A Hybrid MPI/PGAS Finite Element Solver

Niclas Jansson
School of Computer Science and Communication
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
Email: njansson@csc.kth.se

Johan Hoffman
School of Computer Science and Communication
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
Email: jhoffman@csc.kth.se

***Abstract*—We present our work on developing a hybrid parallel programming model for a general finite element solver. The main focus of our work is to demonstrate that legacy codes with high latency, two-sided communication in the form of message passing can be improved using lightweight one-sided communication. We introduce a new hybrid MPI/PGAS implementation of the open source finite element framework FEniCS, replacing the linear algebra backend (PETSc) with a new library written in UPC. A detailed description of the linear algebra backend implementation and the hybrid interface to FEniCS is given. We also present a detailed analysis of the performance of this hybrid solver on the Cray XE6 Lindgren at PDC/KTH including a comparison with the MPI only implementation, where we find that the hybrid implementation results in improvements of up to 33% in communication intensive parts of the solver.**

I. Introduction

The message passing paradigm has been the dominating programming model for writing highly scalable scientific applications for decades, among several implementations, the Message Passing Interface (MPI), has over the years come out as the de facto standard. Based upon a two-sided communication schematics, it is a challenge to handle large amount of fine-grained parallelism. A common optimization is to use non blocking communication, overlapping computation and communication. However, for certain applications the data dependency prevents this, and hence limits the scalability of applications.

Today, at the dawn of exascale computing, some concerns have been raised whether MPI is capable of delivering the needed performance. Therefore, researchers have started to investigate other programming models which could replace MPI, or use some hybrid incarnation combining MPI with something else. One such popular model is the MPI/OpenMP combination, where parts of the MPI tasks are replaced with threads, and hence reduces the number of two-sided communication requests.

Apart from the hybrid models there has also been a push forward on developing new programming models and languages, one good example is the Partitioned Global Address Space (PGAS) languages. Based on the abstraction of a global shared address space (built on top of the distributed global memory) it is a simple and elegant model, especially for algorithms with challenging data dependencies. With its one-sided communication abstraction it is also an efficient model for fine-grained parallelism.

However, despite all the appealing features of PGAS languages, they are seldom used in production codes, which is understandable. Given the tremendous amount of high quality scientific software which has been written in MPI over the past decades, it would be unreasonable to think that time and money are going to be invested in rewriting or replacing this with something new. Therefore, we argue that a reasonable way to prepare old legacy codes for exascale computing is to replace bits and pieces with more scalable one-sided communication, thus creating hybrid MPI/PGAS applications which to our knowledge is a quite unusual combination.

Building on our previous work [1] on optimizing sparse matrix assembly using one-sided communication, we here introduce a fully hybrid MPI/PGAS finite element solver based on the open source framework of FEniCS [2], as an extension of our existing finite element solver DOLFIN/Unicorn [3], [4]. In this work we focus on replacing the MPI based linear algebra parts of the finite element framework with a PGAS implementation, JANPACK [5]. In experiments we find that the hybrid implementation results in improvements of up to 33% in communication intensive parts of the solver.

The outline of the paper is the following; In §II a short background is given and related work are discussed, our finite element solver is presented in §III and §IV. In §V and §VI we present our parallelization strategy and implementation details, our experimental setup and performance evaluation is given in §VII-§IX, and in §X we discuss some shortcomings of our approach. We give conclusions and outline future work in §XI.

II. Background and related work

A large scale finite element simulation can often be decomposed into three major components; mesh partitioning, assembly and solution of linear systems. The first step decomposes the problem by splitting up the large, often unstructured mesh using a graph partitioner, to assign the separated smaller pieces to different processing elements (PEs).

Unstructured meshes are excellent for accurate approximation of complex geometries. However, the lack of underlying structure implies an unstructured communication pattern, which can have a negative effect on the overall performance, in particular for the assembly stage on a large number of PEs, as we have observed in our previous work [6].

