Enhancing Regional Ocean Modeling Simulation Performance with the Xeon Phi Architecture

Chris Lupo and Maria Pantoja and Paul Choboter
California Polytechnic State University
San Luis Obispo, California, USA 93407
Email: {clupo,mpanto01,pchobote}@calpoly.edu

***Abstract*—Ocean studies are crucial to many scientific disciplines. Due to the difficulty in probing the deep layers of the ocean and the scarcity of data in some of the oceans, the scientific community relies heavily on ocean simulation models. Ocean modeling is complex and computationally intensive, and improving the performance of these models will greatly advance and improve the work of ocean scientists. This paper presents a detailed exploration of the acceleration of the Regional Ocean Model System (ROMS) software with the latest Intel Xeon Phi x200 architectures.**

Both shared-memory and distributed-memory parallel computing models are evaluated. Results show run time improvements of nearly a factor of 16 compared to a serial implementation. Further experiments and optimizations, including the use of a GPU acceleration model, are discussed and results are presented.

I. Introduction

The Regional Ocean Modeling System (ROMS) is an open-source, free-surface, primitive equation ocean model used by the scientific community for a diverse range of applications [1]. ROMS employs sophisticated numerical techniques, including a split-explicit time-stepping scheme that treats the fast barotropic (2D) and slow baroclinic (3D) modes separately for improved efficiency [2]. ROMS also contains a suite of *data assimilation* tools, that allow the user to improve the accuracy of a simulation by incorporating observational data. These tools are based on four-dimensional variational methods [3], which generate reliable results, but require more computational resources than without any assimilation of data.

In the roughly fifteen years since ROMS has been available, the microprocessor industry has relied on parallelism rather than clock speeds to improve program performance. In order for the numerical model to provide desired accuracy with reasonable performance, implementations of ROMS currently exist to take advantage of two popular parallel computing paradigms. A shared-memory model using OpenMP enables ROMS to take advantage of modern multi-core processors, and a distributed-memory model using MPI provides access to the computing power of multi-node clusters. ROMS uses a course-grained parallelization paradigm which partitions the computational grid into tiles. Each tile is then operated on by different parallel threads.

The ability to increase the problem size to obtain finer-grain resolution and more precise results is limited by computing performance, even on modern computing systems. This is particularly true when performing data assimilation, where the run-time can be orders of magnitude larger than non-assimilating runs, and where accuracy of a simulation is critical.

Recent research has shown some success with implementations of ROMS that target massively parallel accelerators: including the initial version of the Intel Xeon Phi [4], and using NVIDIA's CUDA framework for Graphics Processing Units (GPUs) [5], [6]. These accelerators, including GPUs, are well suited to many high-performance compute applications. They allow general purpose computation in a massively parallel fashion. GPUs support fast double-precision calculations and error correcting memories, allowing calculations that require high degrees of accuracy to run with performance that can exceed shared-memory implementations using multi-core CPU architectures. However, one factor that limits the performance of GPUs and other accelerators is the communication cost between the host processor and the accelerator.

Another factor that limits the use of accelerators with ROMS is a significant amount of source code refactoring to manipulate the internal data structures and procedures of ROMS to be compatible with the software models that the accelerators require. Feedback from domain experts on prior research with NVIDIA's CUDA framework suggests that many users of ROMS don't want to, and sometimes are unable to, fully comprehend the underlying computer architecture when designing or tuning their models in ROMS. Users of ROMS that are able to make this investment are often unwilling to, as the software refactoring can make the ROMS source code less portable to different computing platforms.

A. Contributions

The contributions of this paper are twofold. First, the performance of ROMS is thoroughly characterized on a modern, massively-parallel architecture, an Intel Xeon Phi x200 series processor. Only the existing parallel paradigms of ROMS are utilized, along with intelligent compiler optimizations, that require zero modifications to the ROMS source code. Second, we explore the use of the OpenACC parallel computing paradigm to target modern GPUs with the minimal amount of source refactoring.

B. Outline

This paper is organized as follows. Section II presents necessary background on ROMS, the Intel x200 architecture, and OpenACC. Section III discusses related work, and Section IV presents our experimental methodology, along with a discussion of the results. Section V contains our conclusions.

