers available on HECToR can be found in the HECToR User Guide [2].

The NEMO code utilises the netCDF (network Common Data Form) library for inputting and outputting its data files which means before compilation can begin the netCDF library must be available or compiled from source. Further information on netCDF can be found at [3, 4].

NEMO can potentially be run on any number of processors providing sufficient memory is available to fit the required subset of data onto a single processor. The number of processors must be specified at compile time as the dimensions of a number of statically allocated arrays are computed based on the processor count. This hard-wiring of the processor count into code means that the code must be recompiled if a different number of processors is used.

The NEMO Makefile is somewhat complex. The internal macro names (those predefined in make) such as `FC`, `FFLAGS`, `LDFLAGS`, etc are not used. Instead, the code authors use their own naming convention and as no comments have been included to describe these the user is left to make their own decision as to what each macro name stands for. Table 1 lists the NEMO Makefile macro names and what they are believed to correspond to using the standard Makefile macro naming conventions.

4.1 PGI

To compile the NEMO model using the PGI compiler the following flags are set in the Makefile:

```
EXEC_BIN = /work/n01/n01/fionanem/NEMO/ORCA025/bin/orca025k64
# netcdf library and includes
NCDF_INC = /home/n01/n01/sga/PACKAGES/include
NCDF_LIB = -L/home/n01/n01/sga/PACKAGES/lib -lnetcdf

P_C = ftn -Mcpp=comment
P_O =
```

NEMO Makefile macro	Standard make macro	Description
P_C	CPP	Pre-processor with default flags
P_P	CPPFLAGS	Pre-processor flags
P_O	-	Pre-processor, specific optimisation flags
M_K	MAKE	Name of make utility
F_C	FC	Specifies Fortran compiler and flags
F_L	LD	Fortran linker
F_O	FFLAGS	Fortran optimisation flags
A_C	AR	Archiver
L_X	LDFLAGS	Fortran linker flags

Table 1: NEMO Makefile macros with corresponding standard Makefile macro name if applicable and description

```
M_K = gmake
F_C = ftn -c
F_L = ftn
A_C = ar -r
F_O =  -O3 -r8  -I $(MODDIR) -I$(MODDIR)/oce  -I$(NCDF_INC)
L_X =  -O3 -r8
```

The Fortran wrapper script `ftn` invokes the `pgf90` compiler and includes the appropriate paths to the MPI library which means that this doesn't need to be added in at link time. `NCDF_INC` and `NCDF_LIB` specify the location of the netCDF library and associated include files. The pre-processor option `-Mcpp=comment` ensures that C-style comments are retained in the pre-processed output. The `-O3` flag specifies the level of optimisation applied. The `-r8` flag ensures that all variables specified as `REAL` are interpreted as `DOUBLE PRECISION`.

For the problem we are concentrating on, processor counts of less than 96 currently will not compile with the PGI compiler. The error message obtained is of the form:

```
/opt/pgi/7.0.4/linux86-64/7.0-4/lib/libpgf90.a(initpar.o)(.text+0x2):
In function `__hpf_myprocnum':
: relocation truncated to fit: R_X86_64_PC32 __hpf_lcpu
```

The reason for this error is that when NEMO is compiled for smaller processor counts, more than 2 GB of address space is required. This is a known feature of the PGI compiler and will be fixed in future releases of PGI compiler and system libraries. At present, the PathScale compiler is the only alternative if runs using smaller processor counts are required.

4.2 PathScale

If address space in excess of 2 GB is required (i.e. using less than 96 processors) then we have to use the PathScale compiler. As mentioned earlier, NEMO uses the netCDF library. The version of netCDF in `sga/` and the version in the package account (under `/usr/local/packages/netcdf`) were both compiled using the PGI compiler. This

means the object files are PGI specific and won't work with the PathScale compiler. Therefore, before compiling NEMO with PathScale we must also compile netCDF with the PathScale compiler to ensure that compatible object files are created. The options used to compile the netCDF library on HECToR were as follows:

```
module swap PrgEnv-pgi PrgEnv-pathscale

export CC='cc -DpgiFortran'
export FC='ftn -cpp -DpgiFortran -fno-second-underscore'
export F90='ftn -DpgiFortran -cpp -fno-second-underscore'
export CXX='CC -DpgiFortran'
./configure --prefix=/home/n01/n01/fionanem/netcdf/3.6.2 --disable-cxx

make
make check
make install
```

The `module swap PrgEnv-pgi PrgEnv-pathscale` command swaps from the default (PGI) programming environment to the the PathScale programming environment. The `-fno-second-underscore` ensures only a single underscore is used when calling external library routines. The `./configure` script must be run with the `--disable-cxx` option. This option prevents the C++ API from being built. If this option is not included then the link stage will fail when attempting to link the shared library `libgcc_s`.

To compile the NEMO model using the PathScale compiler the following flags are set in the Makefile:

```
EXEC_BIN = /work/n01/n01/fionanem/NEMO/ORCA025/bin/orca025k64_path
# netcdf library and includes
NCDF_INC = /home/n01/n01/fionanem/netcdf/3.6.2/include
NCDF_LIB = -L/home/n01/n01/fionanem/netcdf/3.6.2/lib -lnetcdf

P_C = ftn -Mcpp=comment
P_O =
M_K = gmake
F_C = ftn -c -P
F_L = ftn
A_C = ar -r
F_O =  -O3 -r8  -I $(MODDIR) -I$(MODDIR)/oce  -I$(NCDF_INC)
L_X =  -O3 -r8
```

As before, `NCDF_INC` and `NCDF_LIB` specify the paths to the netCDF library compiled using the PathScale compiler suite. The `F_C` flag needs the `-P` flag to ensure that the pre-processor removes lines containing `#` in the output. If these lines are not removed then the Fortran compiler attempts to compile these and the compilation will fail.

5 Running NEMO on HECToR: creating new experiment data, verifying the output and visualising data

A typical NEMO run has a number of input and output files which vary depending on the particular model being solved. The input and output files used for the test configuration are summarised below:

- Input files
  - `namelist` - contains parameters which control the run, e.g. number of timestep, output frequency, which data to compute etc.
  - Various netCDF files which contain information required by the code.
- Output files
  - `ocean.output` General info on run
  - `solver.stat` Output from solver for each time step
  - `time.step` Timing info for each model time step
  - `ice_evolu` Information relating to sea-ice
  - `date.file` Contains info which gives the date range over which the model runs, e.g. O25-TST_1m_19580101_19580101_ for this example.
  - `layout.dat` Contains info relating to the layout used for this run.
  - `*.nc` - netCDF format, one file per processor

The number of netCDF (`*.nc`) output files is determined by the particular output files requested in the `namelist` and the number of processors on which NEMO is executed. For the test configuration, 7 sets of `*.nc` files are output with names of the form:-

```
O25-TST_1m_19580101_19580101_grid_T_0100.nc
O25-TST_1m_19580101_19580101_grid_U_0100.nc
O25-TST_1m_19580101_19580101_grid_V_0100.nc
O25-TST_1m_19580101_19580101_grid_W_0100.nc
O25-TST_1m_19580101_19580101_icemod_0100.nc
O25-TST_00000060_restart_0100.nc
O25-TST_00000060_restart_ice_0100.nc
```

Thus, if NEMO is executed on 221 processors then, on completion, a total of, 221*7 = 1547, `*.nc` files are generated by the run. The `*grid*.nc` and `*icemod*.nc` files contain model information for each time step. The `*restart*.nc` files are files which enable the model to be restarted from a particular point in model time.

5.1 Creating a new set of NEMO experiment data

This section describes how to create a new set of NEMO input data. This is useful as it allows multiple sets of identical input data to be generated in separate directories which is useful when benchmarking the code. Andrew Coward supplied several scripts in the `ORCA/bin` directory which set up new experiment directories. They are used as follows:-

1. Edit `setup_ex1` to change the experiment number from 001 to the appropriate value, e.g. 002, 003 etc.

2. Execute `setup_ex1`. This script creates a new directory with the appropriate number (e.g. `EXP002`) and populates it with symbolic links to the invariant model data. It also copies in the default versions of the namelist control files and several management scripts.

3. Edit `linkcore` to change the experiment number (e.g. `set nex = 002`) as used in step 1. Also ensure that the path to the experiment directory is correctly set.

4. Run `linkcore`. This script creates the symbolic links for to the DFS3 versions of the CORE forcing fields.

With NEMO V3.0 the `setup` and `linkcore` scripts were modified so that they take the name (number or characters) that will be pre-pended to the experiment directory name. E.g. the following commands will create a new run directory called `EXP_V3.0_001`:

```
./setup_experiment 001
./linkcore_experiment 001
```

5.2 Combining the output data

When NEMO executes, each processor writes out the particular part of the ocean on which it operated to a separate netCDF (`.nc`) file. In order to perform any analysis or verification of the output data these netCDF files must first be recombined into a single netCDF file on which the viewers (e.g. nemoplotnc, ncview etc) and diagnostic tools (e.g. CDFTOOLS) can operate. The `nocscombine` code can be used to perform this recombination. To recombine a series of netCDF 3.6.2 files using `nocscombine` the following command can be used:

```
nocscombine -f O25-TST_CU30_19580101_19580101_grid_T_0000.nc -d votemper
```

The `-f` specifies the file series which is to be recombined, the `-d` specifies the particular field (e.g. temperature, salinity, velocity etc) that you wish to extract. If no `-d` is specified then the entire netCDF file will be recombined. The default output file name will be as given to the `nocscombine` tool with the processor number stripped away, e.g. `O25-TST_CU30_19580101_19580101_grid_T.nc` for the example above. If required, the output file can be altered with the `-o` flag. The options for `nocscombine` can be viewed with the `nocscombine -h` command.

5.3 Using CDFTOOLS for verification of NEMO output

Once a single netCDF file has been created as described in Section 5.2 the data can be compared against vanilla (i.e. before any changes have been made to the code) output using the CDFTOOLS package. CDFTOOLS is package of Fortran 90 programs and libraries for performing diagnostic tests on NEMO output. Further details on CDFTOOLS and the individual tools can be found at [5]. One field from each of the 5 output grids (restart files are ignored) is compared against the vanilla output. The grids and fields compared are given below:

```
Grid        Field       Physical property
GRID T      votemper  = temperature (C)
GRID U      vozocrtx  = zonal current (m/s)
GRID V      vomecrty  = meridional current (m/s)
GRID W      vovecrtz  = vertical velocity (m/s)
GRID icemod isssalin  = sea surface salinity (PSU)
```

Comparing a global single field from each of the main output datasets should be sufficient verification that the modified code gives correct results. It would be excessive to compare every single field, besides which the length of time required to run the verification would become prohibitive.

It should be noted that the NEMO output files need to contain some actual data. Running with the default `namelist` settings specifies that data will be output every 300 time steps which is fine for production runs. However, as our test run is only for 60 time steps this means that no real data is actually written out and the CDFTOOLS codes will crash. Sample error output for such a crash is below:

```
 ~/CDFTOOLS/bin/cdfmeanvar votemper_ncdf3.nc votemper T
 npiglo=        1442
 npjglo=        1021
 npk   =          64
 nvpk  =          64
  Problem in getvar for votemper
 ERROR in NETCDF routine, status=          -40
 NetCDF: Index exceeds dimension bound
FORTRAN STOP
```

Thus, to ensure that data is written out the output frequency must be changed from 300 to 30 time steps. This is achieved by changing the value of `nwrite` in the `namelist` file.

Then, the procedure for verifying that the NEMO output is sensible is as follows:-

1. For each dataset (i.e. vanilla and the one to be tested) combine the processor specific output files into a single netCDF file using `nocscombine` as described in Section 5.2.

2. Inside the particular experiment directory, create soft links to several map files `mask.nc, new_maskglo.nc, mesh_zgr.nc and mesh_hgr.nc` which are required by the CDFTOOLS, e.g.

   ```
   ln -s /work/n01/n01/acc/DATA/ORCA025/mask.nc        mask.nc
   ln -s /work/n01/n01/acc/DATA/ORCA025/new_maskglo.nc new_maskglo.nc
   ln -s /work/n01/n01/acc/DATA/ORCA025/mesh_zgr.nc    mesh_zgr.nc
   ln -s /work/n01/n01/acc/DATA/ORCA025/mesh_hgr.nc    mesh_hgr.nc
   ```

   If you forget to create these soft links the following somewhat cryptic error is obtained:-

   ```
   fionanem@nid15875:
   /work/n01/n01/fionanem/NEMO_V3.0/ORCA025/EXP_V3.0_005>
   ```

```
~/CDFTOOLS/bin/cdfmeanvar votemper_ncdf3.nc votemper T
 npiglo=         1442
 npjglo=         1021
 npk   =           64
 nvpk  =           64
 ERROR in NETCDF routine, status=           2
 No such file or directory
FORTRAN STOP
```

3. For each dataset compute the spatial 3D mean (i.e. over all depth levels) temperature and variance using the `cdfmeanvar` tool as follows:-

```
cdfmeanvar vanilla_inputfile.nc votemper T > vanilla_output_votemper_T.txt
cdfmeanvar test_inputfile.nc votemper T > test_output_votemper_T.txt
```

4. Compare the two output files using `diff`, `xxdiff` or similar. Ideally they should be identical or any differences should be explainable.

5. Repeat steps 1-4 for each grid and field to be tested.

As this verification will be necessary following any changes to the code it is sensible to have some scripts to perform it. Such scripts can be found in Appendix A. Essentially the verification scripts compare the NEMO output to vanilla output and note any differences. The verification takes around 20-40 minutes depending on the number of parameters being tested and thus must be run on the serial queue to avoid excessive use of the login node resources.

5.4 Visualising the NEMO output

A visual check of the output data can tell us very quickly whether the code has run successfully and can also be useful in trouble shooting any problem runs. E.g. if a particular part of the model has failed (e.g. the waters around the antarctic region) visualising the results can be much more informative than looking at the differences between the vanilla and problem output files.

Output from NEMO can be viewed using the `nemoplotnc` tool. Prior to using this tool some environment variables must be set to allow colour palette files and default data paths to be set. `COLOUR2_DIR` contains the path to the colour palette files and `DATA_DIRPATH` contains the path to the netCDF file to be read in. These can be set in bash shell as follows:

```
export COLOUR2_DIR = /home/n01/n01/acc/NEMOPLOT/colour2
export DATA_DIRPATH = /work/n01/n01/fionanem/NEMO/ORCA025/EXP001/
  O25-TST_CU30_19580101_19580101_grid_T.nc
export VMASK1 = 1.e20
```

`VMASK1` is used to set the mask value (the value assigned to the land cells) such that the land cells get ignored when the data are plotted. Once these environment variables have been set the `nemoplotnc` command can be executed as follows:

```
~/NEMOPLOTNC/nemoplotnc
```

`nemoplotnc` can be used to view both single processor and combined netCDF files. Figure 2 shows a screen-snapshot from `nemoplotnc` for the temperature field generated as described in Section 5.3.

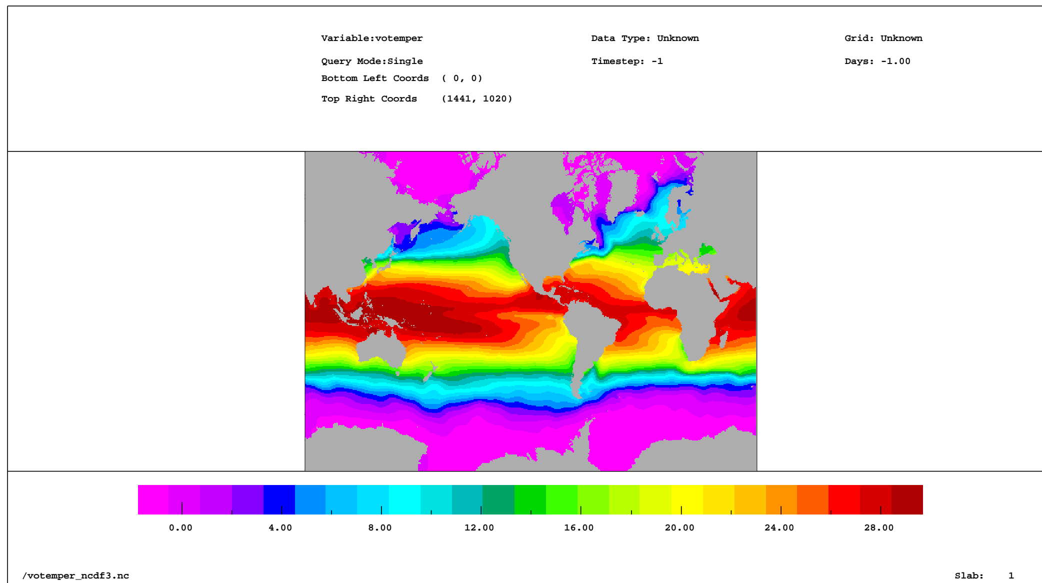


Figure 2: Votemper from NEMO after 30 time steps

6 NEMO performance

To investigate the performance of the NEMO code we run the code using different numbers of processors and different grid configurations. We also investigate whether the particular compiler used makes any difference. The performance of NEMO is measured by examining the `time.step` file output by the code. This reports the model step number and time taken in seconds. The benchmark dataset runs for a total 60 model time steps and so our benchmark data reports the time taken for 60 time steps. All the the results presented in this section correspond to version 2.3 of NEMO. Results from version 3.0 of NEMO are presented separately.

