Science & Technology Facilities Council

Technical Report
DL-TR-2012-00>

# Pq£qx{|uzs ZQY [ r{~ Xm~sq Y¢x¡u9O{~q _omxm~ _¦ ¡qy :Ruzmx ^q|{~¡ of¡tq pO_Q ZQY [ \~{vqo¡

_Y \uowxq ,M^ \{~¡q~

_q|¡qynq~ 201>

**©2012 Science and Technology Facilities Council**

Enquiries about copyright, reproduction and requests for additional copies of this report should be addressed to:

Chadwick Library
Science and Technology Facilities Council
Daresbury Laboratory
Sci-Tech Daresbury
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

**ISSN 1362-0207**

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

# Developing NEMO for Large Multi-core Scalar Systems:

## Final Report of the dCSE NEMO project

*Stephen M. Pickles and Andrew R. Porter, STFC Daresbury Laboratory, July 2012.*

## Abstract

We report our experiences in implementing a set of modifications to the NEMO ocean modelling code intended to improve the code's performance and scalability on HECToR and other large multi-core scalar systems. These modifications were inspired by the coastal ocean modelling code POLCOMS, and involved re-ordering 3-dimensional array indices and associated loop nests, multi-core aware partitioning, and elimination of redundant computation on land. The modifications are invasive, pervasive and mutually dependent, and we under-estimated the effort required to implement them fully in NEMO. We report performance results from our partial implementation, and provide revised estimates of the effort required to complete a full implementation and the likely performance benefits. We also describe by-products of this work that have been incorporated in the production use of NEMO, including dynamic memory allocation, various bug fixes, and a solution to an issue degrading NEMO's performance on HECToR.

## Table of Contents

## 1. Introduction

This is the final report of the HECToR dCSE project "Developing NEMO for large, multi-core scalar systems".

The goal of the project was to apply a set of modifications to the ocean modelling code NEMO, designed to improve the performance, scalability and usability of the code on modern very large multi-core scalar systems such as HECToR. The intended optimisations, already proven in POLCOMS, included:

1. Re-ordering of 3-dimensional array indices and associated loop nests for better cache re-use on scalar processors;
2. Multi-core aware partitioning (MCA) and elimination of redundant computation on land;
3. Provision of an ensemble capability.

Of these, (1) was completed, albeit on a subset of the NEMO source tree, (2) was partially completed, and (3) was dropped. We had underestimated what could be achieved with 12 person-months of effort.

In section 2, we provide sufficient background on the NEMO and POLCOMS ocean modelling codes to motivate this project. In section 3, we describe the project structure, including sequence of tasks, consultation with stakeholders, and performance targets. Section 4 provides detailed technical information about our methodology, aimed primarily at NEMO developers. Section 5 reports performance results, as achieved by the conclusion of the project. In section 6, we present our main conclusions, and make recommendations for follow-up work.

## 2. Background and motivation

### NEMO

NEMO (Nucleus for a European Model of the Ocean) [1] is an ocean modelling code of great strategic importance for the UK and European oceanographic communities.

Although NEMO has been used successfully for a number of years in global and ocean basin applications, its use as a shelf-sea model is less well developed. In order to align the shelf-seas modelling work with the open-ocean, to coordinate effort with the Met Office and to address issues of ocean-shelf coupling, the shelf-sea modelling partners of the UK marine research programme (Oceans 2025; see http://www.oceans2025.org) determined to transition from the use of the Proudman Oceanographic Laboratory Coastal Ocean Modelling System (POLCOMS) [2, 3] to the use of NEMO for modelling shelf seas. When making this decision, they recognised that significant work would be required to bring NEMO to bear on Grand Challenge problems in multi-scale oceanography, particularly in the algorithms used for shallow-sea problems and in the performance of the code and its scalability to many thousands of cores on modern architectures. This project was designed to address the performance and scalability concerns, drawing on techniques previously proven in POLCOMS and applying them to NEMO.

### Related Work

This project builds upon previous work on NEMO and POLCOMS. A dCSE project looked at compiler options, use of netCDF 4.0 compression, and the removal of land-only cells by pre-processing [4, 5]. Our own work with NEMO in the GSUM project [6], and our refinements to the POLCOMS partitioning method [7] in the GCOMS project [3], together established the feasibility of the programme of work undertaken here. In the recently

completed GNEMO project [8], which ran concurrently with this project, we examined the feasibility of porting NEMO to GPUs.

Hybrid MPI+OpenMP parallelisation strategies (see e.g. [9]) have become increasingly important with the commoditisation of multi-core, shared memory nodes, and early experiments applying this approach to NEMO have been conducted at CMCC [10]. The activity is expanding in PRACE-2IP, in which alternative approaches to those employed in [10] are being investigated.

The issue of I/O performance in NEMO is being addressed by the development of asynchronous, parallel I/O within the core NEMO programme.

### Comparison of NEMO and POLCOMS

Both NEMO and POLCOMS are finite-difference ocean-modelling codes, written in Fortran and parallelised using an MPI-everywhere paradigm. Both geographically decompose their domains in the horizontal, but not vertical, dimensions. Each rectangular sub-domain is owned by a single MPI process. The models deal with both 2-dimensional (2D) and 3-dimensional (3D) fields, which are represented as Fortran arrays. These arrays are declared with haloes, and field values on the haloes must be exchanged frequently, which involves nearest-neighbour, point-to-point communications.

Despite these fundamental similarities, NEMO and POLCOMS have some significant differences, arising out of their different origins. NEMO is descended from OPA, which was a deep-ocean code, designed for vector computers. Subsequently, NEMO's developers have unashamedly striven to preserve the code's affinity for vector architectures. On the other hand, POLCOMS was designed for coastal and shelf-seas, and made the transition from vector to massively parallel scalar architectures early in its life cycle [2].

A typical domain of the coastal ocean contains both wet (sea) and dry (land) points. Occasionally, a grid point can oscillate between the wet and dry states, e.g. in the presence of inundation. Computation performed at permanently dry points is redundant. There are two obvious approaches to deal with this redundancy: either perform the redundant computations and mask out the results on dry points, or take steps to eliminate the redundant computations altogether. NEMO takes the former approach; this comes at the price of imbalance in the amount of *useful* computation performed by each processor core. POLCOMS takes the latter approach, explicitly testing the mask at each grid point.

### Decomposition, array index ordering and masks in NEMO

NEMO decomposes its domain into equally sized rectangular sub-domains. Prior to version 3.3, the sizes of the sub-domains were fixed at build-time. It is possible to eliminate sub-domains situated entirely on land by pre-processing; but this is not universally used in production runs.

The focus on vector architectures accounts for NEMO's choice of array-index ordering. In NEMO, 3D arrays are declared with the z-index, or vertical index, or (sometimes) level index, last, e.g.

```
REAL(wp), DIMENSION(jpi,jpj,jpk) :: a
```

Thus within any sub-domain, the `jpi*jpj` field values for any level are contiguous in memory. When the target machine comprises a small number of vector processors, `jpi*jpj` tends to be a large number, which helps effective vectorisation. This is usually accompanied by a trick: many computations only need to traverse the interior of the sub-domain (i.e. excluding the halo points):

```
DO jj=2,jpj-1
   DO ji=2,jpi-1
      …
```

but better vectorisation is achieved by performing redundant computation on the halo points (which will be overwritten at the next halo exchange):

```
#define fs_2 1
#define fs_jpim1 jpi
#define fs_jpjm1 jpj
   DO jj=fs_2, fs_jpjm1
      DO ji=fs_2, fs_jpim1
   …
```

This idiom occurs throughout the NEMO source code and is enshrined in the NEMO style guide.