II. Background

A. ROMS

ROMS has a modern and modular code base, consisting of approximately 400,000 lines of Fortran 90/95 code. The shared-memory option follows OpenMP 2.0. The distributed-memory option allows data exchange between nodes using MPI. Both of these parallel paradigms are integrated in the same code base, though only one may be used at any given time. The parallel paradigm is selected at compile-time.

During the execution of ROMS applications, run time measurements are logged and reported. This information essentially profiles the application, providing both absolute and relative timings for each of the major modules utilized during execution. For the application studied in this paper, the *Model 2D Kernel* is where the large majority of run time is spent, with over 48% of execution time occurring in that module. The implementation of this module is parallelized using both OpenMP and MPI, and is the target of the OpenACC experiments performed in this work. Profile data for the serial execution of ROMS is shown in Figure 1.

```
--------------------------------------------------------------------------
------------------------------ Profile Data ------------------------------
--------------------------------------------------------------------------
 Nonlinear model elapsed time profile , Grid: 01

  Allocation and array initialization ..........        3.235  ( 0.0009 \%)
  Ocean state initialization ...................        1.230  ( 0.0004 \%)
  Reading of input data ........................        0.276  ( 0.0001 \%)
  Processing of input data .....................       20.523  ( 0.0059 \%)
  Processing of output time averaged data ......    20381.832  ( 5.8607 \%)
  Computation of vertical boundary conditions ..       16.813  ( 0.0048 \%)
  Computation of global information integrals ..     6675.357  ( 1.9195 \%)
  Writing of output data .......................      403.165  ( 0.1159 \%)
  Model 2D kernel ..............................   168419.839  (48.4280 \%)
  2D/3D coupling , vertical metrics ............     1111.320  ( 0.3196 \%)
  Omega vertical velocity ......................     1175.540  ( 0.3380 \%)
  Equation of state for seawater ...............      981.042  ( 0.2821 \%)
  3D equations right—side terms ................    11489.107  ( 3.3036 \%)
  3D equations predictor step ..................    32995.365  ( 9.4876 \%)
  Pressure gradient ............................     4641.079  ( 1.3345 \%)
  Harmonic mixing of tracers , S—surfaces ......     2758.494  ( 0.7932 \%)
  Harmonic stress tensor , S—surfaces ..........     4452.490  ( 1.2803 \%)
  Corrector time—step for 3D momentum ..........    60498.191  (17.3959 \%)
  Corrector time—step for tracers ..............    24289.857  ( 6.9844 \%)
                                          Total:   340314.755    97.8553

 All percentages are with respect to total time =      347773.577
                                          ...
```

Fig. 1. ROMS internal profiling data for serial execution

B. The Xeon Phi x200 Architecture

It is now possible to avoid the CPU–Coprocessor communication bottleneck on a massively-parallel architecture. Experiments are performed to quantify the performance of ROMS on the new Intel Many Integrated Core (MIC) architecture, now named Xeon Phi [7]. The newest Intel Xeon Phi x200 series processor is self-booting, and includes integrated on-package memory for significantly higher memory bandwidth than off-chip memory modules. The cores on the Xeon Phi each support four simultaneous threads, and are tiled in pairs. Each core has two 512-bit vector units, and 32 MB of L2 cache shared across a tile. The tiles are linked to each other using a 2D mesh interconnect, which connects to the memory controllers *far memory* (up to 384 GB capacity and 90 GB/sec) and to high bandwidth stacked *near memory* (up to 16 GB of capacity and 400 GB/sec). Different modes of memory addressing allow the use of the combined memory as a single address space or using the near memory as an L3 cache. This on-package interconnect allows the Xeon Phi to be a stand-alone accelerator that doesn't need to be tied to a regular CPU to do useful work. The architecture also eliminates the communication cost between the processor and accelerator. Because the architecture uses the same instruction set as Intel Xeon CPUs, it allows the use of common parallel programming paradigms, including OpenMP and MPI. This means that the original ROMS code can run with no changes. This is a great advantage compared to using CUDA for an NVIDIA GPU architecture that requires extensive changes in the code to get optimal performance. However, to achieve maximum performance, one must make full use of the vector processors and other optimizations.