The number of processors on which NEMO will be run is set in the source file named, `par_oce.F90`. This file also sets up the grid dimensions over which the model will be decomposed. The variables `jpni`, `jpnj`, and `jpnij` specify respectively the number of processors in the i direction, the number of processors in the j direction and the total number of processors. E.g. a 16x16 processor grid which runs on 256 processors would have `jpni = 16`, `jpnj = 16` and `jpnij = 256`.

6.1 Scaling for equal grid dimensions

We begin by investigating the scaling of the code for grids of equal dimension, i.e where `jpni = jpnj`. This restricts us to a relatively small number of processors counts ranging from 64 (8x8) to 1024 (32x32). The PathScale compiler is used for processor counts less than 96. The results are shown in figure 3.

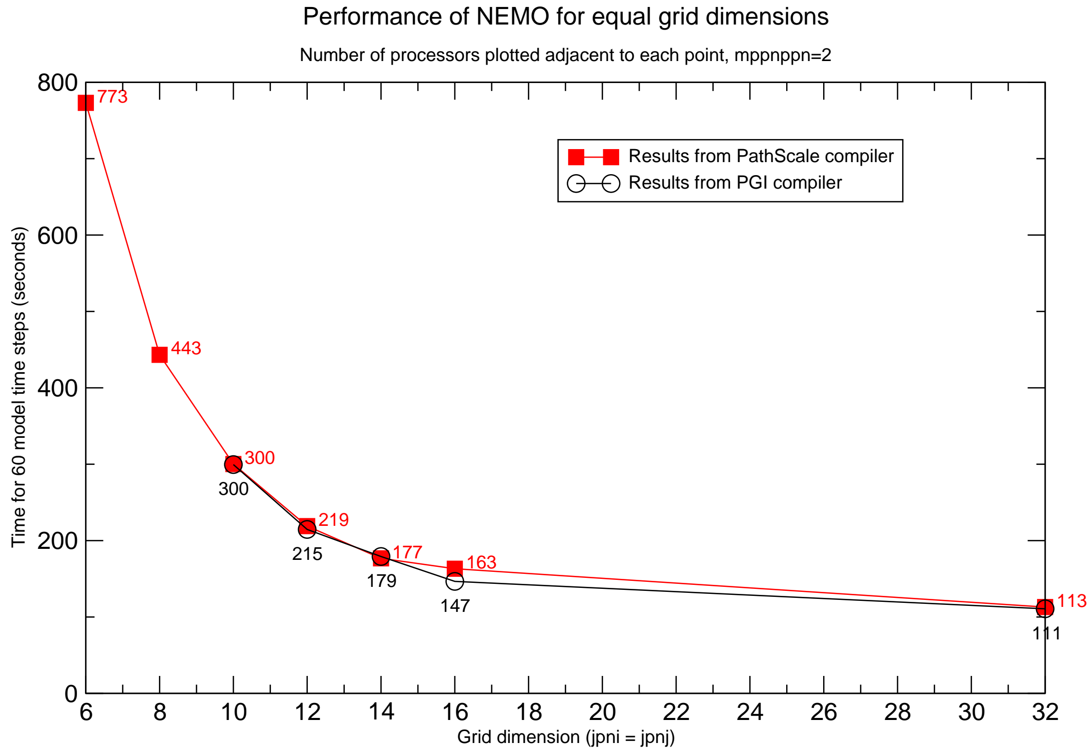


Figure 3: Performance of NEMO when $jpni = jpnj$ for the PGI and PathScale compiler suites.

From figure 3 it is clear that the PGI compiler performs slightly better (a few percent) than the PathScale compiler.

6.2 Single core versus dual core performance

Comparison of single node versus virtual node mode shows that the runtime is generally faster when running in single node mode. The likely reason for this is that single node mode will create less contention for both memory and I/O nodes than running in virtual node mode. Table 2 shows the runtimes for 256 and 221 processors using a 16 by 16 grid for both single and virtual node modes. It should be noted that the 221 processor run has had the land only cells removed. The effect of removing land only cells will be examined in section 6.5.

From table 2 we can see that single node mode is up to 18.59% faster than virtual node mode. As the charging structure on HECToR is per core, single node mode will cost significantly more (almost double) AU's than virtual node mode. Thus, running NEMO in single node mode should only be considered if it's critical to obtain a fast solution.

jpnij	jpni	jpnj	Time for 60 steps (seconds)	
			mppnppn=1	mppnppn=2
256	16	16	119.353	146.607
221	16	16	112.542	136.180

Table 2: Runtime comparison for 60 time steps for single node (mppnppn=1) and virtual node (mppnppn=2). Runs were performed using the PGI compiler

6.3 Performance for different grid dimensions

Using a fixed number of processors we investigate how the shape of the grid affects the performance. We concentrate on 128 and 256 processors with two results from a 512 processor run. All runs are carried out using the PGI compiler suite. The results of this experiment are shown in figure 4.

Figure 4 suggests that for a fixed number of processors the ideal grid dimensions are square i.e. where `jpni=jpnj`. If the number of processors is such that it is not possible to have `jpni=jpnj` (i.e. the number of processors is not a square of an integer) then the results suggest that the values of `jpni` and `jpnj` should be as closer to each other as possible with the value of `jpni` chosen such that `jpni < jpnj`.

6.4 Scaling plot

We also look at the scaling of NEMO from 128 to 1024 processors. Where possible equal dimension grids have been used. In situations where this was not possible, e.g. 128 and 512 processors the grid size has been chosen as close to square as possible and such that $jpni < jpnj$ as this has shown to yield the best performance.

Figure 5 shows the scaling of NEMO for both the PGI and PathScale compilers.

From figure 5 it's clear that NEMO continues to scale out to 1024 processors but the benefit in using more processors is purely a reduction in runtime and not efficient in terms of AU's used. 128 or 256 processors seem to give the best compromise between AU use and runtime. As seen previously, the PGI compiler performs slightly better than the PathScale compiler for all processor counts tested.

6.5 Removing the land only grid cells

So far we have considered decompositions in which all the grid cells are used, i.e. those where the code has `jpnij = jpni x jpnj`. However, many decompositions give rise to grid cells which contain only land. These land only cells are essentially redundant in an ocean model and can be removed. In the code this means that the value of `jpnij` can be reduced such that `jpnij <= jpni x jpnj`. It is anticipated that removing land only cells may improve the performance of the code as branches into land only regions will no longer take place and any I/O associated with the land cells will also be removed. Furthermore, the number of AU's required will be reduced as fewer processors will be required if the land cells are removed.

The NEMO code does not automatically remove the land cells which means the user needs to use the chosen decomposition and then separately determine how many cells contain only land. A tool written by Andrew Coward can be used to determine the

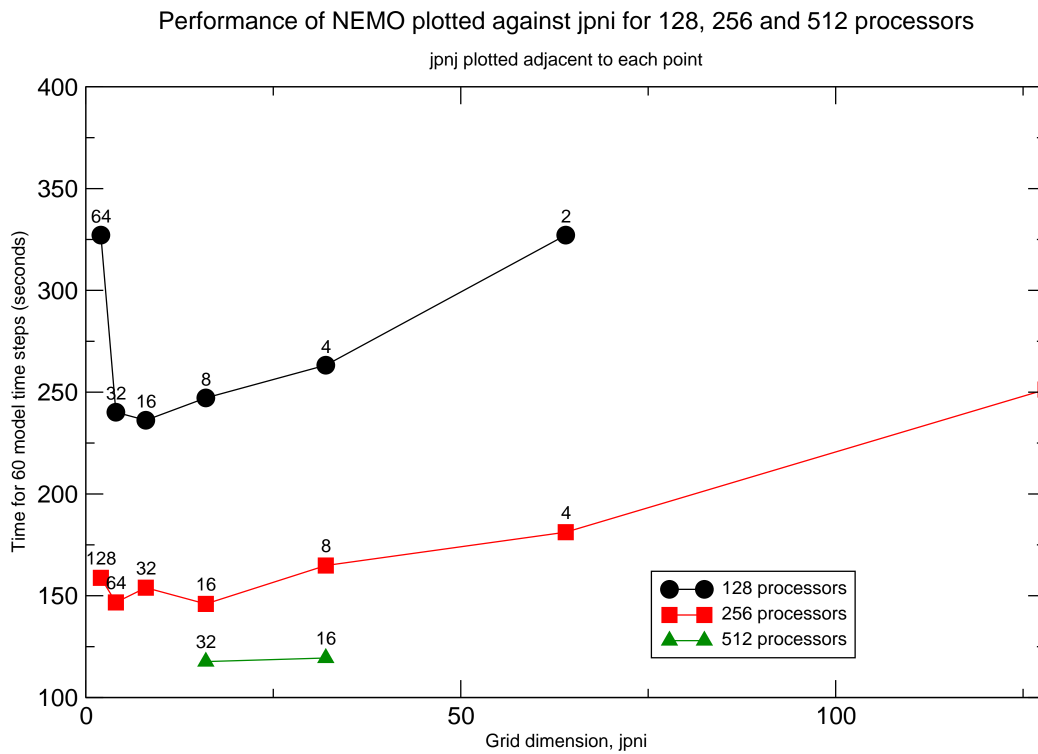


Figure 4: Performance of NEMO on 128, 256 and 512 processors plotted against the number of grid cells in the i direction, $jpni$

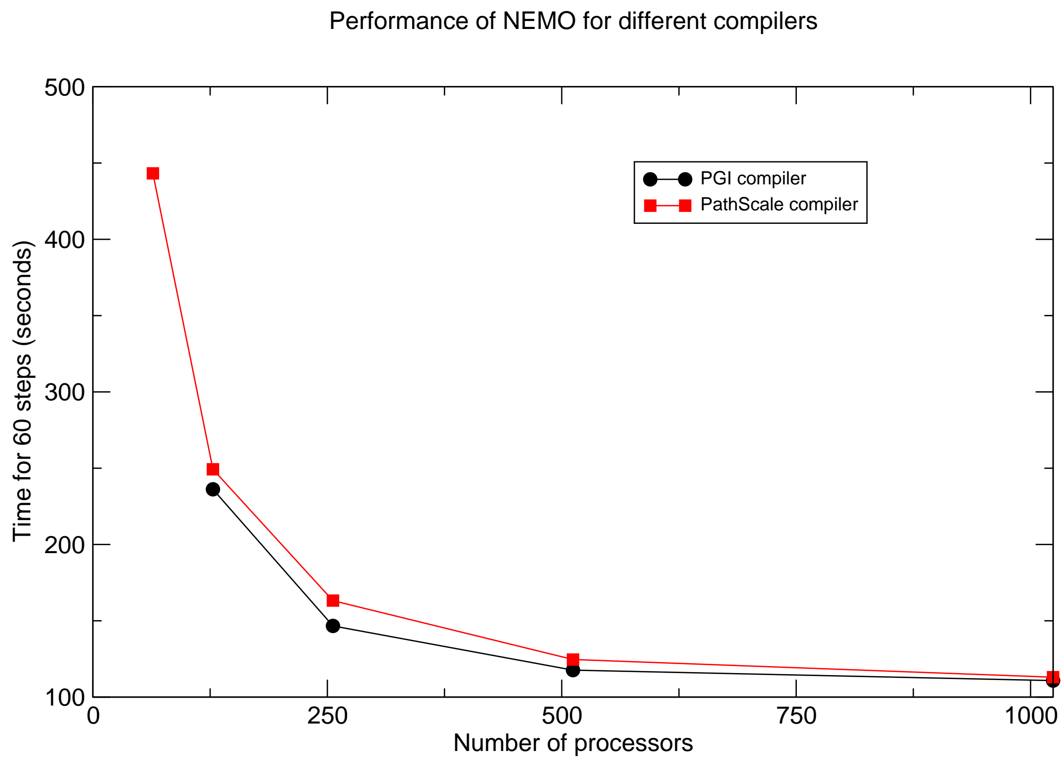


Figure 5: Scaling of NEMO for the PGI and PathScale compilers. The grid dimensions used were respectively; 128 = 8 x 16, 256 = 16 x 16, 512 = 16 x 32 and 1024 = 32 x 32.

number of active (ocean containing) and dead (land only) cells. The procedure for doing this is as follows:

- Use the `nocsprocmap` code to generate the `layout.dat` file for the required decomposition. E.g running the command
  `~acc/NTOOLS/NOCSPROCMAP/nocspmap_r25 -f bathy_meter.nc -i 16 -j 16 -s`
  gives the number of active (i.e. ocean only) regions for a `jpni = 16` by `jpnj = 16` processor grid.
- Alter the appropriate line of `par_oce.F90` so that the value of `jpnij` is reduced such that the the land only squares are removed. For a 16 by 16 grid, there are 35 land only squares and thus `jpnij = 221` instead of 256.

Table 3 gives the number of land only cells for a variety of grid dimension configurations. The reduction in the number of processors required is generally around 10%. For very large (>256) processor counts the reduction can be considerably larger and as much as 25%.

jpni	jpnj	**Total cells**	**Land only cells**	**Percentage saved**
6	6	36	0	0.00%
7	7	49	1	2.04%
8	8	64	2	3.13%
9	9	81	6	7.41%
10	10	100	10	10.00%
11	11	121	13	10.74%
12	12	144	14	9.72%
13	13	169	21	12.43%
14	14	196	22	11.22%
15	15	225	29	12.89%
16	16	256	35	13.67%
20	20	400	65	16.25%
30	30	900	193	21.44%
32	32	1024	230	22.46%
40	40	1600	398	24.88%
16	8	128	117	8.59%
32	16	512	92	17.97%

Table 3: Number of land only squares for a variety of processor grids. The percentage saved gives the percentage of cells saved by removing the land only cells and will correspond to the reduction in the number of AU's required for the computation.

We now investigate whether removing the land only cells has any impact on the runtime of the NEMO code. We hope that by avoiding branches into land only regions and the associated I/O involved with the land cells that the runtime should reduce. For this test we have considered only 128, 256, 512 and 1024 processor grids. The results are given by table 4.

jpni	jpnj	jpnij	**Time for 60 steps (seconds)**
32	32	1024	110.795
32	32	794	100.011
16	32	512	117.642
16	32	420	111.282
16	16	256	146.607
16	16	221	136.180
8	16	128	236.182
8	16	117	240.951

Table 4: Runtime comparison for 60 time steps for models with/without land squares included on 128, 256, 512 and 1024 processor grids.

From table 4 we can see that for 256 processors and above removing the number of land squares reduces the total runtime by up to 10 seconds which corresponds to a reduction of around 7-10%. For a 128 processors run, removal of the land-only cells actually gives a small increase in the total runtime. This difference is within normal repeatability errors and could be a result of heavy load on the system when the test was run. As the runtime does not seem to improve greatly with the removal of the land only cells the main motivation for removing these cells is to reduce the number of AU's used for each calculation. Assuming the runtime is not affected detrimentally then the reduction in in AU usage will be as given by table 3.

The times given in table 4 are the time that the NEMO code reports when it writes the information from time step 60 to disk. This, however is not the whole story. At the end of the run, NEMO also dumps out the restart files required to restart the computation from the final time step. These restart files are significantly larger than the files output at each individual time step and thus take a reasonable amount of time to write out to disk. Unfortunately the code does not output any timings which include the writing of these restart files. One way to get an estimate of the time taken to write out these restart files is to look as the actual time taken by the parallel run as reported by the batch system. The PBS output files gives the walltime in hh:mm:ss. By subtracting the time taken for 60 steps from walltime we can get an estimate of the time taken over and above the step by step output, i.e. we can get an estimate of the time taken to read in the input data and output the final restart files. To get accurate time estimates timers should be inserted into the code but as a first pass this method will let us find out whether there is any variation with processor count. The amount of time that NEMO spends in I/O and initialisation will be discussed in Section 6.9.

