# Parallelisation of Hydro-environmental Model for Simulating Marine Current Devices

Fearghal O'Donncha*, Scott C. James†, Noreen O'Brien*, Emanuele Ragnoli*
*IBM Research - Ireland
Email: feardonn@ie.ibm.com
†Baylor University

***Abstract*—There are many serial simulation codes in the geosciences that cannot take advantage of the computational resources commonly available today. Further, the initial design of many parallel application codes targeted relatively small-scale commodity clusters and does not take full advantage of modern cache-based, blade computing system. In this paper, we discuss the porting of a widely used hydro-environmental code, EFDC, to parallel using both the MPI and OpenMP paradigms. The objective of the research is to better elucidate the strengths and weaknesses of both approaches to further a hybrid parallelisation strategy that leverages the capabilities of both.**

## I. Introduction

Numerical modelling has several advantages in the study of coastal ocean flow processes and events. Chief among these is the reduced cost and ease of deployment of a numerical model compared to field work or other methods of investigation. In addition, it is easier to configure a numerical model to investigate different flow conditions and scenarios. However, with the drive to model more realistic and detailed scenarios, the computational demands of numerical solutions increase, due primarily to finer grid resolution and the simulation of a greater number of passive and active tracers. As a result, the practical ability of numerical models to solve real-world problems is constrained. Parallel computing allows faster execution and the ability to perform larger, more detailed simulations than is possible with serial code. In this paper we discuss the porting of a widely used hydro-environmental code, EFDC, to parallel using both the MPI and OpenMP paradigm. The objective of the research is to better elucidate the strength and weaknesses of each approach to further a hybrid parallelisation strategy that leverages the capabilities of both.

The code is ported using both paradigms and adopts two different parallel strategies, namely coarse-grain and fine-grain parallelisation. Coarse-grain parallelism means that the parallelism in the program is achieved through a decomposition of the target domain into a set of subdomains that is distributed over the different processors of the machine. Fine-grain parallelism means that the parallelism in the program is achieved by distributing the work of the do-loops over the different processors, so that each processor computes part of the iterations. Coarse-grain parallelism typically requires some form of periodic communication between sub-domains to synchronise and update the solution. It can be implemented on both shared- and distributed-memory machines and theoretically scales infinitely. Fine-grain parallelism on the other hand does not require explicit communication as the paradigm is designed for shared-memory machines and each concurrent process has access to the same solution space at all times. It also means that the degree of parallelism is limited by the number of computational threads within each machine and typically cannot scale across machines (without some "under the hood," compiler-level communication).

In this paper we compare the performance of both paradigms. For the MPI study we describe a parallelisation strategy involving surgical changes to the source files to minimise error-prone alteration of the underlying computations, while allowing load-balanced domain decomposition for efficient execution on a commodity cluster. The use of a conjugate gradient solver posed particular challenges due to implicit non-local communication hindering standard domain-partitioning schemes. Our partitioning scheme is motivated by the fact that ocean models tend to have much greater horizontal than vertical length scales. We therefore adopt a two-dimensional domain decomposition approach in the horizontal plane; the vertical remaining unchanged. The work flow is based on a data-parallel execution model with all processors performing the same types of computation but on a different sub-domain. The domain decomposition strategy involved a number of additional considerations. Unlike some computational fluid dynamics problems involving a uniform volume of material, simulations of coastal ocean currents have a mix of land and water regions with significantly different computational costs. This complicates the task of balancing the work load to achieve maximum parallel performance, particularly when there is an irregular, serpentine coastline as in many coastal systems.

Fine-grain parallelism using openMP requires a much more invasive programming approach. Preliminarily, a sophisticated profiling code identified regions that are most computationally intensive. Considering that EFDC comprises over 50,000 lines, this is not a trivial task. The relevant sections are then parallelised using compiler directives inserted at appropriate positions within the code to distribute workload across different threads. While explicit load balancing is not a factor for fine-grain parallelism, efficient performance requires consideration of the overheads introduced by openMP and the relevant computational cost to ensure that these overheads do not become punitive. A further advantage of openMP for parallelisation is the availability of well-established libraries for matrix operations. Parallelism can be realised through graph colouring

978-0-933957-43-5 ©2015 MTS

techniques where the graph defines the data dependencies in matrix assembly. This approach removes data contention, so called critical sections in OpenMP, affording efficient parallelisation. Equivalent structures in MPI are considerably more difficult due to non-local memory access. In addition, shared-memory parallelisation avoids many of the parallel file I/O considerations of distributed memory implementations

