Geosci. Model Dev. Discuss., 6, 4705–4744, 2013
www.geosci-model-dev-discuss.net/6/4705/2013/
doi:10.5194/gmdd-6-4705-2013

Geoscientific Model Development Discussions
Open Access

A distributed computing approach to improve the performance of the Parallel Ocean Program (v2.1)

B. van Werkhoven[1], J. Maassen[2], M. Kliphuis[3], H. A. Dijkstra[3], S. E. Brunnabend[3], M. van Meersbergen[2], F. J. Seinstra[2], and H. E. Bal[1]

[1]VU University Amsterdam, Amsterdam, the Netherlands
[2]Netherlands eScience Center, Amsterdam, the Netherlands
[3]Institute for Marine and Atmospheric research Utrecht, Utrecht, the Netherlands

Received: 5 July 2013 – Accepted: 8 August 2013 – Published: 12 September 2013

Correspondence to: B. van Werkhoven (ben@cs.vu.nl)

Published by Copernicus Publications on behalf of the European Geosciences Union.

Abstract

The Parallel Ocean Program (POP) is used in many strongly eddying ocean circulation simulations. Ideally one would like to do thousand-year long simulations, but the current performance of POP prohibits this type of simulations. In this work, using a new distributed computing approach, two innovations to improve the performance of POP are presented. The first is a new block partitioning scheme for the optimization of the load balancing of POP such that it can be run efficiently in a multi-platform setting. The second is an implementation of part of the POP model code on Graphics Processing Units. We show that the combination of both innovations leads to a substantial performance increase also when running POP simultaneously over multiple computational platforms.

1 Introduction

Physical oceanography is currently undergoing a paradigm shift in the understanding of the processes controlling the global ocean circulation. Two factors have contributed to this shift: (i) the now about twenty-year long record of satellite data and (ii) the possibility to simulate the ocean circulation using models which include processes on the Rossby deformation radius (10–50 km). Resolving this scale captures the instability processes that lead to ocean eddies which subsequently interact and affect the large-scale ocean flow (Vallis, 2006).

The level of realism (in relation to available observations) in simulating the ocean with high-resolution strongly eddying models substantially increases compared to the low-resolution models in which the effects of eddies are parametrized. For example, it leads to a much better simulation of the different oceanic boundary currents, in particular the separation of the Gulf Stream in the Atlantic. Also the degree to simulate the surface kinetic energy distribution, which can be compared with satellite data, markedly improves (Smith et al., 2000; Maltrud et al., 2010).

GMDD
6, 4705–4744, 2013

The use of the strongly eddying models is, even on the supercomputing platforms currently available, still computationally expensive and simulations have a long turn-around time. Typical performances are one to a few model years per 24 h using thousands of cores (Dennis, 2007). Considering the fact that it takes at least 1000 yr to reach a near statistical equilibrium state, innovations to increase the performance of these models and to efficiently analyze the data from the simulations have a high priority.

Today many traditional cluster systems are equipped with GPUs, because of their ability to process computationally intensive workloads at unprecedented throughput and power efficiency rates. Existing software requires modifications such as the expression of fine-grained parallelism before it may benefit from the added processing power that GPUs offer (Bleichrodt et al., 2012). However, it is currently not well known what specific parts of ocean models can benefit the most from execution on GPUs, how the existing software should be revised to efficiently use GPUs, and what impact the use of GPUs will have on performance.

In this paper, we present two innovations to improve the performance of the Parallel Ocean Program (POP). POP is also used as the ocean component of the much used Community Earth System Model (CESM). We have applied our modifications to a standalone version of POP (v2.1). However, we have confirmed through source code inspection that all of our changes are also applicable to and fully compatible with the latest release of CESM (v1.2.0). The main issue is how to adapt POP such that it can run simultaneously (and efficiently) on multiple GPU clusters. First we address alternative domain decomposition schemes and hierarchical load balancing strategies which enable multi-platform simulations such that further scaling can be achieved. Second, we show how POP can be adapted to run on Graphics Processing Units (GPUs) and study the effect of GPU usage on its performance. The source code of our modified version of POP can be obtained from https://github.com/NLeSC/eSalsa-POP/.

GMDD
6, 4705–4744, 2013

2 Load balancing

The model considered here is the global version of POP (Dukowicz and Smith, 1994) developed at Los Alamos National Laboratory. We consider the strongly eddying configuration, indicated by $R_{0.1}$, as used in recent high-resolution ocean model simulations (Maltrud et al., 2010; Weijer et al., 2012). This version has a nominal horizontal resolution of 0.1° using a 3600 × 2400 horizontal grid with a tripolar grid layout, having poles in Canada and Russia. The model has 42 non-equidistant z levels, increasing in thickness from 10 m just below the upper boundary to 250 m just above the lower boundary at 6000 m depth. In addition, bottom topography is discretized using partial bottom cells, creating a more accurate and smoother representation of topographic slopes.

2.1 Domain decompositions and block distributions

POP supports parallelism on distributed memory computers through the Message Passing Interface (MPI). To distribute the computation over the processors, POP uses a three-dimensional mesh, sketched in Fig. 1a. The domain is decomposed into equal sized rectangular blocks in the horizontal direction (corresponding to the longitude and latitude). Each block also contains several layers in the vertical direction (depth). The blocks are then distributed over the available MPI tasks, where each task receives one or more blocks. Blocks consisting only of land points may be discarded from the computation. Below we will assume that a single MPI task is assigned to a processor core (unless stated otherwise).

Each block is surrounded by a halo region (Fig. 1b) that contains a copy of the information of the neighbouring blocks. These halos allow the calculations on each block to be performed relatively independently of its neighbour blocks, thereby improving parallel performance. Nevertheless, the data in the halo regions needs to be updated regularly. This requires a data exchange between the blocks, which leads to communication

GMDD
6, 4705–4744, 2013

between the MPI tasks, the amount of data depending on the width of the halo, the size of the blocks, and the block distribution over the MPI tasks.

In POP, the halo width is typically set to 2. For an example block size of 60×60, the number of elements that need to be exchanged per block in every halo exchange is $4 \times (60 \times 2) + 4 \times 4 = 496$. This number may need to be multiplied by the number of vertical levels, depending on the datastructure on which the halo exchange is performed. Some datastructures, like the horizontal velocity, store a value for every gridpoint at every depth level. As a result, a 3-D halo exchange is required that exchanges elements from every depth level. Others datastructures, such as surface pressure, only consist of a single level. There, a 2-D halo exchange is sufficient.

For neighboring blocks that are assigned to the same MPI task, the data exchange is implemented by an internal copy and no MPI communication is required. Also, no data needs to be exchanged with (or between) land elements. Therefore, the amount of data that needs to be communicated between MPI tasks depends heavily on the way the blocks are distributed over the MPI tasks.

2.2 Existing block partitioning schemes

POP currently supports three algorithms for distributing the blocks over the available MPI tasks, cartesian, rake (Marquet and Dekeyser, 1998), and space-filling curve (Dennis, 2007). The cartesian algorithm starts by organizing the tasks in a two-dimensional grid. Next, the blocks are assigned to these tasks according to their position in the domain. If the number of MPI tasks does not divide the number of blocks evenly in either dimension, some tasks may receive more blocks than others. In addition, some tasks may be left with less work (or even no work) if one or more blocks assigned to it only contain land. As shown in Dennis (2007), load imbalance between tasks can significantly degrade the performance of high-resolution ocean simulations.

The rake algorithm attempts to improve the load balance by redistributing the blocks over the tasks. Note that this requires that the number of blocks is significantly larger than the number of MPI tasks. The rake algorithm starts with a cartesian distribution

GMDD
6, 4705–4744, 2013

and the corresponding two-dimensional MPI task grid. First, the average number of blocks per task is computed. Then, for each row in the task grid, the algorithm takes the first task in the row and determines whether the number of blocks exceeds the average. If so, the excess blocks are passed on to the next task. This process is repeated for all tasks in the row. The process is repeated for all columns of the task grid. As described in Smith et al. (2010), the algorithm "can be visualized as a rake passing over each node and dragging excess work into the next available hole". In an attempt to keep neighboring blocks close together, constraints are placed on block movements that prevent blocks from moving too far from their direct neighbours. Unfortunately, there are instances where the rake algorithm actually results in a worse load balance where blocks get raked into a corner. As a result Dennis (2007) states that "We do not consider the current implementation of the rake algorithm... sufficiently robust."