NEMO uses 3D masks. They are zero wherever the point (x,y) is dry, they are zero beneath the sea floor (which can happen because in NEMO the number of active levels can vary across grid-points), and they have the value 1.0 everywhere else. They are applied multiplicatively in the innermost loop of almost every computation. There is much redundancy in this representation, but it is certainly well suited to vector architectures.

### Decomposition, array index ordering and masks in POLCOMS

POLCOMS deals with the issue of computational load imbalance by decomposing, at run-time, the domain into rectangular sub-domains with approximately equal numbers of wet points. To do this, it uses a recursive k-section method [2]. We recently extended this method such that a number of alternate partitions are evaluated in parallel, and the best partition (according to a cost-function that takes into account various performance-affecting factors including the placement of MPI tasks on multi-core nodes) is selected [7]. We shall refer to this method as multi-core aware partitioning.

In contrast to NEMO, POLCOMS defines its 3D arrays with the z-index first. Thus, in POLCOMS it is columns, not layers, that are contiguous in memory. On scalar machines it is likely that several columns of water from different arrays can be accommodated in cache at the same time. Moreover, this layout is favourable for the 3D halo exchange operation, because packing and unpacking message buffers involves copying contiguous blocks of memory.

POLCOMS used 2D masks. They identify whether a grid-point (x,y) is wet or dry.

Loop nests in POLCOMS are written to match the array index ordering, so that the loop over levels will usually be the innermost. One important consequence of this is that it becomes feasible to write (inside nested loops over ji and jj):[1]

```
IF (tmask(ji,jj) > land) THEN
   DO jk=1, jpk
      …
```

Thus it is possible within POLCOMS to avoid redundant computation on dry points at the cost of a modest overhead of one mask look-up per grid-point. In contrast, the overhead of testing `tmask(ji,jj,jk)` at every point in a 3D array would be prohibitively expensive.

---

[1] For the sake of clarity, we are using the NEMO, not POLCOMS, naming conventions for array dimensions and loop counters.

| Feature | NEMO | POLCOMS |
|---|---|---|
| Representation of 3D fields | Fields stored as 3D arrays, with the vertical index last: a(x,y,z) | Fields stores as 3D arrays, with the vertical index first: a(z,x,y) |
| Decomposition | Decomposed in the x and y dimensions only into equal-sized sub-domains | Decomposed in the x and y dimensions only into sub-domains of unequal size but approximately equal computational load |
| Dry points | Redundant computation on dry points | No computation on dry points |
| Masks | 3D masks, with zeroes on dry points and beneath the sea floor, and ones everywhere else. Usually applied by multiplication. | 2D masks. Usually applied by testing. |
| Optimisation | Vector architectures | Scalar architectures |
| Ensemble capability | None | Yes |

**Table 1. Comparison of code features between NEMO and POLCOMS.**

## 3. Project structure and timeline

### Proposal and project plan

Our work in [7] showed that dramatic improvements to the performance and scalability of the halo exchange operation in POLCOMS can be achieved by multicore-aware partitioning and the elimination of dry points from halo messages. In this project, we aimed to apply the same techniques in NEMO. We knew, from our work in the EPSRC-funded project Towards Generic Scalability of the Unified Model (GSUM; see http://gsum.nerc.ac.uk/) that the partitioning and communications code of POLCOMS can be re-used in NEMO, after some not inconsiderable work [6].

Our proposal for dCSE funding was supported by the National Oceanography Centre (NOC), the UK Met Office, and the NEMO systems team. We were awarded funding for 12 person-months of effort. The project ran for 18 months, completing in June 2012.

Our project plan addressed three main aspects of NEMO code development, each with its own work package:

1. array index re-ordering, to change the layout of all arrays to have the level index first, while permuting the associated loop nests to match. We call the new ordering z-first, and the existing ordering z-last.
2. multicore-aware partitioning and halo exchange optimisation for large, multi-core systems, exemplified by HECToR.
3. provision of an ensemble capability.

The order of work packages was revised after discussions at the NEMO Systems Team Meeting in Southampton in October 2010, and subsequently confirmed with project partners and the dCSE programme management. The full benefits of (2) cannot be realized in the z-last ordering, so we scheduled it after (1). (3) was deemed to be of lower priority, and was therefore scheduled last. We estimated 5 person months (PM) effort for (1), 5 PM for (2) and 2 PM for (3).

At the same meeting, it was agreed that the work would start from a branch of the NEMO source code based on the 3.3 release, augmented with the inclusion of dynamic memory allocation. As we had previously developed a version of NEMO with dynamic memory allocation in the GSUM project [6], we agreed to take the lead in incorporating dynamic memory allocation into NEMO 3.3. The resulting version, NEMO 3.3.1, was made available in January 2011, and formed the basis of all new NEMO developments for 2011. We were given our own branch in the NEMO Subversion repository.

### Performance targets

Based on our project partners' experience with POLCOMS, OCCAM and NEMO, it was expected that (1) would bring improvements in cache re-use and consequent performance benefits on scalar architectures, and (2) should achieve more complete and flexible land-elimination than NEMO's existing method. We therefore set ourselves the target of 10% reduction in wall-clock time of a deep ocean (i.e. no land) test case, and a 40% reduction in wall-clock time of a test case with approximately 50% land (based in part on our knowledge that POLCOMS' partitioning typically achieves about 90% computational load balance). We thought these targets conservative.

### Consultation

We consulted regularly with project partners and external stakeholders. In particular, we:

- attended the NEMO Systems Team Meeting in Southampton, 2010, to present the project plan and invite input;
- attended the NEMO User Meetings in 2011 (Toulouse) and 2012 (Met Office, Exeter), and presented interim results at the latter;
- made regular visits to NOC (Liverpool);
- liaised with external projects and organisations with an interest in NEMO, including PRACE (1IP and 2IP), EC-EARTH/IS-ENES, CMCC (Lecce) and the NERC Ocean Roadmap Project;
- visited the NEMO core development team in Paris in December 2011.

By the time of our meeting with the NEMO core development team in December 2011, it was apparent that the scale of the work required to achieve the project goals was considerably greater than we had originally estimated. The NEMO team encouraged us to be pragmatic in the remainder of the project. Their view was that it was more important to know the answer to the question, "Are these optimisations right for NEMO?" than it was to do a complete conversion of the entire source tree including all of the possible configurations. It was agreed that it would be sufficient to cover only the NEMO core (~100k lines of OPA_SRC, plus some libraries), testing on the GYRE and AMM12 configurations (which have no sea-ice). We have therefore not tested a new version of NEMO's "north-fold" code, which is not exercised by our test configuration. We have also not spent any effort on attempting to update our development branch with new functionality from NEMO 3.4, except for critical bug fixes.

## 4. Methodology

### Preparatory work.

Some preparatory work was required to incorporate dynamic memory allocation into NEMO version 3.3. This work was carried out by Andrew Porter working in close collaboration with the NEMO core development team, and resulted in version 3.3.1, which formed the baseline for this work.

The NEMO repository automatically notifies developers of updates. In order to avoid spamming other developers, we tended to make infrequent commits to our branch, committing only self-consistent sets of changes.

### Array index re-ordering and loop-nest optimisations.

The task of re-ordering the indices of all the main 3D arrays in NEMO is both pervasive and invasive. It involves modifications to almost every file in the NEMO source tree, and the potential for introducing bugs is high. We decided to enlist the aid of Stephen Booth's nearly-forgotten Fortran pre-processor FTRANS [11], which allows the developer to mark up an array with FTRANS directives indicating how its indices are to be permuted in all subsequent references to that array, thereby reducing the potential for error.

We modified the NEMO build-system to include an extra pre-processing step for array-index re-ordering using FTRANS, installation of FTRANS on HECToR, and validation of FTRANS when applied to NEMO. We identified some issues with FTRANS, fixing some (with the aid of Stephen Booth) and devising work-arounds for others. Further details on FTRANS and how we applied it to NEMO are given in Appendix A.