The ability to run four hardware thread contexts simultaneously per core allows flexibility in the thread scheduling for OpenMP applications, and process scheduling for MPI applications. While having multiple threads or processes per core is allowed, contention for shared resources may degrade performance if too many threads or processes are running on a core. The Intel MPI Library and OpenMP run-times provide mechanisms to bind MPI ranks and OpenMP threads to specific cores [8]. Experiments were performed using from one to four hardware threads per core to determine the optimal combinations of MPI ranks and OpenMP threads for ROMS on this architecture.

C. OpenACC

The performance speed up achieved by using the Intel compiler and hardware is compared with an optimization of the ROMS code using a GPU.

Programming for heterogeneous computer systems has been an area of interest for many years for acceleration of scientific computations. This research has resulted in the development of several low-level APIs including CUDA [9], OpenCL [10], and OpenACC [11]. Programming these APIs can take time and usually requires some level of expertise to develop correct and optimized implementations. From these options, OpenACC is the only pragma directive-based programming language designed to allow easier development for a variety of hardware accelerators that include GPUs from different vendors, multicore architectures, and FPGAs. The goal of OpenACC is to improve the execution time of existing code written in Fortran, C or C++, by adding different pragma directives to the code that will allow it to run on the available accelerator. OpenACC allows programmers to quickly develop for new architectures without the need to understand much of the hardware or the need to learn new vendor specific programming languages. However, since the compiler will make most of the decisions, the performance is usually lower than what can be obtained by using hardware-specific programming languages and compilers.

OpenACC support is provided by a number of vendors and is defined by an open standard [11]. Since ROMS is written in Fortran, using OpenACC is a viable option. ROMS is a complex code with hundreds of different files and procedures. Finding which specific procedures are parallelizable is not a simple task. As discussed earlier in the profiling results, the primary computational bottleneck in ROMS is the Model 2D Kernel, which is responsible for more than 48% of the sequential execution time. This is the function optimized in this research. The Model 2D Kernel is implemented mainly in the file `step2d.f90`, which consists on a series of for-loops that are possible to accelerate using OpenACC directives.

In OpenACC, segments of code to be offloaded to the accelerator can be specified using the syntax `!$acc for` or `!$acc kernel`, which is similar to how OpenMP pragmas are described [12].

III. Related Work

Other researchers have investigated the performance of ROMS on Intel Xeon Phi coprocessor accelerators.

Yalavarthi and Kaginalkar present an early attempt to understand the performance of the ROMS model using a hybrid cluster super-computer with 51392 nodes, each node consisting of one Intel Xeon E5-2697v2 CPU and two Intel Xeon Phi 7110P coprocessors [4]. The article compares three different programming models of the architecture; *host* (where the program runs on just the CPU with MPI ranks on the host cores), *native* (where the program runs on just the coprocessor with MPI ranks residing on the coprocessor cores), and *symmetric* (where the CPU and accelerator communicate with each other using MPI). Their results show that host delivers better results, most probably due to the lack of hybrid parallelism in ROMS, preventing the use of shared-memory and distributed-memory parallelism at same time. They leave for future work vectorization and offload modes.

Bhaskarin and Guarav analyze ROMS and the issues impacting its performance on a Xeon Phi coprocessor, and introduce an iterative optimization strategy that results in a 2x speed-up in performance when comparing the base-line code with a native mode on the Xeon Phi [13]. The improvement requires extensive changes in the code that are applied to all source files, not just the main bottleneck of ROMS (the Model 2D Kernel). The optimizations mainly consist of padding loop rows for better data alignment, flattening nested if-else statements, optimizing the tile size hyperparameter, and using large page sizes to avoid Translation Look-aside Buffer (TLB) misses.

Reuter et al. introduce a multi-platform scaling investigation for OpenMP parallelization of the UTBEST3D ocean simulator [14]. They compare the performance of an Intel Xeon versus an IBM Power6 and an Intel Xeon Phi coprocessor using only shared-memory parallelism. They conclude that the Intel Xeon CPUs produce the best run time results. The run-times achieved with the Intel Xeon Phi coprocessors where not satisfactory due mainly to the low clock rate of the cores and lack of vectorization on the code.