The space-filling curve algorithm described in Dennis (2007) uses a combination of Hilbert, meandering Peano, and Cinco curves to partition the blocks (Fig. 2). Conceptually, it draws a single line that visits each of the blocks exactly once. It then splits this line into equal size segments, each segment visiting the same number of blocks. Due to the way the line is drawn, the blocks in each segment are also continuous in the two-dimensional domain. This solution degrades slightly when the land-only blocks are discarded, which introduces "cuts" in the curve. Nevertheless, the space-filling curve algorithm significantly improves the load balance between MPI tasks. A limitation of this approach is that each of the space-filling curves can only partition domains of a specific size. For example, a domain $P \times P$ can be partitioned by a Hilbert curve if $P = 2^n$, or by a meandering Peano curve if $P = 3^m$, where n and m are integers. By using combinations of different curves the set of supported problem sizes can be extended.

2.3 A new hierarchical partitioning scheme

None of the load balancing algorithms described in the previous section takes into account the inherent hierarchical nature of modern computing hardware. This typically consists of multiple cores per processor, multiple processors per node, multiple nodes

GMDD
6, 4705–4744, 2013

per cluster, and even the availability of multiple clusters for a numerical simulation. The communication performance drops as we go up in the hierarchy. The cores in a processor share cache memory and can therefore communicate almost instantaneously, while communication between processors has to go through main memory which is much slower. Communication between processors on different nodes must go through an external network which is orders of magnitude slower, and communication between clusters in different locations is again orders of magnitude slower. Therefore, simply balancing the load for the individual processors (or cores) is not sufficient. In addition, the hierarchy of the target machine must be taken into account to reduce the amount of communication between the components.

Our new hierarchical load balancing scheme, like the rake and space-filling curve algorithms described earlier, assumes that the number of blocks is significantly larger than the number of processors. Instead of simply specifying the number of MPI tasks for which to create a partitioning, the user must now specify a sequence of partitionings. For example, a sequence $2:16:8$ indicates that the blocks must first be partitioned into 2 sets (preferably of equal size), each of which is then partitioned into 16 pieces, that are further divided into 8 pieces. The sequence of partitionings relates directly to the hierarchy that is present in the computational platform. For example, the $2:16:8$ partitioning can be used for an experiment on two clusters, each containing 16 nodes of 8 cores.

Once the user has specified the desired partitioning, the algorithm proceeds by repeatedly splitting the available blocks into N (preferably equal sized) subsets. We try to partition the domain in such a way that the shape of each of the subsets is as close to a square as possible. This will reduce the amount of communication out of each subset in relation to the amount of work inside each subset.

When splitting a domain, multiple solutions may be available which are equivalent from a load balancing perspective. However, the amount of communication required between subsets may vary between these solutions due to assignment of blocks to MPI tasks and the location of land-only blocks. Our algorithm therefore compares these

GMDD
6, 4705–4744, 2013

solutions and selects the one which generates the least communication between subsets.

To explain our algorithm in more detail, we use the simplified example domain shown in the upper left panel (a1) of Fig. 4. This example domain contains 1200 × 1000 grid elements. It is divided into blocks of 100 × 100, resulting in 12 × 10 blocks, of which 20 are land-only blocks. To divide this domain into 10 subsets, the algorithm starts by computing the required number of blocks per subset. The 100 non-land blocks must be divided into 10 subsets, resulting in 10 blocks per subset. Next, the algorithm tries to arrange the desired number of subsets in a (roughly) rectangular grid. The dimensions of this grid, consisting of N subsets, is determined as follows:

```
f:= floor(sqrt(N));
c:= ceiling(sqrt(N))

if (f = c)   We have found a square grid of [f x f]
if (f*c = N) We have found a rectangular grid of [f x c]
if (N < f*c) We have found a rectangular grid of [f x c] - (f*c-N)
if (N > f*c) We have found a square grid of [c x c] - (c*c-N)
```

In the first two cases of the algorithm shown above, a square or rectangular decomposition is available containing exactly N subsets. In the last two cases, the decomposition contains respectively $(f * c - N)$ or $(c * c - N)$ subsets too many. To correct this, we repeatedly remove a single subset from each row until the desired number of subsets is reached. Figure 3 shows four example subdivisions, for values of $N = 4, 6, 8$ and 10, that correspond to each of these four cases. For our example domain we will use the rightmost subdivision in Fig. 3 for $N = 10$ named [3,3,2,2], which represents the number of blocks in each column.

Next, we compute the required number of blocks per column using the average number of blocks per subset and the selected subdivision. For our example, we will use the [3,3,2,2] subdivision as in Fig. 3 and the 10 blocks/subset average, which will result in columns containing [30,30,20,20] blocks. We then split the domain into subsets by traversing the blocks in a vertical zigzag fashion and selecting all non-land block until

GMDD
6, 4705–4744, 2013

the desired number of blocks for that column in reached. The panels (a2–a6) in Fig. 4 show how the example domain is split into the 4 columns. We subsequently split each of the columns in a horizontal zigzag fashion into the desired number of subsets for that column. Figure 4, panels (b1–b5), show an example for the first column, which needs to be split into 3 subsets of 10 blocks each. A similar subdivision is applied to the other columns. The final block distribution for the example domain is shown in Fig. 4c.

As explained above, the subdivision shown in panel (c) of Fig. 4 is only one out of a series of options. Several permutations of the [3,3,2,2] subdivision can be created that are equivalent from a load-balancing perspective, but require a different amount of communication. In addition, the subdivision can also be rotated, thereby initially dividing the domain row-wise instead of column-wise. Finally, when selecting the blocks in a zig-zag fashion (as shown in Fig. 4) a choice can be made from which position to start the selection, top or bottom, or left or right. In our algorithm we simply compute all unique permutations of the subdivision in all possible rotations, with all possible starting points. We then select the solution with the lowest average communication per subset. If multiple equivalent solutions exist, we select the one with the lowest maximum communication per subset. Table 1 shows the best scoring results for all permutations of the [3,3,2,2] subdivision. All solutions use the same number of blocks per task, but the amount of communication varies per solution. Once a domain has been split into the desired number of subsets, the algorithm is repeated for each of these subsets for the next split.

2.4 Hierarchical partitioning of tripole grids

In the application of the hierarchical load balancing scheme to POP, the tripolar grid layout, where the North Pole is replaced with two poles located (on land) in Canada and Russia, needs special attention. In this case, additional communication is required for the blocks located on the line between these poles, as explained in Smith et al. (2010). These blocks are located on the upper boundary of the grid, as shown in Fig. 5a. To support a tripolar grid layout in our hierarchical load balancing scheme, we add

GMDD
6, 4705–4744, 2013

the additional tripole communication to the communication requirements of the subset whenever a subset contains a tripole block. The extra communication will then be taken into account in the search phase of the algorithm. Although this approach will improve the partitioning, the result will not be optimal. As shown in Fig. 5a, two communicating tripole blocks may be located on opposite sides of the grid. This makes it difficult for our partitioning scheme to put these two blocks into the same subset. We overcome this problem by remapping the grid before we start the partitioning (Fig. 5b). By simply moving blocks from one side of the grid to the other, we enable our partitioning algorithm to optimize the tripole communication. Note that this remapping is only performed on the grid used in our partitioning algorithm. No change to POP is required, as POP only uses the result of the partitioning in which the original block numbering is maintained.

3 Results: load balancing

In this section we will compare the performance of our hierarchical algorithm to the cartesian, rake and space-filling curve block partitioning schemes. In our experiments we carry out a 10 day simulation with the $R_{0.1}$ version of POP, as described at the beginning of Sect. 2, and show performance measures averaged over these 10 days.

3.1 Hardware

The *Huygens* (http://www.surfsara.nl) is an IBM pSeries 575, a clustered SMP (Symmetric Multiprocessing) system. Each node contains 16 dual core IBM Power 6 processors running at 4.7 Ghz, resulting in 32 cores per node. As the cores support Simultaneous Multi Threading (SMT), every node appears to have 64 CPUs. Most applications will perform better by using 64 MPI tasks per node (two MPI tasks per processor core). Per node, 128 GB of memory is available (4 GB per core). The nodes are connected using 8x(4xDDR) InfiniBand, resulting in a 160 Gbit s^{-1} inter-node bandwidth.