The reason for this negative behavior is partly due to the programming model used (message passing) and its two-

sided communication abstraction. For a large number of PEs, the need to match send and receive messages will unavoidably increase latency and synchronization costs. Non-blocking communication is often employed to lower this cost, albeit it requires the receiver to occasionally check for messages which introduces latency and additional overhead costs.

One-sided communication, with its low latency, is in theory the perfect solution to these problems, but the question is what languages to use. For legacy codes using MPI, there is hope with the introduction of Remote Memory Access (RMA) operations in MPI 2.0. The library is able to perform true one-sided communication, but unfortunately the API imposes a number of restrictions that limit the usability of these extensions. For a discussion of these constraints, see for example [7].

Since MPI 2.0 is not a reliable solution, one option is the extreme of rewriting the entire code in a language with the one-sided communication abstraction, for example a PGAS language. The possible gains from this approach is convincingly illustrated in [8], [9], where an entire unstructured finite element solver for flow problems is implemented in a PGAS language. Not only does the code scale well, but also the global memory abstraction allows for a simple and efficient implementation of many algorithms.

However, as discussed in the introduction, rewriting a large code base into PGAS is often not feasible, which leaves the hybrid approach most viable. Related work in this area is sparse, most of the work is focused on support in the runtime system [10], [11] or the applicability of hybrid methods [12], with the exception of [13] which presents an entire large scale application rewritten using a hybrid PGAS/OpenMP approach.

The main contribution of our work is to present a path forward from legacy MPI codes with the hybrid MPI/PGAS model.

III. Automated adaptive finite element solver

A. A general finite element solver

Writing high performance software for scientific computations is a delicate task. These codes are often developed on an application to application basis, highly optimized to solve a certain problem. The FEniCS project [2] seeks to automate the scientific software process. Instead of developing one code per application, the goal is to have one general code that solves a large class of problems in an automated fashion. In FEniCS, a solver takes the equation and discretization method as input in a high level language close to mathematical notation. Low-level assembly functions are then generated by a FEniCS compiler. This means that the development of physical models and discretization methods can be done on a high level, implying robustness and enabling high speed of development.

The core part of FEniCS is the Object-Oriented finite element library DOLFIN [14], [15], from which we have developed a high performance branch [16] for distributed memory architectures. DOLFIN handles mesh representation and finite element assembly but relies on external libraries for solving the linear systems. Our high performance branch also extends DOLFIN with parallel mesh refinement and dynamic load balancing capabilities [3].

On top of DOLFIN we have then developed Unicorn [4], [17], a unified continuum mechanics solver with adaptive mesh algorithms based on a posteriori error estimation. In this paper we focus on the Unicorn fluid mechanics solver based on solving the Navier-Stokes equations, which has shown to scale well both strongly and weakly [4], [6].

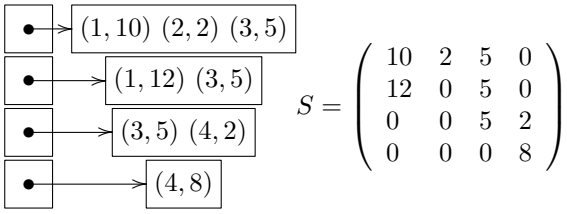

Fig. 1. An illustration of the stack-based representation of the matrix S.

B. Towards a hybrid FEM solver

With the goal of creating a hybrid finite element solver, the question is which part of the framework should be rewritten. Since DOLFIN is heavily Object-Oriented there are several nontrivial constraints on each component. The most natural first step was to rewrite an external library, in this case the linear algebra backend.

In previous work on sparse matrix formats [18] we have found that the most common format Compressed Row Storage (CRS) [19] is sub optimal when it comes to sparse matrix assembly. CRS has an efficient access pattern for Sparse Matrix Vector Multiplication (SPMV) but on the fly insertion of elements can however be costly. Instead we propose a new format based on a stack-based representation, which is similar to the linked-list data structure where each row is represented by a linked list. With the low latency of PGAS, it shows greatly improved assembly rates [1].