To the best of our knowledge, this paper is the first to investigate the performance of ROMS on the Intel Xeon Phi x200 architecture.

IV. Results

In this section, experimental results are presented. Results are divided into experiments run on the Xeon Phi using Intel's compiler technology, and the use of OpenACC on an NVIDIA GPU using the Portland Group's (PGI) compiler technology.

A. Intel Xeon Phi Experiments

With the exception of the vectorization results below, all experiments are run with parameters that are common for research-level uses of ROMS. The three-dimensional grid contains 321x640x32 grid points (the `Lm`, `Mm`, and `N` parameters, respectively). The simulation duration is five days, with a time step of 30 seconds.

In the experiments, ROMS is running the *Upwelling* application to compare execution time of ROMS in shared-memory versus distributed-memory on one Xeon Phi x200 system, while also exploiting vectorization and automatic parallelization features available in the Parallel Studio XE Cluster edition of Intel's compiler suite, version 2017.0.035. The operating system is CentOS Linux release 7.3.1611.

Building ROMS requires the use of the NetCDF library, which in turn requires the HDF5 library. These support libraries had to be compiled using the same Intel compiler that ROMS is built with. The build procedure and environmental setup required to compile ROMS, NetCDF, and HDF5 is not well documented for the Intel Xeon Phi x200 architecture, due to its short time being available. While not included in this paper, the authors can share this procedure with those that are interested.

Simulations are run on an Intel Xeon Phi 7210 processor with 64-cores (256 threads), each with a 1.3 GHz clock speed. Total system memory is 112 GB (96 GB far memory, 16 GB near memory). ROMS was tested using all available parallel paradigms. The serial implementation on the Xeon Phi took approximately 96 hours to complete. For reference, results are also compared to serial performance of identical ROMS simulations on a high-performance Intel Xeon E5-2690v2 CPU with a clock speed of 3.00 GHz.

1) Vectorization: The Intel compiler suite has automatic vectorization on by default, and the largest vector size supported by the underlying architecture is selected using the `-Xhost` compiler option. No changes to the source are required to enable vectorized computations, but the compiler must be conservative in its analysis to ensure that automatic optimizations are correct. Explicit vectorization is also allowed through the use of Single-Instruction-Multiple-Data (SIMD) pragmas, and this may yield better performance than automatic analysis.