Loop nests must be changed to achieve good cache re-use in the z-first ordering. This must be done manually. We introduced a new CPP key in order to be able to generate optimised versions for both orderings from the same source tree. In many cases (such as the example in Figure 1), the necessary transformation is obvious, and can be made at the same time as marking up the arrays.

```
#if defined key_z_first
DO jj = 1, jpj
   DO ji = 1, jpi
      DO jk = 1, jpk
#else
DO jk = 1, jpk
   DO jj = 1, jpj
      DO ji = 1, jpi
#endif
         a(ji,jj,jk) = …
```

**Figure 1. Loop nest optimisations must match index ordering for cache re-use.**

Occasionally, re-ordering a loop nest will change the order of summation and can affect the least significant bits in the results. We did not think it reasonable to pay the performance penalty that would be required to make the z-first formulation bit-reproducible with the z-last formulation.

NEMO's idiom is very "layer-oriented". Often, the use of Fortran array syntax disguises nested loops that are not cache friendly in the z-first ordering.

```
DO jk=1,kpk
   a(:,:,jk) = b(:,:,jk)*c(:,:,jk) …
```

These too must be transformed by hand.

NEMO's layer-oriented idiom encourages the developer to refer to values at the sea surface. For example, sometimes the sea-surface value of a 3D mask is applied inside a triply nested loop:

```
a(ji,jj,jk)=b(ji,jj,jk)*tmask(ji,jj,1)
```

This results in poor cache re-use after index permutation. In the case of masks (which are static), we were able to introduce copies defined only at the sea surface, and refer to these instead.

```
tmask1(:,:)=tmask(:,:,1)
```

Many loop nests are more complicated, and require more thought and testing to re-formulate efficiently in the z-first ordering.

Our goal was to preserve the format of input and output files, for compatibility with the rest of the NEMO tool chain. This constraint required us to take great care when dealing with 3D arrays near the I/O layer. We need to transform arrays from the z-first ordering to the z-last ordering before they are written to or read from disk. This is further complicated by two factors.

1. NEMO supports several different flavours of I/O, all of which must be treated and tested. One of the I/O libraries broke the PGI fortran compiler, which was being confused by variable names which clashed with the Fortran intrinsic `len`. We reported the compiler bug via the HECToR help desk. We worked around the compiler bug by choosing a different variable name, and reported this to the NEMO developers.
2. Sometimes the same I/O routine is used for 3D arrays in which the last dimension (on disk) is not the vertical dimension, and it is typically impossible to distinguish these cases inside the I/O layer. Consequently, we needed to examine carefully every call to the I/O routines.

The nature of this work meant that it was not practical to start testing until a complete pass of the hundreds of source files in OPA_SRC had been transformed to the z-first ordering. Such tasks can become tedious, and considerable discipline is required.

We chose the GYRE configuration, an idealized test case with no land, for this phase of the work. We explicitly tested some CPP keys (particular those that had flagged as being tricky or problematic during the initial transformation phase) additional to those activated by the GYRE test case, but we did not attempt an exhaustive coverage analysis of all combinations of CPP keys in NEMO.[2] As some of the changes to loop nest ordering alter the order of summation, we could not rely on simple bit-wise comparison between the z-first and z-last results. However, we did obtain equivalent results in both orderings and all flavours of I/O. The tool nccmp, which compares two NetCDF files with a user-defined tolerance, proved invaluable here.[3]

This phase of the work was carried out using the PGI compilers, then the default on HECToR.

Having achieved a working z-first version, and found it to be some 20% slower than the z-last version, we embarked on a phase of optimising the z-first version using the GYRE test case. We profiled the code using CrayPat on HECToR.

Our initial profiling showed up a load imbalance such that MPI processes other than rank 0 were spending a rather long time waiting for messages. We tracked the cause of the load imbalance down to a REWIND on the file "time.step" in the subroutine `stp_ctl` in file `stpctl.F90`, performed by the master process at every time step. In our test case, the REWIND was increasing the run-time by 10% (or more, on days when the Lustre file system was less responsive). It turns out that REWIND causes an expensive interaction with the Lustre metadata catalogue. The file "time.step" is monitored by some users to check on the progress of a NEMO run, but it is not required by the NEMO tool chain. We therefore disabled the REWIND in our development version. We reported the issue to the NEMO systems team, to NEMO users of HECToR at NOC and Plymouth Marine Laboratory. In order to preserve the usefulness of "time.step" for monitoring purposes, while reducing the overhead on systems like HECToR, one

[2] The complexity of a full coverage analysis grows exponentially with the number of CPP keys.
[3] nccmp is available from http://nccmp.sourceforge.net/

could do the REWIND every time step for the first 10 (say) time steps only, but less frequently from then on.

As is typical of most climate modelling codes, no single routine in NEMO's profile stands out as dominating the run-time. In NEMO, typically 10-20 routines each account for between 1 and 10% of the run-time. Focussing our attention on these routines, we were able to make further optimisations with the effect that most of the routines ran as fast, or slightly faster, in the z-first ordering. The halo exchanges went noticeably faster with very little work at all.

However, it turned out that one class of routines in particular proved problematic to make performant in the z-first ordering. The problematic routines all involve a tri-diagonal solve in the vertical dimension. Close inspection of these revealed that in the z-first ordering, loop nest permutation followed by some loop fusion not only greatly simplified the code (the z-first versions were typically 25% shorter than the originals) but also gave better cache utilisation (as measured by various CrayPat metrics). However, the z-first versions still ran significantly slower (by 30% or more), than the z-last versions. Why?

The clue came from an unexpected quarter. At the time, one of us was porting one NEMO routine of this class to GPUs for the GNEMO project [8]. He noticed that the Intel Fortran compiler was able to vectorise the z-last version but not the z-first version.

The explanation is now clear. These tri-diagonal solves involve three loops over levels; two loops traverse the levels in ascending order (from sea-surface to sea-floor) and one loop traverses the levels in the reverse order. In all three loops, each iteration depends on a previous iterate. It is this dependency that prevents the compiler from fully exploiting the instruction level parallelism (ILP) available in the processor cores. In contrast, in the z-last ordering, the loop over levels is outer-most, and the compiler can feed the processor cores with independent work from adjacent water columns. We tried trading off reduced cache re-use for increased vectorisation, but the results were not encouraging. We have also made some initial experiments with the transformation known as unroll-and-jam (sometimes called outer loop unrolling) on a simplified kernel. This transformation is more promising (especially on the loop that traverses the levels in the reverse order), and should narrow, but not eliminate, the performance gap between the z-first and z-last versions on routines involving a tri-diagonal solve in the vertical. However, although many compilers have directives that enable unroll-and-jam, it seems that the pattern of the loop body needs to be "just so", and there may be a good deal of "black art" in persuading compilers to make this transformation.

At the end of this phase, the z-first version was achieving approximately (i.e. +/- 3%) the same single-core performance as the z-last version on the GYRE test case, with slightly improved scalability (due to the more efficient halo exchanges, which is more cache friendly in the z-first version than the z-last version). This is less than our target of a 10% reduction in the run-time on a deep ocean test case.

We attribute the shortfall to the unexpected difficulty of getting the tri-diagonal solves to vectorise in the z-first ordering. Before we started this work, we had hoped that operations up and down a water column would be advantageous for the z-first ordering. Indeed, when POLCOMS was translated from z-first to z-last ordering during the 1990s, the cache-based commodity scalar processors of the then emerging MPPs typically had little ILP (apart from a fused multiply-add) to exploit. Today, commodity processors typically offer some degree of ILP (e.g. SSE, AVX, VLIW); although there is no longer a compelling need to maximise the vector length, it still pays to exploit the hardware's available parallelism.

