Geosci. Model Dev. Discuss., 6, C2062–C2067, 2013
www.geosci-model-dev-discuss.net/6/C2062/2013/


Geoscientific Model Development Discussions

# *Interactive comment on* "A distributed computing approach to improve the performance of the Parallel Ocean Program (v2.1)" *by* B. van Werkhoven et al.

**B. van Werkhoven et al.**

ben@cs.vu.nl

Received and published: 9 December 2013

**MS-No.:** gmdd-6-4705-2013
**Version:** First Revision
**Title:** A distributed computing approach to improve the performance of the Parallel Ocean Program (v2.1)
**Author(s):** Van Werkhoven et al.

## Point by point reply to reviewer #2 (Mark Govett)

9 December 2013

C2062

We thank the reviewer for his/her careful reading and for the useful comments on the manuscript.

1. *I liked this paper. It describes a new domain partitioning strategy that minimizes interprocess communications across multiple nodes and yields significant performance benefit. The paper also describes efforts to parallelize select routines to run on the GPU. The time required to copy data between host (CPU) and device (GPU) is significantly more than the computation time of each routine, so several strategies were tried to reduce the copy time (explicit), by using GPU mapped memory (implicit) and cuda streams that overlap communications and computations. GPU runtimes were marginally faster but would benefit much more if more routines were run on the device to avoid or reduce communications time. It is noted the future work will be done to parallelize more routines which will reduce inter CPU-GPU communications.*

   We thank the reviewer for the positive assessment. No changes in the text.

2. *Section 2.3 describes the partitioning algorithm and references figures describing an example of the domain decomposition. In particular, Figure 7 shows significant improvement (in blue) for the two cluster runtimes. However, it seems there are edge cases that might lead to points in the domain that would be stranded in the two-stage (horizontal and vertical) partitioning scheme. This would be potentially exacerbated by land points that could surround ocean points at the edge of the model domain. For example if the domain were 180W to 180E, could there, in some cases, be ocean points at the corner of the domain (eg. NE corner) that are isolated (surrounded by land points) that might be contiguous with ocean points in the NW portion of the domain? Are there other edge cases that might require more complexity in the algorithm than was either explained or*

*shown to incorporate potentially stranded or isolated ocean points?*

The partitioning scheme shown in Figure 4 does not represent the traditional Òflood fillÓ type algorithm (which may skip isolated points). Instead, our partitioning scheme simply skips over any land points encountered when scanning in a certain direction, and continues scanning in a zig-zag fashion until the required number of ocean points have been selected. To avoid confusion we will explain this more clearly in Section 2.2.

3. *Section 2.4: You do not explain the motivation for exploring tripole grids. Are the scientific results better? Do you avoid certain scientific, computational or other issues? Please explain.*

   Tripole grids are used by many global ocean circulation models (including POP). The main benefit is that the grid spacing in the Arctic is much more uniform and the cell aspect ratios are much closer to one than in a dipole grid. We will add this motivation to the revised section 2.4.

4. *Section 4: The POP is a widely used community model written in Fortran 90. It appears select routines were rewritten in CUDA-C (assume you did not use CUDA Fortran but you did not say). You reference the CUDA programming model but that is not sufficient, please give more details. In both cases, the original source code would need to be modified so it can run on the NVIDIA GPU, which means it is less useful to the community of Fortran-based modelers and researchers - because they donÕt want to learn CUDA or maintain two copies (Fortran and CUDA) of the code. Have you considered using the openACC compilers to parallelize the routines, and based on your success parallelize other routines in POP with these compilers? That would be most beneficial to the*



*community in my view.*

The first paragraph of Section 4 of the manuscript explains why we have decided to use CUDA and not any of the existing directive-based parallelization tools, such as OpenACC. Using any directive-based parallelization tools creates another layer of abstraction, which further obscures the insights that we try to get into the performance of the program. That is why using such a tool is not fit for our intended purposes. In the revised section 4 of the manuscript we will shortly expand on this issue based on the following considerations.

There is always a trade-off between productivity and performance, regardless of the programming model. Directive-based parallelization tools like OpenACC may have allowed for a larger part of the program to be executed on the GPU. Since we have not used OpenACC to parallelize parts of POP, we cannot make any assumptions on the resulting performance. On the other hand, if we had not made such an effort to create a very efficient implementation that overlaps computation and communication, we could have translated a much larger part of the parallel ocean program using CUDA as well. We are currently not aware of the possibility to implement staged copy and execution kernels in OpenACC, but if it were possible it would require a collection of directives similar to the collection of calls to the CUDA runtime that are currently responsible for managing this. Regarding the kernel code, the CUDA code is actually surprisingly similar to the Fortran 90 code. This is because Fortran specifies operations on entire arrays, whereas in CUDA each thread computes one element in the output array. The OpenACC parallelized code would leave the kernel code in the same language as the original, but that does not mean that not the same intimate level of knowledge about the underlying architecture is required to modify that parallelized code and reason on its correctness.

5. *You do not report on speedup when communications times (with explicit, implicit or streams) is not included. This would be useful to know particularly if you or others wish to parallelize the rest of the model for GPU. Given there is such a low computational intensity, and the subroutine profile is fairly flat, parallelization of the entire model may be the best approach and as you state, may be a target for future work.*

   We did generate these numbers but have decided not to include them. Unless you rewrite the entire application to execute on the GPU and only migrate data over the PCI Express bus at the start and end of the application, you will never actually see the speedups suggested by these numbers in reality on the application level. No changes in the text.

6. *Section 5.2: Page 4726, lines 25-29 The results you show using more MPI tasks with less data per GPU block outperforms larger blocks and fewer MPI tasks. Can you explain the configuration more clearly?*

   All configurations in section 5.2 use the exact same block size of 60 $\times$ 60 horizontal grid points. The paragraph that the reviewer refers to aims to explain that a trade-off exists when choosing a particular block size. GPU utilization can be increased by using larger block sizes. However, smaller block sizes increase the total number of blocks that consist of only land points that can therefore be discarded, reducing the overall amount of work. Additionally, smaller block sizes allow for more fine-grained load balancing, as this happens at the granularity of individual blocks. This remark is not related to the configurations used in for example Figure 9 or 10, which all use a block sizes of 60 $\times$ 60. In Figure 9, for example, the configuration that uses 4 MPI tasks per node will have (on average) twice as many blocks assigned to each MPI task, as compared to



   the configuration that uses 8 MPI tasks per node. In the revised version of the manuscript we will rewrite section 5.2 to clarify these points.

7. *Are you scheduling multiple MPI tasks per GPU device? If so, you may benefit from the reduced synchronization overhead, because when one MPI task is waiting for block (kernel) level synchronization, another MPI task can be executing on the same GPU device.*

   Indeed, in the configuration with 12 MPI tasks per node, all 12 processes share a single GPU and use it simultaneously. We have inserted the synchronization points in the code in such a way that the host (CPU) thread never has to block waiting for the GPU to complete execution. All GPU execution is overlapped with execution of other routines on the CPU. This is independent of the number of MPI tasks per node. There is no reduced synchronization overhead in this context. Synchronization between the CPU thread and the GPU relates to the fact that the memory written by the GPU is ready for use by the CPU. The synchronization overhead is not proportional to the number of threads or thread blocks running on the GPU; it is solely determined by the waiting times for GPU execution and the corresponding memory transfers to complete.