Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication

Niclas Jansson[1]

Computational Technology Laboratory, CSC/NA
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
njansson@csc.kth.se

Abstract. In parallel finite element solvers, sparse matrix assembly is often a bottleneck. Implemented using message passing, latency from message matching starts to limit performance as the number of cores increases. We here address this issue by using our own stack based representation of the sparse matrix, and a hybrid parallel programming model combining traditional message passing with one-sided communication. This gives an insertion rate up to more than twice as fast compared to state of the art implementations on a Cray XE6.

1 Introduction

In large scale finite element simulations a considerable amount of time is spent in sparse tensor assembly. These tensors are often represented as sparse matrices which can be frequently reassembled for time dependent problems. Efficient tensor assembly is therefore a key to obtain good performance. Sparse matrix formats are designed to have a low memory footprint and for good performance when rows are accessed consecutively, for example in sparse matrix vector multiplication. Finite element assembly on the other hand often insert or add data at random locations in the tensor, resulting in a poor access pattern which can have a tremendous impact on the assembly performance.

Unstructured meshes are excellent for accurate approximation of complicated geometries. The unstructured communication pattern from the mesh dependencies, however, will have a negative effect on the assembly performance for large core counts, and can be the main cause for less scalable codes. Something that we have observed in related work [5].

The reason for this behaviour is partly due to the programming model used. Today, most scientific libraries and applications are parallelized using the *Message Passing Interface* (MPI) [7] or some hybrid incarnation combining MPI with OpenMP. MPI, is what is called two-sided,which means each process can only communicate with send and receive operations, and since these have to be matched to each other it will unavoidably increase latency and synchronization costs in non structured communication, as in sparse matrix assembly on unstructured meshes.

In this paper we address this issue by replacing the linear algebra parts of the finite element library DOLFIN [6] with a one-sided communication linear algebra backend, based on the *Partitioned Global Address Space* (PGAS) programming model, resulting in a hybrid MPI/PGAS application. Using this hybrid model we observe a reduction in assembly time by more than a half.

The outline of the paper is the following; in §2 we motivate our work and give a short background on related work. In §3 we present our sparse matrix format and in §4 we present our parallelization strategy. Performance results are discussed in §5, and we give conclusions and outline future work in §6.

2 Background

Most state of the art linear algebra packages, as for example PETSc [1], optimize the sparse matrix assembly by using non blocking communication, overlapping computation and communication. It can be seen as a simulation of one-sided communication. However, it requires the receiver to occasionally check for messages which introduces latency etc. Non blocking communication also involves software buffering, either on the sending or receiving side which adds to the latency penalty. True one-sided communication is realized in MPI 2.0 with *Remote Memory Access* (RMA) operations, but unfortunately the API imposes a number of restriction that prevents an efficient implementation. For a discussion see for example [2].

Today, at the dawn of exascale computing some concerns have been raised whether MPI is capable of deliver the needed performance. Since a tremendous amount of high quality scientific software has been written in MPI over the past decades, it is unreasonable to think that these are going to be rewritten or replaced with something written in for example PGAS. Therefore we argue that a reasonable way to prepare old legacy codes for exascale 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.

In our previous work on sparse matrix formats [3] we have found that the most common format *Compressed Row Storage* (CRS) [9] is sub optimal when it comes to sparse matrix assembly. CRS has an efficient access pattern for *Sparse Matrix Vector Multiplication* (SPMV) since data is stored in consecutive place in memory. On the fly insertion of elements can however be costly due to reallocation and data movement. Instead we propose a new way of using a stack-based representation. It is similar to the linked-list data structure where each row is represented by a linked list.

3 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

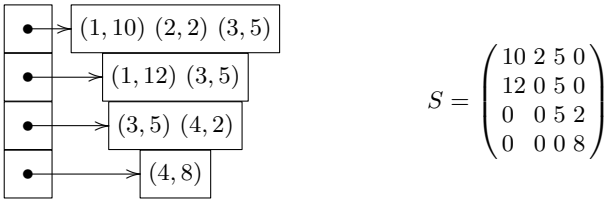

Fig. 1. An illustration of the stack-based representation of the matrix S.

column index c and element value v, hereby referred to as a row-stack, illustrated in Figure 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. 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 below:

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

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

We argue that this representation is more efficient than the CRS format, in particular for random insertion such as finite element assembly. Foremost since the indirect addressing to find the corresponding start of a row is removed. Secondly we do not require these stacks to be ordered. Thus we could push new elements regardless of the column index. In the case of adding a value to an already inserted element, the linear search will still be efficient since each row-stack has a short length equal to the number of non-zeros in the corresponding row.