### Multi-core aware partitioning and halo exchange performance optimisation

In our previous work during the GSUM project [6], we adapted NEMO to use POLCOMS' domain-decomposition methods. This work was done on version 2.3 of NEMO, but never incorporated into the NEMO trunk. We therefore needed to re-apply these modifications to version 3.3.1. This proved straightforward for the GYRE configuration. The only changes required to the existing NEMO source were to the routines called by *nemo_init()* to do the domain decomposition and to the interfaces to the halo exchange routines to call the new routines instead.

Although the GYRE configuration was appropriate for the array-index re-ordering work, we now needed a test case with a significant fraction of land. We originally planned to obtain a suitable coastal configuration from Jason Holt at NOC (Liverpool). However, when we began this phase of the work, the UK Met Office had already contributed a suitable configuration to the NEMO system for inclusion in version 3.4. This configuration, AMM12 (see Figure 2), has the advantages that it is part of the standard release and is supported by the UK Met Office.

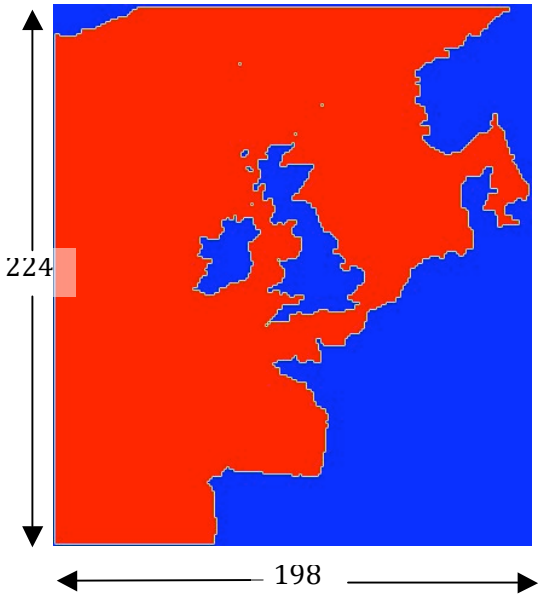


**Figure 2: The AMM12 configuration with grid dimensions.**

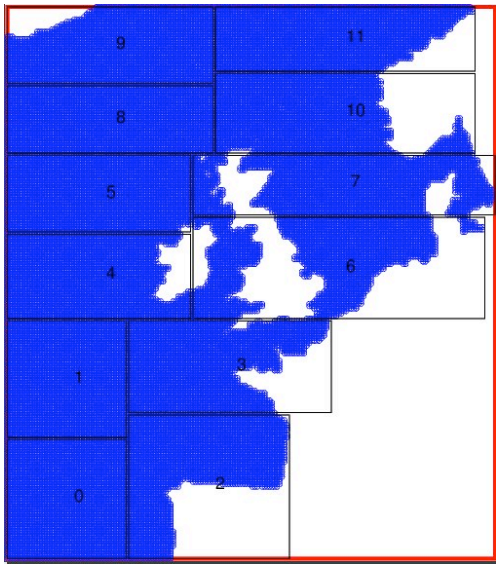


**Figure 3: decomposition of the AMM12 domain over 12 MPI processes. Land-only rows and columns have been trimmed. The numbers give the ID of the MPI process assigned to each sub-domain.**

Although AMM12 was only officially released in version 3.4 of NEMO, its initial merge into the NEMO repository was based on version 3.3.1. One of the greatest difficulties we had in getting a working version of AMM12 was arriving at a consistent set of input data files and namelist file. This is because input data files, unlike the source code and namelist files, are not under revision control, presumably because they are large, binary (NetCDF) files. The publicly available input data set was for version 3.4 of NEMO and some file formatting and naming conventions had changed since version 3.3.1. It took several iterations before Enda O'Dea at the UK Met Office was able to produce a data-set compatible with version 3.3.1.

The AMM12 configuration involves quite a different set of CPP keys from either the GYRE or ORCA2_LIM configurations upon which the code had been tested so far. In particular, it has no cyclic boundary conditions and uses the BDY module to implement

the forcing of fields along the domain boundaries. Therefore getting a correctly working version required some debugging of both the GSUM and z-first aspects of the code.

Finally and most difficult of all, there turned out to be a bug in the bathymetry-smoothing routine, `zgr_sco()`. This bug only manifested when NEMO was run on particular processor counts. The symptom was de-normalized values produced in the `zdf_gls()` routine. We traced this problem back to differences in the smoothed bathymetry produced when NEMO was run on one of these problematic processor counts. These differences occurred whenever a region of bathymetry to be smoothed happened to fall across a domain boundary. In this case, the way in which the smoothing algorithm handled halo exchanges was flawed, causing the bathymetry to remain frozen at domain boundaries. Consequently, the final, 'smoothed' bathymetry differed from that used in the original NEMO run which generated the input data sets, giving rise to unphysical behaviour. We logged the bug in the NEMO issues tracker and reported a fix to the UK Met Office.

In addition to this bug, it became evident that the smoothing algorithm could cause previously wet coastal points to become dry and vice versa. It is probable that this too is a bug. Since our domain decomposition code trims dry columns and rows it is essential that the wet/dry mask that it uses is identical to the one used during the model run. In NEMO the bathymetry-smoothing step is performed once the bathymetry has been distributed over processes. Therefore, to ensure that the domain decomposition was performed using the same mask as the model run, we implemented the workaround of applying the smoothing algorithm to the complete domain bathymetry read from disk, immediately prior to computing the domain decomposition.

Figure 3 shows an example decomposition of the AMM12 domain over 12 MPI processes. As well as sharing the number of ocean points equally between all sub-domains, the algorithm has trimmed any land-only rows and columns from their edges.

Once the AMM12 configuration was working with the GSUM modifications, the next stage was to incorporate the multi-core aware (MCA) partitioning code from POLCOMS. This required only small changes to the initialisation routines. Slightly more work was required to extend the communications routines to eliminate dry points from halo messages. An example of the performance gains made possible by these changes is shown in Figure 4.

The results in Figure 4 were obtained by running each of the possible domain decompositions for a job with 192 MPI processes (as enumerated by the MCA code). Over these 13 different decompositions, the performance of the halo-exchanges varied by almost a factor of two; from 5589 up to 11256 exchanges per second. To get the best performance from the code it is therefore important that the best of the 13 decompositions is chosen. We can investigate the skill of the MCA cost function in predicting the best decomposition by using the score it assigned to each to order them along the *x*-axis in Figure 4.

Even though the absolute performance of the code (as measured by time-steps per second) is limited by load balance, it generally tracks the trend in halo-exchange performance. Since the MCA cost function was originally tuned for POLCOMS we had to modify it slightly to get reliable predictions of the best partition. This is because POLCOMS avoids calculations on land while NEMO currently does not. To account for this, the weight given to the computational cost of land points in a sub-domain was increased from 0.02 to 0.9.

We note that we had a lot of difficulty developing NEMO on phase III of HECToR due to bugs in the Totalview debugger. Line-by-line 'stepping' of the code would result in 'illegal instruction' errors and therefore the only way to move the code forwards was to

set a breakpoint and then continue execution. Unfortunately we were unable to reproduce this bug in a simple test case and therefore the whole NEMO code and configuration had to be supplied to the helpdesk. A bug report was eventually submitted to the Totalview developers. In addition, we frequently found that attempting to plot a 2D field would crash Totalview completely and that Totalview was often unable to display a complete stack trace for NEMO.

