Multi-Core Aware Performance Optimization of Halo Exchanges in Ocean Simulations

Stephen Pickles, *STFC Daresbury Laboratory*

ABSTRACT: *The advent of multi-core brings new opportunities for performance optimization in MPI codes. For example, the cost of performing a halo exchange in a finite-difference simulation can be reduced by choosing a partition into sub-domains that takes advantage of the faster shared-memory mechanisms available for communication between MPI tasks on the same node. I have implemented these ideas in the Proudman Oceanographic Laboratory Coastal-Ocean Modelling System, and find that multi-core aware optimizations can offer significant performance benefit, especially on systems built from hex-core chips. I also review several multi-core agnostic techniques for improving halo exchange performance.*

KEYWORDS: multi-core, halo exchange, performance optimization, scalability, domain decomposition, MPI, combinatorics

1. Introduction

The performance of halo exchange operations limits the scalability of a large class of domain-decomposed scientific codes. Strong scaling, that is to say linear scaling of performance with the number of processor cores on the same dataset, is fundamentally impossible. Nonetheless, optimizing the performance of halo exchanges is a worthy goal, and, I submit, the opportunities for doing so have not been exhausted.

The advent of multi-core brings new challenges. But it also brings new opportunities, arising out of the availability of faster shared-memory mechanisms for communications between processor cores on the same node. Mixed-mode codes, e.g. OpenMPI and MPI hybrids, exploit intra-node shared memory by design. But the effort required to undertake the wholesale conversion of a large, legacy MPI code to a mixed-mode paradigm is often preclusive.

In this paper, I describe my recent efforts to optimize the performance of the halo exchange in the context of one particular ocean modelling code, POLCOMS. Section 2 describes some aspects of the code to help my readers form their own conclusions about the wider applicability of the optimizations in this paper. Section 3 presents various optimizations, including message combination, and the elimination of redundant points.

In section 4, I make contact with multi-core considerations, and show how untapped freedom in the POLCOMS domain decomposition method can be exploited to improve communications locality and hence halo exchange performance. The recent incorporation of 6-core chips into the latest Cray XT systems has significant consequences for the pertinence of these ideas. Any batch job that uses a whole number of 6-, 12- or 24-core nodes will have at least one factor of 3 in its processor grid. We will see how this causes the number of ways of decomposing a domain to explode, and how exploiting this freedom can yield surprising performance benefits.

I used two Cray XT systems to collect the performance results presented here. These are HECToR, the UK's national HPC service, which at time of writing (HECToR phase 2A) was equipped with quad-core Barcelona nodes, and Rosa, at the Swiss National Supercomputing Service (CSCS), which had two 6-core Istanbul chips per node.

2. POLCOMS

The Proudman Oceanographic Laboratory Coastal Ocean Modelling System (POLCOMS) [1] is a finite-difference parallel Fortran code for modelling coastal and shelf seas, on a regular latitude/longitude grid. The domain is decomposed geographically in two dimensions (longitude and latitude), and each sub-domain is assigned to one MPI process. As with many codes of its class, the domain is not decomposed in the vertical dimension. The model deals with both 2-dimensional and 3-dimensional fields, represented as arrays, the haloes of which must be exchanged between neighbouring domains periodically. Most of these arrays have a halo width of one grid-point, although larger haloes are supported.

These properties are typical of many well known environmental codes. Less typical are the ways that POLCOMS (a) declares its arrays, (b) handles 'dry' (land) points, and (c) performs the de-composition into sub-domains. I describe these features here, for they have some impact on the applicability to other ocean codes of the optimizations described in this paper.

In POLCOMS, 2-dimensional arrays are declared with the *x* (longitude) dimension first and the *y* (latitude) dimension second. Three-dimensional arrays are declared with the vertical (level) dimension first, followed by *x* and *y*. In this respect POLCOMS differs from NEMO, which has the level index varying slowest. In this paper, I do not discuss the effect that the choice of array layout has on single-core performance and affinity for scalar and vector architectures; however, I will discuss the consequences of this choice as far as it affects possible optimizations of the performance of the halo exchange operation.

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 flooding. Any computation performed at a point that is *permanently* dry is redundant.[1] There are two obvious approaches to deal with this redundancy: either perform the redundant computations and discard the results, or take steps to avoid performing 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, introducing a set of mask arrays (one for each grid type), and explicitly testing the mask at each grid point before performing computations there. Of course, the act of testing the mask introduces a certain overhead, but this overhead is reduced by POLCOMS' choice of array layout; with the level index running fastest, only one mask look-up is required per grid-point.

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], about which I will have more to say in section 4 below.

[1] Henceforth, I shall use the term *dry* to mean *permanently dry*.

3. Halo exchange optimization

In this section, I describe various attempts to improve the performance of the halo exchange in POLCOMS.