IV. Stack-based representation

A stack-based representation of a sparse matrix is based around a long array A, with the same length as the number of rows in the matrix. For each entry in the array $A(i)$, we have a stack $A(i).rs$ holding tuples (c, v) representing the column index c and element value v, hereby referred to as a row-stack, illustrated in Fig. 1. Inserting an element into the matrix is now straightforward. Namely, find the corresponding row-stack and push the new (c, v) tuple on the top. Matrix updates (a common operation during FEM assembly), such as adding a value to an already inserted element, is also straightforward: Find the corresponding row-stack, perform a linear search until the correct (c, v) tuple is found and add the value to v, as illustrated in Fig. 2.

Compared to CRS, the stack based format removes the indirect addressing needed to find the start of a row. Also these stacks do not need to be ordered. Thus we could push new elements regardless of the column index. In the case of a matrix update, the linear search will still be efficient since

```
for j = 1 : length(A(i).rs) do
  if A(i).rs(j).c == c then
    A(i).rs(j).v += v
    return
  end if
end for
push (c, v) onto the row-stack A(i).rs
```

Fig. 2. Pseudo-code for matrix update ($A_{i,c} += v$).

each stack has a short length equal to the number of non-zeros in the corresponding row.

V. Parallelization strategy

A. Finite element framework

DOLFIN is parallelized using a fully distributed mesh approach, where everything from preprocessing, assembly of linear systems and postprocessing is performed in parallel, without representing the entire problem or any pre-/postprocessing step on one single processing element. Each PE is assigned a whole set of elements, defined initially by the graph partitioning of the corresponding dual graph of the mesh. The vertex overlaps between PEs are represented as ghosted entities.

Since whole elements are assigned to each PE, assembly of linear systems based on evaluation of element integrals can be performed in a straight forward way with low data dependency. Furthermore, we renumber all the degrees of freedom such that a minimal amount of communication is required when modifying entries in the sparse matrix.

B. Linear algebra backend

The parallelization of the linear algebra backend is based on a row wise data distribution. Each PE is assigned a continuous set of rows, such that the first PE is assigned rows $0 \dots n$, the second $n+1 \dots m$, and so forth. Vectors follow the same strategy, such that the first $0 \dots n$ elements are assigned to the first PE. Overlapping rows are not supported for the matrix but in the vector, represented as a list of ghost indices and stored in a local hash table.

1) Matrix assembly: We adhere to the two phase assembly process, where entries are inserted into a matrix or vector in two phases. The first phase only inserts entries belonging to the local PE. All other entries are placed in a local staging area for later processing. In the second phase each PE fetch the part of each other PE's staging areas that corresponds to its own entries. Possible communication contention is reduced by pairing PE's together, such that each PE copies data from the PE with number $\mod(\mathrm{PE}+i, N_{\mathrm{PE}})$, where N_{PE} is the total number of PEs. For static sparsity pattern, we use the initial assembly to gather dependency information such that consecutive assemblies can be optimized, only fetching data from PEs in the list of dependencies, as illustrated in the pseudo-code for the matrix assembly in Fig. 3.

```
if Initial assembly then
  for i = 1 : N_PE do
    src = mod(PE + i, N_PE))
    if length(staging_area(src, PE)) > 0 then
      data = memget(staging_area(src, PE))
      for j = 1 : length(data) do
        add data(j) to matrix
      end for
      add src to list of dependencies (dep)
    end if
  end for
else
  for i = 1 : length(dep) do
    src = dep(i)
    data = memget(staging_area(src, PE))
    for j = 1 : length(data) do
      add data(j) to matrix
    end for
  end for
end if
```

Fig. 3. Pseudo-code for finalization of matrix assembly.