6.6 Compiler optimisations

In this section we investigate whether any compiler optimisations can be used to improve the performance of NEMO. We investigate a number of different compiler flags for both the PGI and PathScale compilers and investigate the performance for a 16 by 16 grid running on 221 processors. Tables 5 and 6 shows the results obtained for the PGI and PathScale compilers respectively.

Tables 5 and 6 show that the best performance is obtained using `-O3 -r8`. More

Compiler flags	Time for 60 steps (seconds)
-O0 -r8	173.105
-O1 -r8	169.694
-O2 -r8	151.047
-O3 -r8	141.529
-O4 -r8	144.604
-fast -r8	fails on step 6
-fastsse -r8	fails on step 6
-O3 -r8 -Mcache_align	155.933

Table 5: Runtime for 60 time steps for different compiler flags for the PGI compiler suite. All tests run with `jpni=16`, `jpnj=16` and `jpnij=221`.

Compiler flags	Time for 60 steps (seconds)
-O0 -r8	325.994
-O1 -r8	203.611
-O2 -r8	154.394
-O3 -r8	152.971
-O3 -r8 -OPT:Ofast	162.148

Table 6: Runtime for 60 time steps for different compiler flags using the PathScale compiler suite. All tests were run with `jpni=16`, `jpnj=16` and `jpnij=221`.

aggressive optimisations either cause the code to slow down or to break entirely, e.g. `fast` or `fastsse` both cause the code to crash.

6.7 Summary of benchmarking study

What have we found out from running these simple benchmarks?

- PGI performs consistently better than PathScale with the latest versions of the compilers (PathScale 3.1, PGI 7.2.5) giving almost identical performance.
- Running in single core mode will give a reduction in the 60 step runtime but this is more than offset by the increased number of AU's required.
- Equal grid dimensions are best and should be used where possible. If equal dimensions can't be used then they should be chosen to be as square as possible and such that `jpni < jpnj`.
- NEMO continues to scale out to 1024 processors but the best performance in terms of runtime versus AU's used is obtained for 128 or 256 processors.
- Removal of land squares reduces the runtime for 60 time steps for most processor counts and greatly reduces the number of AU's required. This is not carried out by default in NEMO and thus many researchers could be using more AU's than necessary.

- Compiler flags above -O3 don't provide any benefit and in some cases break the code entirely - see section 9 for more details.

6.8 Optimal processor count

The results presented in Section 6 suggest that all future work on NEMO should be carried out using code compiled with the PGI compiler suite as it gives the lowest runtimes.

The NOCS researchers ideally want to be able to run an entire model year, i.e. 365 model days, in a 12 hour run on HECToR as this enables them to make optimal use of the machine/queues and also allows them to keep up with the post-processing and data transfer of the results as the run progresses. They can currently achieve 300 model days in a 12 hour run using 221 processors. In this section we investigate whether an optimal processor count which satisfies the desire to complete a model year in a 12 hour time slot can be found. To do this NEMO is executed over a range of processors and the number of model days which can be computed in 12 hours, *ndays*, is obtained from:-

$$ndays = 43200/t_{60} \quad (1)$$

where 43200 is the number of seconds in 12 hours and t_{60} is the time taken to complete a 60 step (i.e. 1 day) run of NEMO. This means we ideally need $t_{60} <= \frac{43200}{365} = 118.36$ seconds. The processor count investigated varies from 159 to 430. In all tests runs have been performed with the land cells removed. The results of this test are summarised in table 7. Figure 6 shows the results in graphical form with the 365 day threshold marked by the dashed line.

In performing this investigation some problems were discovered relating to the computation of land only cells performed by the nocspmap_r25 code. It was found that several processor configurations yielded incorrect numbers of land cells. These have been highlighted in table 7 where the value which was incorrectly computed is given in "()" after the correct number of land cells. If the wrong number of land cells are specified the code fails with an error of the form:-

```
===>>> : E R R O R
        ===========

 Eliminate land processors algorithm

  jpni =           21  jpnj =           22

  jpnij =          380 < jpni x jpnj


  ***********, mpp_init2 finds jpnij=          379
```

6.9 Time spent in file I/O and initialisation

The previous sections have reported performance based on the time taken to complete 60 time steps of the ocean modelling computation. This does not include initialisation

jpni	jpnj	No. of procs	Time for 60 steps (seconds)
13	14	159	177.583
14	14	174	163.633
14	15	187	172.191
15	15	196	157.858
15	16	209	153.450
16	16	221	145.078
16	17	232	137.507
17	17	244	127.705
17	18	260	135.688
18	18	274	127.103
18	19	286	122.639
19	19	304	125.880
19	20	321	118.081
20	20	335	117.830
20	21	349	107.464
21	21	364	113.491
21	22	379(380)	114.175
22	22	398(396)	107.051
22	23	413	123.939
23	23	430(429)	110.871

Table 7: Runtime for 60 time steps for various processor configurations ranging from 159 to 430.

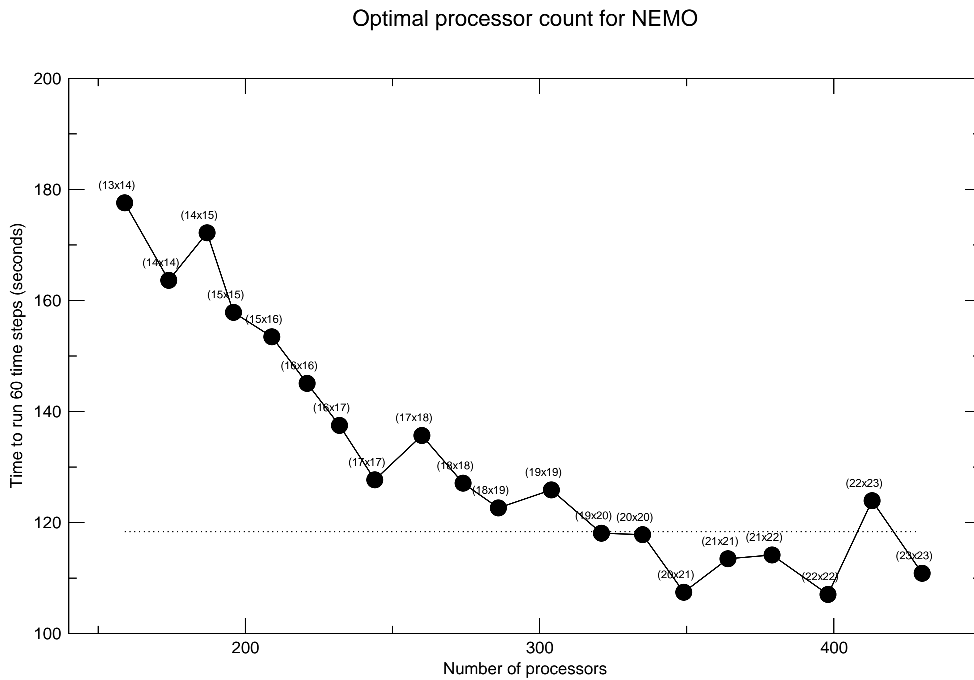


Figure 6: Investigation of optimal processor count for NEMO subject to completing a model year within a 12 hour compute run. The dashed line shows the cut-off point.

time or file I/O time which can be a significant fraction of the total runtime. Figure 7 shows the breakdown of the total runtime for various processor counts.

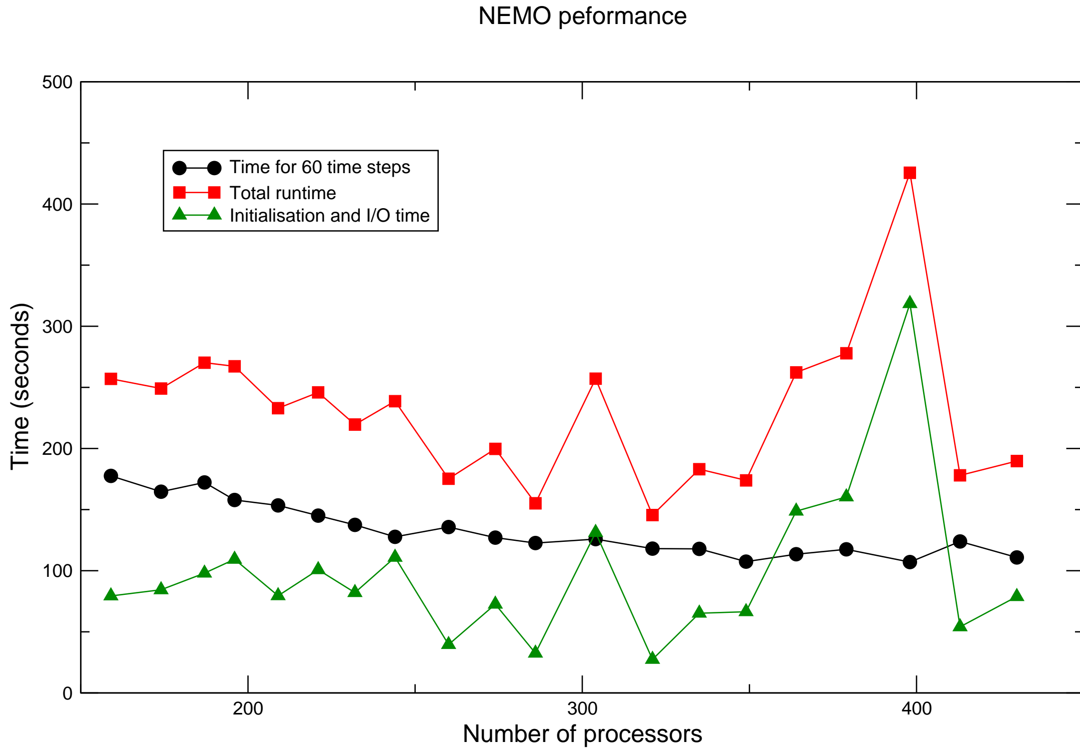


Figure 7: Performance of NEMO showing the breakdown of the total runtime for various processor counts. All tests carried out using PGI compiler.

Up to 250 processors the difference between the total runtime and time for 60 time steps remains approximately constant. Beyond 250 the results are somewhat more erratic with large variations, of up to 200%, occurring between 350-400 processors. As the number of files opened for output increases linearly with the number of processors these variations are perhaps to be expected. The time spent in initialisation and I/O reported by Figure 7 was found to be highly variable with multiple runs producing up to 50% variation. Conversely, the time for 60 model time steps was observed to be relatively stable with variations lying within the expected range for repeated runs (5-10%). The large variation in the initialisation and I/O time occurs because the I/O subsystem on HECToR is a resource shared between other users. Thus, the speed of I/O is governed by the load the system is under at the time when the job runs. If the I/O system is heavily loaded when NEMO attempts to read/write from/to file then the time spent in I/O operations will be increased. This makes predicting the runtime of a NEMO job problematic as the total runtime will be governed by system load whilst the job is

running.

7 Aside on I/O strategies for parallel codes

Parallel codes typically use one of the following I/O strategies:

- Master-slave - where a single (master) processor performs all the reading/writing and broadcasts/gathers data to/from the slave processors. The slave processors are usually idle whilst the reading/writing is taking place. If the time spent in computation is much greater than the time spent in I/O this approach may be acceptable. However, for codes involving significant amounts of I/O this approach could be highly detrimental to the performance. Due to it's ease of implementation, however, this is still the most common form of I/O used in parallel codes. Often codes were designed to run on a relatively small number of processors where such an approach was suitable. However, in recent years, as the number of processors has increased the master-slave I/O approach is becoming less than ideal.

- Multiple masters and groups of slaves or I/O subgroups - similar to the master-slave approach but here we have multiple master processors each gathering data from their own group of slave processors. This approach can reduce the overheads involved in having a single master process carrying out the I/O. It also reduces the memory requirements as the data to be input/output is now distributed between several master processors rather than a single processor. Some synchronisation of the I/O may be required to ensure the data are read/written in the correct order. However, it is anticipated that any synchronisation will be more than offset by the savings made from using multiple master processors.

- Parallel I/O - where each processor writes its own data to a separate file. The files then need to be collected together in the correct order at some later stage either via standard Unix commands (e.g. `cat`) or with a separate code. This approach should be more efficient than the master-slave approach as all the processors are kept busy with none idling. However, there may be limitations on the scalability of this approach. Most operating systems limit the number of files which can be open (for read/write) at the same time. This limit could be as few as 1000 files for some Unix implementations. Some applications may write to several different files and so this places a severe restriction on the number of processors which be used. E.g. if the file limit is 1000 and each processor writes to 10 files then we are limited to running on 100 processors or less. Clearly, this is not ideal. Many applications require many hundreds or thousands of processors and thus a different approach is required.

- MPI-IO Extensions to MPI and part of the MPI-2 standard [6]. Essentially it is a library providing functions which can be used to perform parallel I/O using the MPI libraries. A single file is written to by all processors which avoids the limitations of parallel I/O. Each processor writes directly to its own region of the file which avoids the need for any post-processing. As with parallel I/O all the processors are involved in the read/write operation so no-one remains idle. Not fully implemented by all vendors.

A vast number of I/O benchmarks exist and can be used to obtain performance estimates for the different I/O methods.

7.1 NEMO file I/O

In its current configuration NEMO uses a parallel I/O type approach to read in much of its input data or restart data. Some small configuration files are read in using a master-slave method, e.g. the `namelist` file.

The output is performed by each processor where each processor dumps out it's own section of the ocean model, i.e. the output is performed using parallel I/O. The input/output binary data are typically in netCDF (`*.nc`) format which means that any changes to the I/O strategy must take this into account. NEMO currently uses netCDF version 3.6.2 but it is intended that future versions will use netCDF 4.0 which is anticipated to give improved performance.

The use of netCDF gives portable output files that can be used on different architectures. The size of the NEMO output and the post-processing means that converting to an MPI-IO strategy is simply not feasible and thus we need to do the best we can with the existing parallel I/O implementation.

8 NetCDF performance and installation

NEMO uses netCDF files for both its input and output data. NetCDF stands for network Common Data Form and is a set of interfaces, data formats and software libraries which help read and write scientific data files. Further information on netCDF can be found in [3, 4]. The netCDF libraries allow scientific data to be represented in a machine independent and thus portable format.

NEMO currently uses version 3.6.2 of netCDF. The new release of netCDF (netCDF 4.0) allows HDF5 files to be accessed and also includes parallel I/O capabilities. The first stable release version of netCDF 4.0 became available on 29/06/2008 and as of 23/03/2009 the latest version is 4.0.1-beta3. Below we describe the installation of the stable release which contains all the necessary functionality required for NEMO. Some additional installation and porting details can be found at:
`http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-install/`

Prior to installing netCDF 4.0 both zlib version 1.2.3 or higher and HDF5 version 1.8.1 must be installed as these are prerequisites of netCDF 4.0. The installation of zlib, HDF5 and netCDF 4.0 will be described in sections 8.2, 8.3 and 8.4 respecitvely. The next section gives some performance details relating to netCDF 3.6.2.

8.1 NetCDF 3.6.2 performance on HECToR

The serial performance of netCDF version 3.6.2 is investigated by means of a simple benchmark which reads and writes a netCDF file of varying size (Mbytes). Versions of netCDF compiled with the PGI and PathScale compilers are tested to determine whether the choice of compiler has any influence on the performance. The library is also compiled with various optimisation levels to determine where compiler optimisations can improve the performance.

One of the netCDF tester codes (in `nc_test/large_files.c`) writes and reads a large netCDF file. The size of the file can be altered by varying the value of `I_LEN`. Timers (`MPI_Wtime`) have been inserted into this code to enable the write/read times to be computed. Table 8 gives the write/read time in seconds for various compilers and compiler flags for a file of size 4 Gbytes. The timings are taken from the best (fastest) of three runs.

Compiler	Compiler flags	Write time	Read time
PGI	`ftn` i.e. `-O1`	32.951	28.549
Pathscale	`ftn` i.e. `-O2`	17.652	14.269
PGI	`ftn -O3`	12.823	12.351

Table 8: Comparison of write/read performance of netCDF for various compilers and compiler flags.

Table 8 shows that the performance of netCDF 3.6.2 compiled with the default compiler options is significantly poorer than that compiled with `-O3`. The PathScale compiler gives the best performance for this example.

The variation in performance for varying file sizes has also been investigated for both the PGI and PathScale compiler suites. Figure 8 gives the results of this experiment. From figure 8 it is clear the write/read time varies approximately linearly with file size and that netCDF 3.6.2 compiled with the PathScale compiler is consistently faster than that compiled with the PGI compiler.