Existing API

The existing API of POLCOMS separates a halo exchange operation into two parts. A call to *exchs()* starts a halo exchange, and a call to *exchr()* completes it. A single array is passed as an argument; this may be 2-, 3- or 4-dimensional, integer or 64-bit real. Communication may be overlapped with computation by interleaving computation, which must be independent of the array being exchanged, between the calls to *exchs()* and *exchr()*. The API is designed to allow multiple halo exchanges to be in flight at any one time. The application code makes use of both of these features. The optimizations described here are designed to preserve them.

The behaviour of the halo exchange operation is governed by internal tables that list the bounds of the local array patches that each process sends to and receives from its peers. There are different sets of tables for each supported halo width (currently one and two grid points). The patches involve both wet and dry points. Most of the halo exchanges in the application code are for a halo width of one grid point.

The implementation of the existing API uses MPI derived datatypes, two-sided communications, non-blocking, standard sends and non-blocking receives. These choices give reasonably portable performance, but alternative modes are available as compile time options.

Message combination

The idea behind message combination is to reduce the effect of communications latency. If we can combine the exchange of the haloes of several arrays into a single operation, we can reduce the number of messages. To achieve this, we need to abandon the use of MPI derived datatypes, manage our own buffers, and manually pack array patches into these buffers prior to a send, and perform the corresponding unpack after a receive. It is not obvious whether this should be expected to improve performance. Reducing the number of messages comes at the price of additional memory allocations, de-allocations and copies. It turns out that message combination is often, but not always, advantageous, as will be seen.

Masking, message clipping, and wet patches

Sub-domain haloes often contain dry points. This may be seen by a close examination of Figure 1 below, which illustrates the decomposition of HRCS domain into sub-domains. Eliminating dry points from halo exchange messages should improve the performance of halo exchanges, especially those that are bandwidth limited.

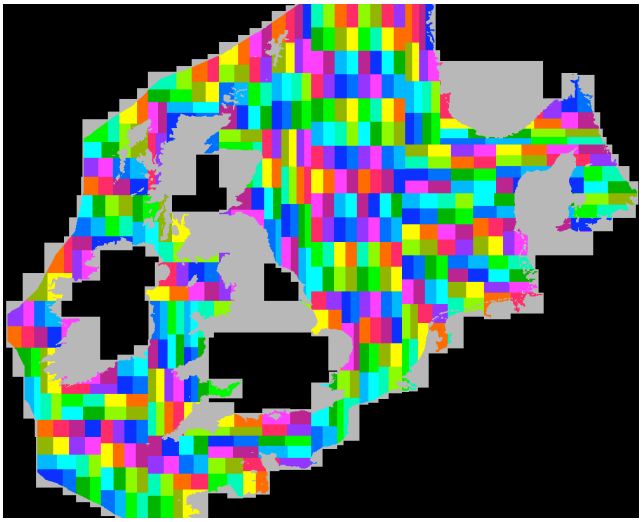


Figure 1. Sub-domain decomposition of the High Resolution Continental Shelf (HRCS) domain, which encompasses the North Sea and the British Isles. Black points are outside the domain. Grey points are dry, but inside the domain. Image courtesy of Mike Ashworth.

One way to eliminate dry points from halo exchanges is to apply a wet-dry mask during message packing and unpacking. As was the case with message combination, this optimization is incompatible with the use of MPI derived datatypes. Once again, testing the mask brings an overhead, and this overhead would be higher if the level index (in 3-dimensional arrays) varied slowest.

Another way to reduce the number of dry points in halo exchanges is to trim, or clip, *exterior* dry points from haloes in the communication tables. Message clipping has two appreciable advantages: it only needs to be done once, and, as it does not preclude the use of MPI derived datatypes, it can be used with the existing API. It is always a good thing to do. However, message clipping cannot eliminate *interior* dry points, and its performance benefits are not always measurable.

A third way to eliminate dry points is to extend the communications tables so that the grid points in each message are described by multiple wet patches, instead of just one patch containing both wet and dry points. This allows more efficient memory copies to be used in packing and unpacking. My implementation completely eliminates interior dry points from halo exchange messages when the haloes are one grid point wide, but the occasional dry point might remain when the haloes are wider. This method generally does as well as, or better than, message masking.

API, Fortran pointers, and compilers

The existing API, which operates only on single arrays, is not suitable for message combination. Accordingly, I have designed a new API that operates on a list of arrays. The list is represented as a Fortran90 derived type, containing Fortran pointers to up to seven 2-dimensional and seven 3-dimensional arrays, with the restrictions that the arrays must be of the same type (integer or 64-bit real) and have haloes of the same size. The API includes functions to set up and tear down the lists. A halo exchange operation is started by a call to *exch_list_start(list, exchange, …)* and completed by a call to *exch_list_finish(list, exchange)*. The first call returns an opaque "exchange" object, represented as a Fortran90 derived type, which encapsulates such things as the MPI tag and buffers used in the halo exchange operation.