GMDD
6, 4705–4744, 2013

The *DAS4* (http://www.cs.vu.nl/das4) is a six-cluster wide-area distributed system. DAS-4 is heterogeneous in design, but in this experiment we will use dual-quad-core compute nodes containing Intel E5620 CPUs running at 2.4 Ghz, resulting in 8 cores per node. The nodes contain 24 GB of memory (3 GB per core). Nodes are connected using QDR InfiniBand resulting in a 20 $\mathrm{Gbit\,s^{-1}}$ bandwidth. We use DAS-4 in a single cluster and two cluster experiment. In the two cluster experiment, the clusters are connected using a internet link with a maximum bandwidth of 1 $\mathrm{Gbit\,s^{-1}}$. The average round trip time between clusters is 2.6 ms. As the link is shared with other users, the available bandwidth and round trip latency may vary over time.

3.2 Using MPI for multiple clusters.

For POP to run on multiple clusters, an MPI implementation is required that is capable of communicating both within and between clusters. This is far from trivial, as clusters are often protected by a firewall that disallows any incoming communication into the cluster. Also, it is common for the compute nodes to be configured such that they can only communicate with the cluster frontend, but not directly with the outside world, as explained in Maassen and Bal (2007). To solve this problem, we created wrapper code that is capable of intercepting the MPI calls in POP. For each intercepted call, the MPI wrapper decides if it should be forwarded to the local MPI implementation, or if it should be send to another cluster. To use the MPI wrapper code, POP needs to be recompiled using a different MPI library but no changes to the POP code itself are required.

To communicate between clusters, one or more support processes, so-called hubs, are used. Each hub typically runs on the cluster frontend, and serves as a gateway to the other clusters. If necessary multiple hubs can be connected together to circumvent communication restrictions caused by firewalls. In Fig. 6, the left panel shows a traditional POP run on a single machine, while the right image illustrates how a hub is used in DAS4 to connect two clusters together. Only a single hub is needed, as all compute nodes in DAS4 can communicate with all head nodes, even those of other clusters.

GMDD
6, 4705–4744, 2013

However, compute nodes cannot directly communicate with compute nodes in other clusters.

3.3 Performance

Table 2 shows the configurations of the partitioning schemes. For each experiment we use 256 MPI tasks. The cartesian distribution uses a 225 × 150 block size, resulting in exactly one block per MPI task (no land blocks are discarded). Both rake and space-filling curve use a block size of 60 × 60 and discard 628 of 2400 blocks (i.e., 26 %). The table also shows the minimum, average, and maximum communication per MPI task, and the amount of traffic generated between the clusters for the two cluster experiment. We will discuss these below. As can be seen from Table 2, the hierarchical domain distribution significantly decreases the amount of traffic between the clusters compared to rake and space-filling curve. As a result, the performance overhead of using two clusters is limited.

The performance results of POP are shown in Fig. 7 in model days/day. On Huygens and single cluster DAS4, the rake and space-filling curve block distribution clearly improve the performance over the cartesian distribution. On Huygens, the performance improvement of space-filling curve is close to the amount of work discarded (23 % vs. 26 %). On DAS4 the improvement is much greater (54 % vs. 26 %) due to the better cache behavior of smaller blocks. The space-filling curve distribution outperforms the rake distribution in all cases, due to the better load balancing characteristics, as shown in Table 2. Figure 7 also shows that for the two cluster DAS4 experiments the performance degrades. Interestingly, the performance reduction for cartesian is only 10 %, while space-filling curve (41 %) and rake (44 %) are much more effected. This difference is caused by the increased communication caused by these two block distributions, as shown in Table 2.

Although rake and space-filling curve both decrease the amount of work per MPI task, they also significantly increase the amount of communication, between tasks. On supercomputers, where POP is traditionally run, this problem is mitigated by high-

GMDD
6, 4705–4744, 2013

speed network interconnects, but in a multi-cluster environment, the internet link between clusters becomes a bottleneck. In Table 2, the column "communication between clusters" clearly shows that compared to cartesian, rake causes a 3.4x increase in the communication between clusters. The increase caused by the space-filling curve is smaller, a factor 2.1, but still significant.

The hierarchical scheme performs slightly better than the space-filling curve scheme on Huygens and single cluster DAS4 (Fig. 7). This is to be expected, as the communication overhead is small on these systems due to the fast local network interconnects. On two cluster DAS4, however, the hierarchical domain distribution provides a significant performance improvement over the existing algorithms. When running on two clusters, the performance drop compared to a single cluster run is only 8 % for the hierarchical domain distribution, compared to 10 % for cartesian, 41 % for space-filling curve and 44 % for rake.

Table 3 shows the speedup on DAS4 compared to a 16-node run on a single cluster. The speedup on 32-nodes on a single cluster is, with a factor of about 1.9, almost perfect. Although the speedup on two clusters (of 16-nodes each) is slightly lower, about a factor 1.7, the performance gain compared to a single cluster is still significant. These results clearly demonstrate that using multiple clusters can be beneficial, especially to increase the number of machines beyond the size of a single cluster.

4 Execution on graphics processing units

This section discusses the main challenges that exist when moving parts of the computation in POP to a GPU. We use the CUDA programming model (Nvidia, 2013) to have fine-grained control over our GPU implementation and to be able to explain and improve performance results. Many different software tools, libraries, (directives-based) parallelization tools, and compilers aim to assist in the development of GPU code. However, it is our goal to gain a deep understanding of the performance behavior of POP, which requires more control over the implementation and in particular how data

GMDD
6, 4705–4744, 2013

is transferred between the host memory and GPU device memory. In the following sections, we use CUDA terminology (Nvidia, 2013), although our methods could just as easily apply to OpenCL (Khronos Group, 2013).

POP consists of a large Fortran 90 codebase and in this paper we therefore limit ourselves to the most compute intensive parts of the program and only offload those computations to the GPU. The main challenge with this approach is to overcome the PCIe bus bottleneck. Whenever computations are to be performed on the GPU the input and output data has to be transferred from host memory through the PCIe bus to GPU device memory and vice versa. The achieved bandwidth to GPUs connected through the PCIe 2.0 bus is approximately 5.7 $\mathrm{GB\,s^{-1}}$ from host to device and 6.3 $\mathrm{GB\,s^{-1}}$ from device to host. This is significantly lower than the bandwidth between host memory and a CPU and the bandwidth between GPU device memory and the GPU. Therefore, it is crucial that we maximize the overlap of data transfers to the GPU with computation and with transfers from the GPU back to the host.

To overlap GPU communication and computation we need fine-grained control over how data is transferred to the GPU. There are several alternative techniques for moving data between host and device using the CUDA Programming model. The most commonly used approach is to simply use *explicit memory copy statements* to transfer large blocks of memory to and from the GPU.

Alternatively, *CUDA streams* may be used to separate the computation into distinct streams that may execute in parallel. This way, communication from one stream can be overlapped with computation and communication in other streams. GPUs with 2 copy engines, such as Nvidia's Tesla K20, can use the PCIe bus in full duplex with explicit memory copies in different streams. This way, communication and computation from different streams can be fully overlapped.

Finally, the *mapped memory* approach uses no explicit copies, but maps part of the host memory into device memory space. Whether this approach is feasible depends on the memory access pattern of the kernel. Typically mapped memory can only be used efficiently if each input and output element is read or written only once by the

GMDD
6, 4705–4744, 2013

GPU function, called *kernel*. Although this approach results in very clean host code, requiring no explicit copy statements, it requires complex kernel implementations with intricate memory access patterns to ensure high performance.

4.1 Targets for GPU implementation

To determine which part of POP to port to the GPU, we must first get an impression of where the most time is spend. It is well-known that the three-dimensional baroclinic solver is the most computationally intensive part of POP (Kerbyson and Jones, 2005; Worley and Levesque, 2003). We therefore limit ourselves to analyzing the performance of the baroclinic solver.

Table 4 gives an overview of the most time consuming functions in POP. These profiling results of are obtained from one month of simulation using the $R_{0.1}$ version (see beginning of Sect. 2) on the DAS-4 cluster (described in Sect. 2). For this experiment we have used a cartesian distribution with blocks of size 255×300 and 8 processes per node on 16 nodes.