The model used for the study, Environmental Fluid Dynamics Code (EFDC), is a widely used, three-dimensional, finite difference, hydrodynamic model (Hamrick, 1992). The study was guided by the following requirements considered key to the success of the parallelisation effort and subsequent operation

1) Assesment of load balancing performance and impact on performance of the parallel code.
2) Limited changes to the large number of source files, to avoid introducing computational errors.
3) Binary regression of the parallel model versus serial simulations, to ensure the simulation runs in parallel exactly as it ran serially. Even a small deviation could mask the presence of an error in the port.
4) Automation of the setup process for a parallel run to allow the original serial models to run properly as parallel code. This involves automatic generation of source code specific to each parallel run of a model, to avoid manual effort and the introduction of errors.
5) Intelligent management of file I/O to (1) minimise performance issues as cores increased and (2) allow compatibility with existing software to post-process serial model output.

## II. Methodology

### A. Model description

EFDC is a public-domain, open-source, modelling package for simulating three-dimensional flow, transport, and biogeochemical processes in surface-water systems. The model is specifically designed to simulate estuaries and subestuarine components (tributaries, marshes, wet and dry littoral margins), and has been applied to a wide range of environmental studies including: surface current processes [1], [2], suspended sediment transport [3], [4], water quality [5], [6] and hydrothermal responses [7]. It is presently being used by universities, research organisations, governmental agencies, and consulting firms [8].

The equations that form the basis for the EFDC hydrodynamic model are based on the continuity and Reynolds-averaged Navier-Stokes equations. Adopting the Boussinesq approximation for variable-density fluid, which states that density differences are sufficiently small to be neglected except where they appear in terms multiplied by gravity, $g$, the governing equations can be expressed as:

$$\frac{\partial \eta}{\partial t} + \frac{\partial Hu}{\partial x} + \frac{\partial Hv}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{1}$$

$$\begin{aligned} &\frac{\partial Hu}{\partial t} + \frac{\partial (Huu)}{\partial x} + \frac{\partial (Hvu)}{\partial y} + \frac{\partial (wu)}{\partial z} - fHv \\ &= -H\frac{\partial (g\eta + p)}{\partial x} - (\frac{\partial h}{\partial x} - z\frac{\partial H}{\partial x})\frac{\partial p}{\partial z} + \frac{\partial (\frac{A_v}{H}\frac{\partial u}{\partial z})}{\partial z} \end{aligned} \tag{2}$$

$$\begin{aligned} &\frac{\partial Hv}{\partial t} + \frac{\partial (Huv)}{\partial x} + \frac{\partial (Hvv)}{\partial y} + \frac{\partial (wv)}{\partial z} + fHu \\ &= -H\frac{\partial (g\eta + p)}{\partial y} - (\frac{\partial h}{\partial y} - z\frac{\partial H}{\partial y})\frac{\partial p}{\partial z} + \frac{\partial (\frac{A_v}{H}\frac{\partial v}{\partial z})}{\partial z} \end{aligned} \tag{3}$$

where $t$ is time; $\eta$ is water elevation above or below datum; $u, v$ are the velocity components in the curvilinear orthogonal coordinates $x$ and $y$ with $w$ representing the vertical component; $H$ is total water depth ($= h + \eta$, where $h$ = water depth below datum); $f$ is the Coriolis parameter; $p$ is excess water-column hydrostatic pressure; and $A_v$ is vertical turbulent viscosity.

The equations governing the dynamics of coastal circulation simulate propagation of fast-moving external gravity waves and slow-moving internal gravity waves. It is desirable in terms of computational economy to separate vertically integrated equations (external mode) from the vertical structure equations (internal mode) [9].

The external mode associated with barotropic, depth-independent, horizontal, long-wave motion is solved using a semi-implicit three time level scheme; the external mode computes surface elevation and depth-averaged velocities, $\bar{u}$ and $\bar{v}$. The internal mode, associated with baroclinic, fully three-dimensional, velocity components is solved using a fractional step scheme combining an implicit step for the vertical shear terms with an explicit discretisation for all other terms; the depth-averaged velocities computed in the external mode equations serve as boundary conditions to the computation of the layer integrated velocities. This approach solves the two-dimensional, depth-averaged momentum equations implicitly in time, hence allowing for the model's barotropic time step to equal the baroclinic time step. The primary limitation of this semi-implicit method is the introduction of an elliptic solver (preconditioned conjugate gradient) to solve implicitly for the free-surface elevation. This has traditionally posed a problem for efficient projection of model codes onto parallel computers due to the inherent non-local conditions of the solver [10]; this issue is addressed in further detail in the next section.