The use of Fortran90 pointers inside the list and exchange objects was strongly indicated by the intention to provide an elegant and convenient API. However, the use of pointers introduced a performance issue in the packing and unpacking of message buffers. Comparing the measured performance of two versions, one using the new API and the other using an interim API that did not involve Fortran pointers, I observed a significant loss in performance, about 15-20% using the PGI compilers.[2] The performance loss was attributable almost entirely to the use of pointers, the problem being that the PGI compiler noticed the possibility of aliasing and refused to apply any of the usual optimizations. Worse, PGI Fortran offered no compiler directive to allow the programmer to assert that the pointers were alias-free. In the end, I managed to recover most of the lost performance by replacing the innermost loops of the message packing and unpacking code by a call to a Fortran77 style subroutine, which did a simple array copy. Now that the presence of pointers was concealed from the PGI compiler, it recognized the memory copy and made the appropriate optimization.

Pre-posting receives

Conventional wisdom has it that on Cray XT systems, it is advantageous to issue the call to MPI_IRECV as early as possible, for if the matching receive has already been posted when the send is processed, the MPI implementation can avoid a memory copy. I introduced a compile-time option to force each task to post all its receives before allocating its send

[2] This discussion is focused on the PGI compilers, preferred by the code owner for reasons not pertinent to this paper. The PathScale compilers did a little better.

buffers and posting its sends, but this by itself is not enough to ensure that when a task processes its sends, its peers have already posted matching receives. Indeed, there is little that can be done to increase the likelihood that this condition is satisfied. The performance benefits, if any, proved too small to measure.

Rank re-ordering

None of the optimizations considered above take communications locality into account. One way to do so is by re-labelling MPI tasks. Now, a brute force search over all permutations is infeasible; there are simply too many to explore. One could try the *mpi_sm_rank_order* experiment in CrayPat, taking care not to collect statistics during program start-up, which may have a different communications pattern from the rest of the program. I devised and implemented a greedy heuristic, which proceeds roughly as follows:

```
construct the sub-domain adjacency graph, with
    edges weighted by communications cost
mark all sub-domains as unallocated
mark all nodes as empty
sort sub-domains by the sum of their edges
do until all tasks allocated or no nodes empty
    select the unallocated sub-domain with the
        highest  communications cost
    assign it to the first empty node
    greedily fill this node with neighbours of the
        sub-domains already in the node,
        preferring neighbours with high weight
        edges
end do
assign un-allocated sub-domains to unfilled
    nodes arbitrarily
```

The MPI rank order lists produced by this rather naïve algorithm usually performed worse than the default ordering on HECToR, based on halo exchange experiments conducted during HECToR's dual core and quad-core phases. However, the algorithm did better on the 12-core system Rosa, occasionally yielding rank orderings that outperformed the default by up to 10%. Although I do not recommend this particular algorithm, its occasional successes suggest that there is scope for further investigation. It was trying to understand why the greedy algorithm performed well in some cases but poorly in others that led me to the ideas discussed in section 4.

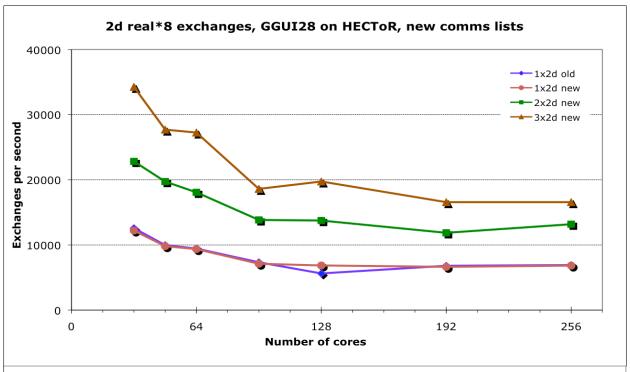
Results

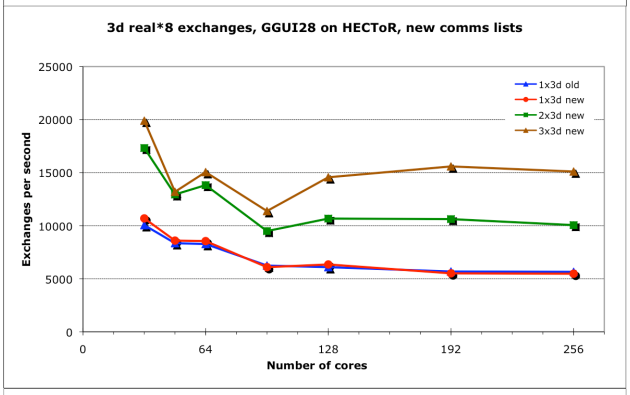
I have measured the performance of the optimized halo exchange on HECToR, during its quad-core phase. Two domains feature in these results. The domain labelled GGUI28 (for the Gulf of Guinea, on the west coast of Africa) is a small domain, contained within a box of 323 grid points in the *x* dimension and 181 in the *y* dimension, with 20 levels. The domain labelled HRCS (High Resolution Continental Shelf, the area surrounding the North Sea and the British Isles) is larger, contained within a box of 1001 grid points in the *x* dimension, 812 grid points in the *y* dimension, and 34 levels. The geometries of the two domains also differ in a way that could conceivably have performance consequences. In GGUI28, dry (land) points are located along the northern and eastern edges, and the active wet points follow the coast in a narrow band (see Figure 6). In HRCS, dry points are found throughout the domain (see Figure 1).