2) Matrix vector multiplication: For the matrix vector multiplication, $y = Ax$ the vector entries in x that are not own by the local PE must be fetched before the multiplication can be performed. In order to optimize the communication we group vector entries together into logical blocks. For each local chunk of the global vector we define a set of blocks by using a load-balanced linear data distribution [20]; Let N_b be the number of logical blocks, N_v be the local size of a vector. Then the length of each chunk can be written as $N_v = N_b L + R$ given that $0 \leq R < N_b$, with:

$$L = \left\lfloor \frac{N_v}{N_b} \right\rfloor \quad R = N_v \bmod N_b$$

During matrix assembly, an algorithm sweep through the matrix columns and marks blocks corresponding to the dependencies. Given the global column index j, the start offset for each local chunk, the block index i, is given by:

$$i = \max\left(\left\lfloor \frac{j - \mathrm{offset}}{L+1} \right\rfloor, \left\lfloor \frac{(j-\mathrm{offset}) - R}{L} \right\rfloor \right)$$

Furthermore, we also want to stress that for problems with static sparsity pattern, this scan is only needed in the initial assembly.

Once the list of dependencies (matvec_dep) has been built the multiplication routine iterates through the list and fetches all marked blocks from remote PEs. When all the missing blocks have been fetched the multiplication is performed without any further communication, as illustrated in the pseudo-code in Fig. 4.

VI. Implementation

DOLFIN is written in C++, parallelized using MPI, uses ParMETIS [21] for mesh partitioning. PETSc [22] is used

for all matvec_dep **do**
  for all marked blocks b_{marked} **do**
    memget(b_{marked}) (non blocking)
  end for
end for
Memory fence
Compute $y = A\,x$

Fig. 4. Pseudo-code for matrix vector multiplication $y = Ax$.

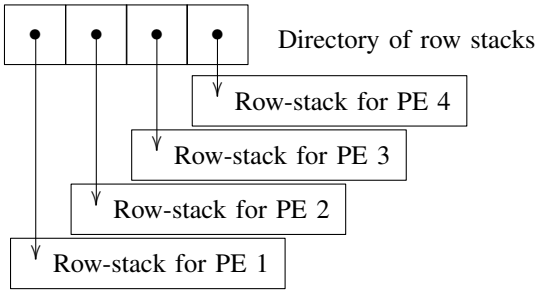

Fig. 5. An illustration of the directory of objects representation.

for linear algebra and will be used as a baseline for our comparison in this paper.

For the new linear algebra backend we used Unified Parallel C (UPC) [23], a C like language that extends ISO C99 with PGAS constructs. In UPC the memory is partitioned into a private and a global space. Memory can then either be allocated in the private space as usual or in the global space using UPC provided functionality. Once memory is allocated in the global space it can be accessed in the same manner on all threads.

A. Directory of objects representation

For performance reasons, memory in UPC is allocated in blocks with affinity to a certain thread. In our application, the determination of an optimal block size is an impossible task. Since a FEM discretization of an unstructured mesh is almost guaranteed to not be evenly divisible with the number of PEs, so how should the block size be chosen. The straightforward solution with fixed block sizes, would then not only waste memory but only causes an irrecoverable load imbalance.

The solution to this problem is to use a technique called directory of objects, where a list of pointers is allocated in the global space such that each PE has affinity to one pointer. Each pointer then points to a row-stack, allocated in the global space with affinity to the same PE as the pointer. This technique enables us to have unevenly distributed global memory, and each piece can grow or shrink independently of each other.

B. Hybrid interface

Mixing different programming languages in scientific code has always been a cause for headache and portability issues. Since there is no C++ versions of UPC we had to use an interface that did not expose the UPC specific data structures to the DOLFIN C++ code. To overcome this problem we access the UPC data types from DOLFIN as opaque objects [24]. On the C++ side we allocate memory for an object of the same size as the UPC data type and the object is never accessed by the C++ code. All modifications are done through the interface to the UPC library. This technique enables us to access exotic UPC types from C++ with only minor portability issues, except to determine the size of the data type for each new platform on which the library is compiled.