Table 4 lists the percentage of the total execution time spent in this function, not including subfunctions. All functions in Table 4, except those from the module pop solversmod, belong to the baroclinic solver. Our profiling results indicate that the baroclinic solver does not contain any true computational hotspots. That is, no individual function consumes a major part of the computation time.

However, the density computations from the equation of state are requested by several different parts both within the baroclinic solver and at the end of each time step. The computation of water densities is required so frequently by the model that their computation time consumes 15.09 % of the total execution time on average.

The functions from the vmix_kpp module in Table 4 are part of the computation of the vertical mixing coefficients for the KPP mixing scheme (Large et al., 1994), which in total consumes about 35.3 % of the total execution time. We therefore focus on obtaining a GPU implementation for the equation of state and for the computation of vertical mixing coefficients, in particular the three functions state, buoydiff (the computation of

GMDD
6, 4705–4744, 2013

buoyancy differences) and ddmix. We focus on buoydiff() and ddmix() since they are among the most compute intensive functions and are responsible for 64.9 % of the calls to state().

It is well-known that kernel-level optimizations focused on increasing computation throughput are generally not worthwhile when memory bandwidth is the primary factor in limiting performance (Ryoo et al., 2008). A frequently used tool for performance analysis on multi and many-core hardware using the Roofline model (Williams et al., 2009) is the *arithmetic intensity*. For example, the Nvidia Tesla K20 GPU has a theoretical peak performance of 1173 GFLOP s^{-1} for double precision and a theoretical peak global memory bandwidth of 208 GB s^{-1}. However, in practice the achieved memory bandwidth is (roughly) 160 GB s^{-1}, as reported by the bandwidthTest tool in the Nvidia CUDA SDK. A rough estimation tells us that an arithmetic intensity of at least 7.3 FLOP byte^{-1} is required for the kernel to become compute-bound. Thus, if the arithmetic intensity is less than 7.3 FLOP byte^{-1}, we know the kernel is memory bandwidth-bound when executed on the K20.

The arithmetic intensity of the state() function is computed as follows. Although POP supports various implementations for the equation of state, we focus on the MWJF 25-term equation of state (McDougall et al., 2003) because it is the most commonly used implementation. The MWJF state() function requires the temperature and salinity tracers as inputs as well as 25 coefficients, of which 6 depend on the water pressure, the rest is constant. The state() function outputs the density of water and optionally also outputs the derivatives of the water density with respect to temperature and salinity. When only the density of water is computed, state() performs 40 floating point operations per grid point with an arithmetic intensity of 2.5 FLOP byte^{-1}, assuming that all 25 coefficients can be stored in on-chip caches and can be fully reused. When all outputs are requested, 89 floating point operations are executed per grid point, resulting in an arithmetic intensity of 5.56 FLOP byte^{-1}. With an arithmetic intensity of either 2.5 or 5.56, the MWJF state() kernel is memory bandwidth-bound. Therefore, we focus

GMDD
6, 4705–4744, 2013

on optimizing the time spent on communication between host and device, rather than kernel-level optimizations.

4.2 Efficient integration of GPU code

We now describe how POP should be revised to efficiently use GPUs. For our discussion, we focus on three functions in POP state(), buoydiff(), and ddmix(). Due to a lack in GPU performance models that consider asynchronous PCIe transfers, it is currently impossible to predict what kind of implementation will be the most efficient. For each function we have therefore implemented three different versions that we call *Explicit*, *Implicit*, and *Streams*. We first describe the three versions in general and then discuss the specific implementations for state(), buoydiff(), and ddmix() in detail.

Explicit is a bulk synchronous implementation that uses explicit memory copy statements to copy all the required input data to GPU and from the GPU for the entire three-dimensional grid. The kernel used in *Explicit* creates a two-dimensional array of threads, i.e. one thread for each horizontal grid point, which iterate the grid points in the vertical dimension. *Implicit* uses mapped memory and therefore requires no explicit memory copy statements. Instead, data is requested by the GPU directly from the host memory and sent over the PCIe bus. The performance of accessing the memory in this way is very sensitive to the order in which data is requested and care must be taken not to create gaps or misalignments from the mapping between threads and data. Therefore, *Implicit* uses a kernel implementation that creates a one-dimensional array of threads with size equal to the number of grid points in the three-dimensional grid. Each thread then computes its three-dimensional index from its one-dimensional thread ID to direct itself to the correct part of the computation. The *Streams* implementation creates one stream for each vertical level and uses explicit copy statements to copy the corresponding vertical level of the input and output variables to and from the GPU. If the computation of one vertical level requires input from multiple vertical levels, CUDA events are used to delay the computation until all inputs have been moved to the device and vice versa. The kernel used in *Streams* is similar to the kernel used in

GMDD
6, 4705–4744, 2013

Explicit, except for the fact that the kernel only computes the grid points of one vertical level. Note that, except for the differences described here, the kernels do not contain any architecture specific optimizations.

While the state() function computes the density of water at a certain vertical level k, the function is mostly used directly surrounded by a loop over all vertical levels. These code blocks can safely be replaced by a call to a single function that directly computes the water densities for all vertical levels. Our *Explicit* implementation uses explicit copies to move the three-dimensional grid of tracer values between host and device and creates one thread for each horizontal grid point, which compute all outputs in the vertical direction. However, this approach is unable to overlap communication to and from the device with GPU computation. It is possible to also parallelize the computation of different vertical levels using CUDA streams. Our *Streams* implementation ensures that GPU computation can be overlapped with GPU communication of different vertical levels, and thus alleviates the PCIe bus bottleneck to a large extent. Because of the simple access pattern in state(), where each input and output element is read or written only once, it is also a good candidate for the highly parallel *Implicit* implementation.

More complex uses of the equation of state are found within the computation of the vertical mixing coefficients for the KPP mixing scheme (Large et al., 1994), in particular, in the computation of buoyancy differences (buoydiff) and double-diffusion diffusivities (ddmix). In POP the vertical mixing coefficients are sequentially computed for all vertical levels. The computation of buoyancy differences at level k requires the density of both the surface level and level $k-1$ displaced to level k, as well as the water density at level k. These values can be computed for each level in parallel as long as all the data is present on the GPU. Overlapping data movement from the host to the GPU with GPU computation and data movement from the GPU to host becomes significantly more difficult, because the tracers for levels 1, $k-1$, and k need to be present on the GPU to compute the buoyancy differences at level k. The *Streams* implementation first schedules memory copies to the GPU for all vertical levels in concurrent streams and then

GMDD
6, 4705–4744, 2013

invokes GPU kernel launches for all levels. However, before the execution of the kernel in stream k can start, the memory copies in stream 1, $k-1$, and k need to be complete. The kernel executing in stream k outputs to different vertical levels for different variables. Therefore, some of the memory copies from device to host in stream k have to wait for the kernel in stream $k-1$ to complete. We use the CUDA event management functions to guarantee no computations or memory transfers start prematurely.

In the ddmix function, the computation of diffusivities at level k requires the derivatives of density with respect to temperature and salinity at level k and $k-1$. That is, the computation of level k reuses the derivatives that were used to compute level $k-1$. At a first glance, it would seem that the computation of all vertical levels cannot be parallelized. The sequential approach prevents that these values have to be recomputed, but inhibits the ability to overlap communication and computation of different vertical levels. Therefore, our implementation also parallelizes the computation in the vertical dimension by introducing double work. The cost of computing the derivatives twice is significantly less than the inability to overlap computation and communication. Similarly to the buoyancy differences computation, the kernel executing in stream k requires the memory copies of stream k and $k-1$ to be complete. Again, CUDA event management functions are used to guarantee no data is copied from the GPU back the host before GPU computations have finished.

5 Performance of POP on GPUs

In this section, we will describe the performance of the $R_{0.1}$ version of POP on a single cluster and on multiple GPU clusters. In the first subsection below, we focus on the performance impact on individual POP subroutines when using a GPU. In the second subsection, we address the performance of the whole POP code on a single GPU and on multiple GPU clusters.