Figures 2 and 3 below show performance results for the optimized POLCOMS halo exchange using these two domains. In each case, the time taken to complete a large number (2000 or so) of *consecutive* halo exchanges was measured. Since exchanging haloes has the effect of loosely synchronizing MPI processes, the reported speeds are effectively those of the slowest process. Results are plotted separately for 2-dimensional halo exchanges (top), and 3-dimensional exchanges (middle). In these charts, I plot speeds in halo exchanges per second, based on the number of effective exchange operations, for (a) the old method, (b) the new method without message combination, (c) and (d) the new methods combining respectively 2 and 3 exchanges in a single operation. I treat "mixed-dimensional" exchanges (bottom) slightly differently, reporting speeds based on the time taken to perform two 2-dimensional and one 3-dimensional exchange, without and with message combination. In each case, the results reported for the old method used the message clipping. In interpreting these results, please be aware that run-to-run fluctuations of about 5% are normal, and an occasional burst of activity on the interconnect, which is a shared resource on Cray XT systems, can suppress performance by much more.

From these results several things are clear. Firstly, it is always beneficial to combine the halo exchanges of 2-dimensional arrays when the application logic permits, as the top-most charts in Figures 2 and 3 show. This is to be expected, as these operations are latency limited (halo messages are typically a few hundred bytes in length). Secondly, it is almost always beneficial to tack a few 2-dimensional exchanges onto a 3-dimensional exchange when the application logic permits.

However, it is *not always* beneficial to combine 3-dimensional exchanges. Although the smaller domain is well behaved (Figure 2), on the larger domain (Figure 3), we see that combining three 3-dimensional exchanges is actually worse than combining only two. This result is surprising and requires further investigation. A possible explanation is that cache conflicts in the message packing and unpacking code come into play as the size of message buffers exceeds a certain threshold.





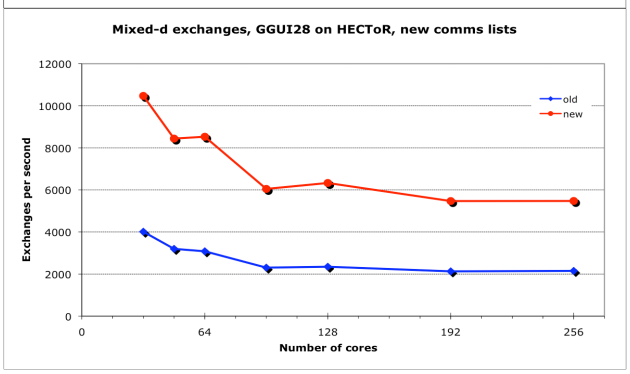
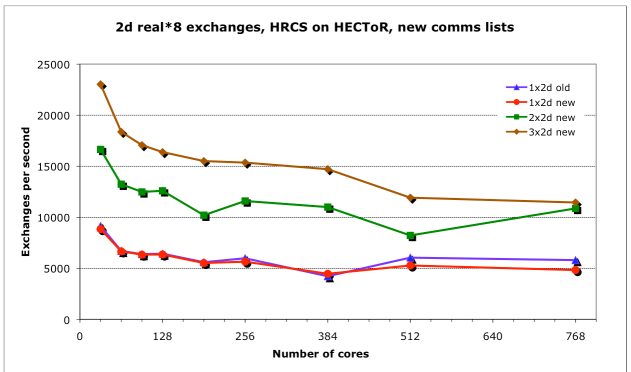
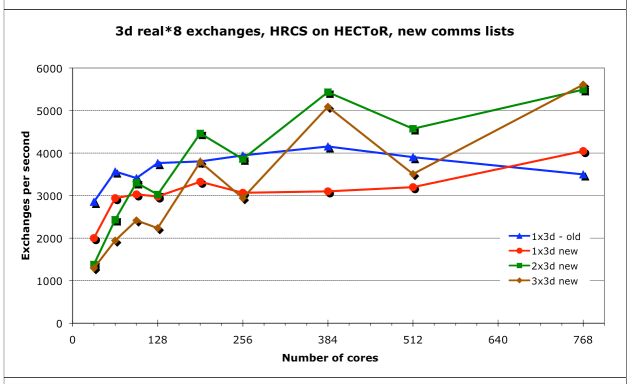


Figure 2. Performance of halo exchanges at various core counts on a small domain, on quad-core Cray XT.

These charts also show how far removed from the ideal of strong scaling the performance of halo exchanges actually is. For strong scaling, the speed of a halo exchange would have to double with each doubling of the number of cores. For 2-dimensional exchanges the speed actually falls as the number of cores increases, due in part to fundamental limitations of latency, and in part to the fact that the number of messages that any process has to send during a halo exchange tends to increase slowly as the number of cores increases and the domain is decomposed more finely. For bandwidth-limited 3-dimensional exchanges (Figure 3, middle), there is initially a modest increase in performance with core count as halo sizes diminish, but this soon tails off.