### B. Parallelisation

Three different parallelisation strategies were investigated:

- An MPI implementation using a coarse-grain domain decomposition strategy.
- Fine-grain parallelism at the loop level of the code using the OpenMP paradigm.
- An hybrid MPI+OpenMP approach that adopts a domain decomposition strategy with OpenMP used to improve parallel performance by reducing communication costs.

The MPI implementation is presented in detail in [11] and will be briefly discussed here with emphasis on the

challenges faced. EFDC is a Fortran 77 code originally designed for deployment on vector computers as opposed to distributed systems. The code was configured to achieve a degree of parallelisation on shared-memory processors through directives specific to vectorised architectures. For performance comparable to vector systems, scalable, cache-based processors achieve speedup through massively parallel partitioning of the problem among many processors working concurrently [12]. The origins of our parallelisation study lie in the rapidly increasing computational requirements of environmental flow models, along with the general availability of distributed clusters, particularly blade systems. To enable real-time modelling of large-scale waterbodies and to ensure maximum transferability across high-performance computing (HPC) facilities, parallel deployment across these commonly available distributed clusters is desirable. Considering the practical nature of geoscientific modelling studies, the fundamental philosophy of the parallelisation effort focused on the question: If X number of computational nodes are available, what is the maximum simulation throughput that can be achieved by a legacy serial code? In this case, the original study did not focus on achieving maximum scalability on a large number of nodes, nor on tuning the number of nodes precisely to the problem at hand. Instead, it addressed the common desire simply to run a given simulation much faster on a given cluster, and with minimal changes to the original serial code. All aspects of the parallelisation and associated load balancing proceeded with this in mind.

In this study, we focus on extending the MPI parallelisation to a high core count. This required addressing both load balancing and the cost of communication with increasing numbers of cores.

The standard domain decomposition implementation stores both fluid and land cells in a full matrix, which is usually decomposed into a number of equal-sized, regular-shaped subdomains for parallel processing. By adding ghost layers of halo points to each of the subdomains, the communication step can be accomplished by sending values to the ghost layers of the neighbouring processors. The mix of land and water regions with such different computational costs complicates the task of balancing the work load to achieve maximum parallel performance. First, memory is wasted by storing information for completely land domains that do not participate in the computation. The second drawback is the potential load imbalance problem caused by typical domain decomposition approaches. For a typical coastal ocean simulation with complex topographic geometry, a regular domain decomposition approach can result in severe load imbalances. Hence, the efficiency of the parallel program decreases significantly due to idle synchronisation time. Normally, the goal of load balancing is to equilibrate the workload for each processor to the degree possible, but the presence of land in EFDC simulations complicates this goal by placing the focus on minimising the maximum work on any given processor.

There are two commonly used non-uniform domain decomposition strategies [13]:

1) Rectilinear partitioning: The grid is split into rectilinear-shaped subdomains such that the workload is balanced. The method has the advantage of being simple to code, and the subdomain boundaries maintain similar shapes just as they do in the standard domain decomposition approaches (i.e., each partition has maximum of one neighbour in each direction).
2) Orthogonal Recursive Bisection (ORB): ORB techniques divide the domain into two with a vertical cut, then cuts each half into two with a horizontal cut, then each quarter is cut vertically, and so on. It has been shown to provide a time-efficient distribution, particularly when the domain is fairly homogeneous [14]. Recursive bisection methods require that the number of processors are $n_p = 2^n$ and the resulting partitioning is no longer rectilinear (i.e., partitions can have more than one neighbour in any direction).

To minimise added complexity to the code while achieving improved throughput, we retained the concept of a rectilinear grid, but allow both a range of tile sizes and the ability to mark some tiles as inactive. We call our algorithm for computing this optimal partition, LORP, for Locally Optimal Rectilinear Partition. The algorithm relies on estimates of the computational and communication costs in the simulation to find, for a given partition, the total number of active tiles, and the cost associated with the most expensive tile, using an empirically based estimate of the computation and communication times derived from a number of runs with different partitions. Figure 1 shows a load balanced partitioning of a domain into a $2 \times 14$ grid configured to run on 16 processors with 12 inactive partitions.

For a given number of desired active partitions, A, the LORP algorithm explores a range of possible partitions with tile counts much larger than A, to allow for possible inactive tiles that contain no water, and hence no computation. The only requirement is that the final partition must contain exactly A active partitions. Because this algorithm accepts any downward step in cost, it is greedy and is not expected to find the true globally optimal partition, but it does explore the optimisation landscape around the starting partition to find one that is locally optimal. LORP is therefore in the class of load balancing algorithms based on choosing an initial nearly balanced configuration, then seeking a more efficient partition similar to it. The iterative search across all possible partitions is trivial at low node counts, but increases exponentially. A search across 32 partition completes in less than 30 seconds on a laptop, but this increases to a number of hours when partitions exceed 128. This, however, can be improved dramatically by providing guidance on the initial partitioning to reduce the search space.