GMDD
6, 4705–4744, 2013

5.1 Performance impact of GPU usage: individual routines

First we evaluate the performance of single functions that were taken out of POP for individual benchmarking. We test our three implementations (*Explicit, Implicit* and *Streams*) for each discussed function of POP on a single node equipped with a Nvidia Tesla K20 GPU in the DAS4 cluster. The Tesla K20 has 2496 CUDA cores running at 705 MHz providing a theoretical peak double precision performance of 1173 GFLOP s^{-1}. The K20 has 5 GB of device memory and a theoretical peak memory bandwidth of 208 GB s^{-1}. The K20 is connected through a PCIe 2.0 bus and has 2 copy engines which enable full duplex use of the PCIe bus for concurrent explicit memory transfers. The grid dimensions used for the experiments discussed here are $229 \times 304 \times 42$. This is the same block size as used to obtain our profiling results, with 2 ghost cells in both horizontal dimensions. For all three implementations the majority of the execution time is spent on transferring the data to and from the GPU. For example, for the *Streams* implementation of state() only 10.3 % of the execution time is spent on GPU computation, and only 19.4 % and 13.3 % for buoydiff() and ddmix(), respectively.

Figure 8 shows the performance results for all three functions with three different GPU implementations. For the state() function the *Implicit* implementation provides the best performance. Although the kernel implementation used by *Implicit* is slightly less efficient than the kernel used by *Explicit*, the total execution time is significantly less because a large part of the memory transfers between host and device and computation are overlapped. While *Streams* achieves overlapping behavior similar to *Implicit*, it is more coarse-grained with one vertical level at a time, rather than individual grid points. That explains why *Implicit* outperforms the *Streams* implementation for the state() function.

The buoydiff() function has a very low arithmetic intensity and therefore the computation again accounts for only a small part of the total execution time. The *Implicit* implementation is slower than *Explicit* because the access pattern in buoydiff() requires several input elements multiple times. As a result, the *Implicit* approach transfers more

GMDD
6, 4705–4744, 2013

data than necessary over the PCIe bus. Although these transfers can be overlapped with computation and with transfers in the opposite direction, the performance penalty for transferring data multiple times reduces the overall performance. The *Streams* approach again benefits from the fact that data transfers and computation can be overlapped, but without the restrictions that come with the *Implicit* approach. The data access pattern in buoydiff() requires that operations in some streams may have to wait for operations in another stream to complete before they can start. The overhead of these synchronizations accounts for on average 3.26 % of the total execution time of the *Streams* implementation.

To parallelize the computation of ddmix() in the vertical dimension the *Implicit* and *Streams* implementations do some double work. That is, some values are computed twice by different threads operating at different vertical levels, whereas a thread in the *Explicit* approach may reuse that value from the computation of a previous vertical level. Therefore, the time spent in computation for *Implicit* and *Streams* is higher than that of *Explicit*. However, due to the overlap of computation and PCIe transfers in both directions, both *Streams* and *Implicit* do outperform the *Explicit* implementation in terms of total execution time. The *Implicit* implementation again suffers from the fact that, although overlapped with communication and computation, data has to be transferred multiple times through the PCIe bus.

In the GPU implementation of the POP we use in the next subsection, the *Implicit* implementation for state() and the *Streams* implementation for buoydiff() and ddmix() is used. As buoydiff() is executed before ddmix() as part of the computation of vertical mixing coefficients, ddmix() reuses the tracers that have been copied to the GPU by buoydiff(). Additionally, for all three functions the execution on the GPU as well as all data transfers are overlapped with the computation of other functions on the CPU. Therefore, the CPU never has to wait for the results of GPU computations.

GMDD
6, 4705–4744, 2013

5.2 Performance of POP on multiple (GPU) clusters

For the evaluation of POP incorporating the implementation of some of the functions on a GPU we use the DAS4 (described earlier in Sect. 3.1). First, 8 compute nodes each containing two quad-core Intel E5620 CPUs (8 cores per node total) running at 2.4 Ghz, 24 GBytes of memory, and a Nvidia GTX480 GPU are used. In addition, we also use 8 compute nodes each containing two six-core Intel E5-2620 CPUs (12 cores per node total) running at 2.0 Ghz, 64 Gbyte of memory, and a Nvidia Tesla K20 GPU each. We employ the new hierarchical partitioning scheme described in Sect. 2.3 using a block size of 60×60.

Figure 9 shows the performance of POP using 4, 8 and 12 MPI tasks per node, with and without GPU. Note that only a single GPU is available in each node. Therefore, the GPU is shared between the multiple MPI tasks on a single node. For the 8-core DAS4 nodes, the performance gained by using the GPU is approximately 20 %, both when using 4 or 8 MPI tasks. This directly corresponds with the execution time consumed by POP code that has been ported to the GPU. The figure also shows that the scalability of POP itself is far from perfect. Running on 8 MPI task per node, only provides a speedup of 1.4 compared to 4 MPI tasks per node, both for the CPU-only and GPU versions.

For the 12-core DAS4 nodes, the performance gained by using the GPU is approximately 15 % when using 4 MPI tasks per node, and 13 % when using 8 or 12 MPI tasks per node. Although this relative performance gain is lower that for the 8-core nodes, the absolute performance gain is much higher, due to the better performance offered by the (newer) six-core CPU and K20 GPUs. In addition, the scalability of POP on the 12 core nodes is also much better, achieving a speedup of 1.9 on 8-cores and 2.6 on 12-cores (both relative to the 4-core experiment).

Additionally, these results show that it is possible to combine the hierarchical partitioning scheme with GPU execution and still obtain a performance increase. This is a remarkable result, as the hierarchical partitioning scheme prefers small block sizes to eliminate many land blocks and distribute load evenly among MPI tasks, while the

GMDD
6, 4705–4744, 2013

GPU code prefers large block sizes to increase GPU utilization. However, GPU utilization is again increased by the fact that all MPI tasks running on a single node share a single GPU for all their GPU computations. It is important to understand that this would not have been possible with larger block sizes, because of the limited size of the GPU memory. As such, the two approaches presented in this paper work in concert to improve the performance of POP.

As a final experiment, we study the performance of POP on multiple platforms including GPUs. For this experiment, we use 8-core DAS4 compute nodes with an Nvidia GTX480 GPU (described in Sects. 3.1 and 5.2).

Figure 10 compares the performance of a 16-nodes single cluster run with a 2×8-node two cluster run. Results are shown for CPU-only and CPU plus GPU experiments. The results show a performance increase when using the GPUs of 15 % on one cluster and 13 % on two clusters. The performance loss when going from one to two clusters is 5 % for the CPU-only version and 6 % for the CPU+GPU version. These results clearly indicate that running POP on multiple GPU clusters is feasible and also beneficial in terms of performance. Moreover, it allows users with access to multiple smaller GPU clusters to scale up to well beyond the size of a single GPU cluster.

6 Summary, discussion and conclusions

High-resolution ocean and climate models are becoming a very important tool in climate research. It is crucially important that multi-century simulations with these models can be performed efficiently. In this paper, we presented a new distributed computing approach to increase the performance of the POP model.

We first have shown that it is possible to optimize the load balancing of POP such that it can run successfully in a multi-platform setting. The new hierarchical load balancing scheme was shown to perform much better than the existing load balancing schemes (cartesian, rake and space-filling curve), mainly due to the reduction in communication between the MPI tasks. Second, it was demonstrated that it is advantageous to port

GMDD
6, 4705–4744, 2013

part of POP to GPUs (and get a performance increase), even though POP itself does not contain any real hotspots and is therefore not an obvious candidate for using GPUs.

In the experiments shown, only three functions in POP were implemented on a GPU. Another substantial portion of the execution time is spent computing the advection of momentum and the horizontal diffusion of momentum and tracers. Obtaining a GPU implementation for these functions is deferred to future work. The advection of tracers also uses the equation of state to compute the potential density referenced to the surface layer, which is used to compute a variety of time averaged fields. Currently, the majority of the execution time is spent on PCIe transfers. When more computation is moved to the GPU more data can be reused, and some intermediate data structures that result from computation may even never have to leave the GPU. In that case, some PCIe transfers can be eliminated completely. In future work we hope to produce a complete GPU implementation of the vertical mixing part of POP.