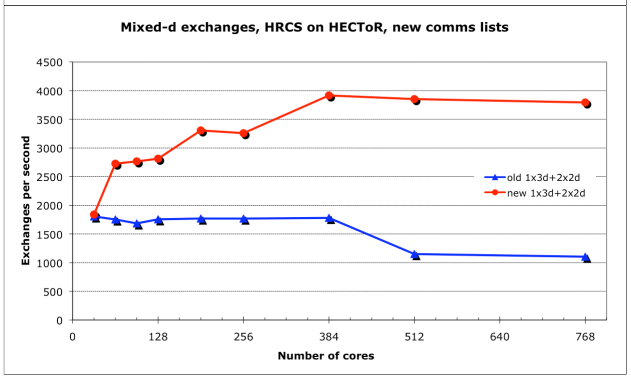


Figure 3. Performance of halo exchanges at various core counts on a large domain, on quad-core Cray XT.

Figure 4 below is based on earlier measurements using message masking and message combination taken on HPCx (the IBM Power5 system, now retired, that preceded HECToR as the UK's national HPC service). It clearly shows the benefit of eliminating dry points.

Comparing Figure 4 with Figure 3 (middle) highlights an interesting difference between HPCx and HECToR. In sharp contrast to the situation on HECToR, the new API always outperformed the old on HPCx, even for the case of 3-dimensional exchanges without message masking and message combination. I attribute this to the relative quality of the implementations of MPI derived datatypes on the two systems.

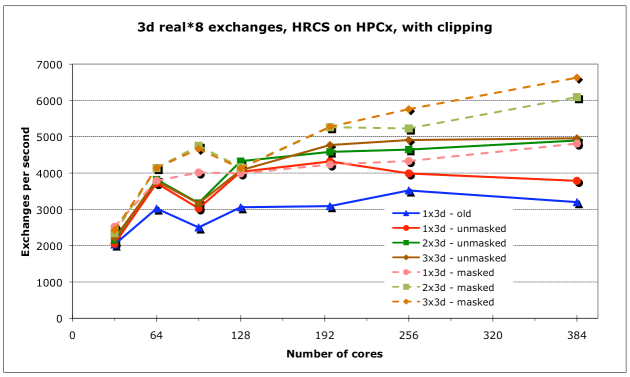


Figure 4. Performance of halo exchanges at various core counts on a large domain, showing the effect of message combination and message masking. Results from HPCx.

There are about 350 halo exchange operations in the POLCOMS application code. So far I have replaced about 20% of these by calls to the new API, focussing on the cases where message combination is feasible and likely to bring performance benefits. Figure 5 below shows the performance improvement, relative to the original code, of certain key routines (velocity advection, scalar advection, barotropic time-step, baroclinic time-step) and the total run-rime. Note that the total run-time is suppressed by the poor scaling of I/O and other routines (not plotted). The fact that the benefit tends to increase with the number of cores suggests that the new API scales better than the old one. The disappointing performance at 96 and 192 cores is probably due to sub-optimal partitions when the processor grid involves a factor of 3, a problem that the optimizations in the next section will address.

Figure 5. Relative performance improvement of the new API on key routines in POLCOMS, measured on a small domain on HECToR.

4. Multicore-aware partitioning

The recursive ***k***-section decomposition method used by POLCOMS starts by finding the most nearly square factorization of the n_c available cores

$$n_c = n_x \times n_y$$
$$n_x = f_1 \times f_2 \times \cdots \times f_x$$
$$n_y = g_1 \times g_2 \times \cdots \times g_y$$

where $\boldsymbol{n_x}$ and $\boldsymbol{n_y}$ are the sizes of the processor grid in the ***x*** and ***y*** dimensions, and the $\boldsymbol{f_i}$ and $\boldsymbol{g_i}$ are their prime factors, arranged (for no particular reason) in descending order. The method cuts the domain into $\boldsymbol{f_1}$ sub-domains containing approximately equal numbers of wet points, and proceeds recursively, cutting each sub-domain into $\boldsymbol{g_1}$, then $\boldsymbol{f_2}$, then $\boldsymbol{g_2}$, and so on, until all factors are exhausted, and n_c sub-domains have been created.[3]

There is nothing in the recursive ***k***-section decomposition method that depends on the order of factors, or how they are split between the *x* and *y* dimensions.

Sub-domains are numbered such that the last ***k***-section applied to any sub-domain results in ***k*** consecutively numbered sub-domains. This numbering scheme has the serendipitous property that the last two bisections will usually divide each sub-domain into four, which tends to yield good communications locality on a quad-core system such as HECToR (phase 2a), when the default MPI task placement method is in use. In contrast, a different partition may be preferable on other architectures. Figure 6 below compares two partitions of a small domain (the Gulf of Guinea) across 24 cores. On a system with 12-core nodes, sub-domains 0-11 are placed on one node, and sub-domains 12-23 are placed on the other. The factorization (3x2,2x2), the original default, leads to reasonable communications locality on quad-core systems, but on 12-core systems, an alternative factorization (2x2x2,3) is preferable; measurements on Rosa confirm that the halo exchange performs at more than twice the speed.

