Bitwise Reproducible task execution on unstructured mesh applications

Bálint Siklósi
3in Research Group,
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Esztergom, Hungary
Email: siklosi.balint@itk.ppke.hu

István Z Reguly
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Budapest, Hungary
Email: reguly.istvan@itk.ppke.hu

Gihan R Mudalige
Department of Computer Science
University of Warwick
Coventry, United Kingdom
Email: g.mudalige@warwick.ac.uk

***Abstract*—Many mesh applications use floating point arithmetic which do not necessarily hold the associative laws of algebra. This could cause the application to become unreproducible. In this paper we present some work on generating a method for unstructured mesh applications to provide bitwise reproducibility between separate runs, even if they are started with different number of MPI processes. We implement our work in the OP2 domain-specific library, which provides an API that abstracts the solution of unstructured mesh computations. We carry out a performance analysis of our method applied on two applications: a simple airfoil application, and a more complex Aero application which uses a finite element method and a conjugate-gradient algorithm. We show a $2.37\times$ to $1.49\times$ slowdown on this applications as a price for full bitwise reproducibility.**

I. Introduction

In the field of high performance computing, there is a need for tools or methods to achieve reproducibility [1]. We might need reproducibility for many cases. When we have an error during a program execution, we might want to reproduce the problem exactly. We might have some contractual obligations, for example in applications from human sciences or financial fields. There might be some really rare random events that we want to study, then we would use a system where we can reproduce the results to the last bit. Comparing the output of a trustworthy sequential run to the output of a parallel run is very common method to determine the correctness of the parallel method. Without reproducibility we might have trouble to determine whether we have some error from the representation or we have a bug in the parallel code.

Many scientific simulations use floating point arithmetic. The floating point arithmetic is non-associative since we need to use roundings, because of the representation limits. The basic standard for floating point representation is the IEEE-754, most of the applications use this format. The standard provides the correct behaviour for all operations, and the necessary roundings too. Since the IEEE floating point standard defines a finite representation size, we need to round the results between the steps of the operations, for example between multiple summations.

It is straightforward that reproducibility comes with a price. This price depends on the application area of which is addressed. The ReproBLAS project [2] uses a different representation, they introduce a binned method. With their method a $5n$ to $9n$ floating point operations overhead is produced when summing n floating point numbers. And also to use this, we would need to rewrite our algorithm, and we would also introduce a dependency on a new library. Lulesh [3] present some work on achieving reproducibility, but only between runs of the same number of MPI processes.

In this paper, we present work to address this problem in the field of unstructured mesh computations, commonly used for the discretized solution of partial differential equations. In section II we introduce the OP2 domain specific language where we implemented our work. In section III we present our approach about how we achieved reproducibility between multiple runs or using different number of MPI processes. In section IV we present the effectiveness on two mini-applications, we summarise our progress and we introduce some further tasks with this topic. All of our work is open source, and can be found at [4].

II. The OP2 domain specific language

The OP2 DSL is the second version of the Oxford Parallel Library for Unstructured mesh Solvers (OPlus), aimed at expressing and automatically parallelising unstructured mesh computations. While the first version was a classical software library, facilitating MPI parallelisation, OP2 can be called an "active" library, or an embedded DSL.

OP2's key components consists of (1) a mesh made up of a number of sets (such as edges and vertices), (2) connections between sets (e.g an edge is connected to two vertices), (3) data defined on sets (such as coordinates on vertices, pressure/velocity on a cell centre) and (4) Computations over a given set in the mesh.

A user initially sets up a mesh and hands all the data and metadata to the library using the OP2 API. The API appears as a classical software API embedded in C/C++ or Fortran. Any access to the data handed to OP2 can only be accessed subsequently via these API calls. Essentially, OP2

```
/* ----- elemental kernel function in res.h ------*/
void res(const double *edge,
         double *cell0, double *cell1 ){
  //Computations, such as:
  cell0 += *edge; *cell1 += *edge;
}
/* ---------- in the main program file -----------*/
// Declaring the mesh with OP2 sets
op_set edges = op_decl_set(numedge, "edges");
op_set cells = op_decl_set(numcell, "cells");
// mppings -connectivity between sets
op_map edge2cell = op_decl_map(edges, cells,
                   2, etoc_mapdata,"edge2cell");
// data on sets
op_dat p_edge = op_decl_dat(edges,
                1,"double",edata,"p_edge");
op_dat p_cell = op_decl_dat(cells,
                4,"double",cdata,"p_cell");
// OP2 parallel loop declaration
op_par_loop(res,"res", edges,
  op_arg_dat(p_edge,-1,OP_ID    ,4,"double",OP_READ),
  op_arg_dat(p_cell, 0,edge2cell,4,"double",OP_INC ),
  op_arg_dat(p_cell, 1,edge2cell,4,"double",OP_INC));
```