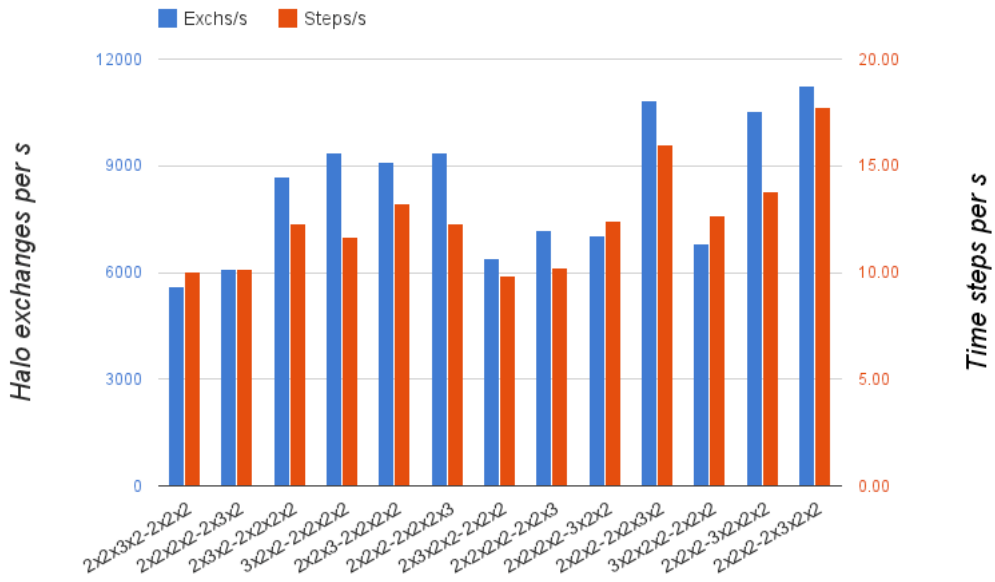


**Figure 4. Performance of the AMM12 configuration run with the possible domain decompositions on 192 MPI processes. The decompositions are ordered along the *x*-axis in order of increasing *predicted* performance (from the MCA scoring function).**

## 5. Results.

As discussed earlier in section 2, the main motivation for re-ordering NEMO's array indices was to allow the elimination of redundant calculations over land. Unfortunately, the amount of effort required to re-order the array references and loop nests was much greater than expected, leaving insufficient time to modify the NEMO source to implement loop-level avoidance of calculations on the dry (land) points that remain after partitioning into sub-domains.

Our recursive k-section partitioning algorithm assumes that dry points do not have any computational cost and therefore counts only wet points. Since this is not yet true in our version of NEMO, a load imbalance arises, adversely affecting the performance results reported here. As an example, the decompositions obtained for the AMM12 domain on 128 processes by the recursive k-section and the default NEMO algorithms are shown in Figure 5.

Trimming off the land-only areas in the decomposition in Figure 5(a) has resulted in sub-domains that are generally smaller than those in the regular example in Figure 5(b). However, there are some sub-domains (such as numbers 57 and 122) that are much larger than the average (e.g. 74) because they contain land. Until calculations on dry points are fully eliminated, these larger sub-domains will be the rate-limiters for the computational aspects of the code. Nonetheless, the combination of smaller cells with the removal of land points from halos means that the halo swaps should perform better for the recursive k-section version of NEMO, which we call RK-NEMO.

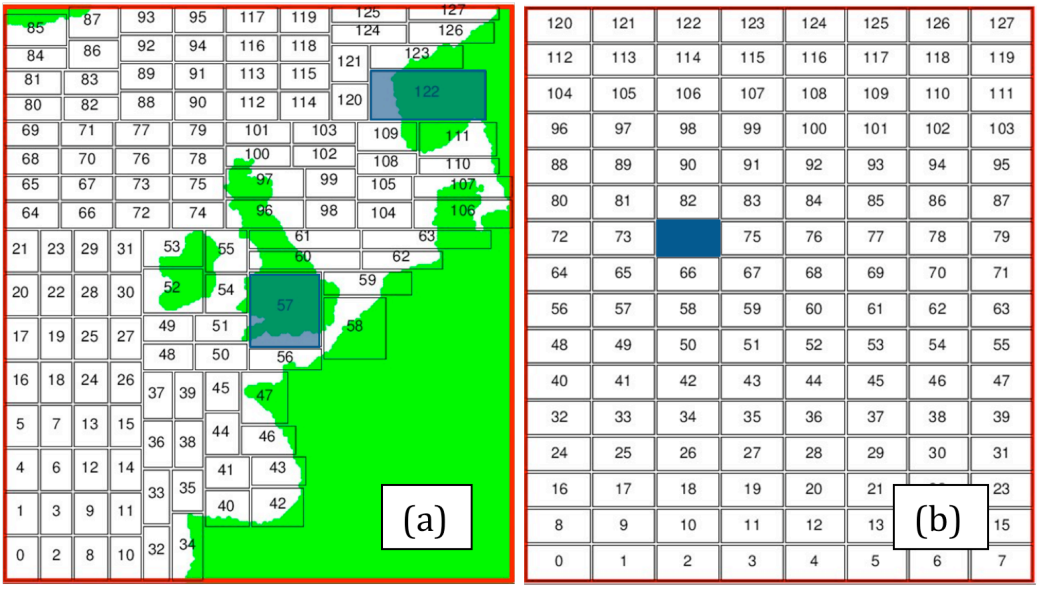


**Figure 5. Comparison of the domain decompositions produced by the recursive k-section algorithm (a) and the original NEMO code (b) for 128 processes. Sub-domains highlighted in blue illustrate the load-imbalance problem in (a).**

In Figure 6 we compare the performance of the new version of NEMO with that of the original for the AMM12 configuration. On up to 128 MPI processes (one core per process), RK-NEMO is typically slightly faster than the original, irrespective of the ordering of array indices. Beyond this, the load imbalance discussed above makes RK-NEMO uncompetitive. The performance of RK-NEMO relative to the original is plotted more explicitly in Figure 7. It is surprising that RK-NEMO is faster than the original on 128 processes despite the load imbalance. We attribute this improvement to the reduction in time spent in halo-swaps.

The strong scaling performance of a code like NEMO is very sensitive to the performance of the halo exchange operation. Our MCA optimisations strive to improve this by minimising off-node communications. We have also minimised the quantity of data that must be exchanged by excluding dry points from halo regions.

However, a halo exchange also involves packing data from the halo region into a contiguous buffer that can be passed to the MPI library. The performance of this packing (and corresponding unpacking) task is crucial to the performance of the halo exchange. In NEMO the vast majority of halo exchanges involve three-dimensional, double-precision arrays. The halos for these are 2D slabs in the *x*-*z* or *y*-*z* planes meaning that the packing and unpacking process is always dealing with full columns in the z direction. Packing/unpacking is therefore much more efficient in the z-first ordering and the data in these columns is contiguous in memory. This is demonstrated by the results in Figure 8 where the halo exchanges in the *z*-first case always outperform, sometimes significantly, both those in the *z*-last case and those in NEMO in its original form.

Having the *z* index first means that all triply nested loops over the *x, y* and *z* dimensions will have the *z* loop innermost. It is this innermost loop that an optimising compiler will normally choose to vectorise. Since the 2D domain decomposition leaves the *z* dimension unchanged, the trip count of the vectorised inner loop is independent of the number of MPI processes. In contrast, with the *z* index last, it is the *x* loop that is vectorised; as the number of MPI processes is increased, the extent of the innermost loop shrinks and computational efficiency drops off.