4 Parallelization strategy

For the parallel implementation of the stack-based representation we used *Unified Parallel C* (UPC) [10], 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.

4.1 Directory of objects representation

In theory the stack-based (or CRS based) representation can easily be implemented in UPC by allocating the entire list of row-stacks in global space. How-

ever, this is not possible. For performance reasons memory in UPC is allocated in blocks with affinity to a certain thread. Hence we would either waste memory or force the matrix dimension to be even divisible by the number of threads used.

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 thread has affinity to one pointer. Each pointer then points to a row-stack, allocated in the global space with affinity to the same thread as the pointer. This technique enables us to have unevenly distributed global memory, and each piece can grow or shrink independently of each other.

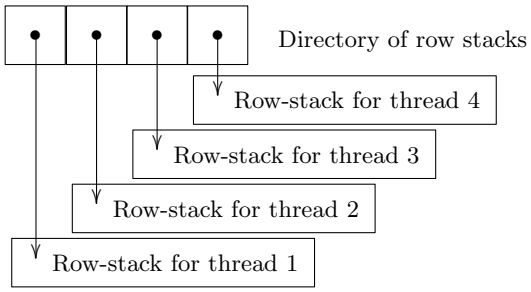

Fig. 2. An illustration of the representation of row stacks as a directory of objects.

4.2 Parallel matrix operations

Operating on the matrix in parallel is almost the same as for the serial implementation. The big difference is how matrix add/insert operations handle non thread-local elements. One solution is to allow the thread which update an entry to update it in the row-stack on the thread to which the entry has affinity to. The possibility of data races makes this approach error prone, which of course can be solved by adding locks around certain regions, and pay a certain latency fee for acquiring the locks.

Instead, we use a lock-free approach where Algorithm 1 is modified such that if an entry does not belong to a thread it is placed in a staging area (offthread row-stack) one for each thread, as illustrated in Algorithm 2 below.

After all entries have been added or inserted into the matrix, each thread copies the relevant staging areas from each other using the remote memory copy functionality in UPC, and add/insert them into the local row-stack. In order to minimize communication contention we pair threads together as illustrated in Algorithm 3 below.

4.3 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

Algorithm 2 *Lock free matrix update:*

```
if (owner(A(i).rs) != threadid)
  stage(owner(A(i).rs),c,v)
else
  for j = 1:length(A(i).rs)
    if (A(i).rs(j).c == c)
      A(i).rs(j).v += v
      return
    end
  end
  push (c, v) onto the row-stack A(i).rs
end
```

Algorithm 3 *Matrix finalization:*

```
for i = 1:nthreads
  src = mod((threadid + i ), nthreads)
  if length(offthread-stack(src, threadid)) > 0
    data = memget(offthread-stack(src,threadid))
    for j = 1:length(data)
      add data(j) to matrix
    end
  end
end
```

UPC we had to use an interface that did not expose the UPC specific data structures to DOLFIN's C++ code. To overcome this problem we access the UPC data types from DOLFIN as opaque objects [8]. 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 minor portability issues, except to determine the size of the data type for each new platform the library is compiled on. During runtime we use a flat model and map MPI ranks to UPC threads one-to-one. How this mapping is created and managed is left to the runtime system. In order to minimize possible runtime problems, we ensure that each programming model is separated such that no communication overlaps, e.g. inserting MPI barriers before and after sections with UPC calls.

5 Performance results

We have conducted a performance study in order to determine if sparse matrix assembly can gain anything from one-sided communication and if the hybrid MPI/PGAS model is feasible for optimizing legacy MPI codes. For our experiments we choose two partial differential equations (PDE) both resulting in dif-

ferent kind of communication and computational cost. We measured the time to recompute the stiffness matrix, thus assuming that the sparsity pattern is known a priori. Insertion rate r (per core) were also measured, computed as

$$r = \frac{N}{t \cdot c}$$

where, N is the number of measured matrix updates, t the time it took to assembled the matrix and c the numbers of cores.

Benchmark A: Laplace equation in 3D In the first benchmark we compute the stiffness matrix corresponding to the continuous linear Lagrange FEM discretization of Laplace equation:

$$-\Delta u = 0$$

This benchmark corresponds to a worst case scenario, since the stiffness matrix can be computed with minimal work. Hence, this benchmark tests the communication cost more than the insertion rate. For this experiment we use an unstructured tetrahedral mesh of the unit cube consisting of 317M elements.

