Geosci. Model Dev. Discuss.,
https://doi.org/10.5194/gmd-2020-257-AC3, 2021


Geoscientific Model Development Discussions

# *Interactive comment on* "Parallelizing a serial code: open–source module, EZ Parallel 1.0, and geophysics examples" *by* Jason Louis Turner and Samuel N. Stechmann

**Jason Louis Turner and Samuel N. Stechmann**

jlturner5@wisc.edu

Received and published: 8 January 2021

Dear Anonymous Referee,

Thank you for reading our paper and providing these helpful comments. Below are detailed responses to your comments. *Your comments are shown in black italics*, the authors' response is in blue, and the authors' changes in the manuscript are in green.

*This paper presents the authors' efforts on developing a tool that can help the process of parallelizing a serial code, at the level of MPI-based parallelization. This is a traditionally interesting topic, even more interesting at the current stage of migrating*



*from homogeneous multi-core clusters to heterogeneous clusters with GPUs or other many-core accelerators.*

*This is overall a quite interesting work. The authors have also done a good job to provide not only a tool but also demonstrations in a number of different applications or kernels.*

*My major question is about the comparison with other similar efforts in the field. Parallelization, especially MPI-based parallelization, has been around for many decades. Groups from both computer science and application domains have existing projects that try to derive languages, compiler, tools to support better and easier parallelization. Therefore, an introduction of such a tool should come with a comprehensive overview of existing efforts. Also, for demonstrating the efficiency and performance of the proposed tool, comparisons should be made on both the parallel performance achieved, and the extra coding efforts needed. In the current paper, we only see results of the parallel code in the proposed tool, but not sure how good it is when compared to other similar tools, or languages, such as the Unified Parallel C project from Berkeley.*

Now included in the revised manuscript are four lengthy paragraphs, for a total of about 1.5 pages of text, to provide an overview of and comparisons with existing efforts. The first of the four paragraphs is new and is intended to describe the goals of EZP, for contrast with other existing efforts. The second of the four paragraphs was mostly present in the original manuscript, but it has been expanded and modified in the revised manuscript, and it describes efforts for automatic parallelization. The third of the four paragraphs is another new paragraph, and it focuses on interactive parallelization tools. Lastly, the fourth of the four paragraphs is a new paragraph that focuses on alternatives to MPI such as Unified Parallel C and Coarray Fortran. In these paragraphs, we have tried to more clearly describe the goal of EZP to create a parallel code that looks as similar as possible to the original serial code, by grouping together the MPI commands into a few subroutines calls.

(LINES 44 - 94)

The goal of EZP is to break the parallelization process into a few high-level tasks. For the scenarios of interest described above, where the setup involves PDEs on a 2D or 3D domain, the high-level tasks include, for instance, a domain decomposition to break the domain into several partitions (see Section 2). For each of the high-level tasks, the goal of EZP is to provide a corresponding subroutine. Then the subroutine can be called to accomplish the task, and the user does not need to worry about the parallel programming details. For instance, a simple command `CALL DOMAIN_DECOMPOSITION` could be inserted into a code, and the domain decomposition will be accomplished without the user needing to write their own parallel programming algorithms. The EZ Parallel subroutine that accomplishes the domain decomposition contains approximately 150 lines of code, including defining MPI derived datatypes to be used in some inter-core communications and error handling. As an analogous situation that is already in widespread use, consider the calculation of Fourier transforms. Most users do not write their own Fourier transform algorithms, in parallel or in serial. Instead, a user would typically take advantage of the many established Fourier transform codes that have been written, which allow the user to type a simple command such as `y = FFT(x)` and obtain the Fourier transform. No knowledge is needed of the underlying details of the Fourier transform code or algorithm. The goal of EZP is to provide a similar type of high-level functionality for some parallel programming tasks, in order to ease the process of parallelizing a serial code.

A related but different type of tool is automatic parallelization, which is intended to take a serial code and, without needing the user to change the code at all, automatically implement parallelization in its execution (Griebl, 2004; Bondhugula et al., 2008; Benabderrahmane et al., 2010; Kraft et al., 2018). Such tools are typically designed to treat a variety of different codes with great generality. For instance, the code could be for playing the game of chess, or some other artificial intelligence application. In their simplest forms, auto-parallelizers detect “for loops” in the code and, if possible, they



break up the tasks using multi-threading or parallelization. Some freely available tools are Cetus (Dave et al., 2010), ComPar (Mosseri et al., 2020), Par4All (Amini et al., 2012), and Polaris (Blume et al., 1994). Some commercial tools that use these and other strategies include the Matlab Parallel Computing Toolbox™ and Matlab Parallel Server™ or Matlab Distributed Computing Server™. A disadvantage of Matlab is that it is often slower than other programming languages (Fortran, C++, etc.) for PDEs with time-stepping. For those other languages, the Intel® Fortran and C++ Compilers also have automatic parallelization capabilities, and so does the freely available GNU Compiler Collection (GCC). The commercial tools are not freely available and therefore may not be an option for many users. Auto parallelization is typically limited to shared-memory computer architectures, which limits the amount of parallelization that can be achieved. Also, for some codes, auto parallelization may not achieve parallelism or may not provide a great amount of speedup. Despite these limitations, it may be a good choice for some users. However, some users may also have concerns about using a “black box” to automatically parallelize a code. Some users prefer to have some control over and some knowledge of the parallelization process. For such users, EZP can help with parallelization with a minimal amount of user effort, while also engaging the user in the parallelization process, and allowing the user to further optimize the parallelization if desired.