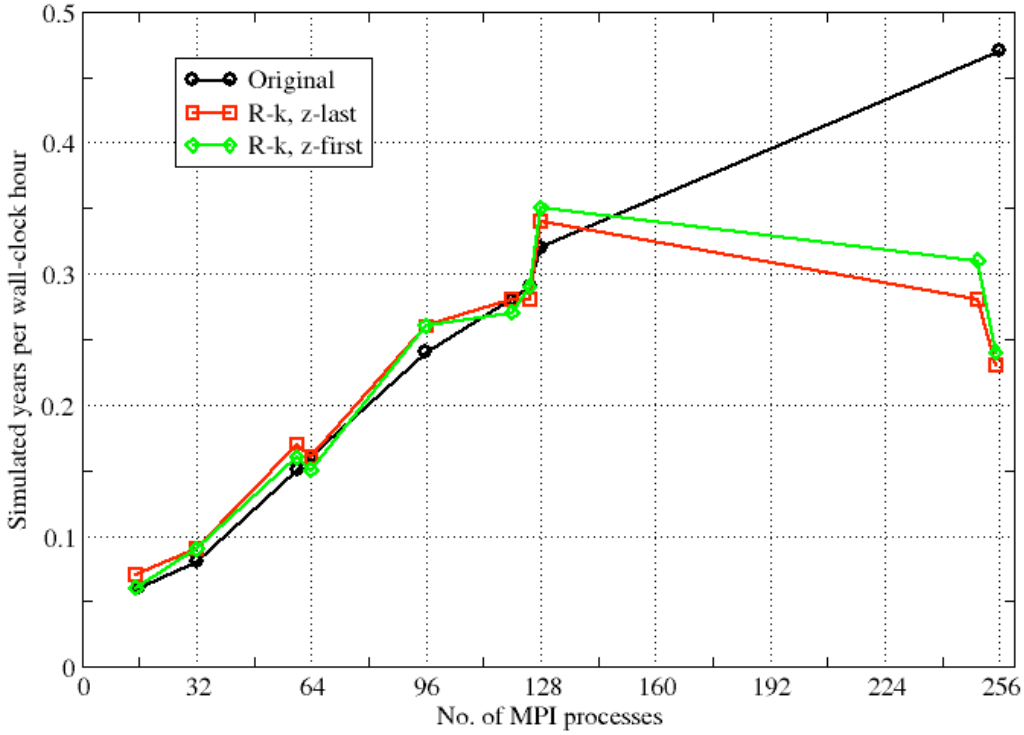


**Figure 6. Performance comparison of the original NEMO with the version using recursive k-section partitioning. Two sets of results for this version are shown: one where the z (level) array indices are last (as in the original code) and one where they are first. The results are for the AMM12 configuration on Phase III of Hector.**

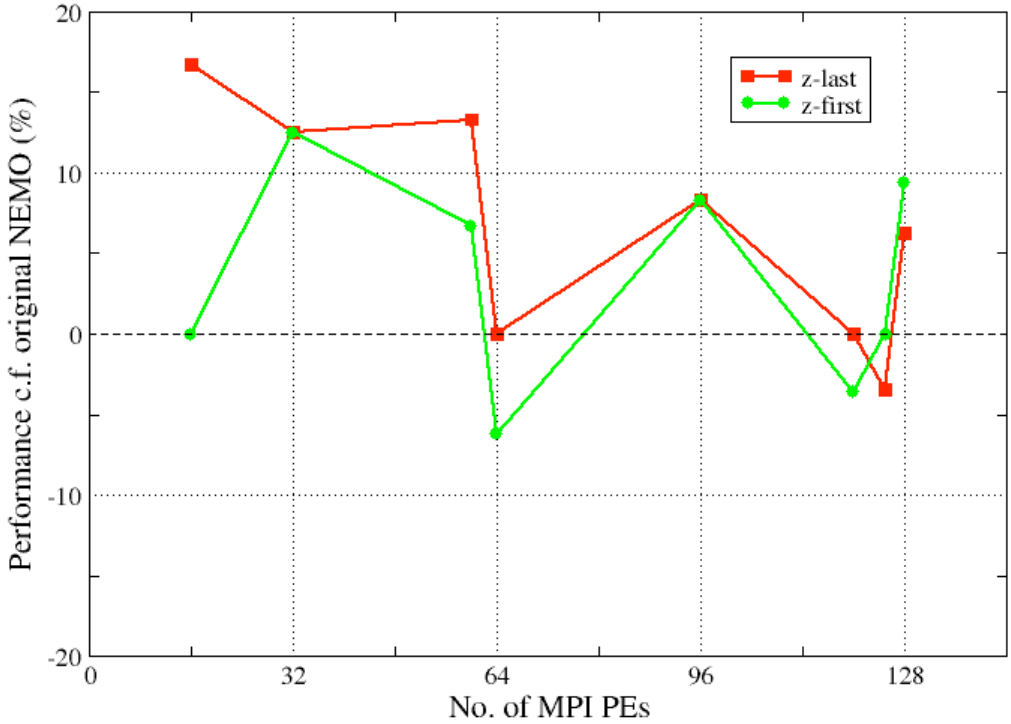


**Figure 7: Performance of RK-NEMO relative to the original. Details are the same as in Figure 6 above.**

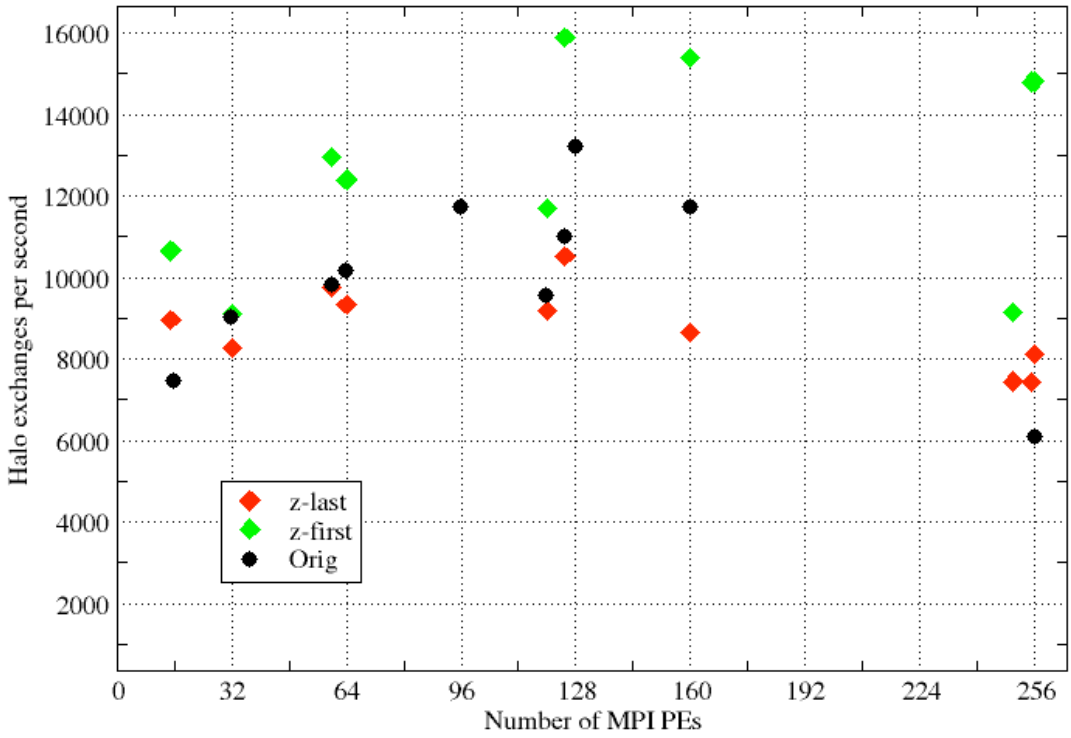


**Figure 8: Comparison of halo exchange performance for a real 3D array in NEMO and in the *z*-first and *z*-last forms of RK-NEMO. Results are for the AMM12 configuration on Hector.**

## 6. Conclusions

### Impact

The most immediate impact of this work is seen in:

1. Introduction of dynamic memory allocation into the NEMO trunk;
2. Identification of various bugs (including some fixes) in NEMO, FTRANS, Totalview, and the PGI Fortran compiler;
3. Identification of the performance overhead of REWIND on HECToR, which triggers an expensive interaction with the Lustre file system. Our solution was immediately transferable to production NEMO work on HECToR.