The software presented in this paper has the same portability properties as the original POP. The GPU code is written in CUDA, which is a widely used language for GPU Computing applications. To increase portability across different GPU architectures no architecture specific optimizations have been included. OpenCL is a well-known alternative to CUDA that aims at a wider set of many-core compute devices and different compilers are available for different platforms. However, there are no real linguistic differences between CUDA and OpenCL and porting the code will be a simple engineering effort, automated source-to-source translation tools are also available. The use of both extensions (domain decomposition or GPU functions) can be enabled, disabled, and controlled individually through the well-known pop_in namelist file.

Finally, we have shown that the combination of these two approaches also improves performance. Although we demonstrated this only for the DAS4 cluster, it opens up the possibility to submit a POP job in the near future over multiple supercomputing platforms (with or without GPUs). The new hierarchical load balancing scheme and the MPI wrapper methodology are crucial elements for maintaining the performance of

GMDD
6, 4705–4744, 2013

POP. Future work is to port more of POP to GPUs and to scale up the multi-cluster experiments to production size hardware.

Acknowledgements. This publication is part of the eSALSA project (An eScience Approach to determine future Local Sea-level chAnges) of the Netherlands eScience Center (NLeSC), Institute for Marine and Atmospheric research Utrecht (IMAU) at Utrecht University, and VU University Amsterdam. This publication was supported by the Dutch national program COMMIT. We thank SURFsara (http://www.surfsara.nl) for the support in using the Dutch national super computer Huygens.

References

Bleichrodt, F., Bisseling, R., and Dijkstra, H. A.: Accelerating a barotropic ocean model using a GPU, Ocean Model., 41, 16–21, doi:10.1016/j.ocemod.2011.10.001, 2012. 4707

Dennis, J. M.: Inverse space-filling curve partitioning of a global ocean model, IPDPS 2007, IEEE International, 1, 1–10, doi:10.1109/IPDPS.2007.370215, 2007. 4707, 4709, 4710, 4736

Dukowicz, J. K. and Smith, R. D.: Implicit free-surface method for the Bryan-Cox-Semtner ocean model, J. Geophys. Res., 99, 7991–8014, doi:10.1029/93JC03455, 1994. 4708

Kerbyson, D. J. and Jones, P. W.: A performance model of the parallel ocean program, Int. J. High Perform. C., 19, 261–276, doi:10.1177/1094342005056114, 2005. 4719

Khronos Group: OpenCL, available at: http://www.khronos.org/opencl/ (last access: August 2013), 2013. 4718

Large, W. G., McWilliams, J. C., and Doney, S. C.: Oceanic vertical mixing: a review and a model with a nonlocal boundary layer parameterization, Rev. Geophys., 32, 363–403, doi:10.1029/94RG01872, 1994. 4719, 4722

Maassen, J. and Bal, H. E.: Smartsockets: solving the connectivity problems in grid computing, in: Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC), Monterey, CA, USA, 1–10, doi:10.1145/1272366.1272368, 2007. 4715

Maltrud, M., Bryan, F., and Peacock, S.: Boundary impulse response functions in a century-long eddying global ocean simulation, Environ. Fluid Mech., 10, 275–295, doi:10.1007/s10652-009-9154-3, 2010. 4706, 4708

GMDD

6, 4705–4744, 2013

Marquet, C. P. and Dekeyser, J. L.: Data-parallel load balancing strategies, Parallel Comput., 24, 1665–1684, doi:10.1016/S0167-8191(98)00049-0, 1998. 4709

McDougall, T. J., Jackett, D. R., Wright, D. G., and Feistel, R.: Accurate and computationally efficient algorithms for potential temperature and density of seawater, J. Atmos. Ocean. Tech., 20, 730–741, doi:10.1175/1520-0426(2003)20<730:AAACEAF>2.0.CO;B2, 2003. 4720

Nvidia: CUDA Programming Guide, available at: http://docs.nvidia.com/cuda/ (last access: August 2013), 2013. 4717, 4718

Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and Hwu, W.-M. W.: Program optimization space pruning for a multithreaded GPU, in: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, doi:10.1145/1356058.1356084, pp. 195–204, 2008. 4720

Smith, R., Jones, P., Briegleb, B., Bryan, F., Danabasoglu, G., Dennis, J., Dukowicz, J., Eden, C., Fox-Kemper, B., Gent, P., Hecht, M., Jayne, S., Jochum, M., Large, W., Lindsay, K., Maltrud, M., Norton, M., Peacock, S., Vertenstein, M., and Yeager., S.: The Parallel Ocean Program (POP) Reference Manual: Ocean Component of the Community Climate System Model (CCSM), 2010. 4710, 4713, 4735

Smith, R. D., Maltrud, M. E., Bryan, F. O., and Hecht, M. W.: Numerical simulation of the North Atlantic Ocean at $\frac{1}{10}^{\circ}$, J. Phys. Oceanogr., 30, 1532–1561, 2000. 4706

Vallis, G. K.: Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation, Cambridge University Press, Cambridge, UK, 2006. 4706

Weijer, W., Maltrud, M. E., Hecht, M. W., Dijkstra, H. A., and Kliphuis, M. A.: Response of the Atlantic Ocean circulation to Greenland Ice Sheet melting in a strongly-eddying ocean model, Geophys. Res. Lett., 39, L09606, doi:10.1029/2012GL051611, 2012. 4708

Williams, S., Waterman, A., and Patterson, D.: Roofline: an insightful visual performance model for multicore architectures, Commun. ACM, 52, 65–76, doi:10.1145/1498765.1498785, 2009. 4720

Worley, P. and Levesque, J.: The performance evolution of the parallel ocean program on the Cray X1, in: Proceedings of the 46th Cray User Group Conference, 17–21, 2003. 4719

GMDD
6, 4705–4744, 2013

Table 1. Permutations of the [3,3,2,2] example distribution, showing the number of assigned blocks and the communication per task in grid points per level. The entries are sorted by average communication per task. The top most entry provides the best solution.

permutation	blocks per task	communication per task (min/avg/max)
(3,3,2,2)	10	1440/2186/2888
(2,3,3,2)	10	1244/2187/2888
(2,2,3,3)	10	1240/2188/3100
(2,3,2,3)	10	1240/2188/3300
(3,2,2,3)	10	1240/2229/3720
(3,2,3,2)	10	1440/2265/2876

GMDD
6, 4705–4744, 2013

Table 2. Configuration of the cartesian, rake and space-filling curve and hierarchical distributions.

algorithm	block size	blocks per core (min/max)	blocks discarded	communication per task (min/avg/max)	communication between clusters (messages/volume)
cartesian	225 × 150	1/1	0 (of 256)	0/1267.4/2408	22.3 M/99.0 GB
rake	60 × 60	5/8	628 (of 2400)	748/1940.5/3936	77.9 M/337.4 GB
space-filling curve	60 × 60	6/7	628 (of 2400)	1007/1707.7/2960	41.0 M/212.7 GB
hierarchical	60 × 60	6/7	628 (of 2400)	504/1394.9/2584	20.0 M/82.5 GB

GMDD
6, 4705–4744, 2013

Table 3. Speedup on DAS4 for one and two cluster configurations using a hierarchical domain distribution.

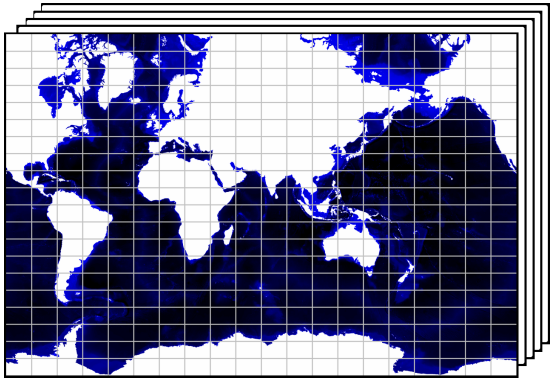
Configuration	Performance (modeldays/day)	Speedup
1 cluster, 16 nodes	82	1.0
1 cluster, 32 nodes	155	1.9
2 clusters, 16 nodes each	142	1.7

GMDD
6, 4705–4744, 2013

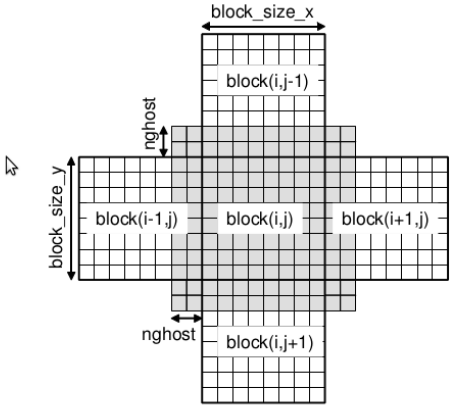
Table 4. List of the most compute intensive functions in POP, covering 76.48 % of the total computation time. The reported time does not include time spent in functions called by this function.