Two types of inter-machines communications are needed to keep the computation consistent with the sequential code: (1) point-to-point communication to update halo values among neighbouring machines and (2) global communication at each time of the iterative conjugate gradient solver. Naturally, the global communication was of primary concern as it presents

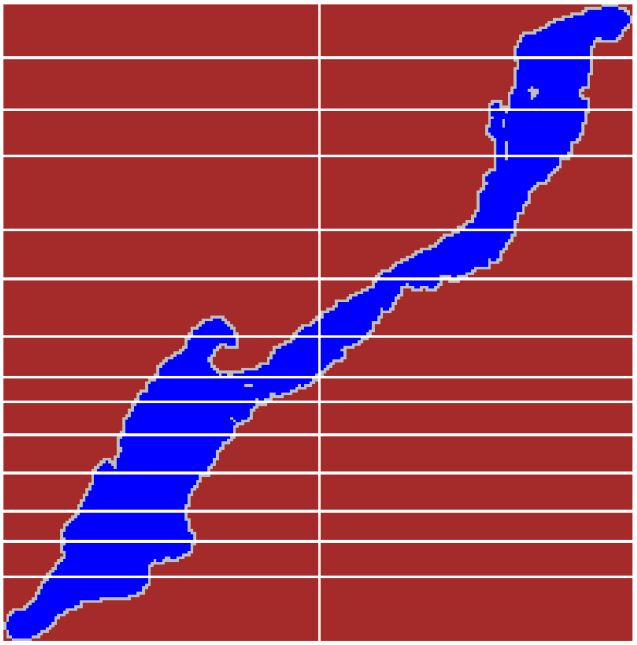

Fig. 1. Rectilinear domain decomposition demonstrating a LORP. Active water cells are blue while red denotes inactive land cells. Note that the partitioning and the algorithmic implementation comprise 16 partitions, 12 of which are inactive and removed from the computation. This produces a considerable improvement over a more naive regular partitioning wherein each partition is allocated directly to a processor. The entire domain consists of $220 \times 221$ grid cells distributed over 16 processors to maximise throughput.

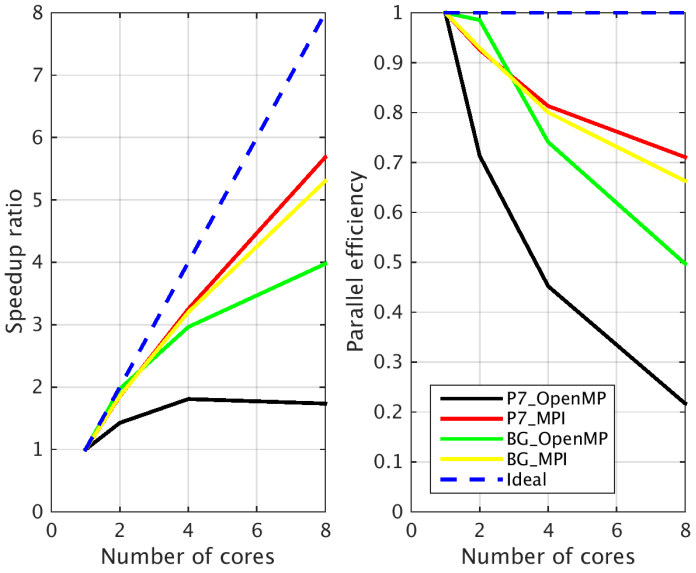


Fig. 2. Comparison of the speedup ratio and parallel efficiency when deploying both MPI and OpenMP computing paradigm on the P775 and BG/Q architecture. Scaling is for the idealised TC example consisting of $133 \times 20$ grid cells and low processor counts.

a bottleneck for each core, which is exacerbated at increasing number of cores. A number of possible approaches exist for parallelising the semi-implicit solver of the code:

- The parallel solution of the conjugate gradient on each processor requires a global MPI reduction call and a point-to-point communication at each iteration of the solver. Typically 10 to 40 iterations are required for convergence [15].
- An alternative approach that avoids the need for communication at each iteration is to have all processors communicating the required variables to a single processor. The solver then proceeds in an identical manner to the serial code after which the computed value of surface elevation is distributed across all partitions. The implementation requires an all-to-one/one-to-all communication and is limited in practise to a low node count.
- An implementation without an iterative conjugate gradient solver eliminates global communications. A widely used algorithm for solving large, sparse, linear system of equations in a parallel environment is Chebyshev iteration, which does not require global reduction operations [16].
- Implement an explicit solution scheme in place of the semi-implicit discretisation. This has the disadvantage of requiring a smaller time step, but transfers more simply to parallel processors.