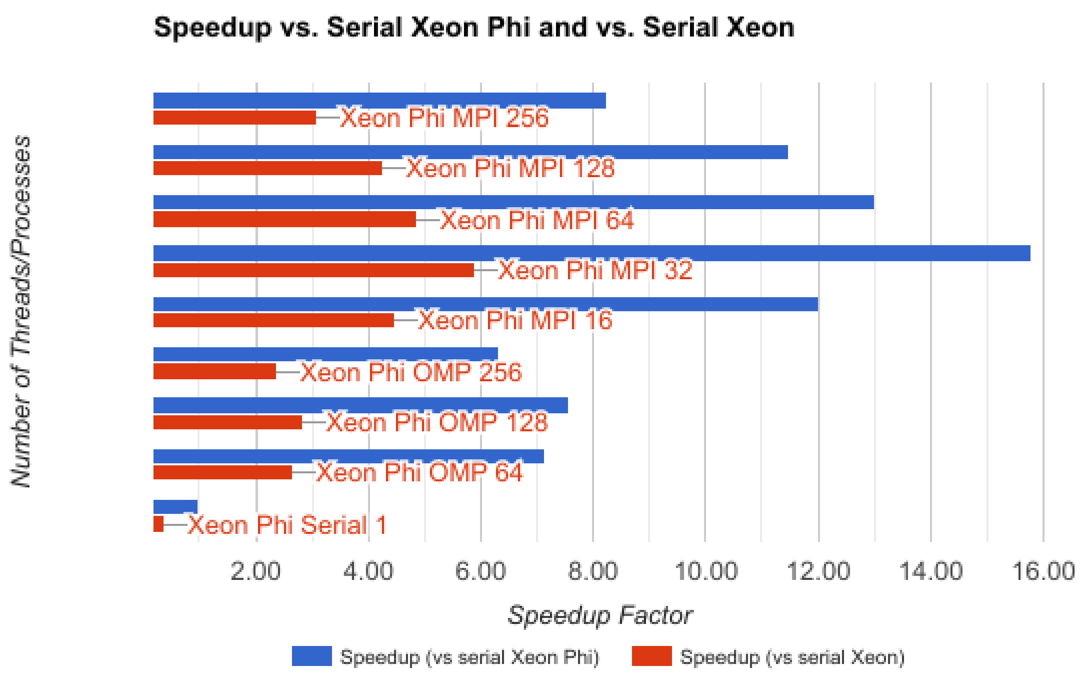

Fig. 2. Simulation speedup compared to serial implementations

The effect of vectorization was measured on a small data set. Without vectorization, the Upwelling model of ROMS took 658.9 seconds, and the same experiment with automatic vectorization enabled took 599.6 seconds, a 10% speedup.

Since vectorization is a level of parallelism that is orthogonal to other parallel computing paradigms such as OpenMP and MPI, automatic vectorization is left enabled in all subsequent results presented here.

2) OpenMP: ROMS was compiled and run using the OpenMP parallel execution model. Experiments were run using various thread affinity settings, and for 64, 128, and 256 threads. The optimal OpenMP thread configuration for these experiments was 128 threads with balanced thread affinity. This configuration had a runtime that was 7.6 times faster than the sequential implementation on the Xeon Phi x200, and 2.8 times faster than the sequential Xeon CPU run time. These results are not completely intuitive, as it is difficult for the ROMS user to know where the tradeoff is between fully using the computational resources of the architecture while still avoiding contention for resources.

3) MPI: In a separate set of experiments distinct from the OpenMP tests, ROMS was compiled and run with the MPI parallel execution model. Numbers of processes from 16 to 256 were tested, incrementing by powers of two. For numbers of processes less than or equal to 32, one process was scheduled per tile on the Xeon Phi, to prevent processes from sharing the L2 cache on each tile. For 64 processes, one process was scheduled per core, while for 128 and 256 processes, two and four processes were scheduled per core respectively. The optimum configuration identified from these experiments was 32 processes with one process per tile. This configuration resulted in a run time speedup of 15.8 compared to the sequential Xeon Phi implementation, and a factor of 5.9 compared to the sequential Xeon CPU run time. Again, this result in not intuitive, and doesn't correlate to the OpenMP experiments.

Table I shows the execution times of the OpenMP and MPI experiments. As can be seen, the distributed-memory MPI implementation performs better than the shared-memory OpenMP one. This by itself is not completely unexpected, as the two paradigms are implemented differently within ROMS, and parallelize different portions of the simulation. Results also show that there is resource contention resulting in a slowdown when multiple processes run on a single core for distributed MPI, but there is advantage to sharing up to two threads per core for OpenMP.

Figure 2 shows the speedup of each simulation experiment compared to a serial implementation on the Xeon Phi, and a serial implementation on an Intel Xeon CPU. The performance advantage of the Xeon Phi processor is significant, allowing faster turn-around on simulations on a compute system that is similar in cost, size, and energy usage to a standard high-performance workstation.

4) Automatic Parallelization: Based on the OpenMP and MPI experiments, it can be seen that the optimal number of threads for the shared-memory OpenMP implementation is greater than the optimal number of processes for the distributed-memory MPI implementation. This presented a

TABLE I
EXECUTION TIMES

# Processes/Threads Paradigm	256 MPI	128 MPI	64 MPI	32 MPI	16 MPI	256 OMP	128 OMP	64 OMP	1 Serial
Time (sec.)	42174	30337	26695	22029	28959	55045	45884	48735	347789

question as to whether the MPI implementation could benefit from some amount of automatic shared-memory parallelization.

The Intel compiler can be given commands to attempt to automatically parallelize the source code, usually by targeting loops where it can be guaranteed that each iteration of the loop is independent. If such a guarantee cannot be made using static analysis, the compiler will not parallelize the loop. The compiler flags used in these experiments are shown in Figure 3.

```
—parallel
—par—num—threads=2
—par—affinity=balanced
—par—schedule—auto
—par—runtime—control3
```

Fig. 3. Intel compiler flags to enable automatic parallelization with two threads

Experiments were run to attempt automatic parallelization in addition to the MPI implementation of ROMS. Two threads per MPI process were created, and the performance was measured. Figure 4 shows the run times for each of the experiments, with 16, 32, and 64 MPI processes.