The results given table 8 suggest that using an optimised version of netCDF 3.6.2 may be beneficial to NEMO. To test this, NEMO was recompiled using a version of netCDF 3.6.2 compiled with `-O3`. Note, to ensure object file compatibility, NEMO must be compiled with the same compiler suite that is used to compile netCDF. The runtime was found to be almost identical to that obtained with the unoptimised version of netCDF 3.6.2 which is unsurprising. The test codes are serial, whereas NEMO is a parallel code. Furthermore, NEMO writes out many files simultaneously rather than one single large file and also does computation whereas the test code is purely carrying out I/O operations. Even although the netCDF optimisation level appears to have little effect on the peformance of NEMO we will still compile netCDF 4.0 using `-O2` as it may be beneficial to other users of the library.

8.2 Installing zlib 1.2.3

Zlib is freely available compression library which can be utilised by HDF5 1.8.1 or later. Further information on zlib can be found at [7]. The following options were used to compile zlib 1.2.3 on HECToR. The optimisation level was set to `-O2` as this should give a good compromise between performance and code stability.

```
make distclean
export FC='ftn -O2'
export F90='ftn -O2'
export F95='ftn -O2'
export CC='cc -O2'
```

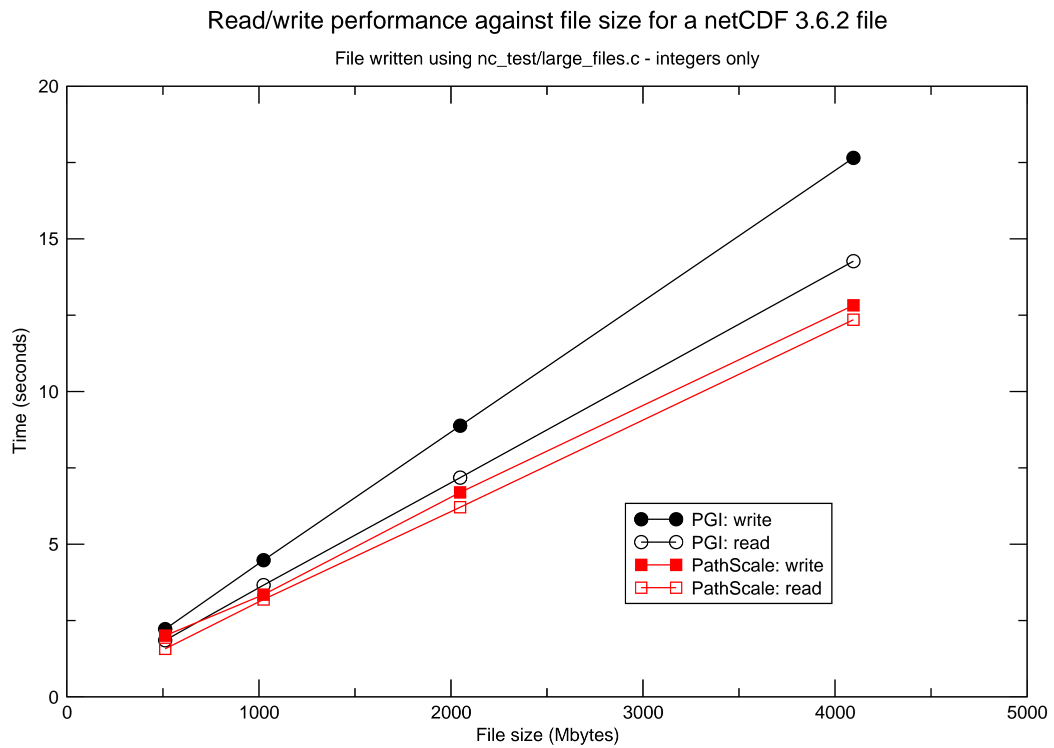


Figure 8: Variation of write/read time with filesize for netCDF 3.6.2 compiled with the PGI and PathScale compiler suites.

```
export CXX='CC -O2'
export LOGFILE=build_zlib_pgi_opt.txt

./configure --prefix=/home/n01/n01/fionanem/local/optimised &> $LOGFILE
make >> $LOGFILE
make check >> $LOGFILE
make install >> $LOGFILE
cp -rp ~/local/optimised /work/n01/n01/fionanem/local/.
```

At the end of the installation the library files are copied to the the `/work` file system as part of the HDF5 and netCDF installations must be carried out via the batch system which can only access the work file system. The output from the `make check` is also examined to ensure that all the testers complete successfully.

8.3 Installing HDF5 1.8.1

HDF5 is a set of tools and libraries that allows extremely large and complicated data collections to be managed. The file format used by HDF5 is designed to be portable. Further information on HDF5 can be found at [8].

HDF5 can be installed both with and without parallel I/O (i.e. MPI-IO) capabilities. The following options were used to compile a serial (i.e. without parallel I/O) version of HDF5 1.8.1 on HECToR.

```
export FC='ftn -O2'
export F90='ftn -O2'
export F95='ftn -O2'
export CC='cc -O2'
export CXX='CC -O2'
export RUNSERIAL="aprun -q"
export LOGFILE=build_hdf5-1.8.1_noparallel.txt

./configure --prefix=/work/n01/n01/fionanem/local/noparallel \
--disable-shared --enable-static-exec  --enable-fortran \
--disable-stream-vfd  --disable-parallel \
--with-zlib=/work/n01/n01/fionanem/local
--with-szlib=/work/n01/n01/fionanem/local  &> $LOGFILE
```

Some of the executables need to be executed on the backend (due to issues with the `getpwuid` function causing a segmentation violation when executed on the login nodes). Therefore, the `make`, `make check` and `make install` commands are all run via the batch system running on a single processor. The `RUNSERIAL` environment variable is used to tell the build that `aprun` must be used to launch executables. Initial attempts to build the library failed with a number of errors relating to libtool. The error message is of the form:

```
../libtool: line 1531: 0: Bad file descriptor
libtool: link: `H5.lo' is not a valid libtool object
make[2]: *** [libhdf5.la] Error 1
```

```
make[1]: *** [all] Error 2
make: *** [all-recursive] Error 1
```

The addition of `#!/bin/bash` to the batch script seems to resolve this problem. The following commands are executed via a batchscript:

```
# Ensure we don't run out of space for compile
export TMPDIR=/work/n01/n01/fionanem/tmp
export LOGFILE=build_hdf5-1.8.1_noparallel.txt
export CHECKFILE=check_hdf5-1.8.1_noparallel.txt
make >> $LOGFILE
make check > $CHECKFILE
make install >> $LOGFILE
```

The `TMPDIR` variable is set to ensure that we don't run out of tmp space during the build. After the build is complete the output from `make check` is examined to ensure that all the testers pass. One of the testers fails - this is a known issue on the Cray X1 and assumed to also be an issue on the Cray XT4. The error message reported by the `make check` is as follows:

```
Testing h5dump --xml -X : tempty.h5             *FAILED*
```

The failed tester is invoked by the `testh5dumpxml.sh` script in tools/h5dump. This error is reported in the release notes supplied with the snapshot release and version 1.8.1, see [9] for further details. Essentially the error occurs because a single colon is misinterpreted by the operating system. If the command is run via the command line, e.g. `./h5dump --xml -X : tempty.h` then the tester runs successfully.

To compile a parallel version of HFD5 1.8.1 the following options were used.

```
make distclean
export FC='ftn -O2'
export F90='ftn -O2'
export CC='cc -O2'
export CXX='CC -O2'
export RUNSERIAL="aprun -q"
export RUNPARALLEL="aprun -n 4"
export LOGFILE=build_hdf5-1.8.1_parallel.txt

./configure --prefix=/work/n01/n01/fionanem/local/parallel \
--disable-shared --enable-static-exec  --enable-fortran \
--disable-stream-vfd  --enable-parallel \
--with-zlib=/work/n01/n01/fionanem/local \
--with-szlib=/work/n01/n01/fionanem/local &> $LOGFILE
```

As with the serial build the `make`, `make check` and `make install` are performed on the backend. The `RUNPARALLEL` environment variable ensures that the parallel runs are performed on 4 processors. The following commands are executed via a batchscript:

```
# Ensure we don't run out of space for compile
export TMPDIR=/work/n01/n01/fionanem/tmp
export LOGFILE=build_hdf5-1.8.1_parallel.txt
export CHECKFILE=check_hdf5-1.8.1_parallel.txt
make >> $LOGFILE
make check > $CHECKFILE
make install >> $LOGFILE
```

All the test codes pass with the exception of `--xml -X : tempty.h5` as described above.

8.4 Installing netCDF 4.0

Once zlib 1.2.3 and HDF5 1.8.1 have been successfully installed the installation of netCDF 4.0 can begin. As with HDF5 1.8.1, both serial and parallel versions of netCDF 4.0 are required. The serial version of the library is built using the serial version of HDF5 and the parallel version built using the parallel version of HDF5.

The serial build of netCDF 4.0 is relatively straightforward. The following commands allow a serial version of netCDF 4.0 to be compiled:

```
make distclean
export FC='ftn -O2'
export F90='ftn -O2'
export F95='ftn -O2'
export CC='cc -O2'
export CXX='CC -O2'
export NM=nm
export CPPFLAGS=-DpgiFortran
export LOGFILE=build_netcdf4.0_noparallel.txt
export CHECKFILE=check_netcdf4.0_noparallel.txt

./configure --enable-netcdf-4 \
--with-hdf5=/work/n01/n01/fionanem/local/noparallel \
--with-zlib=/work/n01/n01/fionanem/local \
--with-szlib=/work/n01/n01/fionanem/local --disable-cxx \
--disable-parallel-tests \
--prefix=/work/n01/n01/fionanem/local/noparallel &> $LOGFILE
```

The `CPPFLAGS` variable is a macro which is required by the PGI compiler suite - otherwise the build fails. The `--disable-cxx` prevents the C++ API from being built. The build fails when attempting to link the shared `libgcc_s` otherwise. The `--disable-parallel-tests` ensures that the parallel components of the library and testers do not get built.

The `make`, `make check` and `make install` can all be executed on the login nodes as no parallel elements are involved. The tester codes all pass without error.

Building the parallel version of netCDF 4.0 proved to be more problematic due to various cross-compilation issues. At several stages of the build process, executables are generated and then run. Unlike HDF5, netCDF 4.0 does not contain any environment

settings for cross-compilation (e.g. the `RUNSERIAL` and `RUNPARALLEL` mentioned above). Any parallel executables are either invoked via `mpiexec` which is not valid for HECToR or on the command line via `./exename`. This means that any steps of the build process which run parallel executables must be extracted and run separately via the batch system.

The first such problem arises during the `make`, where `ncgen` is executed in order to generate the `ctest.c` and `ctest64.c` files. As `ncgen` is a parallel executable it cannot run on the login nodes. The error message reported is:

```
[unset]: _pmi_init: _pmi_preinit encountered an internal error
Assertion failed in file /tmp/ulib/mpt/nightly/3.0/042108/xt/trunk/mpich2/..
.. src/mpid/cray/src/adi/mpid_init.c at line 119: 0
aborting job:
```

The solution is to execute the two runs of `ncgen` on the backend via a batchscript and then to continue the `make` on the login node once the batch job has completed.

A similar problem occurs during the `make check` where 18 testers fail for the same reasons. The error messages are of the form:

```
[0] assertion: st == sizeof ident at file mptalps.c line 93, pid 25085
FAIL: tst_dims
[0] assertion: st == sizeof ident at file mptalps.c line 93, pid 25090
FAIL: tst_files
...
Testing parallel I/O with HDF5...
SUCCESS!!!
PASS: run_par_tests.sh
==========================================
18 of 36 tests failed
Please report to support@unidata.ucar.edu
==========================================
```

Again, the error occurs because the tester codes are parallel (i.e. contain MPI calls) and cannot run on the login nodes of HECToR. As before, the solution is to run these eighteen testers on the backend via a batchscript.

The flags used to compile the parallel version of netCDF are summarised below:

```
make distclean # Ensure we start with a clean install
export FC='ftn -O2'
export F90='ftn -O2'
export F95='ftn -O2'
export CC='cc -O2'
export CXX='CC -O2'
export NM=nm
export CPPFLAGS=-DpgiFortran
export LOGFILE=build_netcdf4.0_parallel.txt
export CHECKFILE=check_netcdf4.0_parallel.txt
```

```
./configure --enable-netcdf-4 \
--with-hdf5=/work/n01/n01/fionanem/local/parallel \
--with-zlib=/work/n01/n01/fionanem/local \
--with-szlib=/work/n01/n01/fionanem/local --disable-cxx \
--enable-parallel-tests \
--prefix=/work/n01/n01/fionanem/local/parallel &> $LOGFILE
```

The `CPPFLAGS` and `--disable-cxx` are as described for the serial installation. The `--enable-netcdf-4` ensures that the netCDF 4.0 features are enabled. The `--enable-parallel-tests` ensures that the parallel tests are executed.

After configuration completes the procedure for compiling and testing the parallel version of netCDF 4.0 is as follows:

- Run `make` on the login node - it fails when attempts to execute `ncgen`
- Submit a batchscript which runs the two instances on `ncgen`, e.g.

  ```
  aprun -n $NPROC ../ncgen/ncgen -c -o ctest0.nc ./../ncgen/c0.cdl > ./ctest.c
  aprun -n $NPROC ../ncgen/ncgen -v2 -c -o ctest0_64.nc ./../ncgen/c0.cdl > ./ctest64.c
  ```

- Once batchscript completes, re-start the `make` which should now complete successfully
- Run `make check` on the login node - 18 testers will fail
- Submit a batchscript which runs the 18 parallel testers and wait for this to complete.
- Once the testers have executed successfully run the `make install` on the login node.

The serial and parallel tester codes are all found to run successfully confirming that our installation of both the serial and parallel versions of netCDF 4.0 has been successful.

8.5 NOCSCOMBINE performance on HECToR for different versions of netCDF

In this section we compare the performance of the `nocscombine` tool when compiled with different versions of netCDF. Various versions of netCDF 4.0 were compiled throughout the project (e.g. beta releases prior to the final stable release version) and results are included for a variety of these along with the final release version results. The results are summarised in table 9. For each test the following command was executed:

```
nocscombine -f O25-TST_CU30_19580101_19580101_grid_T_0000.nc -d \
   votemper -o outputfile.nc
```

Each run was carried out in batch with the timings reported in table 9 being the best of three runs. The runs were performed consecutively ensuring that the same processing core was used for each. Despite this, considerable variation in runtimes was observed,

NetCDF version	`nocscombine` time (seconds)	File size (Mbytes)
3.6.2	343.563	731
4.0-unopt	86.078	221
4.0-opt	85.188	221
4.0-opt*	76.422	221
4.0-beta2	84.758	221
4.0-beta2*	77.055	221
4.0-release	85.188	221
4.0-release*	78.188	221
4.0-Cray	92.203	221
4.0-release-classic	323.539	731

Table 9: Comparison of `nocscombine` performance for various versions of netCDF. The * indicates that the system version of zlib was used.

as much as 100% in some cases. As I/O is a shared resource on the system we have no control over other user activities so these variations are perhaps not surprising.

In table 9, 3.6.2 denotes the release version of netCDF 3.6.2 and uses the version available via the package account on HECToR, e.g. the version accessed via the `module load netcdf` command. Version 4.0-unopt denotes the Snapshot release dated 29th April compiled with default optimisation (i.e. `-O1`). 4.0-opt is the same Snapshot release compiled with optimisation set to `-O2`. 4.0-beta2 denotes the final beta2 version compiled with `-O2`. 4.0-release denotes the final release version compiled with `-O2`. 4.0-Cray denotes the version supplied by Cray which became available on HECToR during March 2009. Version 4.0-release-classic is netCDF 4.0 run in classic (i.e. netCDF 3.6.2 style) mode. The * denotes versions which have been compiled using the system version of zlib (version 1.2.1) rather than version 1.2.3.

Examining the results in table 9 we see that netCDF 4.0 clearly outperforms netCDF 3.6.2 both in terms of runtime performance and in terms of the amount of disk space required. The size of the file output by netCDF 4.0 is 731/221 = 3.31 times smaller than that output by netCDF 3.6.2. The runtime difference between the versions (c.f. version 3.6.2 with 4.0-release) is 343.563/85.188 = 4.03. This tells us that the runtime savings do not just result from the reduced file size. It's possible that there are some algorithmic differences between the versions or perhaps the dataset now fits into cache better thus reducing memory latency. The compression and chunking used by netCDF 4.0 may also be improving the performance. Interestingly, the Cray version of netCDF 4.0 is slower (92.203 seconds version 85.188 or 78.188 for the different zlib versions) than any of the versions compiled as part of the dCSE project. Whilst the difference is 17.92% or 8.23% depending on which version of zlib was used this is still significant enough to warrant compiling a local version if your code spends sufficient time in netCDF routines.