Benchmark B: Convection-diffusion in 2D The second benchmark computes the matrix for the convection-diffusion equation:

$$\dot{u} + \nabla \cdot (\beta u) + \alpha u - \nabla \cdot (\epsilon \nabla u) = f$$

on a 214M element continuous linear Lagrange FEM discretization of the unit square. In this benchmark, the assembly cost for each element is significantly higher. Since we are using vector elements more data is also inserted per element. Hence, this benchmark tests both the communication and the insertion rate for a more realistic problem.

All experiments have been tested on the 1516 node Cray XE6 *Lindgren* at PDC/KTH. The PDEs were solved using the finite element software DOLFIN 0.8.2-hpc, compiled with the Cray C++ compiler version 7.4.0. DOLFIN supports several different linear algebra backends. For this work we choose the Cray provided PETSc 3.1.0 as the base line for our experiments. Our UPC based sparse matrix library called JANPACK [4] was also compiled with the 7.4.0 version of the Cray C compiler and then linked together with DOLFIN forming a true hybrid MPI/PGAS application. In Figure 3 and Table 1 we present the insertion time. In Table 2 insertion rates (normalized to one core) are presented.

In Figure 3 we can see that our claims are true. One-sided communication indeed has a lower overhead than traditional message passing. This is mostly visible for benchmark A (left part) where PETSc's performance flattens out when the number of cores gets large, while the less latency affected UPC code continues to perform well. We also observe that when the computation and

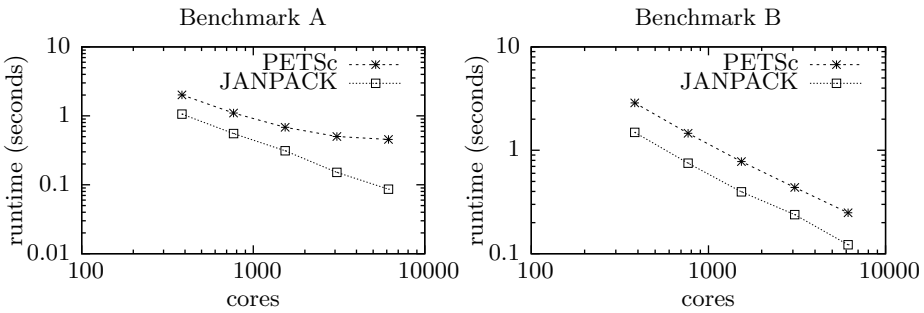

Fig. 3. Assembly time for benchmark A (left) and benchmark B (right).

Table 1. Assembly time (in seconds) for both benchmarks.

	Benchmark A		Benchmark B	
cores	PETSc	JANPACK	PETSc	JANPACK
384	2.010	1.060	2.870	1.490
768	1.100	0.553	1.460	0.750
1536	0.681	0.311	0.778	0.396
3072	0.501	0.151	0.437	0.239
6144	0.455	0.086	0.249	0.122

Table 2. Insertion rate (Mega entries/second) for both benchmarks.

	Benchmark A		Benchmark B	
cores	PETSc	JANPACK	PETSc	JANPACK
384	6.60	12.5	6.99	13.5
768	6.01	11.9	6.87	13.4
1536	4.90	10.6	6.44	12.7
3072	3.30	10.9	5.74	10.5
6144	1.82	9.60	5.03	10.3

communication costs are well balanced (as in benchmark B and for low core counts in benchmark A) JANPACK insert elements twice as fast as PETSc. Part of this is due to the less costly communication in UPC, but also due to our own sparsity format which has proven to outperform PETSc in previous single core studies [3].

6 Summary and future work

In this work we have investigated the feasibility of optimizing finite element solvers with one-sided communication. Our results show that one can gain a

factor of two in speedup by switching from traditional message passing sparse matrix assembly to a one-sided communication using the PGAS programming model. The approach of developing a hybrid MPI/PGAS application also demonstrates that old legacy code can be optimized further with PGAS without rewriting the entire application.

For many applications good SPMV performance is equally or even more important than matrix assembly. Since this is the first step towards solving nonlinear transient problems, where assembly performance matters since matrices are reassembled for each time step, tuning the stack based representation for good SPMV performance are left as future work. Also, we acknowledge that replacing PETSc in our solvers is a serious undertaking, future work also includes implementation of Krylov solvers and efficient preconditioners in UPC such that we can use our sparse matrix library for solving large industrial flow problems.

7 Acknowledgments

The author would like to acknowledge the financial support from the Swedish Foundation for Strategic Research and is also grateful for the large amount of computer time given on *Lindgren* during its test phase. The research was performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC - Center for High-Performance Computing.

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. 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.
3. 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/.
4. N. Jansson. JANPACK, 2012. http://www.csc.kth.se/~njansson/janpack.
5. 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, 2011.
6. A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.
7. MPI Forum. Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/.
8. 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.
9. Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.
10. UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.