We have also provided a branch of NEMO that, while falling short of our original goal of delivering improved performance in production runs on HECToR, can nonetheless inform strategic decisions about future NEMO developments.

### Reflections

In hindsight, it is clear that we grossly under-estimated the effort required to complete our proposed programme of work. With 12 person-months of effort, we achieved: a working index-reordered version of the NEMO core, including associated loop nest optimisations, and integration of multi-core aware partitioning and associated communications optimisations into the NEMO code base. We ran out of time to make another pass over the code to eliminate, at loop level, the redundant computation on dry points that remain after partitioning; but this would be essential if the full benefits of our strategy are to be realised. We also had to drop the ensemble capability work.

We were surprised at how invasive and pervasive the changes necessary to recover performance in the z-first ordering proved to be. Some operations (e.g. 3D halo exchanges) do indeed go noticeably faster in the new ordering. Many operations go at

similar speed in either ordering, or only marginally faster in the z-first ordering, even on scalar cache-based processors. The class of routines that involve a tri-diagonal solve in the vertical dimension proved particularly problematic in the z-first ordering. There remains some scope for further optimisation of these routines in the z-first ordering, but it is clear that the z-last ordering will remain favourable for these operations.

We think that our original performance targets of a 10% reduction in run-time on a deep ocean test case, and a 40% (or better) reduction in run-time on a coastal test case (50% land) are still achievable. Achieving the former would depend either on finding an effective recipe for the tri-diagonal solves, and/or finding other opportunities for optimisation in the z-first ordering (this is not unlikely – NEMO's layer-oriented idiom often disguises the potential for simplification in the z-first ordering). Achieving the latter would depend on a fairly complete implementation of loop-level avoidance of dry points. There are two candidate approaches: (a) explicitly test a point to see if it is wet (following the style of POLCOMS), or (b) look up the loop bounds at each point. Of these (b) has the advantage of making it possible also to eliminate redundant computation beneath the sea floor in shallow regions; to exploit this fully, further modifications to the partitioning logic would be required. The penalties of both approaches should be offset at least partially by the obviation of the need to load and multiply by a 3D mask.

Although we are currently able to produce z-first and z-last orderings of NEMO from the same source tree, we do not think this is a viable long-term option. The patterns of loop nests required for performance in the two orderings are quite different. Similarly the idiom of loop-level avoidance of dry points in the z-first ordering point is very different to the idiom of multiplicative application of a mask in the current NEMO. There would be a considerable overhead in having to maintain and test both versions, and it does not seem reasonable to expect developers to do so.

We do not as yet have any evidence either way to say which of the z-first or z-last orderings is more favourable for an MPI+OpenMP hybrid version of NEMO. We hope to gain some insight into this question from our work in PRACE-2IP.

### Recommendations

Our revised estimate of the work required to implement our strategy on the whole of NEMO would be of the order of 3 FTE-years (including testing). This is a lot to ask, given that, on the evidence that we can provide today, the case for doing so is not compelling. We would not recommend embarking on such an extensive programme of work without first taking steps to confirm the likely benefits of dry-point elimination, and comparing this with the current method of eliminating all-land cells by pre-processing. A reasonable estimate could probably be obtained without a complete implementation of loop-level dry-point elimination, by applying it in (say) 10-20 of the more expensive routines, and extrapolating with the aid of a carefully contrived performance model.

If the NEMO developer's committee decide to adopt the z-first ordering, we would advise an approach along the following lines. In order not to delay concurrent developments too long (because the changes are truly pervasive), we would recommend first agreeing all the recipes and style rules first (learning from our experiences), then have a co-ordinated team of say 6 programmers working together for 6 months in a "freeze and sprint" strategy.

If on the other hand the NEMO developer's committee decide against switching to the z-first ordering, there are still things that we could do to reduce the amount of redundant computation on land in the z-last ordering, at run-time, instead of by pre-processing. As a result of this work, we already have, in a branch of NEMO, all the machinery for halo-exchanges on partitions of variably sized rectangular sub-domains. As it stands, the multi-core aware recursive k-section partitioning from POLCOMS assumes that the

computational cost of any sub-domain goes as the number of wet points. As observed above, this does not hold without loop-level avoidance of dry points, which is not viable in the z-last ordering. We believe that a POLCOMS-like partitioning algorithm can be developed that is better suited to the z-last ordering. An investigation along these lines could be completed with modest effort, and offers the potential of more flexible and more effective dry-point elimination than NEMO's current pre-processing method.

## Acknowledgments

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR – A Research Councils UK High End Computing Service – is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd.

We are grateful to numerous people for assistance, advice, encouragement and stimulating discussions. We thank in particular Rachid Benshila, Sebastien Masson, Claire Levy, and Gurvan Madec of the NEMO team; John Siddorn, Dave Storkey, and Enda O'Dea of the Met Office; Jason Holt, Andew Coward and Hedong Liu of the National Oceanography Centre; Italo Epicoco, Sylvia Mocavero and Giovanni Aloisio of CMCC; Mike Ashworth of STFC; and Stephen Booth of EPCC.

## References

[1] G. Madec, "NEMO ocean engine", 2008, *Note du Pole de modélisation*, Institut Pierre-Simon Laplace (IPSL), France, No 27 ISSN No 1288-1619.

[2] M. Ashworth, J.T. Holt and R. Proctor, "Optimization of the POLCOMS hydrodynamic code for terascale high-performance computers", *Proceedings of the 18th International Parallel & Distributed Processing Symposium*, Sante Fe, New Mexico, 26-30 Apr 2004.

[3] Jason Holt, James Harle, Roger Proctor, Sylvain Michel, Mike Ashworth, Crispian Batstone, Icarus Allen, Robert Holmes, Tim Smyth, Keith Haines, Dan Bretherton, Gregory Smith, "Modelling the Global Coastal-Ocean", *Phil. Trans. R. Soc. A*, March 13, 2009 367:939-951; doi:10.1098/rsta.2008.0210

[4] Fiona Reid, *NEMO on HECToR - A dCSE Project*, 4th May 2009, HECToR web site, http://www.hector.ac.uk/cse/distributedcse/reports/nemo/nemo_final_report.pdf.

[5] Fiona J. L. Reid, "The NEMO Ocean Modelling Code: A Case Study", *Proceedings of the Cray User Group*, Edinburgh, 2010.

[6] M. Ashworth and A. Porter, "Towards Generic Scaling of NEMO, the oceanographic component of the Unified Model", Final Report for the GSUM project, 2009.

[7] Stephen Pickles, "Multi-Core Aware Performance Optimization of Halo Exchanges in Ocean Simulations", *Proceedings of the Cray User Group*, Edinburgh, 2010.

[8] A. R. Porter, S. M. Pickles, M. Ashworth, "Final report for the gNEMO project: porting the oceanographic model NEMO to run on many-core devices", *Daresbury Laboratory Technical Reports*, DL-TR-2012-001 (2012).

[9] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications", *Scientific Programming*, Vol. 9, No 2-3, 2001, 83-98.

[10] Italo Epicoco, Silvia Mocavero, and Giovanni Aloisio, "NEMO-Med: Optimization and Improvement of Scalability", CMCC Research Paper No. 96, 2011.

[11] Stephen Booth, "FTRANS User Guide", 1996.

## Appendix A. On the use of FTRANS in NEMO.

FTRANS is a Fortran pre-processor, written by Stephen Booth of EPCC in the mid 1990s [11]. It supports (1) global replacement of identifier names, and (2) array index re-ordering. We only use the array index-reordering feature in this work.