Results from these experiments show that in no case was automatic parallelization able to improve performance compared to single-threaded MPI processes. This is most likely due to overhead created in the scheduling and management of threads, and possibly some additional resource contention. What is interesting is that 32 MPI processes, each with two threads, outperformed 64 MPI single-threaded processes. This is another non-intuitive result, as the total number of running threads is the same.

B. OpenACC

As discussed in Section II, OpenACC allows the user to insert simple pragmas around sections of parallel code to offload those computations to an accelerator.

The experimental platform used for OpenACC tests included an NVIDIA GTX TITAN X GPU, an Intel i7 CPU, 32 GB RAM, running Ubuntu Linux 14.04 LTS as the operating system.

In the experiments done in this research, just adding the pragmas actually caused a slow-down of execution time of the ROMS simulation by approximately 12%. Usually, a slowdown is due to data transfer between the CPU and device (GPU). In some cases, these data transfers are unnecessary, and can be explicitly eliminated by adding specific information about which data should be used for input, output or both and which data should be transferred to the GPU. These data directives were also inserted, but performance was still degraded with OpenACC compared to the serial implementation.

These results are in line with the performance results seen in related work running ROMS using the Xeon Phi as a coprocessor. Data transfer is expensive in ROMS because some of the data arrays are very large. This result is also in line with the experimental results attempting to automatically parallelize sections of the ROMS code. ROMS may have some data dependencies in loop iterations that the compiler can't resolve.

An additional attempt to use OpenACC was made by targeting existing regions of shared-memory parallelism in ROMS, and converting OpenMP pragmas to OpenACC alternatives. After transforming several of the OpenMP Parallel pragmas into pragmas in OpenACC, this solution had better performance than the original sequential code, but just by 1%. The original OpenMP implementation far outperformed the OpenACC implementation identified in this work. These transformations weren't a trivial task to implement.

V. CONCLUSION

This paper presents the first performance evaluation of the ROMS ocean modeling software package on the Intel Xeon Phi x200 architecture, and compares that performance to using OpenACC to run portions of ROMS on a modern GPU.

Extensive experimentation was done to identify an optimal parallel computing model, and configuration of the number of parallel threads and processes for ROMS. Shared-memory OpenMP implementations, distributed-memory MPI implementations, and accelerated OpenACC implementations are quantitatively compared.

The best performing configuration was found to be the use of 32 MPI processes, one process running per tile on the x200 architecture, with automatic vectorization enabled, and automatic parallelization disabled. This implementation was found to be a factor of 15.8 times faster than the serial implementation on the Xeon Phi, and 5.9 times faster than the serial implementation on a modern high-performance Xeon CPU.

A significant conclusion to draw from this research is that excellent parallel performance is possible for users of ROMS with absolutely no modifications to the ROMS source code. Tuning performance of other accelerated computing models, including OpenACC in this work, and CUDA in prior work, takes weeks of effort, with no guarantee of performance improvement, and results in a code base that is much less portable than the original code.