% time	function	module	#calls	computes
15.09	state	state_mod	29562112	density of water and derivatives
6.69	hdiffu_del4	hmix_del4	4865280	horizontal diffusion of momentum
5.79	advu	advection	4865280	advection of momentum
5.33	bldepth	vmix_kpp	115840	ocean boundary layer depth
5.25	hdifft_del4	hmix_del4	4865280	horizontal diffusion of tracers
4.62	chrongear	pop_solversmod	115840	preconditioned conjugate-gradient solver
4.07	ri_iwmix	vmix_kpp	115840	viscosity and diffusivity coefficients
3.83	vmix_coeffs_kpp	vmix_kpp	115840	vertical mixing coefficients
3.66	impvmixt_correct	vertical_mix	115840	implicit vertical mixing corrector step
3.34	blmix	vmix_kpp	115840	mixing coefficients within boundary layer
3.27	impvmixt	vertical_mix	231680	implicit vertical mixing of tracers
3.27	clinic	baroclinic	4865280	forcing terms of baroclinic momentum
3.17	advt_centered	advection	4865280	tracer advection using centered differencing
3.12	btropoperator	pop_solversmod	14705152	applies operator for the barotropic solver
3.10	baroclinic_driver	baroclinic	115840	integration of velocities and tracers
2.88	ddmix	vmix_kpp	115840	add double-diffusion diffusivities

GMDD
6, 4705–4744, 2013



(a)



(b)

Fig. 1. (a) Sketch of the blockwise subdivision of the domain in POP. **(b)** The halo regions of a block; image from Smith et al. (2010).

GMDD
6, 4705–4744, 2013

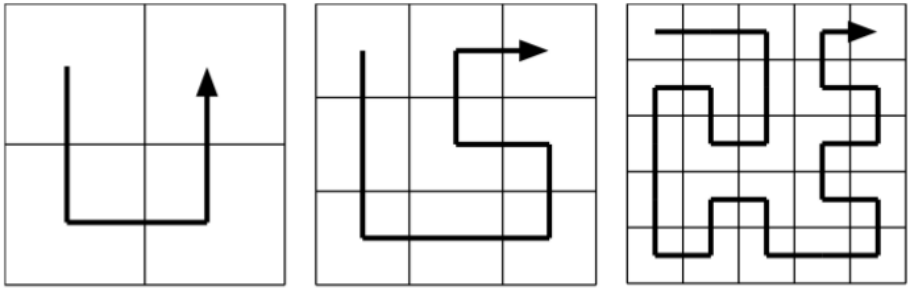


Fig. 2. Examples of the space-filling curve load balancing algorithm, with the Hilbert (left panel), meandering Peano (middle panel) and Cinco (right panel) curves; image from Dennis (2007).

GMDD
6, 4705–4744, 2013

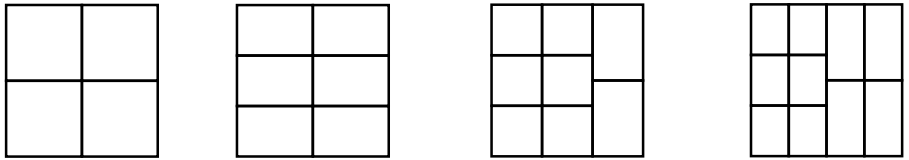


Fig. 3. Example subdivisions of a square into 4, 6, 8, and 10 rectangular sections.

GMDD
6, 4705–4744, 2013

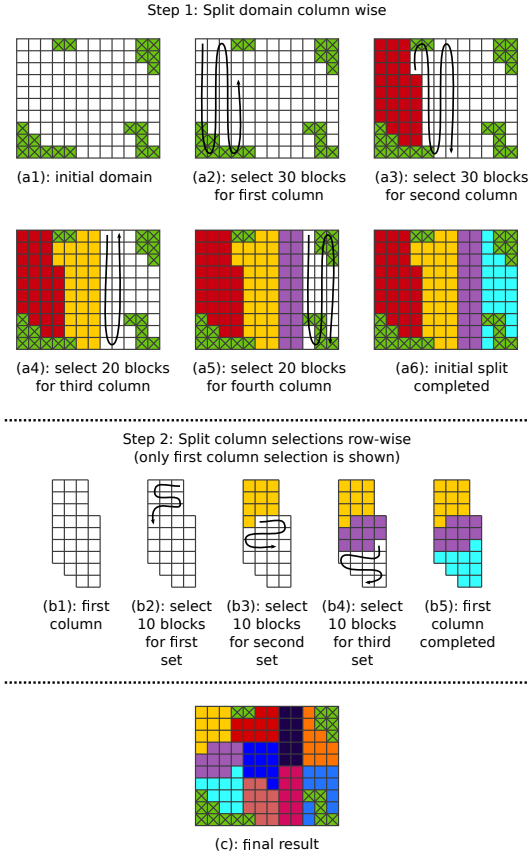


Fig. 4. Description of the hierarchical load balancing scheme for an example of 12 × 10 blocks, of which 20 are land-only blocks, as shown in panel **(a1)**. The initial columnwise split is shown in panels **(a2)**–**(a6)**, the next row wise split in the panels **(b1)**–**(b5)** and the final results is shown in panel **(c)**.

GMDD
6, 4705–4744, 2013

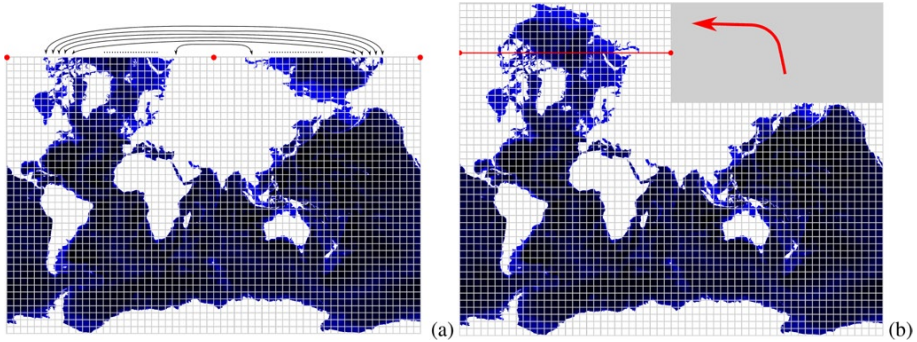


Fig. 5. (a) A subdivision of the topography into 60 by 40 blocks. The two tripoles are depicted by the red dots on the upper boundary. Note that the left-most and right-most dots represent the same tripole; the tripole communication is (partially) shown by the arrows. **(b)** A remapping of the grid that moves an area of 30 × 7 blocks. The original tripole boundary is shown as a red line.

GMDD
6, 4705–4744, 2013

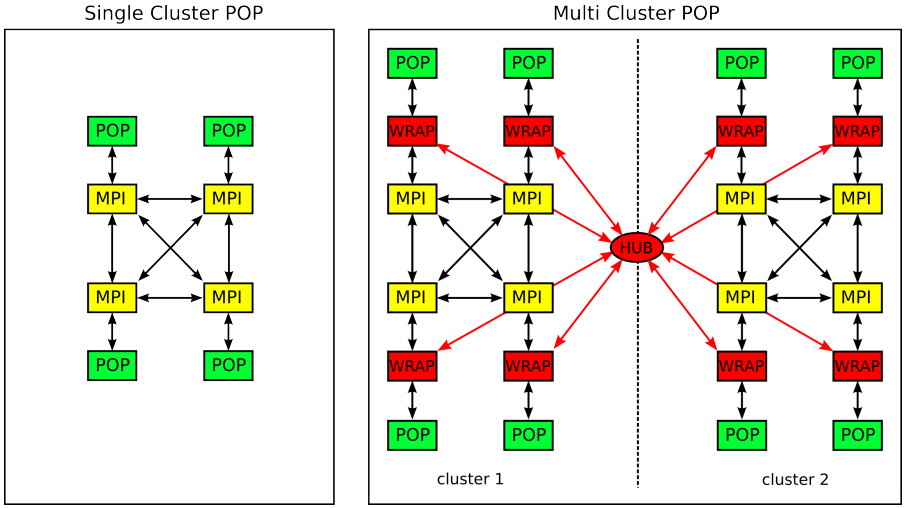


Fig. 6. An example of POP running without the MPI wrapper on a single cluster (left panel) and with the MPI wrapper on a multi cluster (right panel).