The level of optimisation used to compile the netCDF library appears to have minimal effect. The system version of zlib (version 1.2.1), outperforms version 1.2.3. However, as netCDF 4.0 clearly states that version 1.2.3 or later is required it is potentially risky to use the older version as functionality required by netCDF 4.0 maybe missing.

In order to compare directly the performance of netCDF 3.6.2. and 4.0 we also tested netCDF 4.0 in classic mode. To output classic format using netCDF 4.0 the following changes must be made to the `make_global_file4.F90` code:

- Change `nf90_create` call from: `status = nf90_create( trim(ofile), NF90_HDF5, ncid )` to `status = nf90_create( trim(ofile),NF90_CLOBBER,ncid)`. Here the `NF90_CLOBBER` ensures that netCDF classic format is used.
- Change the `nf90_set_fill` call from: `status = nf90_set_fill(ncid, NF90_FILL, oldfill)` to `status = nf90_set_fill(ncid, NF90_NOFILL, oldfill)`.
- Comment out the `nf90_put_att` command which sets the fill value for the netCDF 4.0 file. Strictly, as no filling occurs it's not actually necessary to do this but avoids uneccessary computations.

Comparing the results we see that netCDF 4.0-release in classic mode is approximately 5.8% faster than netCDF 3.6.2. Therefore it's possible that some improvements to the algorithms have been made between versions.

In summary, based on the results obtained from the NOCSCOMBINE code using netCDF 4.0 instead of netCDF 3.6.2 will likely give significant performance improvements for NEMO. The amount of disk space used could be reduced by a factor of 3 and the time taken to write this information to disk could be reduced by a factor of 4. The time taken to compress and uncompress the data at the post-processing stages still needs to be quantified but the early results are promising. Section 9.4 discusses the implementation of netCDF 4.0 in NEMO.

9 NEMO V3.0

Version 3.0 of the NEMO code became available in the autumn of 2008 and after addition of the NOCS specific features this version was used for the remainder of the project. Many of the same tests carried out in section 6 were repeated with NEMO V3.0.

9.1 Compilation

Compilation of NEMO V3.0 on HECToR was relatively straight-forward. When compiling for both the XT4 and X2 some issues with the `make clean` not removing various files and libraries were discovered. The file `libioipsl.a` does not get removed and a number of `*.mod` and `*.o` files are not removed. This can create problems when swapping between compilers or between XT4/X2 builds as incompatible `.mod` or `.o` files persist which causes the build to fail. The problem can be solved by the addition of: `$(IOIPSL_LIB) $(LIBDIR)/*.mod $(LIBDIR)/*.o *.mod` to the `clean:` macro in the top level NEMO Makefile in `/WORK`.

A diff of the original and modified Makefile gives:

```
<               $(RM) model.o $(MODDIR)/oce/*.mod $(MODEL_LIB)
$(SXMODEL_LIB) $(EXEC_BIN)
---
>               $(RM) model.o $(MODDIR)/oce/*.mod $(MODEL_LIB)
> $(SXMODEL_LIB) $(EXEC_BIN) $(IOIPSL_LIB) $(LIBDIR)/*.mod $(LIBDIR)/*.o
```

```
> *.mod
```

This modification ensures that all the `.a`, `.o` and `.mod` files are removed with `make clean`. These suggested modifications were passed back to the NOCS researchers.

9.2 Performance for different compiler flags

A number of different compiler flags have been tested for NEMO V3.0. The results are summarised in Table 10. These tests were all carried out using a 16 by 16 processor grid with the land cells removed, that is `jpni=16`, `jpnj=16` and `jpnij=221`. Both cores were used for all tests, i.e. `mppnppn=2` was specified in the batch script.

Compiler flags	**Time for 60 steps (seconds)**
`-O0`	163.520
`-O1`	157.123
`-O2`	138.382
`-O3`	139.466
`-O4`	137.642
`-fast -O3`	fails with segmentation violation
`-fast -O3` on PGI 7.2.3	fails with segmentation violation
`-O2 -Munroll=c:1`	runs 139.568
`-O2 -Munroll=c:1 -Mnoframe`	runs 138.862
`-O2 -Munroll=c:1 -Mnoframe -Mlre`	fails step 1
`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline`	fails step 1
`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse`	seg fault
`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse`	seg fault
`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align`	seg fault
`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align -Mflushz`	seg fault
`-O2 -Munroll=c:1 -Mnoframe -Mautoinline -Mscalarsse -Mcache_align -Mflushz`	runs??

Table 10: Runtime for 60 time steps for different compiler flags for the PGI compiler suite. Version 7.1.4 used unless stated otherwise. All tests were run with `jpni=16`, `jpnj=16` and `jpnij=221`.

Increasing the level of optimisation from `-O0` to `-O2` gives an increase in performance. Optimisation of `-O2` up to `-O4` gives minimal improvement. The `-fast` flag results in a segmentation violation. As this flag invokes a number of different optimisations we tested each of these in turn to ascertain which particular flags cause the problem. The command `pgf90 -help -fast` lists the optimisations invoked by `-fast`, e.g.

```
fionanem@nid15879:~> pgf90 -help -fast
Reading rcfile /opt/pgi/7.1.4/linux86-64/7.1-4/bin/.pgf90rc
-fast     Common optimizations;
          includes -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline
          == -Mvect=sse -Mscalarsse -Mcache_align -Mflushz
```

The `-Munroll=c:1` flag enables loop unrolling which `c:1` ensuring that all loops with a length of 1 or more are completely unrolled. The `-Mnoframe` flag prevents the compiler from generating code which fits in a stack frame. The `-Mlre` flag allows loop carried redundancy elimination to occur - i.e. variables redundant within a loop are removed. The `-Mautoinline` flag automatically enables function inlining in C/C++ and thus does not apply to NEMO. The `-Mvect=sse` flag allows vector pipelining to be used with SSE instructions. The `-Mscalarsse` flag generates scalar SSE code with xmm registers - this flag also implies `-Mflushz`. The `-Mcache_align` flag ensures that objects are aligned along cache boundaries. The `-Mflushz` flag sets the SSE instructions to "flush-to-zero" which ensures that numbers approaching zero get automatically zeroed.

From Table 10 we see that the addition of the flags `-Mlre` and `-Mvect=sse` cause the code to crash at runtime. All other flags invoked by `-fast` do appear to not cause significant issues. The `-Mlre` causes the zonal velocity to become very large suggesting that the loop redundancy elimination may have removed a loop temporary that was actually required. The reason for the failure when `-Mvect=sse` is added is unknown. Ultimately the addition of the additional flags doesn't give significant performance improvements over `-O2` or `-O3` and thus `-O3` will be used in future.

9.3 Performance of NEMO V3.0

The scaling on NEMO V3.0 is similar to that of version 2.3. For 398 and 794 processor runs the total runtime was found to be highly unstable, varying as much as 400%. Due to these variations, the results shown by figure 9 have been taken from the best of 5 runs.

The variability of the total runtime appears to be highly dependent on the system load. This makes predicting the length of a run or indeed attempting to specify an appropriate wallclock time difficult. Anything that can be done to improve this situation will be hugely beneficial to the researchers.

9.4 Converting NEMO to use netCDF 4.0

This section describes the procedure for adapting NEMO V3.0 to use netCDF 4.0. To compile NEMO V3.0 with netCDF 4.0 the main NEMO Makefile must be altered such that the `NCDF_INC` and `NCDF_LIB` variables point to the location of the netCDF 4.0 include files and libraries. The paths to the HDF5, zlib or szip include files and libraries must also be added. E.g. including the following in the three main NEMO Makefiles ensures that the netCDF 4.0 library is used:

```
H5HOME = /work/n01/n01/fionanem/local/noparallel
ZLIBHOME = /work/n01/n01/fionanem/local
NCDF_INC = /work/n01/n01/fionanem/local/noparallel/include \
  -I$(H5HOME)/include -I$(ZLIBHOME)/include
```

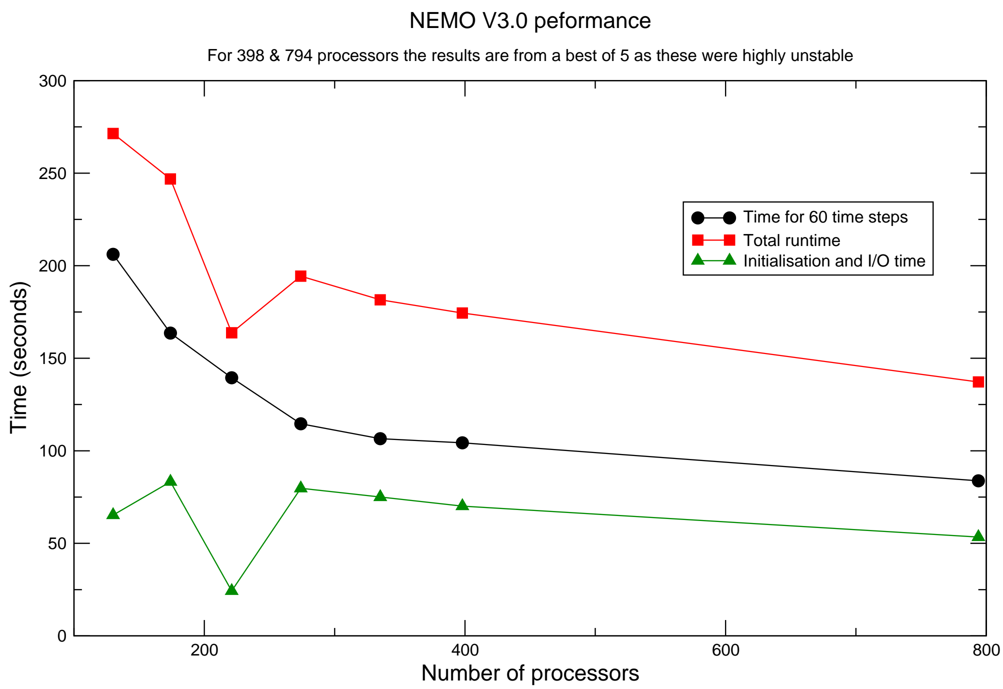


Figure 9: Performance of NEMO V3.0 showing the breakdown of the total runtime for various processor counts. All tests carried out using PGI compiler

```
NCDF_LIB = -L/work/n01/n01/fionanem/local/noparallel/lib \
  -lnetcdf -L$(H5HOME)/lib -lhdf5 -lhdf5_hl -L$(ZLIBHOME)/lib \
  -lz -lsz -L$(H5HOME)/lib -lhdf5
```

After re-compilation the code is executed and the output verified as being correct using the method described in section 5.3.

The procedure above uses netCDF 4.0 in classic mode and is the first step in converting NEMO to use netCDF 4.0. However, what we actually want to do is to convert NEMO to use netCDF 4.0 with compression and chunking enabled. This is best tackled as a two stage process. The first step is to generate netCDF 4.0 format output files without any compression/chunking. The second step is to add chunking and/or compression and thus take advantage of the full functionality of netCDF 4.0.

For step one we need to tell netCDF that we want to take advantage of the new features. This can be achieved by making some minor modifications to the NEMO code. In particular, all subroutine calls to `NF90_CREATE` need to be modified such that each instance of `NF90_NOCLOBBER` is replaced with `NF90_HDF5`. The two source files which require alteration are:-

- `IOISPL/src/histcom.f90`
- `IOISPL/src/restcom.f90`

In addition, the file `IOIPSL/src/fliocom.f90` also requires the variable `m_c` to be set to `NF90_HDF5`. At present the mode of this variable has been set directly after the conditional block from lines 859-884. E.g. we replace

```
i_rc = NF90_CREATE(f_nw,m_c,f_e)
```

with

```
m_c = NF90_HDF5
i_rc = NF90_CREATE(f_nw,m_c,f_e)
```

With these modifications the code is then recompiled and tested. The main output files should now be in netCDF 4.0 format which can be verified by attempting to read one of the output files with versions of `ncdump` [1] which have been compiled with netCDF 3.6.2 and with netCDF 4.0, e.g.

1. Using `ncdump` from version 3.6.2 of netCDF (e.g. `module load netcdf/3.6.2`) attempt to read one of the NEMO output files,
   e.g. `ncdump O25-V3_5d_19580101_19580101_grid_U_0001.nc` if this is successful (i.e. the contents of the file are displayed) then the file is a classic netCDF 3.X format file. If it is not an netCDF 3.X file then you'll get an error message of the form: `ncdump: O25-V3_5d_19580101_19580101_grid_U_0001.nc: NetCDF: Unknown file format`
2. Now try `ncdump` from netCDF 4.0. The version installed under `/work/n01/n01/fionanem/local/noparallel/bin/ncdump` can be used or that

[1] `ncdump` is part of the netCDF library and is a executable which can be used to convert a netCDF file into a text file.

added to your path when the new centrally installed netCDF 4.0 is loaded (use `module load netcdf` to achieve this). If the file is in netCDF 4.0 format then the contents of the file will be displayed. Please note: netCDF 4.0 can also read netCDF 3.X files so just testing with one version of `ncdump` is not sufficient.

The modifications described above allow netCDF 4.0 format files to be output from NEMO. However, as yet no compression or chunking has been included. To use the chunking/compression features of netCDF 4.0 additional modifications must be made to the code. These modifications are outlined below:

1. Declare new variables relating to the chunking and compression, e.g.

   ```
   INTEGER, dimension(4) :: chunksizes
   INTEGER :: chunkalg, shuffle, deflate, deflate_level
   ```

   `chunksizes` is an array containing the chunksize to be used for each dimension of the dataset.

2. Initialise the chunking and compression variables, e.g.

   ```
   chunksizes(1) = 10
   chunksizes(2) = 10
   chunksizes(3) = 10
   chunksizes(4) = 1
   deflate_level = 1 ! Turn compression on
   deflate = 1       ! Level of compression
   shuffle = 1       ! Allow shuffling
   chunkalg = 0      ! Turn chunking on
   ```

   Here chunksize is chosen such that it is less than the number of data points within that dimension. It should be noted that for three-dimensional fields such as the ice data (i.e. `*icemod*.nc` files) this may not be ideal.

3. Following each call to `nf90_def_var`, add new calls to `nf90_def_var_chunking` and `nf90_def_var_deflate` to ensure that chunking and compression is applied to each output variable. An example of the original code would be:

   ```
   iret = NF90_DEF_VAR (ncid,lon_name,NF90_FLOAT,dims(1:ndim),nlonid)
   ```

   with the modified code example including the following two lines immediately after the call to `NF90_DEF_VAR`, e.g.

   ```
   iret = NF90_DEF_VAR_CHUNKING(ncid,nlonid,chunkalg,chunksizes)
   iret = NF90_DEF_VAR_DEFLATE(ncid,nlonid,shuffle,deflate, deflate_level)
   ```

Modifications were made to `../../IOIPSL/restcom.f90`, `../../IOIPSL/histcom.f90` and `../../IOIPSL/fliocom.f90`. However, after testing the code it became apparent than only the changes to the `../../IOIPSL/histcom.f90` file are actually required for

the test model. Furthermore, the changes were not added for the NF90_DEF_VAR call at line 1022 (unmodified code) as this was found to truncate the *icemod*.nc file.

