Improving Parallel Performance of FEniCS Finite Element Computations by Hybrid MPI/PGAS

Niclas Jansson
High Performance Computing and Visualization
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
njansson@kth.se

Johan Hoffman
High Performance Computing and Visualization
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
jhoffman@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 general 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 significant improvements in performance of the solver.

Keywords

Hybrid Programming Models, PGAS, UPC, FEM, CFD

1. 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 completely new. Therefore, we argue that one 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 [14] 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 [20], as an extension of our existing finite element solver DOLFIN/Unicorn [15, 9]. In this work we focus on replacing the MPI based linear algebra parts of the finite element framework with a PGAS implementation, JANPACK [13]. In experiments we find that the hybrid implementation results in significant improvements in performance of the solver.

The outline of the paper is the following; In §2 a short background is given and related work are discussed, our finite element solver is presented in §3 and §4. In §5 and §6 we present our parallelization strategy and implementation details, our experimental setup and performance evaluation is given in §7-§9, and in §10 we discuss the performance results and highlight some shortcomings of our approach. We give conclusions and outline future work in §11.

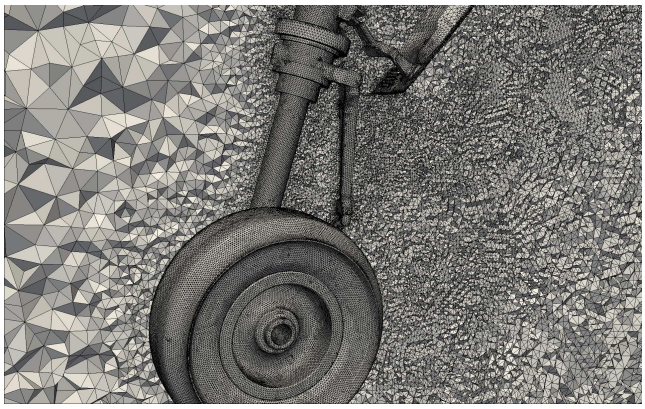


Figure 1: An illustration of the unstructured mesh around a Gulfstream G550 nose landing gear [5].

2. 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 as illustrated in Figure 1. This 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 [16].

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 language 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 [3].

Since MPI 2.0 is not a reliable solution today, one option is the extreme of rewriting the entire code in a language using the one-sided communication abstraction, for example a PGAS language. The possible gains from this approach is convincingly illustrated in [17][18], 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 [2][19] or the applicability of hybrid methods [7], with the exception of [25] which presents an entire large scale application rewritten using a hybrid PGAS/OpenMP approach.

The main contribution of our work is to demonstrate the applicability of PGAS in terms of performance and present a path forward from legacy MPI codes with the hybrid MPI/PGAS model.

3. AUTOMATED SCIENTIFIC COMPUTING

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 [20] 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.

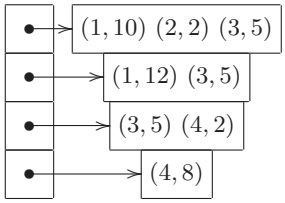
The core part of FEniCS is the Object-Oriented finite element library DOLFIN [21], from which we have developed a high performance branch [12] for distributed memory architectures. DOLFIN handles mesh representation and finite element assembly but relies on external libraries for solving the linear systems. Our high performance branch also extends DOLFIN with parallel mesh refinement and dynamic load balancing capabilities [15].

On top of DOLFIN we have then developed Unicorn [9], 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 [9][16].

3.1 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 replace one of the external libraries, in this case the linear algebra backend.

In previous work on sparse matrix formats [11] we have found that the most common format Compressed Row Storage (CRS) [26] is sub optimal when it comes to sparse matrix assembly, due to the high cost of on the fly insertion of elements. CRS has a very efficient access pattern for Sparse Matrix Vector Multiplication (SPMV), and since most applications only assembles the matrix once, the poor assembly performance is often neglected. However, in implicit finite element solvers, the matrices often needs to be reassembled



$$S = \begin{pmatrix} 10 & 2 & 5 & 0 \\ 12 & 0 & 5 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Figure 2: An illustration of the stack-based representation of the matrix S.

in every time-step. Therefore assembly performance is as important as SPMV performance since the assembly cost can not be amortized by a fast multiplication routine. Therefore, instead of using CRS 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 [14].