The second component of the study investigated the performance of an OpenMP parallel version of the code. The objective of the study was to better understand the thread-level expense of the simulation and how it relates to number of active threads. In particular, it focused on whether OpenMP was competitive with MPI for low numbers of processors. OpenMP realises a multi-threaded paradigm such that each thread accesses shared and private data items. Parallel regions are used to parallelise do loops in the code. Distribution of the entire work is done by the compiler, but any privatisation of data or syncronisation points in the code required to avoid race conditions must be specified by the programmer. This is a much more invasive form of parallelisation than the domain decomposition approach, but it allows the programmer to make incremental developments. In addition, fine-grained parallelism avoids load balancing considerations as distribution occurs at the loop level.

Extending this work, we combine MPI and OpenMP using a hybrid parallel approach to assess comparative performance. We compare the MPI+OpenMP implementation against the pure MPI approach to determine how the different paradigms perform on a core-per-core basis. The hybrid approach combines domain decomposition at the MPI level with loop-level parallelism implemented by OpenMP. The selection of the optimum number of MPI processes and OpenMP threads is one of the main considerations of this study and it will be guided by the performance of the hybrid model together with the standalone MPI and OpenMP versions

## III. Results

Performance tests were conducted on a both an IBM Power 775 (P775) supercomputing server and a IBM Blue Gene/Q

(BG/Q) system. Each P775 compute node consisted of 4 Power 7 chips providing 32 cores with clock speeds of 3.84 GHz and symmetric multithreading allowing four threads per core. Hence 64 or 128 processes could be deployed on each node depending on SMT settings of 2 or 4 respectively. Compute nodes are contained within a drawer with direct connectivity provided between each of the nodes [17].

IBM Blue Gene/Qs are massively parallel supercomputers composed of several building blocks.

- At the heart of the Blue Gene/Q, there is a PowerPC A2 chip running at 1.6 GHz with 16 cores for applications, one for the operating system and one as spare. Each chip has a theoretical peak performance of 204.8 Gflop/s. Each core is able to execute up to 4 hardware threads. Each hardware thread can be used for a standard software thread (i.e., an OpenMP or pthread thread) and also for an MPI process
- To maximise the usage of the execution units, it is necessary to run more than one hardware thread per core. Executing several hardware threads per core yields performance gains most of the time.

The clock speed of a compute core is relatively low (1.6 GHz) and also, therefore, its individual performance. This greatly reduces electrical consumption by each core while simultaneously multiplying the number of cores that can be used at the same time for a fixed quantity of energy (electric consumption scales by frequency cubed). The machine performance, therefore, arises from the large number of compute cores with low clock speeds.

Experiments have been performed on both an idealised test-case scenario where conditions could be readily controlled, and a typical application that provided insight into load balancing and other practical considerations. The idealised test case (referred to as TC) was a rectangular harbour consisting of $133 \times 20$ grid cells with five layers in the vertical. In addition, scaling tests were conducted for a typical model deployment, Lake George in upstate New York. Previous studies have applied and validated fundamental model predictions against measured physical parameters for this waterbody [18]. The model domain consisted of $450 \times 957$ grid cells in the horizontal at a resolution of 20 m, along with 20 layers in the vertical. Benchmarking of model performance focused on strong scaling (i.e., the overall spatial domain size remains constant regardless how many processors are used) because it is the most relevant measure if turnaround time for a fixed problem is important [19]. In practical hydro-environmental studies, the size of the problem is based on physical considerations, and the desire is to solve in the minimum time possible using as many processors as available. Weak scaling is of interest because it relates closely to the scalability of the underlying algorithm [20].

To quantitatively compare the different parallel configurations, both the speedup ratio, $S$, and the parallel efficiency, $E$,