The code was then re-tested and the size of the output files compared with those created without and chunking/compression. For the original code some sample output file sizes were as follows:

```
ls -l *0100.nc
-rw-r--r--  1 fionanem n01 47364120 Apr 17 09:40 O25-V3_00000060_restart_0100.nc
-rw-r--r--  1 fionanem n01  2286160 Apr 17 09:40 O25-V3_00000060_restart_ice_0100.nc
-rw-r--r--  1 fionanem n01  7102636 Apr 17 09:40 O25-V3_CU30_19580101_19580101_grid_T_0100.nc
-rw-r--r--  1 fionanem n01  3246008 Apr 17 09:40 O25-V3_CU30_19580101_19580101_grid_U_0100.nc
-rw-r--r--  1 fionanem n01  3246013 Apr 17 09:40 O25-V3_CU30_19580101_19580101_grid_V_0100.nc
-rw-r--r--  1 fionanem n01 15709982 Apr 17 09:40 O25-V3_CU30_19580101_19580101_grid_W_0100.nc
-rw-r--r--  1 fionanem n01  1118067 Apr 17 09:40 O25-V3_CU30_19580101_19580101_icemod_0100.nc
```

For the modified version the corresponding files sizes were:

```
ls -l *0100.nc
-rw-r--r--  1 fionanem n01 47364120 Apr 21 13:01 O25-V3_00000060_restart_0100.nc
-rw-r--r--  1 fionanem n01  2286160 Apr 21 13:01 O25-V3_00000060_restart_ice_0100.nc
-rw-r--r--  1 fionanem n01  1765659 Apr 21 13:01 O25-V3_CU30_19580101_19580101_grid_T_0100.nc
-rw-r--r--  1 fionanem n01   932602 Apr 21 13:01 O25-V3_CU30_19580101_19580101_grid_U_0100.nc
-rw-r--r--  1 fionanem n01   939433 Apr 21 13:01 O25-V3_CU30_19580101_19580101_grid_V_0100.nc
-rw-r--r--  1 fionanem n01  2569717 Apr 21 13:01 O25-V3_CU30_19580101_19580101_grid_W_0100.nc
-rw-r--r--  1 fionanem n01   536500 Apr 21 13:01 O25-V3_CU30_19580101_19580101_icemod_0100.nc
```

Comparing the output file sizes we see that the four *grid*.nc and *icemod*.nc files have reduced in size by up to 3.55 times relative to the original data. The two restart files are not affected as no compression/chunking has been applied to these. The overall effect of the chunking/compression for the test model is summarised by table 11.

File name	Disk usage for netCDF 3.X (MBytes)	Disk usage for netCDF 4.0 (Mbytes)	Reduction factor
grid_T.nc	1500	586	2.56
grid_U.nc	677	335	2.02
grid_V.nc	677	338	2.00
grid_W.nc	3300	929	3.55
icemod.nc	208	145	1.43

Table 11: Effect of chunking and compression on the size of the NEMO 3.0 output files. For each file name the usage is the sum of all 221 individual processor output files.

The results presented in table 11 demonstrate that a significant reduction in disk usage (∼4 Gbytes for this example) can be achieved by using the chunking and or compression features of netCDF 4.0 within NEMO. For the test model, no significant improvement in the runtime is observed however this is to be expected as the restart files dominate in terms of their size. In our test model we run for 60 time steps, outputting data every 30 time steps with a restart model written output after 60 time steps. However, for a production run of NEMO the model would typically run for in excess of 10,000 time steps with output every 300 steps and a restart file written every 1800 time steps. Therefore an improvement in runtime would be expected due to the reduction in output file size. The actual size of any improvement will depend on the output frequency chosen, which in turn depends on the parameters being modelled and the particular science studied.

Conversion of the restart files to netCDF 4.0 and a full investigation of the optimal chunksize/compression parameters were not possible due to a shift in focus requested by the researchers where we were asked to concentrate on the nested AGRIF models.

10 NEMO AGRIF nested model running/debugging

In this section we describe the use of nested models in NEMO, investigate their performance and discuss the issues associated with getting them to run successfully on HECToR. An introduction to nested models is given in section 10.1 with the two different models (referred to as BASIC and MERGED) discussed in sections 10.2 and 10.3 respectively.

10.1 Introduction to the nested model problem

WP2 of the dCSE project involves investigating the performance and ease of implementation of nested models within NEMO. The first step is therefore to compile and run a nested version of NEMO where a finer model is run inside the main (coarser) ocean model - e.g. a 1° region nested inside a 2° model. Any number of nested regions can be used and within each region multiple levels of nesting are also possible. Figure 10 illustrates this with an example showing the main model, A, containing two nested regions B, and C with region C having one level of nesting and region B having two levels of nesting.

Nested models in NEMO use the Adaptive Grid Refinement in Fortran (AGRIF) pre-processor code called, `conv`, to generate the code for the nested regions. By default, the NEMO code uses arrays of dimension `(jpi, jpj)` where `jpi` and `jpj` are set as parameters within the code. The pre-processor completely re-structures the code by inserting interface routines which pass array information from a special dynamically allocated AGRIF data type the size of which is determined based on the desired size of the nested region. Essentially the AGRIF pre-processor allows the same ocean model to be run on grids with different resolutions in space and/or time. Further details on running and setting up nested models within NEMO can be found at [10].

Two test models are supplied by the NOCS team, BASIC and MERGED. BASIC is a largely unmodified version of the NEMO source code which attempts to use the AGRIF pre-processor to set up a nested model. The BASIC model is a 2 degree model inside which a 1 degree model is run. Only 1 nested region is used in the BASIC model. The MERGED model is an extension of BASIC model including the NOCS specific code changes. Unlike the BASIC model is has two nested regions. It also runs at a higher resolution with the outer model resolution being 1 degree and the nested regions being $\frac{1}{4}$ of a degree.

The nested models produce similar output files to the un-nested version of NEMO (see section 5 for details) with the name of the files pre-pended by `1_` for the nested version. For example the normal, un-nested run output file is called `ocean.output` and the corresponding run output from the nested region is called `1_ocean.output` if one nested region is used and `2_ocean.output` if two nested regions are used.

We have been unable to get the nested models fully running on HECToR. We have resolved the issues with the BASIC model but as yet the MERGED model does not run successfully on HECToR. We have worked extensively with Steven Alderson from

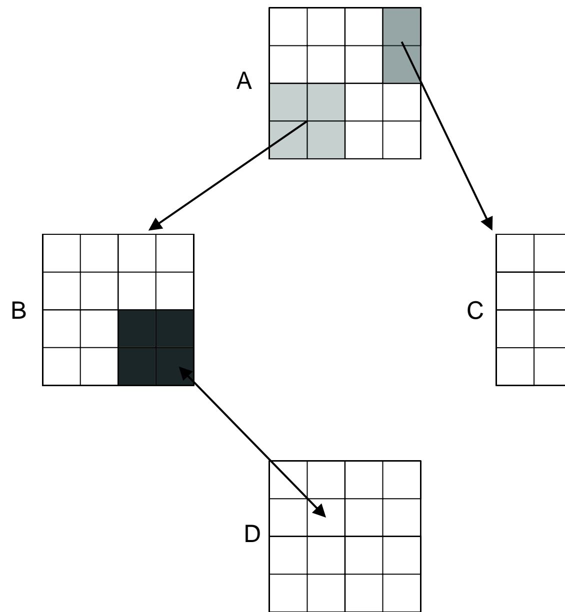


Figure 10: Schematic showing a possible NEMO grid, A, containing two nested regions B and C. Nested region B contains a second level of nesting. Note that the schematic is not drawn to scale, in practice the B, C and D regions would be smaller in physical size and grid spacing than region A. However, their increased resolution may mean they actually contain similar number of actual grid points.

NOCS to resolve this problem. Currently (and since NERC projects left HPCx) the NOCS team internal SGI cluster is the only system on which their version of the NEMO AGRIF MERGED model is running. This system uses the Intel compiler suite. Getting the MERGED model to run is a crucial part of WP2.

10.2 BASIC nested model

We begin by investigating the BASIC nested model. The code was initially compiled using the same options used by NOCS to ensure that the errors they obtained could be replicated. Initially, an optimisation level of `-O3` and the PGI compiler suite was used as this combination was found to perform best for the un-nested NEMO code.

The original version of the code (compiled with `-O3`) stops on time steps 27 and 54 (as reported by `time.step` and `1_time.step` files). The code exits normally when the zonal velocities become too large - i.e. the model has become unstable. The `ocean.output` file reports the following:

```
===>>> : E R R O R
        ===========

 stpctl: the zonal velocity is larger than 20 m/s
 ======
kt=    27 max abs(U):   23.37    , i j k:  123   3  18
```

Because the error output appears in the un-nested output file this tells us that something is wrong with the un-nested part of the model/calculation.

Before investigating this further the optimisation level was reduced to `-O0` and the code re-run. Setting the optimisation level to `-O0` removes all optimisations and should allow us to detect any possible bugs which could be causing the code to fail. Reducing the optimisation level to `-O0` results in the code running to completion, finishing on time steps 5475 and 10950 as desired. To investigate which subset of the source code is affected by the optimisation level the optimisation level is increased systematically. Table 12 summarises the results.

Optimisation used			`time.step`	`1_time.step`
`WORK/Makefile`	`AGRIF/Makefile`	`IOIPSL/src/Makefile`		
-03	-O	-O	27	54
-O0	-O0	-O0	5475	10950
-O1	-O0	-O0	5475	10950
-O1	-O1	-O1	5475	10950
-O1	-O2	-O2	5475	10950
-O1	-O3	-O3	5475	10950
-O2	-O1	-O1	27	54
-O3	-O1	-O1	27	54

Table 12: Code progress for different optimisation levels for the AGRIF BASIC model. `time.step` and `1_time.step` report the time step at which the run ended. A successful run should end on steps 5475 and 10950.

It appears that increasing the optimisation level of the main `/WORK/Makefile` from `-O1` to `-O2` changes the code in such a way as to cause a crash. From the PGI documentation the key differences between the optimisation levels are as follows:

- `-O0` Level zero specifies no optimisation. A basic block is generated for each language statement.
- `-O1` Level one specifies local optimisation. Scheduling of basic blocks is performed. Register allocation is performed.
- `-O2` Level two specifies global optimisation. This level performs all level-one local optimisation as well as level two global optimisation. If optimisation is specified on the command line without a level, level 2 is the default.

Basically `-O2` includes global optimisations such as induction recognition and loop invariant motion. Neither of these *should* cause the code to blow up but clearly something is going wrong. It's possible the optimisation highlights an existing bug in the code or that the compiler is doing something it shouldn't. It's also possible that the optimisation brings out an existing numerical instability, e.g. calculations being performed in a slightly different order which results in the velocity increasing too fast. The later hypothesis can be tested by reducing the time step used to see if this allows the code to run successfully.

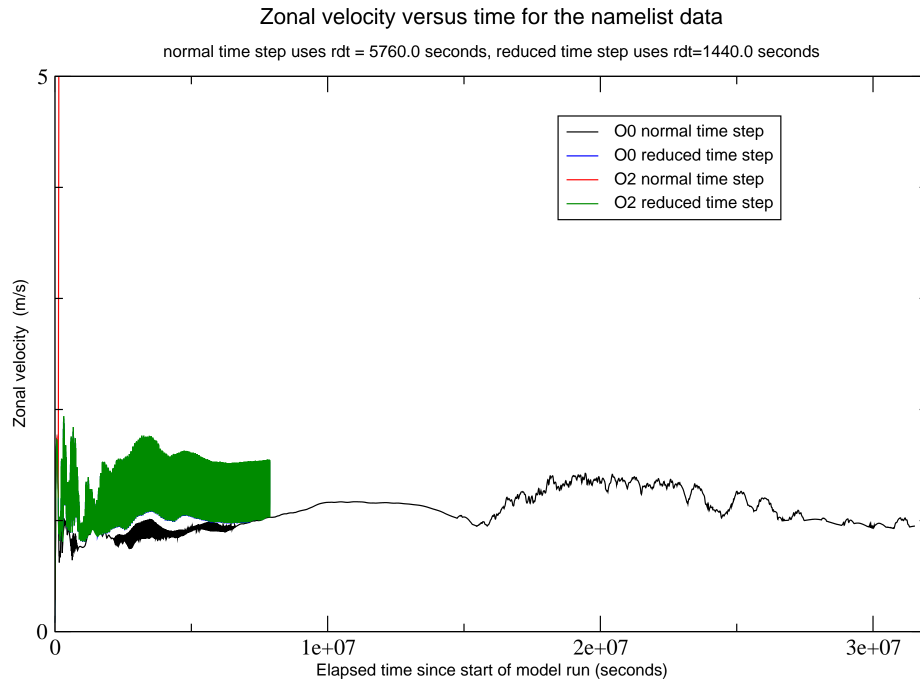
The time step is specified in the namelist file(s) via the `rdt` parameter. For the BASIC model the time step used in the nested model is half that used in the non-nested model. The initial values of `rdt` for the non-nested and nested models are respectively 5760.0 seconds and 2880.0 seconds. To ascertain whether the time step is too large the model can be re-run with a smaller time step. Initially the time steps were reduced by half to 2880.0 and 1440.0 respectively, however, the zonal velocity still increased too fast. Reducing the time steps but a factor of four relative the the original, i.e. to 1440.0 and 720.0 seconds allows the run to successfully.

We can gain a better understanding of what is going on by plotting out the zonal velocity as a function of model time. The zonal velocity can be obtained by modifying the `stp_ctl` subroutine in `stpctl.F90` such that it outputs the zonal velocity every time step and not just when a problem occurs. A diff of the original and modified code gives:

```
diff stpctl.F90 stpctl.F90.orig
< !FR dump out velocity to file for the failed step
<              WRITE(99,*) kt, zumax, ii, ij, ik
< !FR dump out velocity to file for the failed step
150,153d146
< !FR dump out velocity to file for every time step
<        WRITE(99,*) kt, zumax, rdt
< !FR dump out velocity to file for every time step
```

Figure 11 shows the zonal velocity plotted against model time for the non-nested (i.e. `namelist` model). Figure 12 shows the zonal velocity plotted against model time for the nested (i.e. `1_namelist`) model.

From figures 11 and 12 it is clear that the problem with the velocity occurs for the non-nested model (see red line on figure 11). The original time step of 5760.0 and 2880.0 seconds for the non-nested and nested models is simply too large. Reducing the time step



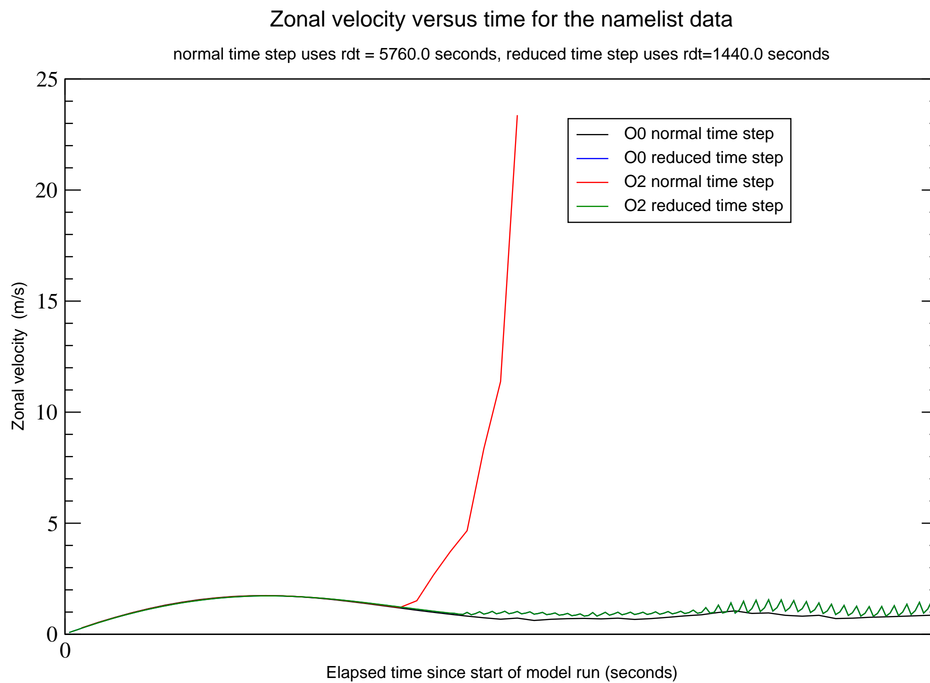
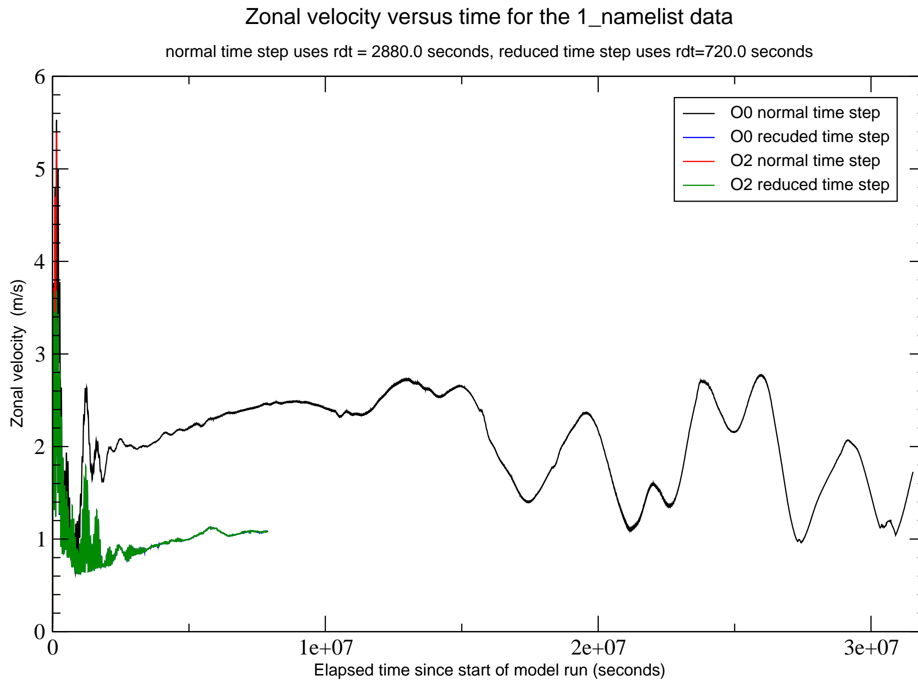


Figure 11: Zonal velocity against model time for the non-nested model. The top figure shows the full plot with the bottom figure zoomed in on the first few time steps.