We employ a flat runtime model, mapping MPI ranks to UPC threads one-to-one. Hence, MPI rank n would be assigned to UPC thread n. This was handled automatically by the Cray runtime system (see §VIII). Also, in order to minimize possible runtime problems we ensured that no UPC and MPI communication overlapped.

VII. Benchmark problem

As mentioned in §III we restrict our test to the Navier-Stokes solver in Unicorn, but we stress that the framework is general and can be used for any kind of application that DOLFIN can handle. Our chosen problem is turbulent flow past a circular cylinder at high Reynolds number ($Re = 10^4$).

The solver, is an implicit LES flow solver, based on the General Galerkin (G2) method [25], which corresponds to a standard Galerkin finite element method with numerical stabilization based on the residual of the equations.

We compute the solution for the G2 formulation by solving a nonlinear system of algebraic equations for each time-step using a fixed-point iteration. With velocity given by the previous iteration we first solve for the pressure, which is then used to solve for the momentum. Since we use an implicit time-stepping method both the pressure and momentum matrices need to be reassembled for each fixed-point iteration.

Both pressure and momentum are solved for using a preconditioned BiCGSTAB Krylov subspace method. As preconditioner we used the best (as in wall time not convergence rate) for each linear algebra backend. For PETSc this turned out to be block Jacobi for both equations, where each block is solved for with ILU(0). With JANPACK the optimal combination was a simple diagonal Jacobi preconditioner for the momentum and a block Jacobi for the pressure were each block is solved for with a diagonal ILU (D-ILU) [26].

VIII. Experimental Platform

This work was performed on a 1516 node Cray XE6, called Lindgren, located at PDC/KTH. Each node consists of two 12-core AMD "Magny-Cours" running at 2.1 GHz, equipped with 32GB of RAM. The Cray XE6 is especially well suited for our work since its Gemini interconnect provides hardware accelerated PGAS support. The interconnect can transmit using two different methods; Fast Memory Access (FMA) and Block Transfer Engine (BTE). In general FMA is intended for short messages, and involves the PE in the communication. However, several transfers can be performed at once. BTE is better suited for long messages and transmits the data asynchronously, offloading the work from the PE to the Gemini chip.

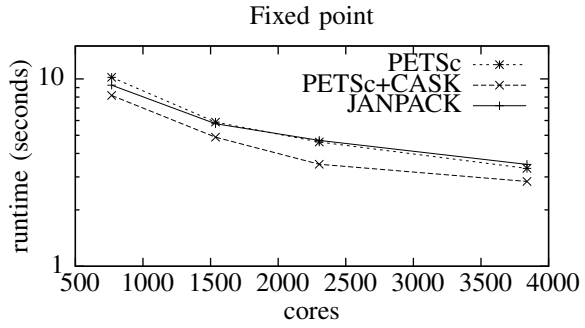

Fig. 6. Average time for computing a time-step (solving the fixed-point iteration).

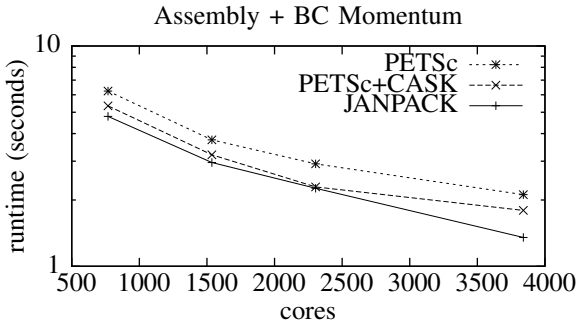

Fig. 7. Average time to assemble and applying boundary conditions to the momentum matrix.

We used the Cray Compiler Environment (CCE) version 7.4.0 to compile everything. For the new PGAS based linear algebra backend, we used the C compiler with UPC enabled (`-hupc`) and compiled a library. For DOLFIN we used the C++ compiler to compile another library for the finite element framework, but without any PGAS flag. Finally, when compiling the flow solver we had to add the UPC flag during linking in order to create a working executable.