Figure 1. Specification of an OP2 parallel loop

makes a private copy of the data internally and restructures its layout in any way that it sees fit to obtain the best performance for the target platform. Once the mesh is set up, computations are expressed as a parallel loop over a given set, applying a "computational kernel" at each set element that uses data that is accessed either directly on the iteration set, or via at most one level of indirection. The type of access is also described - read, write, read-write or associative increment. Additionally, users may pass mesh-invariant data to the elemental computational kernel and also carry out reductions.

The various parallelisation and performance of production applications using OP2 has been published previously in [5], [6] demonstrating near-optimal performance on a wide range of architectures including multi-core CPUs, GPUs, clusters of CPUs and GPUs. The generated parallelisation makes use of an even larger range of programming models such as OpenMP, OpenMP4.0, CUDA, OpenACC, their combinations with MPI and even simultaneous heterogeneous execution.

Figure 1 shows the declaration of the mesh and subsequent definition of an OP2 parallel loop. In this example, the loop is over the set of `edges` in a mesh carrying out the computation per edge defined in the function `res`, accessing the data on edges `p_edge` directly and updating the data held on the two cells, `p_cell`, adjacent to an edge, indirectly via the mapping `edge2cell`. The `op_arg_dat` provides all the details of how an `op_dat`'s data is accessed in the loop. Its first argument is the `op_dat`, followed by its indirection index, `op_map` used to access the data indirectly, arity of the data in the `op_dat` and the type of the data. The final argument is the access mode of the data, read only, increment and others (such as read/write and write only not shown here).

In a direct loop all iterations are independent from each other and as such the parallelization of such a loop does not have to worry about data races. However in indirect loops there is at least one `op_dat` that is accessed using an indirection, i.e via an `op_map`. Such indirections occur when the `op_dat` is not declared on the set over which the loop is iterating over. In which case an `op_map` that provides the connectivity information between the iteration set and the set on which the `op_dat` is declared over is used to access (read or write depending on the access mode) the data. This essentially leads to an indirect access.

We test and measure our methods on two OP2 applications (1) Airfoil, a standard finite volume CFD benchmark code and (2) Aero, a finite element 2D nonlinear steady potential flow simulation.

Airfoil [7] is an industrially representative CFD code, written using OP2's C/C++ API. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid. It is a finite volume application that solves the 2D Euler equations using a scalar numerical dissipation.

Aero [8] is a 2D non-linear steady potential flow simulation of air moving around an airfoil developed based on standard finite element methods. It uses a quadrilateral grid similar to that used by the Airfoil application but uses a Newton iteration to solve the non-linear equations defined by a finite element approximation. Each Newton iteration requires the solution of a linear system of equations. The assembly algorithm is based on quadrilateral elements and uses transformations from the reference square to calculate the derivatives of the first-order basis functions. Dirichlet type boundary conditions are applied on the far-field, and the symmetric sparse linear system is solved with the standard conjugate-gradient (CG) algorithm.

III. Reproducible indirect increments

To achieve reproducibility, we had to introduce a fixed execution order, which is used every time with any settings. On figure 2. two example incrementing orders can be seen on a single cell, by executing through the edges by using an `edge_to_cells` mapping. Since the associative laws of algebra do not necessarily hold for floating-point numbers, $cell0 = e0 + e1 + e2 + e3 \neq cell0 = e1 + e3 + e0 + e2$. In OP2 we can have the different orders, if we run the application two times and the edges are listed with different IDs. This can happen for example over MPI, when the library partition the whole mesh into parts, and then it renumbers all members for other performance reasons.

Our solution for these issues needs two parts: (1) OP2 needs to prepare the backend to use a specific order, (2) the code automatically generated for the application needs to use this order. A brief pseudocode of the first part can be seen on Figure 4 and the second part can be seen on Figure 5.