[3] There are some possible partitions that this method does not reach. If the recursive k-section routine was changed to take a single list of $\boldsymbol{n_f}$ factors and a corresponding list of $\boldsymbol{n_f}$ dimensions (*x* or *y*) in which to apply the cut, then the $(n_f + 1)!$ in equation (1) would become $2^{n_f} n_f!$ instead.

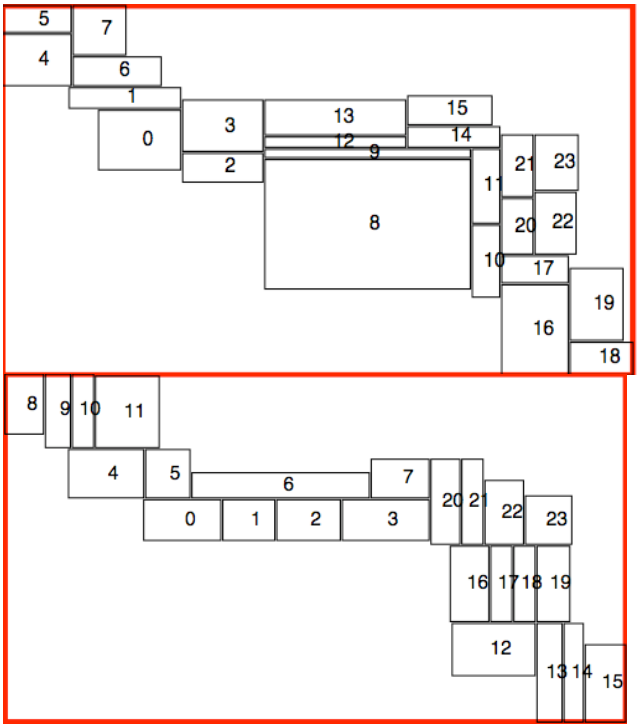


Figure 6. Two different partitions of the same domain on 24 processors. Above: the default factorization (3x2,2x2) leads to a partition with good communications locality on quad-core systems. Below: a different factorization (2x2x2,3) is preferable on a system with 12-core nodes.

Thus, each distinct factorization induces a different partition, and each of these can have different properties such as computational load balance or communications performance. How many distinct factorizations are there? How much do the performance properties of the induced partitions vary? Can we select the factorization with the best performance properties, and do this cheaply at run-time?

A digression into combinatorics

It is not difficult to calculate the number ***N*** of distinct ways to factorize n_c cores. Let ***d*** denote the number of distinct prime factors of n_c, and let m_i denote the multiplicity of the ***i***th distinct factor. Then

$$n_f = \sum_{i=1}^{d} m_i$$

is the total number of prime factors, and there are $n_f!$ permutations of these. So we have:

$$N(n_c) = \frac{(n_f + 1)!}{\prod_{i=1}^{d} m_i!} \qquad (1)$$

Here, the additional $n_f + 1$ in the numerator counts the number of ways to split the factors between ***x*** and ***y***. The denominator corrects for over-counting: for each factor that appears with multiplicity ***m***, there are ***m!*** identical permutations. In terms of multinomial coefficients, this result reads:

$$N(n_c) = (n_f + 1)\binom{n_f}{m_1, m_2, \ldots, m_d}$$

Figure 7 below plots the number ***N*** of distinct partitions as a function of the number of cores, up to 2048 cores. Notice that ***N*** rarely attains or exceeds n_c. The exceptions occur at n_c = 2, 6, 12, 60, 120, 180, **360**, **720**, 840, **1080**, 1260, 1440, 1680,…

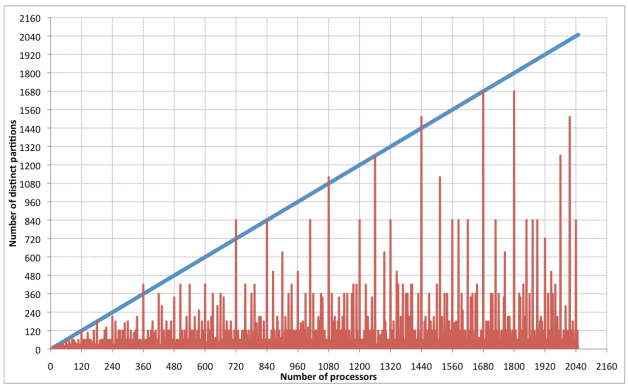


Figure 7. Distinct factorizations of the POLCOMS processor grid as a function of the number of cores.