GMDD
6, 4705–4744, 2013

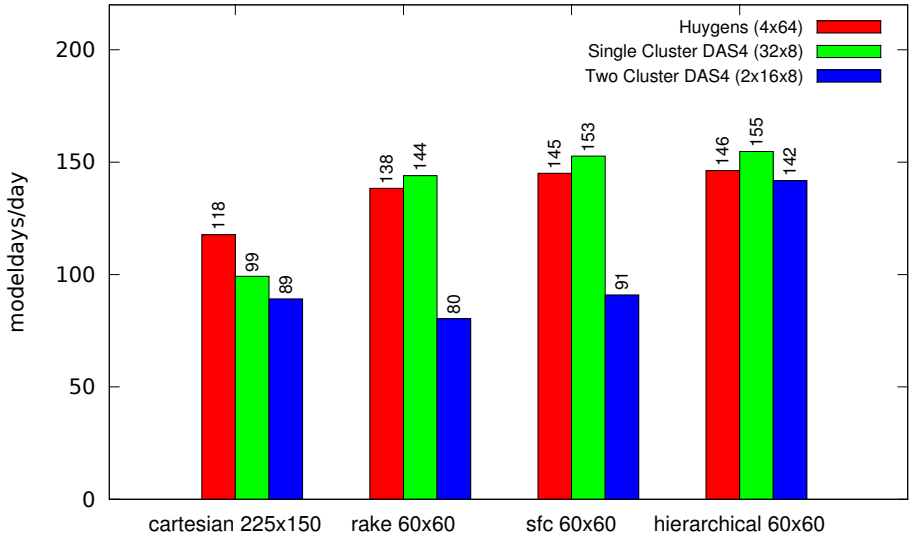


Fig. 7. Performance comparison of POP using cartesian, rake, space-filling curve and hierarchical block partitioning schemes on three different hardware configurations, each using 256 MPI tasks.

GMDD
6, 4705–4744, 2013

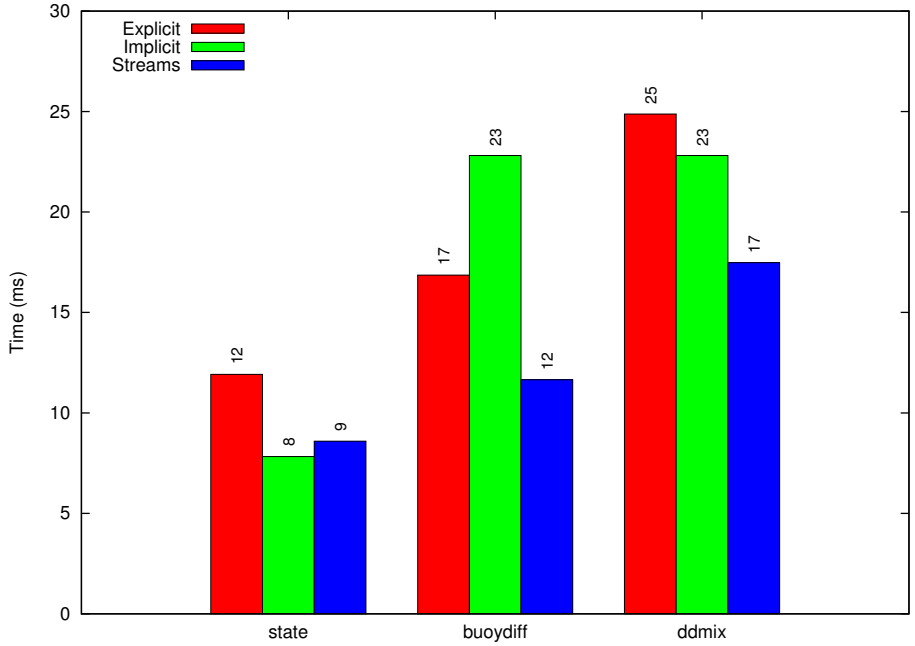


Fig. 8. Performance results for the three POP functions on a GPU with three different implementations as obtained on the Tesla K20 GPU with a 229 × 304 block size.

GMDD
6, 4705–4744, 2013

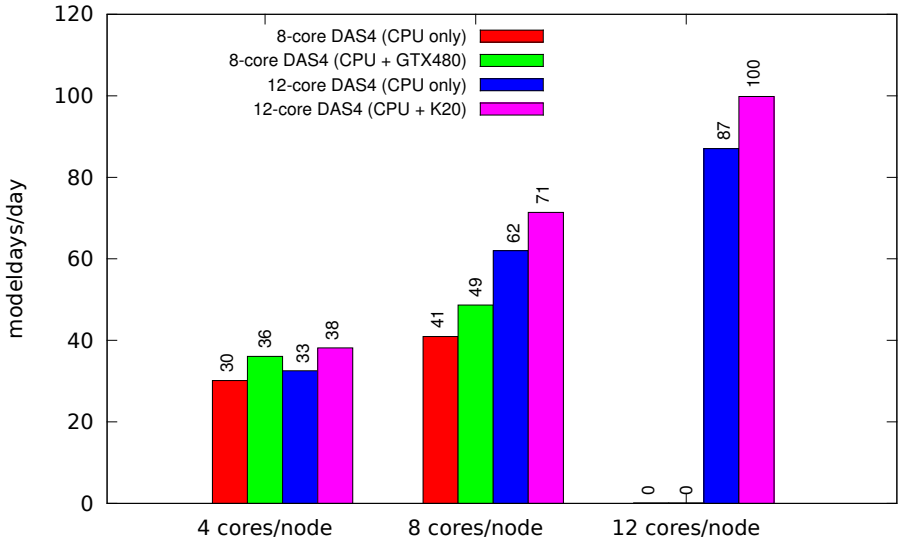


Fig. 9. Performance of POP using 8 computes nodes of the DAS4 cluster, with and without GPUs, using hierarchical partitioning with 60 × 60 block size.

GMDD
6, 4705–4744, 2013

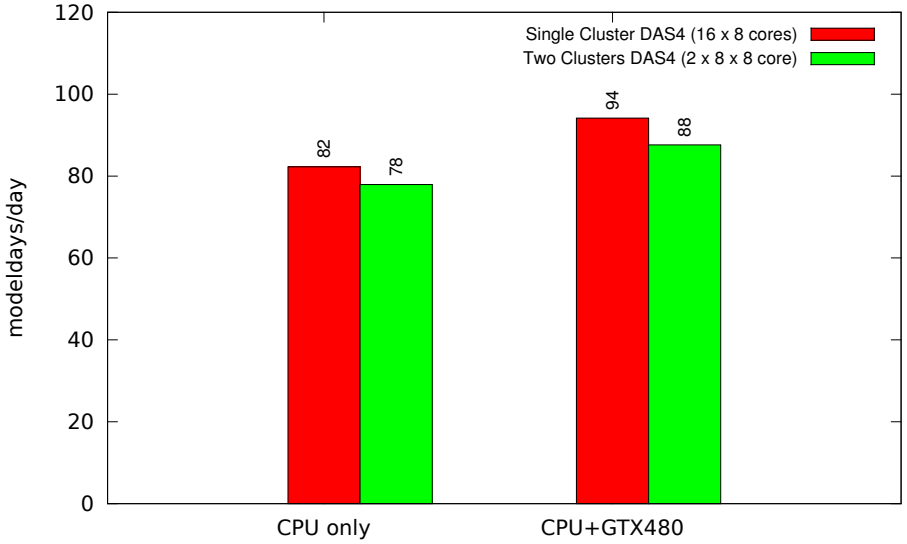


Fig. 10. Performance of POP using 16 computes nodes of the DAS4 cluster, on one or two clusters, using hierarchical partitioning with 60 × 60 block size.

GMDD
6, 4705–4744, 2013