For the referenced MPI solver we used two different implementations of PETSc version 3.1.05. First the reference implementation from the projects website and secondly Cray's own implementation, heavily optimized using the Cray Adaptive Sparse Kernels (CASK) library.

IX. Performance analysis

We evaluated the performance of our solver by computing solutions to the benchmark problem in §VII on an unstructured tetrahedral mesh, consisting of approximately 11M vertices and 59M elements. To collect performance data we ran the benchmark on 768 up to 3840 PEs, and for all runs we used 2MB huge pages.

A. Fixed-point iteration

We let the solver compute several time-steps and measured how long each step took. In Fig. 6 the average time for computing a time-step is presented. From the results we can see that our hybrid solver is performing on par with PETSc, achieving up to 95% of the reference implementations efficiency at highest concurrency. However, in Fig. 6 we also see how well PETSc+CASK performs. With CASK, PETSc's performance is increased by 17%. In order to understand why our solver only achieved 95% efficiency we need to break down the fixed-point iteration into smaller pieces. Recall from the description of the solver in §VII, that we solve two systems in each iteration and since it is an implicit scheme we also need to assemble two matrices.

B. Matrix assembly

In Fig. 7 we give the average time for assembling and applying the boundary condition on the larger momentum matrix. For this communication intensive part we see a clear benefit of using low-latency PGAS. In Fig. 7 we see that our hybrid implementation is fastest, beating the reference implementation by almost 57% and the CASK implementation with 33% for the largest number of PEs. These result are in good agreement with our previous benchmarks [1], where we found that for communication dominated problems (Poisson), JANPACK inserts elements almost up to 5.3 times faster than PETSc (CASK). For computational dominated problems we found that for convection-diffusion, JANPACK inserts elements twice as fast compared to PETSc.

The excellent assembly performance for JANPACK is solely due to the low latency of PGAS. In the finalization step of sparse matrix assembly, where all data not belonging to a certain PE is moved to the correct owner, there is unavoidable need for a lot of communication. The common optimization techniques for message passing libraries is based on non blocking communication, such that matrix assembly is initiated and the cost of transferring data is covered by local computations. However, in our solver there is no such overlap, since we need to assemble the entire matrix before we can proceed with any computations. Therefore, the low latency of PGAS in combination with the low insertion cost of the stack based format enable us to beat PETSc in this part of the solver.

C. Linear solvers

For the linear solver we measured the most interesting quantity, namely how long it took to converge to a solution. Since we used different preconditoners, this was the only reasonable thing to measure. In Fig. 8 the average convergence time for the momentum equation is presented. Here we see that our solver scales reasonable well compared to PETSc (both versions), but is in general slower, due to the less tuned matrix vector multiplication kernel in JANPACK.

However, it turned out that it was crucial to tune how many logical blocks we used to divide the vector into (see §V). With too few blocks, fetching off-PE dependencies (see Fig. 4) took longer time due to the larger amount of data to copy. In general we observed that a small block size improved the performance, which agree with the observation made in [27]. We also made the same observation that one needs to tune the threshold value to switch between FMA and BTE (controlled by the environment variable `PGAS_OFFLOAD_THRESHOLD`).

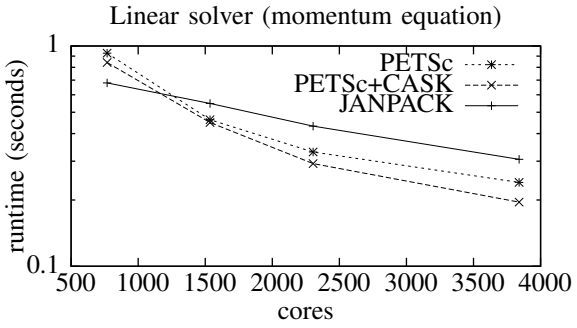

Fig. 8. Average time to solve the momentum equation.