Choosing the "best" partition

Knowing that the number of distinct partitions of a domain rarely exceeds the number of cores available, suggests the idea of visiting each partition in parallel, and choosing the "best" partition, at run-time. In the following Fortran-style loop over permutations, ***N*** is the number of distinct permutations, permutations are numbered from 0 to ***N***-1, and rank and size have the obvious meanings.

```
do i=rank, N-1, size
    determine the factors of the i^th permutation
    compute the corresponding partition
    evaluate a cost function for this partition
end do
select the permutation with the best cost function
re-compute the partition for this permutation
```

Selecting the permutation with the best cost function can be achieved by a single call to MPI_ALL_REDUCE using the MPI_MINLOC pre-defined operation.

Visiting distinct permutations

Ideally, one wants to visit the i^{th} *distinct* permutation without enumerating all N of them. It would be sufficient to have a method for visiting the i^{th} distinct permutation of (not necessarily distinct) objects, for the factor of $n_f + 1$ in equation (1) is easily dealt with separately. However, this problem is rather subtle, and deserves some discussion.

It is well known that the problem of visiting the i^{th} permutation of distinct objects can be treated using a *factoradic* method; by representing the permutation number in a variable radix basis where the radices are the factorials, the permutation can be recovered from the permutation number by relatively straightforward arithmetic operations involving the radices. But in our case, any factor may be repeated, and we want to visit only distinct factorizations. Unable to find a published algorithm dealing with this problem, I devised my own.

It helps to think of the problem as placing m_1 copies of the first distinct object into n_f available bins, then placing m_2 copies of the second distinct object into the $n_f - m_1$ available bins remaining, and so on, until all objects have been placed (and all bins filled). I employ a variable radix basis to keep track of the number of ways of distributing m_i copies of the i^{th} distinct object into the available bins, then traverse these combinations in lexicographic order. Full details are available on request.

Constructing the cost function

The cost function used in this work is based on the following considerations. Computation time on any processor is dominated by the number of wet points, but the number of dry points brings a small performance overhead due to the need to test the mask arrays, and incidentally determines the memory requirements. Communications time will be dominated by the performance of the halo exchange. Most haloes have a width of 1 grid point. The performance of MPI messages between processes on the same multi-core node will be, at least on modern MPI implementations (including CrayXT systems), considerably faster than messages between processes on different multi-core nodes. The overall run-time t will be limited by the slowest MPI process. These considerations lead to:

$$t \propto \max\left(c_{wet} n_{wet} + c_{dry} n_{dry} + c_{off} n_{off} + c_{on} n_{on}\right)$$

where the maximum is taken over sub-domains, n_{wet} and n_{dry} are respectively the number of wet and dry points in a sub-domain, n_{off} and n_{on} are the number of points in a sub-domain's halo that are exchanged with sub-domains placed on a different (n_{off}) or the same (n_{on}) multi-core node, and the c_* are tuneable coefficients.

The calculation of c_{off} and c_{on} requires knowing which MPI processes are placed on which nodes. Unfortunately, there is no way of doing this that is both portable, and standard.[4] In this work, I simply pass in the number of cores per node (n_{cpn}) as a run-time argument, and assume that the first n_{cpn} processes are on the first node, the next n_{cpn} processes are on the second node, and so on; this is the default on Cray XT systems.

This form of the cost function is a little naïve. It neglects, for example, the contribution of MPI latency to halo exchange performance. This could be remedied easily enough by counting the number of on-node and off-node messages sent and received by each process. But it does have the virtue of being quick to calculate.

5. Results and Discussion

Instead of tuning these coefficients carefully, which is a non-trivial exercise, I used the values $c_{wet} = 1$, $c_{dry} = 0.05$, $c_{off} = 5$, $c_{on} = 1$ in this study. These choices are not entirely arbitrary, but probably over-emphasize the importance of communications.

In order to study the effect of the choice of partition on the halo exchange performance, I have conducted runs at various core counts on HECToR (phase2a, CrayXT4, 4 cores per node) and Rosa (Cray XT5, 2x6 cores per node). For the purposes of calculating n_{off} and n_{on}, I treat Rosa as a 12-core system ($n_{cpn} = 12$). The first run at each core count uses the partitioning that yields the "best" cost function given the above coefficients, but also identifies the partitions that yield the best and worst values for alternate cost-functions:

$$t_{wet} = \max\left(n_{wet}\right)$$

$$t_{dry} = \max\left(n_{dry}\right)$$

$$t_{off} = \max\left(n_{off}\right)$$

$$t_{on} = \max\left(n_{on}\right)$$

$$t_{comms} = \max\left(c_{off} n_{off} + c_{on} n_{on}\right)$$

Subsequent runs explore the performance properties of alternate partitions. Figure 8 below shows the variation in the speed of a mixed-dimensional halo exchange (the haloes of two 2-dimensional and one 3-dimensional array

[4] MPICH2 offers some experimental topology enquiry functions, but does not provide Fortran bindings to these.

are exchanged in a single operation) on the high-resolution (HRCS) domain on the Cray XT5 system. Notice that the variation in performance (up to a factor of 3 or more) tends to be higher when there are more partitions to choose from; 256 cores, historically a typical choice on single-, dual- and quad-core architectures, is particularly bad.