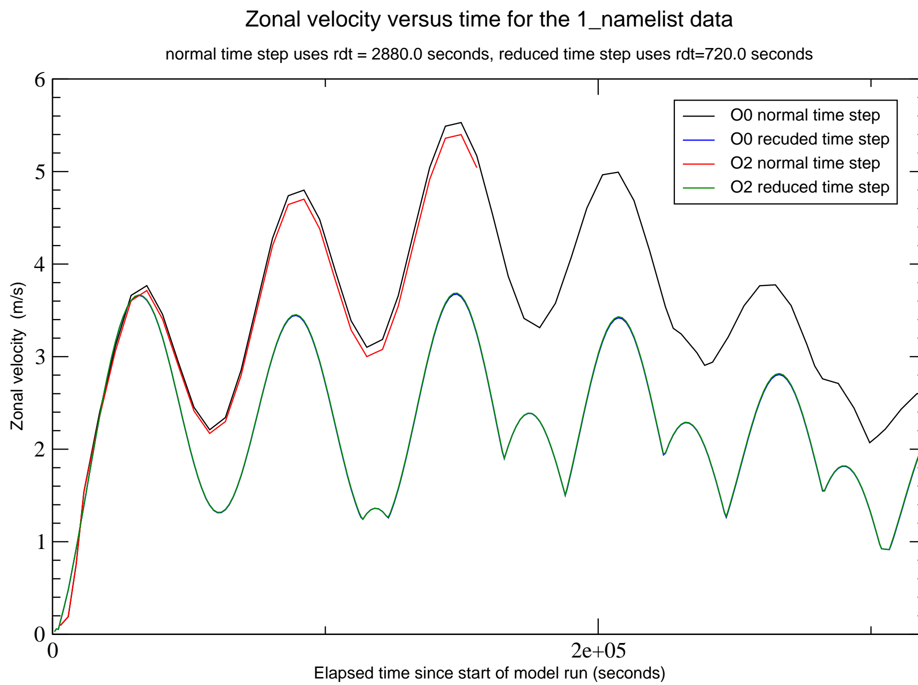


Figure 12: Zonal velocity against model time for the nested model. The top figure shows the full plot with the bottom figure zoomed in on the first few time steps.

by a factor of 4, to 1440.0 and 720.0 seconds prevents the zonal velocity from blowing up. With the reduced time step the `-O0` and `-O2` versions of the code behave very similarly. The zonal velocities are almost identical (the blue and green lines overlap on figures 11 and 12).

It seems there are two choices for running the AGRIF models; run with a code compiled with `-O1` to avoid the numerical instabilities or reduce the time step. The second solution is safer as the model could become unstable even with `-O1` or less. The researchers have reduced the time step as suggested and confirmed that the results appear sensible when compared with those obtained on their SGI cluster.

10.2.1 BASIC nested model performance

We have also investigated the performance of the BASIC model. Table 13 gives the total runtime for several different optimisation levels running on 64 processors. The shortest

Optimisation used	Time for 5475 steps (seconds)
`-O0 -O0 -O0`	1920
`-O1 -O1 -O1`	2283
`-O2 -O2 -O2`	1402
`-O3 -O2 -O2`	1336
`-O3 -O3 -O3`	1366

Table 13: Total runtime for the BASIC nested model over 5475 time steps for different optimisation flags using the PGI compiler. The 3 levels of optimisation were applied respectively to the `Makefile`, `AGRIF/Makefile` and `IOIPSL/src/Makefile`. All tests were run with `jpni=8`, `jpnj=8` and `jpnij=64`.

runtime is achieved when either `-O3,-O2,-O2` or `-O3, -O3, -O3` levels of optimisation are used with very little difference between these. These results are consistent with the non-nested version version of NEMO (c.f. sections 6.6 and 9.2).

The code was also tested on 16 up to 128 processors to examine the scalability of the nested model. For each of these runs an optimisation level of `-O3 -O2 -O3` was used as this gave the best performance as detailed above. The scaling results for the BASIC model are given in table 14. From table 14 we see that the 64 processor run gives the

No. of processors	Time for 5475 steps (seconds)	Cost per time step (AU's)
16	4397	0.0172
32	2164	0.0169
64	1366	0.0208
128	1636	0.0510

Table 14: Total runtime of the BASIC nested model over 5475 time steps for different processor counts. Optimisation levels of `-O3`, `-O2` and `-O2` were applied to the `Makefile`, `AGRIF/Makefile` and `IOIPSL/src/Makefile` respectively.

fastest runtime for this particular model. However, the 32 processor run is actually more efficient in terms of the AU usage per model time step. Providing this longer runtime is

acceptable, researchers may wish to run this model on 32 processors in order to minimise the cost (in AU's) per model time step.

10.3 MERGED nested model

Having found the cause and a solution to the BASIC nested model crashing we now focus on the MERGED model. This model is more complex because it allows for two levels of nesting and also includes the code required by NOCS to carry out their research.

The MERGED AGRIF model also initially failed with an error of the form (from the (`ocean.output` file):

```
===>>> : E R R O R
        ===========

  stpctl: the zonal velocity is larger than 20 m/s
  ======
kt=     4 max abs(U):   50.06    , i j k:  243  56  53
```

and also from the (`1_ocean.output`) file:

```
===>>> : E R R O R
        ===========

  stpctl: the zonal velocity is larger than 20 m/s
  ======
kt=    16 max abs(U):   461.8    , i j k:  314   3  30
```

According to the code output files (`*ocean.output* and *time.step`) the model stops on time steps 4, 16 and 64 for the levels of nesting.

As with the BASIC model different compiler options were explored to see if the optimisation level was a factor in causing the velocities to grow uncontrollably. The MERGED model was compiled incrementally with optimisation levels from `-O3` down to `-O0`. For each optimisation level the code stops in the same manner as described above.

The time step, represented by the value of `rdt` in the `namelist`, `1_namelist` and `2_namelist` files was also varied to see if reducing it gave any improvement. Table 15 summarises the results.

Time step, rdt in seconds					
`namelist`	`1_namelist`	`2_namelist`	`time.step`	`1_time.step`	`2_time.step`
3600.0	900.0	225.0	4	16	62
1600.0	400.0	100.0	4	16	64
400.0	100.0	25.0	6	24	96

Table 15: Time steps at which MERGED AGRIF model crashes for different time steps

From table 15 it seems that regardless of the time step used the MERGED AGRIF model continues to crash early on in the model run. This suggests that unlike the BASIC model the choice of time step is not the issue. The zonal velocity was extracted as

described above in section 10.2. Figure 13, shows the zonal velocity plotted as a function of the elapsed model time for three different values of time step, `rdt`. From figure 13 its clear that the problem lies with the outer-most (i.e. coarsest) model (`namelist`) with the model using `1_namelist` (i.e. first level of nesting) also affected but to a lesser degree.

Further investigation of the output files suggests that something actually goes wrong prior to time step 4. The `mpp_output.0*` files are found to contain `NaN` values as early as time step 2. The cause of these NaN values is currently unknown but clearly they should not be present if the code is running correctly.

Locating the cause of these NaN values will likely be fundamental in getting the merged model to run on HECToR. Unfortunately, the PGI compiler doesn't allow direct trapping of NaN values. The `-Ktrap` flag can be used to trap a number of other numerical problems, e.g. denormalised operand, divide-by-zero, overflow, underflow and inexact. Trapping of NaN values may be possible by instrumenting the code with the `isnan` function which is a logical function which returns true if a NaN value is detected and false otherwise. To use this function the code must be instrumented with `isnan()` calls everywhere where a NaN value is suspected.

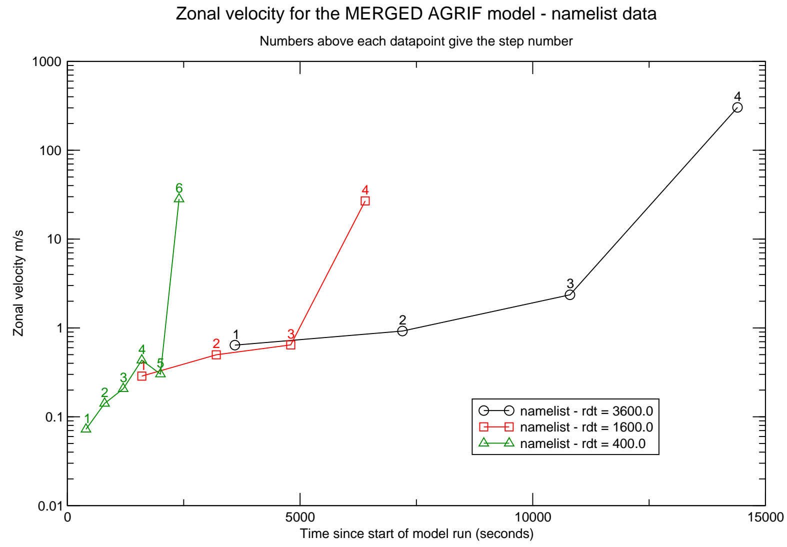
The PathScale compiler, however, provides options which may help with NaN detection particularly if the NaN values are arising from uninitialised values. The two flags of interest are:

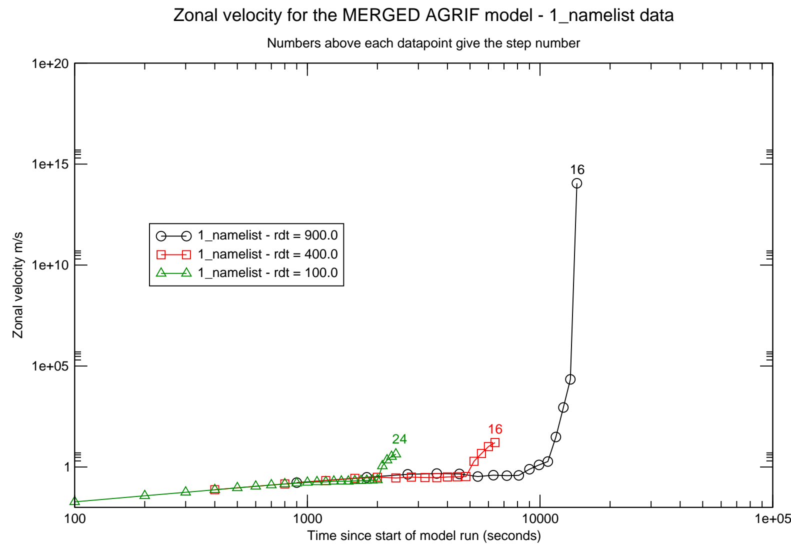
- `-trapuv` - traps uninitialised values by explicitly setting them to NaN
- `-zerouv` - zeros all initialised values

Various attempts have been made to compile the merged model with the PathScale compiler. The compiler fails with an internal error. The output from version 3.0 is below:

```
ftn -freeform -c -Dkey_agrif -Dkey_agrif_nolim -Dkey_trabbl_dif
-Dkey_mpp_mpi -Dkey_orca_r1=64 -Dkey_lim2 -Dkey_dynspg_flt
-Dkey_diaeiv -Dkey_ldfslp -Dkey_traldf_c2d -Dkey_traldf_eiv
-Dkey_dynldf_c3d -Dkey_dtatem -Dkey_dtasal -Dkey_tradmp
-Dkey_trabbc -Dkey_zdftke -Dkey_zdfddm -O0 -r8
-module ../../../lib -I../../../lib -I../../../lib/oce
-I/home/n01/n01/fionanem/netcdf/3.6.2/include \
OPAFILES/lib_mpp.F90 || { if [ -f lib_mpp.L ] ;
then mv lib_mpp.L
../../../tmp ; fi ; false ; exit ; }
/opt/xt-asyncpe/1.0c/bin/ftn: INFO: linux target is being used
pathf90-3.0 INTERNAL ERROR: /opt/pathscale/lib/3.0/mfef95 died
due to signal 11

Please report this problem to <support@pathscale.com>.
Problem report saved as
/home/n01/n01/fionanem/.ekopath-bugs/pathf90-3.0_error_HgpoTn.i
Please review the above file and, if possible, attach it to
your problem report.
make: *** [../../../lib/oce/libopa.a(lib_mpp.o)] Error 1
```





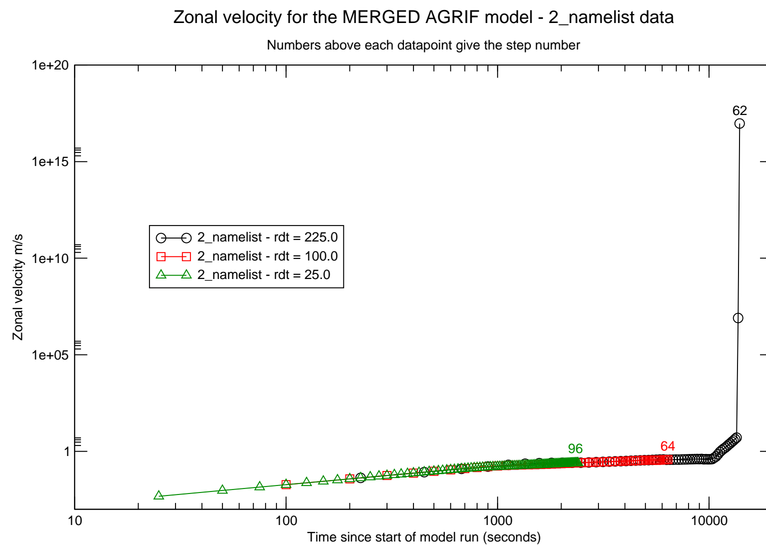


Figure 13: Zonal velocity against model time for the upper most model represented by the `namelist` file.

Version 3.1 was also tested, again it fails on file `lib_mpp.F90` but with a slightly different error.

```
ftn -freeform -c -O0 -r8 -module ../../../lib -I../../../lib
-I../../../lib/oce -I/home/n01/n01/fionanem/netcdf/3.6.2/include \
lib_mpp.f|| { if [ -f lib_mpp.L ] ; then mv lib_mpp.L ../../../tmp ;
fi ;
false ; exit ; }
/opt/xt-asyncpe/1.0c/bin/ftn: INFO: linux target is being used
Signal: Segmentation fault in IR->WHIRL Conversion phase.
"lib_mpp.f": Error: Signal Segmentation fault in phase IR->WHIRL
Conversion -- processing aborted
*** Internal stack backtrace:
pathf90-3.1 INTERNAL ERROR: /opt/pathscale/lib/3.1/mfef95 died due
to signal 4
make: *** [../../../lib/oce/libopa.a(lib_mpp.o)] Error 1
```

These errors suggest an internal problem with the PathScale compiler and as such will need to be referred back to the compiler developers for a bug fix. This has been submitted as HECToR query Q29941 and is currently under investigation.

With the PGI compiler, it transpires the `-Msave` flag has the side-effect of initialising variables to zero. Compiling with this flag results in the code hanging or taking an inordinate amount of time to run (1 step complete in an hour on 256 processors) which makes its use impractical.

We have also tried compiling with the `-Mbounds` flag which performs array bounds checking at compile and runtime. Running the executable with `-Mbounds` caused the code to crash with an error message stating that one of the array indices was negative. The affected file was `fldread.F90`. Removing `-Mbounds` and then re-running whilst writing out the affected indices demonstrated that no negative values occur. It's possible therefore, that the `-Mbounds` compiler flag, has altered the code in some way as to create or highlight a new problem which doesn't appear when the flag is omitted. The results of this are somewhat inconclusive.