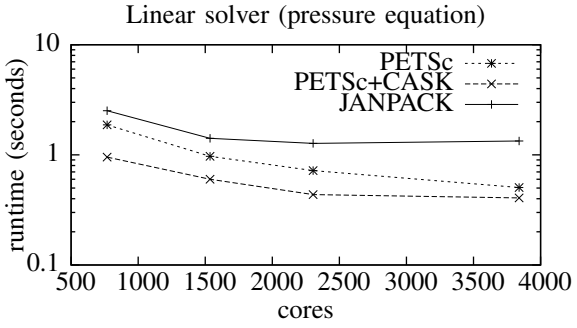

Fig. 9. Average time to solve the pressure equation.

As in [27] we achieved the best result by delaying the switch to BTE by increasing the threshold to 1MB. In our case it was also desirable to use FMA together with the non-blocking remote memory fetch (`upc_memget_nbi`), an UPC extension provided by Cray. We could then initiate several transfers at once in the algorithm and wait (at the memory fence) until they complete (see pseudo-code in Fig. 4). However, using non-blocking memory fetches turned out to be slightly dangerous. With too small block sizes the large amount of initiated remote memory fetches could either exhaust the Gemini's resources or in the worst case cause irrecoverable transaction errors.

For the pressure equation the average convergence times are presented in Fig. 9. Here the results for our hybrid solver is less than encouraging, and in fact the pressure solver is the limiting factor. We believe that this is due to a combination of a sub optimal implementation of our D-ILU preconditioner in combination with a slightly lower matrix vector multiplication performance. Although the slower performance we want to point out that our pressure solver has a similar scalability behaviour as PETSc with CASK (see Fig. 9), hence the problem lies in our sparse matrix representation and is not due to PGAS.

X. Discussion

From the performance evaluation we have shown that the hybrid solver's overall performance is acceptable (within 95% of PETSc), but there are some drawbacks of our approach that need to be addressed. The main bottleneck lies in the alternative stack-based representation. Initially designed to improve matrix assembly, its alternative format has proven to be more challenging to use when implementing efficient linear algebra kernels. However, since the main goal of this paper is to prove the applicability of hybrid MPI/PGAS methods, this is a minor issue. Overall we see that PGAS and in particular UPC can be faster than MPI, in contrary to the conclusions in [28], [29]. But in order to be fair, these evaluations were performed on platforms without hardware accelerated one-sided communication.

Besides the implementation details, our hybrid approach also has some drawbacks when it comes to productivity. We have to acknowledge that PGAS offers great potential when it comes to ease of programming and expressiveness. However, this has not yet influenced current available development tools, which have a direct impact on the performance. Furthermore, the lack of support for celestial linking of MPI and UPC libraries in most application development environments also limits the portability of our approach. Hopefully, these are things that will improve in the future.

XI. Summary and Conclusion

In this paper we have presented an alternative programming model combining traditional message passing with low latency one-sided communication. We demonstrate the applicability of our model by evaluating the performance of a hybrid MPI/PGAS finite element solver on a Cray XE6. The new hybrid model has potential with large improvements in communication intensive parts of the solver.

Our PGAS based linear algebra backend used an alternative sparse matrix representation which shown good performance for sparse matrix assembly, but lower performance for matrix vector multiplication. Since the main contribution of this paper was to present a path forward from legacy MPI codes, optimizing sparse matrix vector multiplication is left as future work.

To summarize, our results demonstrate that a hybrid MPI/PGAS model can improve the performance of communication bound problems, and offer an alternative to complete rewrite of legacy MPI codes when preparing them for future platforms.

Acknowledgment

The authors would like to acknowledge the financial support from the Swedish Foundation for Strategic Research, the European Research Council and the Swedish Research Council. The research was performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC - Center for High-Performance Computing.

References

[1] N. Jansson, "Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-sided Communication," ser. VECPAR'12, July 2012, to appear (accepted).

[2] FEniCS, "FEniCS project," 2003, http://www.fenicsproject.org.