Besides fully automatic tools, other tools have been created to be more interactive but still somewhat automatic. Examples include ParaScope (Balasundaram et al., 1989), the Parallelizing Assistant Tool (PAT) (Appelbe and Smith, 1990), FORGE (Levesque and Wagenbreth, 2011), and the Interactive Parallelization Tool (IPT) (Arora et al., 2014). The interactive tools can provide valuable feedback to a user and can help the user learn more about parallelization, in comparison to automatic tools which are typically not designed for such purposes. Like automatic tools, the interactive tools are commonly designed to be very general. In the present paper, it is not the general case of parallelization that is considered, but the specific case of parallelization of PDE solvers using domain decomposition. In this specific case, following the methods de-

scribed below, a desirable feature can be achieved: the parallel code looks essentially the same as the original serial code. One of the main differences from the original serial code is the addition of some new subroutine calls, since much of the MPI functionality can be grouped together into a few subroutines. In this way, one goal of EZP is to preserve the readability of the code as much as possible.

As another route to simplifying parallel coding, some alternatives to MPI and OpenMP have been developed as parallel languages that extend serial programming languages such as C and Fortran. Examples that use the global address space (GAS) model include Unified Parallel C (UPC) (Chen et al., 2003) and Coarray Fortran (Numrich and Reid, 1998). These examples aim to offer a programming environment that is more user-friendly compared with the message passing of MPI. It would be interesting to try to include UPC and/or Coarray Fortran capabilities into EZP instead of MPI, and to try to achieve the same goals as MPI-based EZP—namely, a parallel code that looks nearly identical to (or as similar as possible to) the original serial code, aside from a few subroutine calls to handle the parallelization. For this initial work, MPI was chosen in part because it is in more widespread use.

*Another minor complaint is about the lack of support for parallelism at thread level. My understanding is that the backbone of the tool is based on MPI, which is still the best way to go for parallelization across different nodes. But the current hardware trend is to have more and more computing power within a node, and you can easily have a heavy CPU with around 50 cores, or a GPU with thousands of CUDA cores. It would be a more interesting tool if these cases can be covered, or at least discussed. For example, for a node with around 100 cores, how would such a tool perform against an OpenMP approach?*

Thank you for these suggestions. It would indeed be interesting to include CUDA capabilities into EZP, and this is now mentioned in the revised manuscript. Also, new results have been added to show the scaling performance on a single node with a large number of processors. We have access to a single node with 64 cores.



New Figure 5 and new description single-node scaling performance (LINES 239 - 259)

We performed two series of tests to investigate the scaling performance of the implementation of the EZP library to parallelize a serial two-level QG equation code: across multiple nodes with several cores per node and on a single node with several cores. We conducted the multi-node tests on the high performance computing (HPC) cluster that is serviced by the University of Wisconsin-Madison's Center for High Throughput Computing (CHTC) and the single-node tests on the Cori System that is serviced by the National Energy Research Scientific Computing Center (NERSC). For the multi-node tests, we utilized a dedicated partition equipped with 35 nodes, each with a single 10-core (20 thread) Intel Xeon E5-2670 v2 CPU at 2.50 GHz. For the single-node tests, we utilized the KNL partition, whose nodes are a single-socket Intel Xeon Phi Processor 7250 processor with 68 cores at 1.4 GHz.

Simulations were run with a random initial condition on a grid with dimensions $N_x = N_y = 2048$ and $10^2$ time-steps. The physical parameters in the simulation correspond to the atmosphere in mid-latitudes, specifically: $\beta = 2.5$, $k_d = 4$, $U = 0.2$, $\kappa = 0.05$, and $\nu = 2.98023 \times 10^{-22}$ (Qi and Majda, 2016). File output was disabled for all tests following the practice of Arabas et al. (2015), and we analyzed the average execution time per time-step for consistency between runs. The multi-node tests used a range of two to 128 cores, utilizing eight cores per node for runs of eight or more cores. We chose not to utilize all 20 available cores per node due to the known lack of optimization of `MPI_ALLTOALL` on shared-memory systems (Kumar et al., 2008). Due to the memory-intensive nature of the six-stage AARK method used, there was not enough memory available for a single-core run with $N_x = N_y = 2048$. The single-node tests used a range of one to 64 cores on a single node. The results of these tests are included in Figs. 4 and 5.

We observed a strong scaling efficiency of $73.0\% \pm 7.8\%$ for the multi-node tests and a strong scaling efficiency of $94.8\% \pm 11.1\%$ for the single-node tests, which implies that doubling the number of cores across several nodes leads to an approximate $40\%$

reduction in execution time and double the number of cores across a single node leads to an approximate $48\%$ reduction in the execution time.

(LINES 313 - 314)

It would also be interesting to explore asynchronous MPI, or to pursue similar software involving CUDA or OpenMP, or other languages such as Julia or Python.

We hope that these responses are satisfactory and would like to again thank you for your efforts and constructive feedback. The authors welcome any further constructive comments.

Sincerely,

Jason Torchinsky [Formerly “Turner"] (jlturner5@wisc.edu)

Samuel Stechmann

---

Interactive comment on Geosci. Model Dev. Discuss., https://doi.org/10.5194/gmd-2020-257, 2020.