4. 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 Figure 2. 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 Algorithm 1.

Algorithm 1: Matrix update $(A_{i,c} += v)$.

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

Compared to CRS the stack based format removes the indirect addressing needed to find the start of a row, and has in general a lower reallocation cost since each stack can be grown independently. 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 each stack has a short length equal to the number of non-zeros in the corresponding row.

However, the representation's SPMV performance is lower than CRS mostly due to the lack of locality when iterating over all the rows. Therefore, for problems with a static sparsity pattern we rearrange the row-stacks such that they are placed in consecutive memory locations. Hence, we convert the format into something similar to CRS. Also, since the conversion is only performed once, after the initial assembly, the extra cost is negligible for a solver after a number of time steps.

5. PARALLELIZATION STRATEGY

5.1 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.

5.2 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 $0, \ldots, n$, the second $n+1, \ldots, m$, and so forth. Vectors follow the same strategy, such that the first $0, \ldots, n$ 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.

5.2.1 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 fetches the part of each other PE's staging area 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 Algorithm 2.

5.2.2 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 [6]; 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:

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

Algorithm 2: Finalization of matrix assembly.

```
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
            add src to list of dependencies (dep);
        end
    end
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
    end
end
```

During matrix assembly, an algorithm sweeps 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:

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

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 Algorithm 3.

Algorithm 3: Matrix vector multiplication $y = Ax$.

```
forall the matvec_dep do
    forall the marked blocks b_marked do
        memget(b_marked) (non blocking) ;
    end
end
Memory fence;
Compute y = A x;
```

6. IMPLEMENTATION

DOLFIN is written in C++, parallelized using MPI, and uses ParMETIS [27] for mesh partitioning. PETSc [1] is used 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) [29], 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.

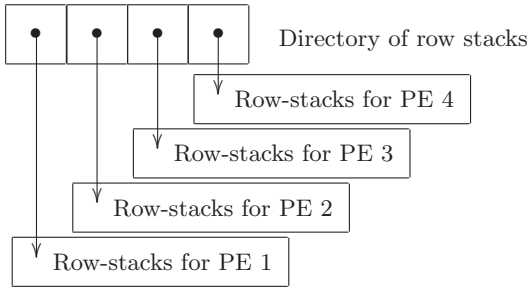


Figure 3: An illustration of the directory of objects representation.

6.1 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 not to be evenly divisible with the number of PEs, so how should the block size be chosen. The straightforward solution with fixed block sizes would not only waste memory but also 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, as illustrated in Figure 3.

6.2 Hybrid interface

Mixing different programming languages in scientific code has always been a cause for headache and portability issues. Since there is no C++ version of UPC we had to use an interface that did not expose the UPC specific data structures to the DOLFIN's C++ code. To overcome this problem we access the UPC data types from DOLFIN as opaque objects [23]. 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.

For example, the code given in Figure 4(a) illustrates how an UPC matrix (`jp_mat_t`) and its initialization function (`jp_mat_init`) are defined. As mentioned above, since shared pointers are illegal in C++, we have to redefine the function on this side as illustrated in Figure 4(b).

This technique enables us to access exotic UPC types from C++ with only minor portability issues, but it is necessary to determine the size of the data type for each new platform on which the library is compiled.

```
typedef struct {
  shared[] row_stack *shared *a_dir;
  row_stack *rs;
  ...

} jp_mat_t;

int jp_mat_init(jp_mat_t *restrict A,
                uint32_t m,
                uint32_t n);
```

(a) Definition of the matrix on the UPC side.

```
extern "C" {
  int jp_mat_init(char *restrict A,
                  uint32_t m,
                  uint32_t n);

}

char A[192]; /* sizeof(jp_mat_t) */

jp_mat_init(A, M, N);
```

(b) Interface to the matrix on the C++ side.

Figure 4: An illustration of the hybrid interface.

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 §8). Also, in order to minimize possible runtime problems we ensured that no UPC and MPI communication overlapped.