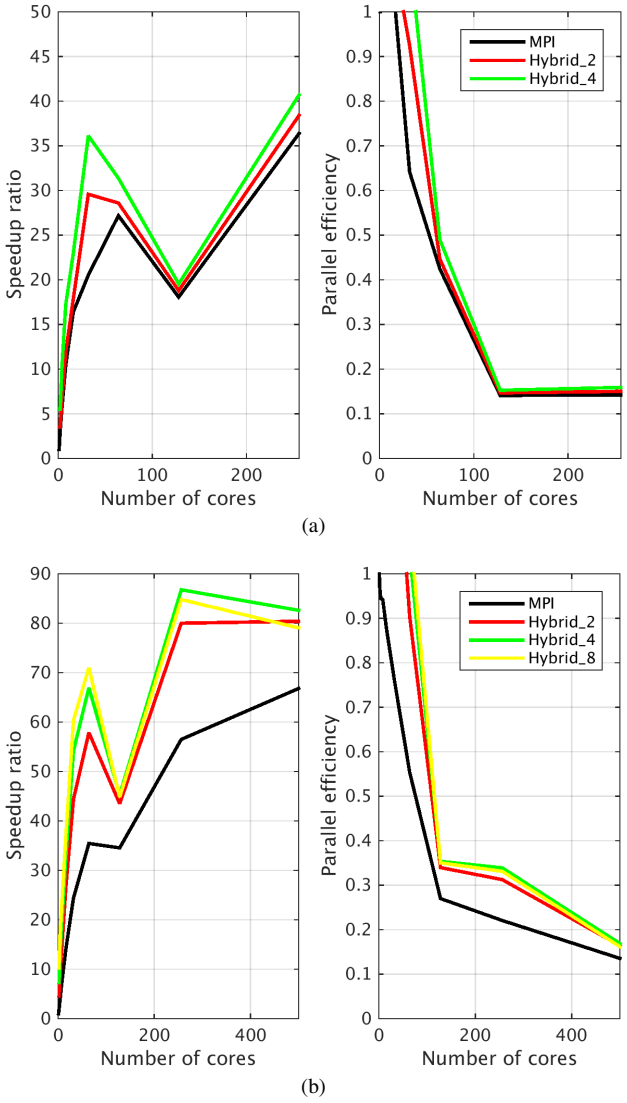


Fig. 3. Comparison of the speedup ratio and parallel efficiency when deploying both MPI and OpenMP computing paradigm on the P775 and BG/Q architecture. Scaling is for a typical application (LG) consisting of $450 \times 957$ grid cells.

of the code were computed:

$$S = \frac{T_1}{T_P} \qquad E = \frac{S}{N_p} \tag{4}$$

where $T_1$ is the time for simulation with the serial code; $T_p$ is the simulation time on $p$ processors; and $N_p$ is the number of processors.

Initially, the OpenMP and MPI implementations were compared for the TC to better understand fundamental performance and limitations. Comparisons were made on both the P775 and the BG/Q as they represent two entirely different compute environments. Figure 2 compares scaling for the different scenarios. On the BG/Q system, OpenMP outperforms MPI at low process counts. As expected however, the parallel efficiency of the loop-level parallelisation drops considerably as the thread count increases. The performance

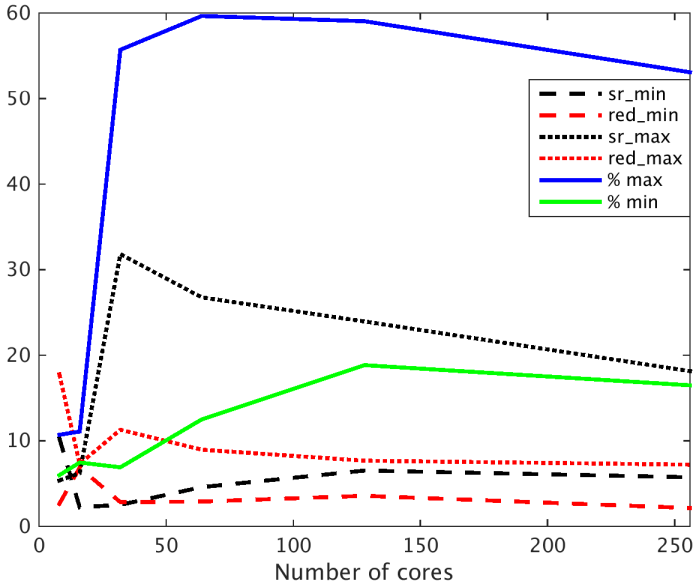


Fig. 4. The time spent in MPI calls during 200 time steps of model simulation versus the number of cores, as measured by the mpitrace utility. Time on the $y$-axis is in seconds. MPI calls are separated into point-to-point communications (sr_max, sr_min) and global reduction communications (red_min, red_max). In addition we present communication metrics for two cases: minimum and maximum time spent in communication. The minimum metric is the processor that spends the least amount of time in communication (dotted line) while the maximum value is the processor that spends the maximum amount of time in MPI calls (dashed line).

gain achieved from parallelisation is exceeded by thread initialisation overheads. Also, the problem size assigned to each thread is reduced, decreasing the ratio of computation and parallel overheads.