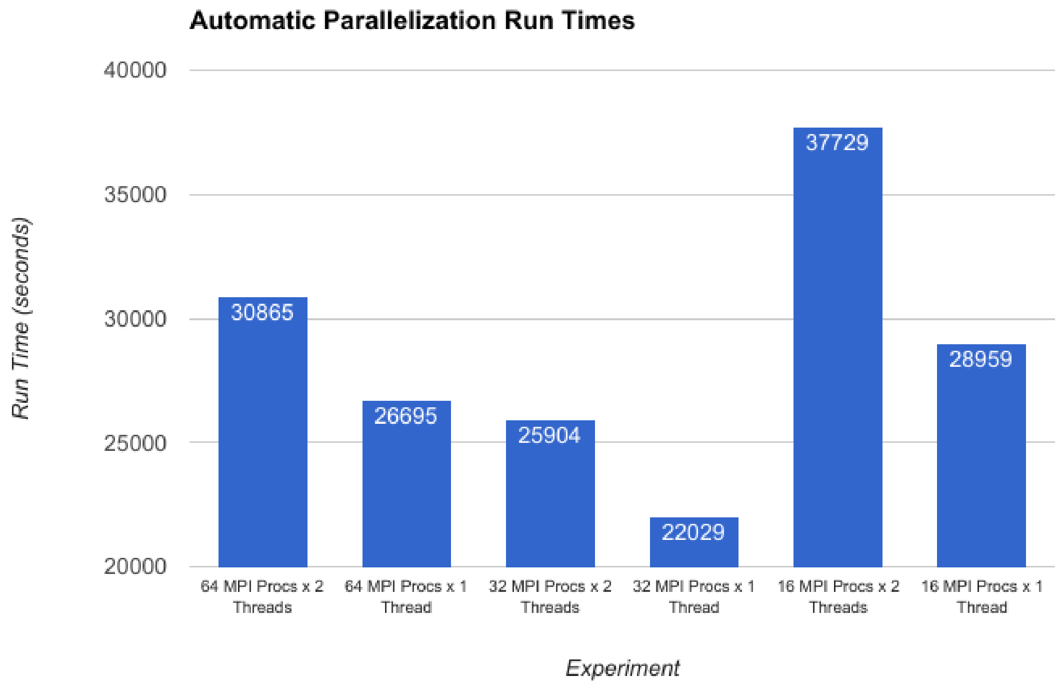

Fig. 4. Run times with and without automatic parallelization of MPI processes

The improvements in the execution time of the ocean simulation tool will allow ocean scientists to use higher precision and more realistic models that should allow them to better understand and predict ocean currents, sea level changes, temperature, etc.

References

[1] ROMS. [Online]. Available: http://www.myroms.org

[2] A. F. Shchepetkin and J. C. McWilliams, "The Regional Oceanic Modeling System (ROMS): A split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean Modelling*, vol. 9, no. 4, pp. 347 – 404, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500304000484

[3] E. D. Lorenzo, A. M. Moore, H. G. Arango, B. D. Cornuelle, A. J. Miller, B. Powell, B. S. Chua, and A. F. Bennett, "Weak and strong constraint data assimilation in the inverse Regional Ocean Modeling System (ROMS): Development and application for a baroclinic coastal upwelling system," *Ocean Modelling*, vol. 16, no. 3-4, pp. 160 – 187, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500306000916

[4] S. Yalavarthi and A. Kaginalkar, "An early experience of regional ocean modelling on Intel many integrated core architecture," in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–6.

[5] J. Mak, P. Choboter, and C. Lupo, "Numerical ocean modeling and simulation with CUDA," in *OCEANS 2011, MTS/IEEE KONA - Oceans of Opportunity: International cooperation and partnership across the Pacific*, September 2011.

[6] I. Panzer, S. Lines, J. Mak, P. Choboter, and C. Lupo, "High performance regional ocean modeling with GPU acceleration," in *OCEANS 2013, MTS/IEEE San Diego - An Ocean in Common*, September 2013.

[7] *Intel Xeon Phi Processor x200 Product Family Datasheet*, Intel Corporation, 8 2016, rev. 001.

[8] Process and thread affinity for Intel© Xeon Phi$^{\text{TM}}$ processors. [Online]. Available: https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200

[9] CUDA zone. [Online]. Available: https://developer.nvidia.com/cuda-zone

[10] OpenCL.org: The community site. [Online]. Available: http://www.opencl.org

[11] The OpenACC application programming interface. [Online]. Available: http://www.openacc-standard.org

[12] OpenMP specification for parallel programming. [Online]. Available: http://www.openmp.org/

[13] G. Bhaskaran and P. Gaurav, "Optimizing performance of ROMS on Intel Xeon Phi," *Procedia Computer Science*, vol. 51, pp. 2854 – 2858, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915012557

[14] B. Reuter, V. Aizinger, and H. Köstler, "A multi-platform scaling study for an OpenMP parallelization of a discontinuous Galerkin ocean model," *Computers and Fluids*, vol. 117, pp. 325 – 335, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045793015001759