7. BENCHMARK PROBLEM

As mentioned in §3 we restricted 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 [10], 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 in 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 where each block is solved for with a diagonal ILU (D-ILU) [24].

8. 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.

We used the Cray Compiler Environment (CCE) version 8.0.6 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. First as a reference implementation, PETSc version 3.1 from the project's website and secondly Cray's own implementation of version 3.2, heavily optimized using the Cray Adaptive Sparse Kernels (CASK) library.

9. PERFORMANCE ANALYSIS

We evaluated the performance of our solver by computing solutions to the benchmark problem in §7 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 6144 PEs, and for all runs we used 2-8MB huge pages, depending on the number of PEs used.

9.1 Fixed-point iteration

We let the solver compute several time-steps and measured how long each step took. In Figure 5 and Table 1 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 slightly better performance than the optimized implementation at highest concurrency. Furthermore, in Figure 5 we also see how well PETSc+CASK performs. With CASK, PETSc's performance is increased by almost a factor of two. In order to better understand our solvers performance we break down the fixed-point iteration (the main time stepping loop) into smaller pieces. Recall from the description of the solver in §7, that we solve two systems in each iteration and since it is an implicit scheme we also need to assemble two matrices.

9.2 Matrix assembly

In Figure 6 and Table 2 we give the average time for assembling the larger momentum matrix. For this communication intensive part we see a benefit of using low-latency PGAS. In Figure 6 we see that our hybrid implementation is fastest,

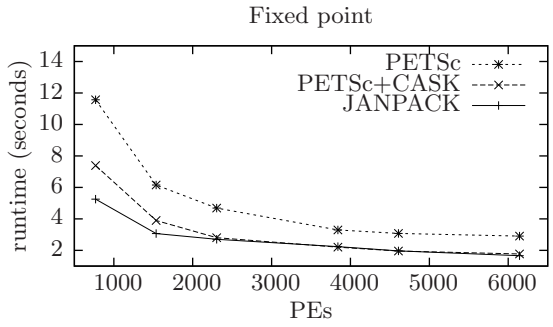


Figure 5: Average time for computing a time-step (solving the fixed-point iteration).

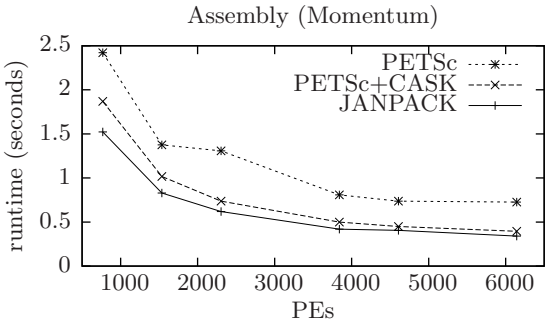


Figure 6: Average time to assemble the momentum matrix.

Table 1: Average time for computing a time-step (solving the fixed-point iteration).

PEs	PETSc	PETSc+CASK	JANPACK
768	11.567	7.3918	5.2520
1536	6.1522	3.9003	3.0700
2304	4.6828	2.8087	2.7063
3840	3.3004	2.2044	2.2413
4608	3.0729	1.9611	1.9688
6144	2.9051	1.7725	1.6615

Table 2: Average time to assemble the momentum matrix.

PEs	PETSc	PETSc+CASK	JANPACK
768	2.4217	1.8687	1.5224
1536	1.3759	1.0171	0.8300
2304	1.3072	0.7373	0.6193
3840	0.8083	0.4994	0.4196
4608	0.7372	0.4491	0.4052
6144	0.7267	0.3955	0.3407

performing twice as fast as the reference implementation, and also slightly faster than the CASK implementation for the largest number of PEs. These result are in good agreement with the observations in our previous benchmarks [14], where we found that for communication dominated problems, JANPACK inserts elements almost up to four times faster than PETSc (with CASK). For computational dominated problems we found that JANPACK inserts elements twice as fast compared to PETSc.

The good assembly performance for JANPACK is partly 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, often with an unstructured communication pattern. 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 (solving the linear systems). Therefore, the low latency of PGAS in combination with the low insertion cost of the stack based format enables us to achieve an overall better performance than PETSc in this part of the solver.