The second stage of the study applied the code to an actual case study (LG). Due to the large problem size, we focus on the MPI and hybrid MPI+OpenMP implementations to compare performance at higher node counts. Again, the code was deployed on the P775 and BG/Q architecture and we evaluated $S$ and $E$. Figure 3 presents scaling plot for both system deployments.

Results demonstrate that for this deployment, strong scaling is achieved up to about 64 cores. On the P775 we achieve parallel speedup of 31.37 and a parallel efficiency of 0.49. Similar scalability is achieved on the BG/Q system up to about 100 cores. Deploying on 128 cores provides speedup of 45.2 at a parallel efficieny of 0.35

An assesment of the communication overheads induced by MPI synchronisations provides insight into parallel bottlenecks. Of particular interest is the costs incurred by point-to-point communication (i.e. MPI_SEND and MPI_RECV) versus global, all-to-all communication (i.e. MPI_REDUCE). Point-to-point communication occurs at the end of each time step when the ghost layers of each domain are updated with values from the neighbouring partition. The preconditioned conjugate gradient solver required within the barotropic module involves both a global all-to-all reduction operation in addition to a point-to-point synchronization at each iteration. A second global communication is required within the flooding-and-drying routine of the code. Each partition makes a check on each cell within its domain to assess if any wet cells have become dry. A synchronisation across processors is required to activate a flooding-and-drying call if any wet cells within the global domain have become dry.

To better understand MPI procedures within the model, we use *mpitrace* to assess the time that the program spends making MPI calls. *Mpitrace* is a profiling and tracing tool which allows you to analyse the MPI communication events within your program, giving an overview of the makeup of your code and potentially an insight into any communication bottlenecks [21]. Figure 4 presents communication overhead for the model deployed on up to 256 cores, on the Blue Gene platform. To better assess communication effects we present the time spent in communication for the two extreme cases of: processor that spends *least* amount, and *most* amount of time in communciation. It is reasonable to assume that the main reason for differences is due to load imbalances. When the processor with the least computational load makes a global reduction call, it must wait for the slowest processor with the highest load to reach the synchronisation point and send its own chunk of data. Figure 4 also includes the precentage of the total simulation time that is spent in MPI synchronisation.

Analysis of MPI function calls illustrate the large variance in time spent in communication between the best and worst case task assignemnts. The max time spent in communication is $50 - 60\%$ compared to a minimum of $10 - 20\%$. If we assume load imbalances to be the primary contributing factor to the former then it seems reasonable to take the minimum time spent in communication to be representative of actual communication time excluding any idle processor time spent waiting. Interestingly, communication time does not increase significantly as number of cores increases. The communication time for both point-to-point and global communications is relatively flat, independent of number of cores. The flat profile of communications does however impact on the proportion of time spent in communication; as the number of cores increases, total computation time decreases and hence the proportion of time spent in communication increases.

## IV. Discussion and Conclusions

This paper describes the challenges of obtaining good scaling of a semi-implict code at a high node count. Results demonstrate that above $64 \sim 100$ cores, parallel performance drops significantly. For both a simple test case (TC) and larger application (LG), the code scales well at low numbers of cores. At higher numbers of cores, parallel efficiency diminishes as communication overheads introduced by the semi-implicit algorithm become punitive. Implementing a purely explicit numerical algorithm makes scaling at high core counts easier as communication is only neighbour-to-neighbour. However, for semi-implicit solvers, global reduction calls represent a challenging obstacle. Synchronisation introduced at each time step incurs a latency and communciation penalty; it also

exacerbates any load imbalances as computation is constrained to the slowest process.

Results presented here suggest that parallel performance at high node counts with a semi-implicit solver is a challenging undertaking. Load balancing is a critical consideration to avoid bottlenecks. Extending the rectilinear partitioning presented here to a more flexible orthogonal bisectioning technique is worthwhile as it helps to equilibrate load distributions across the fixed number of processors [22]. In addition, the replacement of the iterative conjugate gradient solver with an alternative that does not require global communication is likely to improve performance (e.g., Chebyshev iteration is a widely used iterative solver that does not require global reduction operations [16]). The preconditioned conjuate gradient technique has been shown to provide a more computationally efficient solution in fewer iterations. However, a detailed comparison of the performance of both for many-core simulations is worthy of further study.

## REFERENCES


[1] F. O. Donncha, E. Ragnoli, S. Zhuk, F. Suits, and M. Hartnett, "Surface flow dynamics within an exposed wind-driven bay: Combined hf radar observations and model simulations," in *Oceans, 2012*. IEEE, 2012, pp. 1–8.