FTRANS was written at a time when there was something of an industry in migrating Fortran codes from vector machines to Massively Parallel Processors built from commodity scalar processors, which were then relatively new, and when Fortran90 was in its infancy.

### Usage

FTRANS allows a developer to mark up a Fortran source file with directives indicating which arrays have indices that can be permuted by the FTRANS pre-processor. FTRANS will then ensure that all indices are permuted consistently in all references to these arrays. Clearly, this has the virtue of saving the developer considerable effort, and eliminating some classes of human error.

```
!FTRANS a :I :I :z
!FTRANS b :I :I :z :
REAL(wp), ALLOCATABLE :: a(:,:,:), b(:,:,:,:)
ALLOCATE(a(jpi,jpj,jpk), b(jpi,jpj,jpk,2))
DO jk = 1, jpk
   DO jj = 1, jpj
      DO ji = 1, jpi
         a(ji,jj,jk) = b(ji,jj,jk,1)*c(ji,jj) + …
```

**Figure 9. Fortran source fragment before FTRANS pre-processing**

For example, pre-processing the Fortran fragment of Figure 9 with the command line:

```
cpp <args> | ftrans -f -9 -I -w -s :z :I -
```

yields the result illustrated in Figure 10. The important flag here is "`-s :z :I`" which instructs FTRANS to permute indices so that all indices labelled `z` precede all interchangeable indices identified by the label `I`. The meaning of the other flags is as follows: `-f` means expect free format Fortran; `-9` means expect Fortran 90; `-I` means follow Fortran includes; `-w` means expect wide lines (up to 132 characters); and the trailing hyphen means send the output to stdout.

```
!FTRANS a :z :I :I
!FTRANS b :z :I :I :
REAL(wp), ALLOCATABLE :: a(:,:,:), b(:,:,:,:)
ALLOCATE(a(jpk,jpi,jpj), b(jpk,jpi,jpj,2))
DO jk = 1, jpk
   DO jj = 1, jpj
      DO ji = 1, jpi
         a(jk,ji,jj) = b(jk,ji,jj,1)*c(ji,jj) + …
```

**Figure 10. As above, but after FTRANS pre-processing.**

The NEMO build system already has a pre-processing phase, which involves a script that runs an AGRIF pre-processing step followed by the C pre-processor. We were able to modify this script to insert the FTRANS pre-processing step after the C pre-processor.

### Advantages and disadvantages

FTRANS makes it possible for us to generate z-first and z-last versions of NEMO from the same source tree. This means that we can readily switch between the two versions

during development, which proved useful for both debugging and performance comparison purposes.

We were also able to merge minor changes and bug fixes from the trunk to our development branch without much difficulty, and we did this several times during the early stages of the project. However, as we added to the source more and more alternative blocks of code, optimised for the z-first ordering, we could no longer rely on an "svn merge" to do the right thing. This is because the probability of an amendment being applied to only one of the alternative code blocks increases.

This is why FTRANS rarely becomes a permanent fixture of any code. Eventually, the cost of maintaining and testing what is essentially two versions of every source file grows to the point where it becomes preferable to decide on a particular ordering and stick with it. Fortunately, FTRANS itself can help with the final simplification, but this is much easier when it's possible to run FTRANS *before* any other pre-processor.

Using FTRANS does slightly increase the complexity of the build process, which now depends on it, and in turn on lex and yacc (or flex and bison).

### Deficiencies and workarounds

When FTRANS was written, Fortran90 was still in its infancy. By the time that the use of Fortran90 was widespread, many legacy codes had already made the transition from vector to scalar architectures, and FTRANS fell into disuse. It is therefore understandable that FTRANS has not kept up with developments in the Fortran language.

We encountered several issues when applying FTRANS to the task of re-ordering array indices in NEMO. We reported a few bugs (and a couple of fixes) to Stephen Booth, who fixed them promptly in new versions of FTRANS.[4] Version 3.4, as used in this work, still harbours a few limitations or deficiencies. In particular:

**1. Long lines.** There is an upper limit of 132 characters on the line length in the Fortran standard. Most compilers (including PGI and Cray Fortran) do not enforce this limit and will happily compile Fortran sources with longer lines than this. FTRANS, however, will produce a fatal error. In NEMO, there are some lines that expand, after pre-processing with the C pre-processor, to longer than 132 characters.

2. FTRANS does not permute indices inside **DIMENSION** attributes of an array declaration, which causes a problem with lines such as:

```
REAL(wp), DIMENSION(jpi,jpj,jpk) :: a, b
```

Our workaround is to transform this to:

```
!FTRANS a b :I :I :z
REAL(wp) :: a(jpi,jpj,jpk), b(jpi,jpj,jpk)
```

Unfortunately, this is in breach of the NEMO style guide. We did it anyway.

3. Symbols declared in **modules** are not visible to FTRANS when processing the source files that use them. We worked around this deficiency by introducing, for every module <modulename>.F90 that has public arrays with permuted indices, a corresponding header file <modulename>_ftrans.h90 containing FTRANS directives, and including it (using #include directives) wherever the module is used. Unfortunately, this forces us to run FTRANS after the C pre-processor.

---

[4] One of these bugs was the handling of lines with trailing comments following a continuation symbol, a common practice in NEMO.

4. Fortran's **scoping rules** for variable declarations are different to those of FTRANS. An FTRANS symbol stays in scope until the end of the source file. Care must be taken when a symbol in one subprogram needs its indices reordered, and the same symbol in a subsequent subprogram in the same source file should be left alone (something that often happens when a subroutine is overloaded with multiple interfaces for 2- and 3-dimensional array arguments). To work around this, we sometimes needed to force FTRANS to clear its symbol table by using the

```
!FTRANS CLEAR
```

directive at the end of a subroutine (which incidentally causes FTRANS to generate warnings about any unreferenced variables), then re-include all the <module_name>_ftrans.h90 header files.[5]

5. Under certain conditions, FTRANS will discard a trailing comment. This is of little consequence when FTRANS pre-processing is part of the build system, but it is a nuisance when you want to use FTRANS to produce a new source tree from the old.

One must also be wary of **renamed module variables**. For example:

```
USE <module>, ONLY :: zwrk => ua
```

Here, if `ua` is index-reordered, an additional FTRANS directive will be needed for `zwrk`. Similar issues apply to Fortran **pointers**.

There are some other traps for the unwary. Anything that deals explicitly with or makes assumptions about the shape of an array may need special attention; the RESHAPE intrinsic is a case in point.

Some innocuous-looking operations may become erroneous after pre-processing with FTRANS. Consider:

```
a(:,:,1:jpl)=b(:,:,1:jpl)
```

Are the two expressions conformable if one array has its indices permuted but the other does not?

NEMO sometimes uses pointers to array sections as workspace, which can go wrong if the pointer's target has its array indices permuted. For example:

```
REAL(wp), POINTER, DIMENSION(:,:) ::    z_qlw
z_qlw  => wrk_3d_4(:,:,1)
```

Sometimes in NEMO, a subprogram declares arrays with dimensions that are passed by argument:

```
INTEGER,  INTENT(IN)     :: kpi, kpj, kpk
REAL(wp), INTENT(INOUT)  :: a(kpi,kpj,kpk)
```

It is sometimes impossible to tell from inspection of the body of the subprogram whether the indices of the array argument should be permuted or not. Indeed there are cases in NEMO where a subprogram is sometimes called with typical 3D arrays (the third index is the vertical level, and the array indices should be permuted), and is sometimes called with a different bound on the third dimension (e.g. the third index identifies a 2D tracer, and the array indices are not permuted). We generally inserted defensive assertions in such routines, and had sometimes also to examine every call to the routine in question.

[5] We sometimes wished for the ability to CLEAR a list of symbols from the FTRANS symbol table.