The Totalview debugger should be able to provide some helpful information. E.g. tracing through the code, watching variables etc. Unfortunately it has so far not been possible to obtain any symbolic information when the nested version of NEMO (or indeed any code) is compiled with the PGI compiler. The problem has been replicated with a simple helloworld type and filed as a bug with Cray/Totalview via HECToR Q22386. The PathScale compiler does not suffer from this problem but as we cannot compile the code with PathScale this doesn't help.

A number of attempts have been made to compile the code on the TDS as this has the latest versions of the compilers, system libraries, operating system etc. However, no PathScale licence is present on the TDS and the Totalview licence also appears to be invalid and therefore limited progress was possible on the TDS.

11 Conclusions and future work

In this section we summarise the work carried out during the NEMO dCSE project and present the main conclusions. Additional work carried out during the project is also summarised along with some suggestions for future work.

11.1 Summary of work and conclusions

Two different versions of NEMO (2.3 and 3.0) have been compiled and tested on HECToR. The performance of these versions has been investigated and an optimum processor count suggested based on the researchers' requirements for job turn-around. The performance of both the PathScale and PGI compilers has been investigated along with an investigation of how the performance varies with the choice of compiler flags. The NEMO code has been found to scale up to 1024 processors with the best performance in terms of runtime versus AU usage being obtained between 128 and 256 processors. Running NEMO in single core mode is found to be up to 18.59% faster than dual core mode, however, the reduction is not sufficient to warrant the increased AU usage. The choice of grid dimensions has been investigated and is found to be optimal for square grids. Where square grids are not possible choosing the dimensions such that the number of cells in the horizontal direction is less than the number in the vertical direction (i.e. choosing `jpni < jpnj` within the code) gave the best performance. Removal of the land only squares from the computations gave significant reductions to the AU usage, by as much as 25% at larger processor counts. The runtime was also found to decrease, albeit by a lesser extent. Profiling of the code suggests that NEMO spends a considerable amount of time in initialisation and file I/O and thus any reduction that can be made in this area will be beneficial.

NetCDF 4.0, HDF5 1.8.1, zlib 1.2.3 and szip have been installed and tested as part of this project. Initially, beta releases were used until the final release versions became available in June 2008. NetCDF 4.0 is found to give a considerable reduction to both the amount of I/O produced and the time taken in I/O when using the NOCSCOMBINE tool. In addition, the version of netCDF 4.0 installed as part of this project is found to be between 8-20% faster than that installed centrally (via modules) on the system.

NEMO has been converted to use netCDF 4.0 for its main output files resulting in a reduction in output file size of up to 3.55 times relative to the original netCDF 3.X code. For the test model no significant runtime improvement is observed. It is expected that a real research type run should benefit more due to the different frequency of output involved. The restart files have not been converted to use netCDF 4.0.

The BASIC nested model has been compiled and tested and problems with the time step interval identified and rectified. The performance of the BASIC nested model has been investigated with the optimal processor count (in terms of AU usage per time step) found to be 32. The more complex MERGED nested model has not yet run successfully on HECToR. The code compiles but crashes due to the velocity becoming extremely large and NaN values occurring. Various compiler and debugger problems were experienced making identifying the reason for this crash very problematic. These issues have been reported to Cray (HECToR queries Q29941 and Q22386 both described within the report) and are currently awaiting resolution.

11.2 Other work

Below, is a list summarising additional work also carried out as part of the NEMO dCSE project:

- Presentation at HECToR User Group meeting, September 2008
- Presentation at HPCx Parallel I/O Workshop, December 2008
- Article for EPCC News [11]
- Contributed to article for Scientific Computing World [12]
- Highlighted bugs with Totalview/PGI and the PathScale compiler (HECToR queries Q29941 and Q22386).

11.3 Future work

Some suggestions for future work on this subject are given below:

- Full investigation of the optimal chunking parameters when using netCDF 4.0 with NEMO.
- Identify whether converting the restart files to netCDF 4.0 format is feasible.
- Further work on the MERGED AGRIF model subject to having access to a working debugger or bug fix for the PathScale compiler.

Acknowledgements

The Cray Centre of Excellence staff are gratefully acknowledged for their help with various aspects of this NEMO dCSE project.

A NEMO output comparison scripts

This appendix contains the scripts used to compare the NEMO output files following any changes to the code. The scripts are called `test_output.sh` and `get_parameters.sh`. Their functionality is as follows:-

- `test_output.sh` - driver script which calls `get_parameters.sh` once for every parameter which needs to be extracted and compared. The `TEST_OUTPUT_DIR` variable will need to be changed to point to the directory containing the output to be verified.
- `get_parameters.sh` - extracts parameters from NEMO netCDF output files and uses NOCSCOMBINE to combine the into a single file before using the CDFTOOLS cdfmeanvar to compare the two sets of data.

The usage is as follows:

```
./test_output.sh
```

However, this would normally be run within a serial batch script to avoid using excessive resources on the login nodes.

Source for test_output.sh

```
##
## Copyright (c) The University of Edinburgh 2009. All Rights Reserved.
##
#!/bin/bash
#
# Script to test the NEMO output against the vanilla output to ensure that
# changes have not damaged the numerical output
#
# This script will generally be run via the serial queues as it could
# potentially take a number of minutes.

export PATH=$PATH:~/scripts
# Give list of files to be processed/created:
filelist="votemper vozocrtx vomecrty vovecrtz isssalin"

# VANILLA_OUTPUT_DIR = directory containing original NEMO 3.0 output
# created with netCDF 3.6.2 in Classic mode.
export VANILLA_OUTPUT_DIR=/work/n01/n01/fionanem/NEMO_V3.0/ORCA025/EXP_V3.0_005

# TEST_OUTPUT_DIR = directory containing test dataset
export TEST_OUTPUT_DIR=/work/n01/n01/fionanem/NEMO_V3.0/ORCA025/EXP_V3.0_015

# Check whether vanilla output exists and if not create it - we need one
# condition per file as different input data are involved for each test

cd $VANILLA_OUTPUT_DIR
for file in $filelist
do

  if [ ! -e ${file}_ncdf3.nc ]
  then

    echo "Creating file ${file}_ncdf3.nc"

    if [ $file = "votemper" ]
    then
#      echo "RUNNING get_parameters.sh O25-V3_CU30_19580101_19580101_grid_T_0000.nc 3 votemper"
      get_parameters.sh O25-V3_CU30_19580101_19580101_grid_T_0000.nc 3 votemper
    fi

    if [ $file = "vozocrtx" ]
    then
      echo "Running..."
#      echo "RUNNING get_parameters.sh O25-V3_CU30_19580101_19580101_grid_U_0000.nc 3 vozocrtx"
      get_parameters.sh O25-V3_CU30_19580101_19580101_grid_U_0000.nc 3 vozocrtx
    fi

    if [ $file = "vomecrty" ]
    then
      echo "Running..."
#      echo "RUNNING get_parameters.sh O25-V3_CU30_19580101_19580101_grid_V_0000.nc 3 vomecrty"
      get_parameters.sh O25-V3_CU30_19580101_19580101_grid_V_0000.nc 3 vomecrty
    fi

    if [ $file = "vovecrtz" ]
    then
```

```
      echo "Running..."
#      echo "RUNNING get_parameters.sh O25-V3_CU30_19580101_19580101_grid_W_0000.nc 3 vovecrtz"
      get_parameters.sh O25-V3_CU30_19580101_19580101_grid_W_0000.nc 3 vovecrtz
    fi

    if [ $file = "isssalin" ]
    then
      echo "Running..."
#      echo "RUNNING get_parameters.sh O25-V3_CU30_19580101_19580101_icemod_0000.nc 3 isssalin"
     get_parameters.sh O25-V3_CU30_19580101_19580101_icemod_0000.nc 3 isssalin
    fi

  else
    echo "The file ${file}_ncdf3.nc already exists"
  fi
done

# Combine .nc files and create test datasets:
cd $TEST_OUTPUT_DIR
for file in $filelist
do
  echo "About to remove file ${file}_ncdf4.nc before new run"
# Uncomment when running in batch/new tests
#  rm ${file}_ncdf4.nc
done

# Uncomment for new run. All 5 files need to be created for each new test
get_parameters.sh O25-V3_CU30_19580101_19580101_grid_T_0000.nc 4 votemper
get_parameters.sh O25-V3_CU30_19580101_19580101_grid_U_0000.nc 4 vozocrtx
get_parameters.sh O25-V3_CU30_19580101_19580101_grid_V_0000.nc 4 vomecrty
get_parameters.sh O25-V3_CU30_19580101_19580101_grid_W_0000.nc 4 vovecrtz
get_parameters.sh O25-V3_CU30_19580101_19580101_icemod_0000.nc 4 isssalin

# Now we've extracted a dataset from each type of output file need to compare with
# The corresponding vanilla output and ensure the numerical values match up.

testoutputfile="test.output"        # Contains output from the comparison
rm $TEST_OUTPUT_DIR/$testoutputfile # Remove old file before comparison

for file in $filelist
do
  rm tmpfile # Ensure the tmpfile is removed prior to any comparison of the output
  echo "Testing file = ${file}_ncdf3.txt" >> $testoutputfile
  diff $VANILLA_OUTPUT_DIR/${file}_ncdf3.txt $TEST_OUTPUT_DIR/${file}_ncdf4.txt >> $testoutputfile
  diff $VANILLA_OUTPUT_DIR/${file}_ncdf3.txt $TEST_OUTPUT_DIR/${file}_ncdf4.txt >> tmpfile
  if [ -s tmpfile ]
  then
    echo "There is a difference between the test files" >> $testoutputfile
  else
    echo "No differences" >> $testoutputfile
  fi
  echo " " >> $testoutputfile
done
echo "*** All tests completed ***" >> $testoutputfile

# Tidy up - could remove tmpfile here to ensure it doesn't exist between tests.
# rm tmpfile
```

Source for get_parameters.sh

```
##
## Copyright (c) The University of Edinburgh 2009. All Rights Reserved.
##
#!/bin/bash
#
# Script to extract parameters from NEMO netCDF output so that
# output can be verified and compared etc
#
# $1 = name of the file series to be combined and tested
# e.g.  ORCA025-N200_5d_19580101_19580101_grid_T_0000.nc

EXPECTED_ARGS=3

if [ $# -ne $EXPECTED_ARGS ]
then
  echo " "
  echo "Script for extracting parameters from NEMO netCDF output"
  echo "so that output can be compared between runs"
  echo " "
  echo "Usage: `basename $0` {filename.nc netCDF_version(3 or 4), variable (e.g. votemper)}"
  exit $E_BADARGS
fi

if [ $2 -ne 3 -a $2 -ne 4 ]
then
  echo "You must specify whether the file is netCDF3 of netCDF4 format"
  exit
fi

# Set paths to different versions of nocscombine/cdftools - need different
# versions for Classic and netCDF4 files - control selection with input $2

if [ $2 -eq 3 ]
then
  export NOCSCOMB_EXE=/home/n01/n01/fionanem/NOCSCOMBINE/nocscombine
  export CDFTOOLS_DIR=/home/n01/n01/fionanem/CDFTOOLS/bin
else
  export NOCSCOMB_EXE=/home/n01/n01/fionanem/NOCSCOMBINE/nocscombine4_release_nozlib
  export CDFTOOLS_DIR=/home/n01/n01/fionanem/CDFTOOLS/bin_nc4_release
fi

# Give the output netCDF file a name - name is based on variable being
# extracted $MYVAR and netCDF version (3 or 4)
export MYVAR=$3
export OUTFILE_NC=${MYVAR}_ncdf${2}.nc
export OUTFILE_TXT=${MYVAR}_ncdf${2}.txt

# Run NOCSCOMBINE to create large .nc file
echo "Running NOCSCOMBINE on $1 with $NOCSCOMB_EXE"
echo "DEBUG: RUNNING: $NOCSCOMB_EXE -f $1 -d $MYVAR -o $OUTFILE_NC"
$NOCSCOMB_EXE -f $1 -d $MYVAR -o $OUTFILE_NC

# Extract mean of variable using cdfmeanvar
echo "Running cdfmeanvar on $OUTFILE_NC, creating output $OUTFILE_TXT"
echo "DEBUG: RUNNING: $CDFTOOLS_DIR/cdfmeanvar $OUTFILE_NC $MYVAR T > $OUTFILE_TXT"
$CDFTOOLS_DIR/cdfmeanvar $OUTFILE_NC $MYVAR T > $OUTFILE_TXT
```

B Some notes on HDF5 datasets

HDF5 is a set of tools and libraries that allows extremely large and complicated data collections to be managed. The file format used by HDF5 is designed to be portable. Further information on HDF5 can be found at [8].

An HDF5 dataset is an object comprised of a collection of data elements and metadata. In addition the dataset may have optional attribute objects.

- Data elements - one dimensional or multi-dimensional arrays. Can be specified types (integer, real, double, char) or compound type (c.f. C like structs)
- Metadata - describes the data elements, data layout and all information necessary to read/write (e.g. chunking/compression used) and interpret the data.
- Attributes - optional, meta data object which can be used to describe the nature and/or intended use of a data set.

When an HDF5 dataset is created a number of properties of the dataset are set:

- `name` - name of dataset usinb alphanumeric ASCII characters
- `dataspace` - defines the number of dimensions, the current extent in each dimension and the maximum allowed extent in each dimension.
- `datatype` - a dataset has a datatype associated with it which describes the layout of the raw data in the file. The file datatype is set when the dataset is created and cannot be changed.
- `storage properties` - control how the data is stored and whether any chunking or compression is used. The storage properties are set when the dataset is created and cannot change.

Most of these dataset properties are permanent, they cannot be changed during the lifetime of the dataset. The key exception is the, dataspace which can be expanded up to its maximum dimensions.

Data Transfer - e.g. how does the data get from the application to a physical file? Essentially the HDF5 library implements data transfers through a pipeline which includes:

- Data transformations
- Chunking
- I/O operations
- optional filters, e.g. compression, can also be added to the pipeline

Storage allocation in the file, early, incremental, late - may need consideration for parallel I/O.

B.1 HDF5 Filters

There are a number of different filters which can be applied to an HDF5 dataset:

- N-bit filter - essentially compresses the data by removing the unused bits before storing the data on output. The data is then unpacked on input restoring the missing bits. Quite complex to use but may save diskspace. Checks would be required such that no information is lost from the data.
- Scale-offset filter - performs a scale and offset on each data value truncating the resulting value to a fixed number of bits before storing. E.g the operation performed is $d_{new} = a * d_{old}(i) + b$ where d_{old} is the original data value, d_{new} is the new data value, a is the scale and b is the offset. `minimum-bits` determines the minimum number of bits that will be used. For integer data the filter is lossless (unless too small a value for `minimum-bits` is selected. However, for floating point data, the filter translates the floating point data to integer data (the filter is lossy - information is lost due to its action) and so is not useful for NEMO.
- Szip filter - the Szip compression

References

[1] *NEMO Home page*
http://www.nemo-ocean.eu/

[2] *User Guide to the HECToR Service (Version 1.0)*,
http://www.hector.ac.uk/support/documentation/userguide/hectoruser/hectoruser.html

[3] *Network Common Data Format (netCDF) web page*,
http://www.unidata.ucar.edu/software/netcdf/

[4] *NetCDF User's Guide for Fortran 90*,
http://www.unidata.ucar.edu/software/netcdf/f90/documentation/f90-html-docs/

[5] *CDFTOOLS web page*,
http://meolipc.hmg.inpg.fr/CDFTOOLS/cdftools-2.1.html

[6] Message Passing Interface Forum, 2008. *MPI: A Message-Passing Interface Standard, Version 2.1*, High Performance Computing Center Stuttgart (HLRS). http://www.mpi-forum.org/docs/mpi21-report.pdf

[7] *Zlib web page*
http://www.zlib.net/

[8] *HDF5 web page*
http://hdf.ncsa.uiuc.edu/HDF5/

[9] *HDF5 1.8.1 Release Notes*
ftp://ftp.hdfgroup.org/HDF5/prev-releases/ReleaseFiles/release5-181

[10] *NEMO Web page - AGRIF nesting tool (accessible only to registered users)*
http://www.nemo-ocean.eu/index.php/Using-NEMO/Pre-and-post-processing-packages/Pre-Processing/AGRIF-nesting-tool

[11] *EPCC news article - NEMO: improving ocean modelling*,
http://www.epcc.ed.ac.uk/downloads/epcc_news/EPCCNews62.pdf

[12] *Scientific computing world article - Weathering Well*,
http://www.hpcprojects.com/features/feature.php?feature_id=228