[2] F. ODonncha, M. Hartnett, S. Nash, L. Ren, and E. Ragnoli, "Characterizing observed circulation patterns within a bay using $hf$ radar and numerical model simulations," *Journal of Marine Systems*, vol. 142, pp. 96–110, 2015.

[3] P. Thanh, M. D. Grace, and S. C. James, "Sandia national laboratories environmental fluid dynamics code: sediment transport user manual," *Sandia National Laboratories, Livermore, CA, SAND2008-5621*, 2008.

[4] S. C. James, C. A. Jones, M. D. Grace, and J. D. Roberts, "Advances in sediment transport modelling," *Journal of Hydraulic Research*, vol. 48, no. 6, pp. 754–763, 2010.

[5] S. C. James and V. Boriah, "Modeling algae growth in an open-channel raceway," *Journal of Computational Biology*, vol. 17, no. 7, pp. 895–906, 2010.

[6] S. C. James, V. Janardhanam, and D. T. Hanson, "Simulating ph effects in an algal-growth hydrodynamics model1," *Journal of Phycology*, vol. 49, no. 3, pp. 608–615, 2013.

[7] T. Khangaonkar, Y. Zhaoqing, C. Degasperi, and K. Marshall, "Modeling hydrothermal response of a reservoir to modifications at a high-head dam," *Water International*, vol. 30, no. 3, pp. 378–388, 2005. [Online]. Available: http://cat.inist.fr/?aModele=afficheNcpsidt=17100949

[8] Z. Ji, *Hydrodynamics and water quality: modeling rivers, lakes, and estuaries*. John Wiley & Sons, 2008.

[9] A. F. Blumberg and G. L. Mellor, "A description of a three-dimensional coastal ocean circulation model," *Coastal ane Estuarine Sciences*, vol. 4, pp. 1–16, 1987.

[10] S. Griffies, C. Boning, F. Bryan, E. Chassignet, R. Gerdes, H. Hasumi, A. Hirst, A. Treguier, and D. Webb, "Developments in ocean climate modelling," *Ocean Modelling*, vol. 2, no. 3, pp. 123–192, 2000.

[11] F. O'Donncha, E. Ragnoli, and F. Suits, "Parallelisation study of a three-dimensional environmental flow model," *Computers & Geosciences*, vol. 64, pp. 96–103, 2014.

[12] S. M. Griffies, R. C. Pacanowski, M. Schmidt, and V. Balaji, "Tracer conservation with an explicit free surface method for z-coordinate ocean models," *Monthly Weather Review*, vol. 129, no. 5, pp. 1081–1098, 2001.

[13] C. Pan, J. F. Prins, and C. T. Miller, "A high-performance lattice boltzmann implementation to model flow in porous media," *Computer Physics Communications*, vol. 158, no. 2, pp. 89–105, 2004.

[14] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and experience*, vol. 3, no. 5, pp. 457–481, 1991.

[15] J. M. Hamrick, "User's manual for the environmental fluid dynamics computer code," Virginia Institute of Marine Science, Tech. Rep., 1996.

[16] M. H. Gutknecht and S. Röllin, "The chebyshev iteration revisited," *Parallel Computing*, vol. 28, no. 2, pp. 263–283, 2002.

[17] R. Rajamony, M. W. Stephenson, and W. E. Speight, "The power 775 architecture at scale," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 183–192.

[18] E. Ragnoli, F. O'Donncha, J. W. Short, I. A. Hannoun, S. A. Nierzwicki-Bauer, L. A. Treinish, A. P. Priano, and C. W. Boylen, "Thermoclinic assessment of a preliminary circulation model for lake george in the jefferson project," in *HydroInformatics International Conference*, 2014.

[19] A. Deane, G. Brenner, D. R. Emerson, J. McDonough, D. Tromeur-Dervout, N. Satofuka, A. Ecer, and J. Periaux, *Parallel Computational Fluid Dynamics 2005: Theory and Applications*. Elsevier, 2006.

[20] Z. J. Wang, *Adaptive high-order methods in computational fluid dynamics*. World Scientific, 2011, vol. 2.

[21] *IBM Parallel Environment Developer Edition Server for Blue Gene/Q Systems: IBM High Performance Computing Toolkit*, IBM, 2008.

[22] M. Reumann, B. G. Fitch, A. Rayshubskiy, M. C. Pitman, and J. J. Rice, "Orthogonal recursive bisection as data decomposition strategy for massively parallel cardiac simulations," *Biomedizinische Technik/Biomedical Engineering*, vol. 56, no. 3, pp. 129–145, 2011.