To achieve reproduciblity, practically we swap the structure of execution for those arrays, where the order of

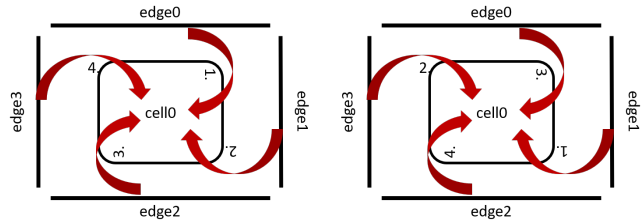


Figure 2. Example orders of incrementing a cell in airfoil.

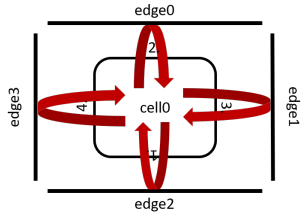


Figure 3. Swapped structure of execution

summation does matter. We still calculate for the rest of the arrays on the original way, but for the sensitive ones we just temporarily store the increments. Then from iterating through edges and increment cells, we iterate through cells and ask for increments from each edges with a fixed global order and apply those on the cell. This swap can be seen on figure 3.

To achieve this swap, a few preparations must be done in the backend. If there are more MPI processes, then the global IDs of each elements must be communicated between the processes. If an element is owned by the given process, then its global ID can be looked up from OP2's `g_index` array. If an element is not owned, then its global ID must be imported from that MPI process, which owns it. This global ID list will be the common seed in every process, which will produce the fixed execution order. After the IDs are shared, the next step is to create a reversed mapping for every map. The reversed mapping is needed, so we can iterate through on the cells and in each iteration we can access the edges connected to the given cell. This reversed map uses local indices which might be in different order in different MPI ranks. That is why we need to reorder them by using their previously shared global indices. Another modification done on the reversed map, that it actually stores indices of a temporary array where the increments from the edges are stored for a cell. With other words: if the k^{th} element in line n (k^{th} edge connection of cell n) of the reversed map is x, then it means that in the temporary increments array at location x can be found the increment for cell n from edge k.

After the reversed map with the correct order is created, then the generated `op_par_loop` code must be modified to use this twisted method. The main change ideas can be seen on figure 5. After some initialisation part, all elements of a temporary array must be set to zero, for the storage of the separate increments. Then we can call the kernel function for all edges to access elements defined on the cells. If a

exchange global IDs
for $m = 0$ to OP_map_index **do**
  create reversed mapping for map m
  for $i = 0$ to set_to_size **do**
    sort the reversed connections of i by global IDs
  end for
end for

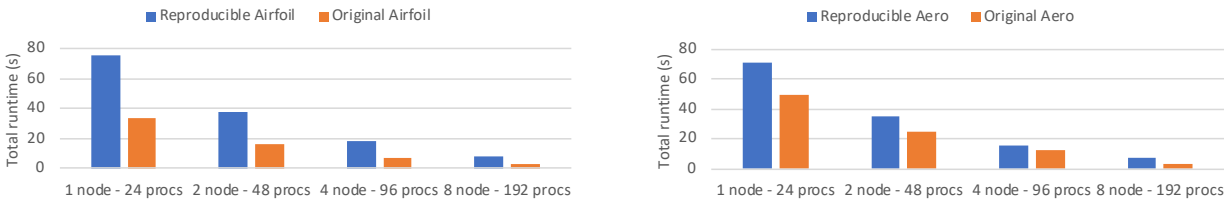
Figure 4. Algorithm of generating incrementing order

for $n = 0$ to set_from_size **do**
  $tmp\_incs[n] \leftarrow 0$
end for
for $n = 0$ to set_from_size **do**
  prepare regular access indices for OP_READ and OP_WRITE parameters
  call kernel function, using the tmp_incs array for OP_INC parameters
end for
for $n = 0$ to set_to_size **do**
  for all connection i of n **do**
    execute the temporary increments on the final location of the data
  end for
end for