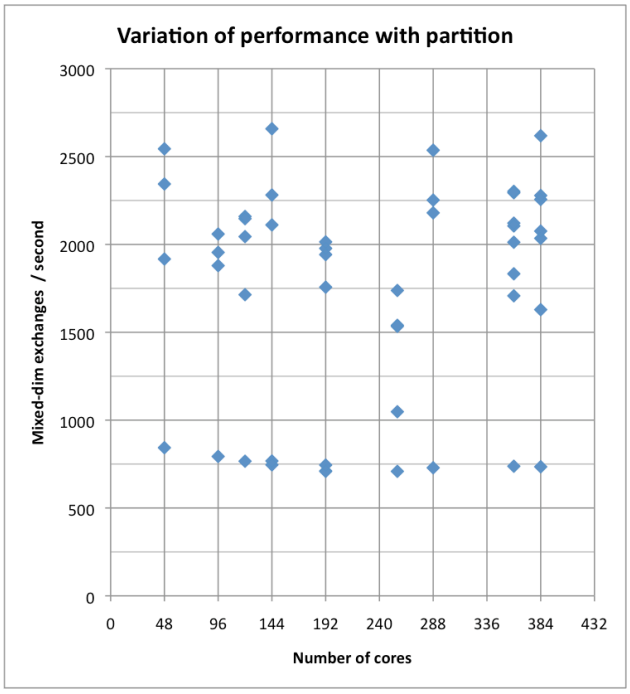


Figure 8. Variation of performance of mixed-dimensional halo exchanges on different partitions. Results from Rosa (2x6-core) on the HRCS domain.

To give some indication of how successful the cost functions are at predicting performance, I note that the partition selected by optimizing t_{comms} had the best measured performance 5 times (out of the 9 core counts studied), was within 15% of the best 3 times, and chose poorly once. The partition selected by optimizing t had the best measured performance once, was within 15% of the best 5 times, and within 25% of the best 3 times.

The behaviour of 3-dimensional exchanges is similar to that shown in Figure 8 above. However, 2-dimensional exchanges behave quite differently. The partitions on which (bandwidth limited) 3-dimensional exchanges perform worst are typically those in which the domain is decomposed only in one dimension (x or y). But the same partition can actually be optimal for (latency limited) 2-dimensional halo exchanges, as such a decomposition completely eliminates messages in the other (y or x) dimension.

6. Summary and Conclusions

I have described a set of optimizations that have helped improve the performance of the halo exchange in POLCOMS, and may be applicable to other MPI-based ocean modelling codes. Some of these come at the price of abandoning MPI derived datatypes — a big sacrifice on Cray XT systems — but still yield net gains in performance.

I have also shown how untapped freedom in the POLCOMS domain decomposition method can be exploited to improve communications locality and hence halo exchange performance, by evaluating cost functions for alternate partitions in parallel. The overhead of doing so is completely negligible. The extent of the variation of halo exchange performance shows that there are significant gains to be had, and the arrival of multi-core nodes based on 6-core chips makes these ideas both timely and pertinent.

Some aspects of this work are still in progress. There remain unresolved, possibly cache-related, issues to do with the performance of message packing and unpacking. Extending the cost function to take communications latency into account should be straightforward; it remains to be seen whether this will yield a better predictor of performance. As I hinted in a footnote in section 4, the current recursive k-section decomposition algorithm does not reach as many partitions as it could. I have not exhausted the possibilities of rank re-ordering. Finally, more work is required on quantifying the impact of these optimisations on the performance and scalability of the code as a whole, and on the closely related problem of tuning the cost function.

Acknowledgments

This work was started during the GCOMS project [3], funded by NERC Grant NE/C516001/1. I am grateful to the Swiss National Supercomputing Centre (CSCS) for computing time on their Cray XT5 system (Rosa), and to Mike Ashworth, Andrew Porter, Kevin Roy and Jason Holt for helpful discussions. All of the errors in this paper are my own.

About the Author

Stephen Pickles is a Computational Scientist in the Advanced Research Computing Group, Computational Science and Engineering Division, STFC Daresbury Laboratory, Daresbury Science and Innovation Campus, Warrington, Cheshire WA4 4AD, UK. Email addressed to stephen.pickles@stfc.ac.uk has been known to reach him.

References

[1] Holt, J.T. and James, I.D., "An s-coordinate density evolving model of the North West European Continental Shelf. Part 1 Model description and density structure", *Journal of Geophysical Research*, 106(C7): 14015-14034 (2001).

[2] Ashworth, M., Holt, J.T. and Proctor, R., 2004. Optimization of the POLCOMS Hydrodynamic Code for Terascale High-Performance Computers, *Proceedings of the 18th International Parallel & Distributed Processing Symposium*, 26th-30th April 2004, Santa Fe, New Mexico.

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