9.3 Linear solvers

For the linear solvers we measured the most interesting quantity, namely how long it took to converge to a solution. Since we used different preconditioners, this was the only reasonable thing to measure. In Figure 7 and Table 3 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 for larger number of cores. The performance drop between 2304 and 3840 cores coincide with the change in page size. Beyond 2304 cores it was necessary to use more then 2MB huge pages for the PGAS code. But since we wish to make a fair comparison we kept the faster 2MB page size for the PETSc implementations for all number of cores.

Furthermore, with our choice of preconditioner, it turned out that it was crucial to tune how many logical blocks we used to divide the vector into (see §5). With too few blocks, fetching off-PE dependencies (see Algorithm 3) 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 agrees with the observation made in [28]. In our case it was desirable to use 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 Algorithm 3). 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 Figure 8 and Table 4. Here the results for our hybrid solver is more encouraging. Overall performance is faster than PETSc (both versions) for all number of cores. The performance drop at 2304 cores are visible here as well but compared to the momentum solver the pressure solver recovers and performs better and better when increasing the number of cores. The most likely explanation is that the penalty of using larger huge pages is hidden behind the large number of Krylov iterations needed to converge the pressure equation, compared to the momentum equation.

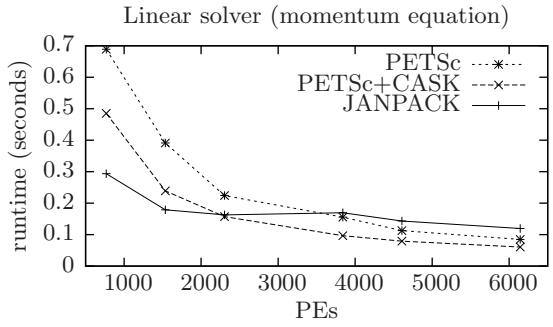


Figure 7: Average time to solve the momentum equation.

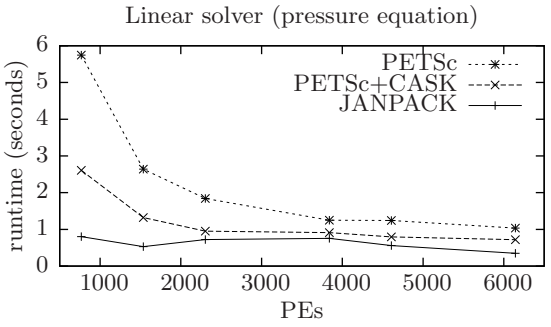


Figure 8: Average time to solve the pressure equation.

Table 3: Average time to solve the momentum equation.

PEs	PETSc	PETSc+CASK	JANPACK
768	0.6891	0.4853	0.2936
1536	0.3910	0.2390	0.1787
2304	0.2241	0.1571	0.1627
3840	0.1556	0.0965	0.1694
4608	0.1126	0.0791	0.1434
6144	0.0846	0.0604	0.1197

Table 4: Average time to solve the pressure equation.

PEs	PETSc	PETSc+CASK	JANPACK
768	5.7463	2.6077	0.8026
1536	2.6410	1.3211	0.5317
2304	1.8398	0.9532	0.7215
3840	1.2490	0.9115	0.7550
4608	1.2427	0.7972	0.5601
6144	1.0350	0.7183	0.3492

10. DISCUSSION

Comparing performance of different software is a delicate task, and this case is not an exception. Not only do we compare two different parallelization methods but also completely different sparse matrix representation and linear solvers. From the performance evaluation we have shown that the hybrid solver's overall performance is good, on par with Cray's heavily optimized version of PETSc, which is in itself a good achievement. Of course we can still improve our numerical kernels, especially preconditioners and parallel matrix vector multiplication. But we wish to empathizes that our choice of sparse matrix representation is not the cause of any performance bottleneck. Since one of the main goal of this paper was to prove the applicability of hybrid MPI/PGAS methods, this tuning is left as future work.

Overall we see that PGAS and in particular UPC can be faster than MPI, in contrary to the conclusions in [4][22]. However, it might be fair to mention that the evaluations in these references were performed on platforms without hardware accelerated one-sided communication.