Figure 5. Algorithm for applying the order of increments

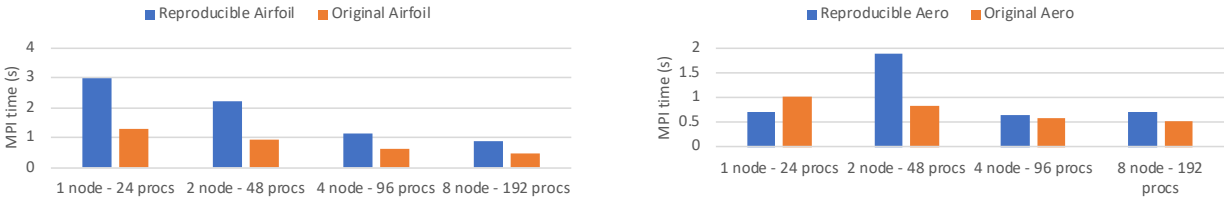
parameter is accessed through an `OP_READ` or `OP_WRITE` method, then the execution order does not matter, so we can use the original method of directly storing the new state on the data. If the parameter is incremented (`OP_INC`), then we need to store each increment values in the `tmp_incs` array. After all increments are calculated, we need to apply those on the actual data. For that, we start a new loop on the cells and by using the reversed mapping with the fixed ordering the cell can collect and apply the increments on itself. This method works with other type of mappings, not just with `edges_to_cells`.

Another difficulty of combining reproducibility with MPI is reducing into a single variable. If we use different number of MPI processes, then we could sum different number of elements which again can produce different results. To solve this issue, we introduced another temporary storage. If a kernel's input parameter is one for reducing, then we give a temporary storage point to store the increment for the result. Then in each MPI process, we reduce these increments reproducibly by using the ReproBLAS library. First we create a local ReproBLAS's `double_binned` variable for every MPI process, then we use `binnedBLAS_dbdsum` to collect those into the `local_sum`. After that, we use reproBLAS's method to call an `MPI_Allreduce` with `binnedMPI_DBDBADD` function. After that we convert back the final result to normal double and we store it at its original location.



(a) (b)

Figure 6. Measured full runtimes of the (a) Airfoil and the (b) Aero applications



(a) (b)

Figure 7. Measured MPI runtimes of the (a) Airfoil and the (b) Aero applications

IV. Results

We have tested our result on ARCHER, the UK's national HPC facility which is a Cray XC30 supercomputer. ARCHER compute nodes contain two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors. We used strong scaling on 1, 2, 4 and 8 nodes to check the performance on different levels. Each time the size of the Airfoil and Aero meshes are 2880000 nodes and 5757200 edges.

On figure 6 we can see the performance difference of the reproducible method compared to the original version on the Airfoil and Aero applications. On the Airfoil, there is an average of $2.37\times$ slowdown, and a $1.49\times$ on the Aero. As it was expected, reproducibility comes with a significant slowdown effect, although if the application is computationally more intensive then the runtime difference decrease. On the side of the bandwidth, there is only a $1.4\times$ decrease measured on the Airfoil application. On figure 7 we can see the effect of the modified MPI gather method. On the Airfoil, there is an average of $2.14\times$ slowdown, and a $1.37\times$ on the Aero. Future works consit of extending this method to OpenMP and Cuda. Also during this progress, a new type of problem came up. If a kernel is not just incrementing a variable, but reads and rewrites it, then the kernel call from one edge must be executed, not just temporarily stored. This problem needs a solution to be able to really execute the kernel calls in a predefined fixed order.

Acknowledgments

Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

[1] "Sc'15 bof on "reproducibility of high performance codes and simulations: tools, techniques, debugging"," 2015.

[2] J. Demmel, P. Ahrens, and H. D. Nguyen, "Efficient reproducible floating point summation and blas," Tech. Rep. UCB/EECS-2016-121, EECS Department, University of California, Berkeley, Jun 2016.

[3] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[4] "OP-DSL: The Oxford Parallel Domain Specific Languages," 2015. https://op-dsl.github.io.

[5] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. Kelly, and D. Radford, "Acceleration of a full-scale industrial cfd application with op2," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016.

[6] I. Z. Reguly, D. Gopinathan, J. H. Beck, M. B. Giles, S. Guillas, and F. Dias, "The volna-op2 tsunami code (version 1.0)," *Geoscientific Model Development Discussions*, 2018.

[7] M. Giles, D. Ghate, and M. Duta, "Using automatic differentiation for adjoint cfd code development," *Computational Fluid Dynamics Journal*, vol. 16, 01 2008.

[8] O. Zienkiewicz, R. Taylor, and J. Zhu, *The Finite Element Method: its Basis and Fundamentals (Seventh Edition)*. Oxford: Butterworth-Heinemann, seventh edition ed., 2013.