[3] N. Jansson, J. Hoffman, and J. Jansson, "Framework for Massively Parallel Adaptive Finite Element Computational Fluid Dynamics on Tetrahedral Meshes," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. C24–C41, 2012.

[4] J. Hoffman, J. Jansson, R. V. de Abreu, N. C. Degirmenci, N. Jansson, K. Müller, M. Nazarov, and J. H. Spühler, "Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for deforming domains and complex geometry," *Computers & Fluids*, no. 0, 2012, in press.

[5] N. Jansson, "JANPACK," 2012, http://www.csc.kth.se/~njansson/janpack.

[6] N. Jansson, J. Hoffman, and M. Nazarov, "Adaptive Simulation of Turbulent Flow Past a Full Car Model," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, State of the Practice Reports*, ser. SC '11, 2011.

[7] D. Bonachea and J. Duell, "Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations," *Int. J. High Perform. Comput. Networking*, vol. 1, pp. 91–99, 2004.

[8] A. A. Johnson, "Computational Fluid Dynamics Applications on the Cray X1 Architecture: Experiences, Algorithms, and Performance Analysis," in *CUG 2003*, 2003.

[9] ——, "Using Unified Parallel C to Enable New Types of CFD Applications on the Cray X1E," in *CUG 2006*, 2006.

[10] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick, "Hybrid PGAS runtime support for multicore nodes," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:10.

[11] J. Jose, M. Luo, S. Sur, and D. K. Panda, "Unifying UPC and MPI runtimes: experience with MVAPICH," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:10.

[12] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with MPI and unified parallel C," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 177–186.

[13] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges, "Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 78:1–78:11.

[14] A. Logg and G. N. Wells, "DOLFIN: Automated finite element computing," *ACM Trans. Math. Softw.*, vol. 37, no. 2, pp. 1–28, 2010.

[15] A. Logg, G. N. Wells, J. Hake *et al.*, "DOLFIN: A C++/Python finite element library," 2011, http://launchpad.net/dolfin.

[16] N. Jansson, "High performance adaptive finite element methods for turbulent fluid flow," Licentiate Thesis, KTH Royal Institute of Technology, School of Computer Science and Communication, 2011, TRITA-CSC-A 2011:02.

[17] J. Hoffman, J. Jansson, M. Nazarov, and N. Jansson, "Unicorn," 2011, http://launchpad.net/unicorn/hpc.

[18] N. Jansson, "Data Structures for Efficient Sparse Matrix Assembly," Computational Technology Laboratory, Tech. Rep. KTH-CTL-4013, 2011, http://www.publ.kth.se/trita/ctl-4/013/.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[20] E. F. V. de Velde, *Concurrent Scientific Computing*, ser. Texts in Applied Mathematics. Springer-Verlag, 1994, vol. 16.

[21] K. Schloegel, G. Karypis, and V. Kumar, "ParMETIS, Parallel graph partitioning and sparse matrix ordering library," University of Minnesota, Departement of Computer Science and Army HPC Research Center, 2011.

[22] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2009, http://www.mcs.anl.gov/petsc.

[23] UPC Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[24] A. Pletzer, D. McCune, S. Muszala, S. Vadlamani, and S. Kruger, "Exposing Fortran Derived Types to C and Other Languages," *Comput. Sci. Eng.*, vol. 10, no. 4, pp. 86 –92, july-aug. 2008.

[25] J. Hoffman and C. Johnson, *Computational Turbulent Incompressible Flow*, ser. Applied Mathematics: Body and Soul. Springer, 2007, vol. 4.

[26] C. Pommerell, "Solution of large unsymmetric systems of linear equations," Ph.D. dissertation, Swiss Federal Institute of Technology, Zürich, Switzerland, 1992.

[27] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, "A preliminary evaluation of the hardware acceleration of the Cray Gemini Interconnect for PGAS languages and comparison with MPI," in *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, ser. PMBS '11. ACM, 2011, pp. 13–14.

[28] C. Coarfă, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47.

[29] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 174–184.