Despite our good results, our hybrid approach has some drawbacks. 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 makes debugging and profiling a not so productive process. 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, and we are currently limited to Cray's compiler environment. Hopefully these are things that will improve in the future. Finally, despite the experimental toolchain we have in this paper shown that the common conception [8] of PGAS's poor performance is not always true, especially on machines with hardware accelerated PGAS support.

11. SUMMARY AND CONCLUSION

In this paper we have presented an alternative programming model combining traditional message passing with low latency one-sided communication within the same applications. Reducing the threshold for adopting new programming model in legacy applications. We demonstrate the applicability of this approach with a hybrid MPI/PGAS implementation of a finite element solver based on the FEniCS framework, where the linear algebra backend is replaced with one based on PGAS. The performance was evaluated on a Cray XE6, were the hybrid model demonstrate great potential with significant performance improvements in the solver, which performs on par or better compared to the MPI based solver using Cray's optimized version of PETSc.

The results presented in this paper has encouraged us to further develop and optimize our PGAS based linear algebra library, and we strongly believe that more complicated numerical algorithms can benefit from using the PGAS abstraction. One such example is our current work on algebraic multigrid solvers, where we see a clear benefit from using PGAS in the coarsening processes, which involves a lot of fine-grained parallelism and unstructured communication. Future work also includes an increased hybridization of FEniCS, moving more of the components from MPI to PGAS, e.g. mesh adaption.

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

12. ACKNOWLEDGMENTS

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.

13. REFERENCES

[1] 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.

[2] 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*, PGAS '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.

[3] 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*, 1:91–99, 2004.

[4] 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*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[5] R. V. de Abreu, N. Jansson, and J. Hoffman. Computation of aeroacoustic sources for a complex nose landing gear geometry using adaptivity. In *Proceedings of the Second Workshop on Benchmark problems for Airframe Noise Computations (BANC-II)*, Colorado Springs, 2012.

[6] E. F. V. de Velde. *Concurrent Scientific Computing*, volume 16 of *Texts in Applied Mathematics*. Springer-Verlag, New York, NY, USA, 1994.

[7] 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*, CF '10, pages 177–186, New York, NY, USA, 2010. ACM.

[8] J. J. Dongarra and A. J. van der Steen. High-performance computing systems: Status and outlook. *Acta Numerica*, 21:379–474, 4 2012.

[9] 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*, -(0):–, 2012. In press.

[10] J. Hoffman and C. Johnson. *Computational Turbulent Incompressible Flow*, volume 4 of *Applied Mathematics: Body and Soul*. Springer, 2007.

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

[12] 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.

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

[14] N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-sided Communication. In *High Performance Computing for Computational Science – VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012. In press.

[15] N. Jansson, J. Hoffman, and J. Jansson. Framework for Massively Parallel Adaptive Finite Element Computational Fluid Dynamics on Tetrahedral Meshes. *SIAM J. Sci. Comput.*, 34(1):C24–C41, 2012.

[16] N. Jansson, J. Hoffman, and M. Nazarov. Adaptive Simulation of Turbulent Flow Past a Full Car Model. In *State of the Practice Reports*, SC '11, pages 20:1–20:8, New York, NY, USA, 2011. ACM.

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

[18] A. A. Johnson. Using Unified Parallel C to Enable New Types of CFD Applications on the Cray X1E. In *CUG 2006*, 2006.

[19] 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*, PGAS '10, pages 5:1–5:10, New York, NY, USA, 2010. ACM.

[20] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[21] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[22] 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*, pages 174–184, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] A. Pletzer, D. McCune, S. Muszala, S. Vadlamani, and S. Kruger. Exposing Fortran Derived Types to C and Other Languages. *Comput. Sci. Eng.*, 10(4):86 –92, july-aug. 2008.

[24] C. Pommerell. *Solution of large unsymmetric systems of linear equations*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1992.

[25] 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*, SC '11, pages 78:1–78:11, New York, NY, USA, 2011. ACM.

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

[27] K. Schloegel, G. Karypis, and V. Kumar. ParMETIS, Parallel graph partitioning and sparse matrix ordering library, 2011.

[28] 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*, PMBS '11, pages 13–14. ACM, 2011.

[29] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.