ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

HYBRID PARALLELISATION OF AN ALGORITHMICALLY DIFFERENTIATED ADJOINT SOLVER

Pavanakumar Mohanamuraly[1], **Jan C. Hückelheim**[1] **and Jens D. Mueller**[1]

[1]Queen Mary University of London
Mile End Road, E1 4NS, London, United Kingdom
{p.mohanamuraly | j.c.hueckelheim | j.mueller}@qmul.ac.uk

Keywords: Parallel computing, MPI, OpenMP, Algorithmic differentiation, Source transformation, Adjoint CFD

Abstract. *We present a novel approach to parallelise an unstructured node-based finite volume solver using a hybrid MPI and OpenMP paradigm. The basic ingredients of this approach are, (i) zero-halo partitioning of the unstructured mesh and (ii) shared node residual accumulation. These two ingredients preclude the need to explicitly exchange the state information across partitions, allowing the computations to run independently in each partition. As a consequence, we retain the original loop structure for the numerical flux kernels, which can be parallelised using OpenMP directives. Due to the hand-assembly of reverse-differentiated routines in our adjoint solver, the adjoint MPI recipes presented in earlier work can not be applied without modifications. We present a modified adjoint MPI treatment for two MPI operations in the context of our hand-assembled nonlinear iterative solver, namely: (i) shared node accumulation and (ii) in-place all-reduce summation. We demonstrate the parallelisation approach on our in-house solver* mgopt*, which uses the Tapenade AD tool to generate the adjoint code. We show preliminary scalability results for a simple* $2d$ *problem.*

1 PDE constraint optimisation

The present work aims at parallelising the primal and adjoint solvers used in the context of PDE based optimisation, namely aerodynamic shape optimisation problems. The problem can be formulated as follows,

$$\min_{\alpha} J(u, \alpha) \tag{1}$$

$$s.t. \quad R(u, \alpha) = 0 \tag{2}$$

Where, J is referred to as the objective or cost function, which is minimised subject to PDE constraint R. The variable u is called the state and α is the control or design variable. Gradient based optimisation methods require one to calculate the quantity $\frac{dJ}{d\alpha}$. When the number of design variables (dimension of α) is higher than number of output variables (dimension of J), it is computationally efficient to obtain adjoint sensitivity $\frac{dJ}{d\alpha}^T$ given by equation 3. The cost of calculating the adjoint sensitivity is proportional to the dimension of the output J and independent of the dimension of the input α. Algorithmic differentiation (AD) in the reverse or adjoint mode is used to obtain the transposed values. Tapenade[11] AD tool is used in this work to obtain the terms of the adjoint equation 4, which solves for the so-called adjoint variable v.

$$\frac{dJ}{d\alpha}^T = \frac{\partial J}{\partial \alpha}^T + \frac{\partial R}{d\alpha}^T v \tag{3}$$

$$\left(\frac{\partial R}{\partial U}\right)^T v = \left(\frac{\partial J}{\partial U}\right)^T \tag{4}$$

In the present context, the PDE constraint is the compressible steady Navier-Stokes equation, α is the shape design parameter and u is the fluid state. Node-based finite volume discretisation is used in the present work to solve both the primal and the adjoint system. Details of the numerical method and implementation are available in reference [4]. The distributed parallelisation of the node-based finite volume discretisation using Message Parsing Interface (MPI) is presented in the next section.

2 Zero-halo partitioning and MPI adjoints

A popular choice for distributed parallelisation in scientific codes is the use of halo or ghost layers. They are an extra layer(s) of cells placed adjacent to the mesh partition boundary to mimic cells from neighbouring mesh partition. The adjoint of the MPI calls for this approach can be obtained using the framework described in reference [1]. An overview of the halo-layer approach is depicted in figure 1. The main drawback of this approach is that the MPI operations are not self-adjoint, i.e., the primal and adjoint have different MPI call sequence. In a source transformed AD, this forces one to replace MPI calls within the code with its adjoint equivalent. Currently, there are no tools available to automate this task and one has to manually implement the forward and reverse MPI calls. The AMPI[1] library reduces this burden by providing MPI wrappers for both forward and reverse mode. The wrappers typically tape appropriate MPI function arguments in the case of operator overloading or push/pop to stack in the case of source transformation. Lastly, the halo approach gives rise to artificial increase in number of computations per partition by duplicating operations at the halo boundary. This in-turn reduces the strong scaling efficiency. In most practical scientific codes, graph based mesh partitioning tools are used, which try to minimise the graph edge cuts (communication

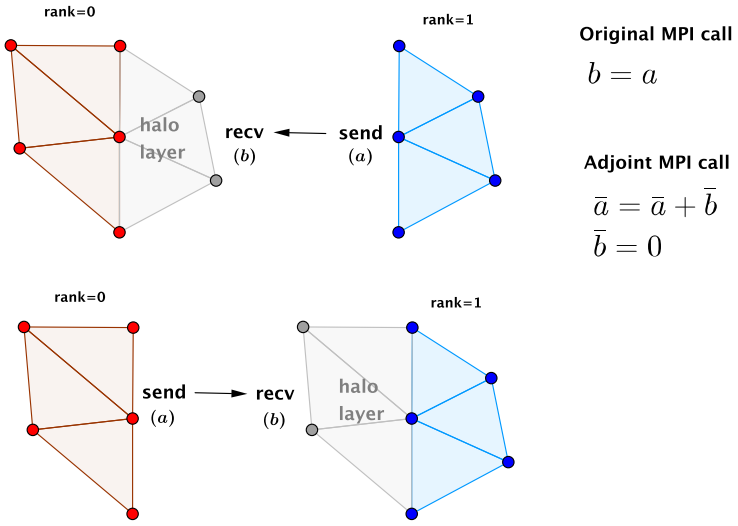


Figure 1: Parallel implementation based on halo layers and their MPI forward-adjoint equivalents

cost) maintaining good load balance. In reference [6], the authors show that for large number of partitions the load balancing severely degrades for graph based methods. The problem is exacerbated in the adjoint [7], which amplifies the load imbalance in the primal causing further reduction in scaling.

In a zero-halo layer approach, the partition local fluxes are computed and accumulated at the partition shared nodes/cells as shown in figure 2. At the partition boundaries, flux calculated at one partition face exactly cancels with its neighbouring partition face pair. As a result, one imposes a no-flux or empty boundary condition at the partition boundary faces. In addition, one should ensure that the flux residuals are accumulated at the shared nodes across partition boundaries to account for the flux residual from the adjacent partition. As a result, flux accumulation at shared-nodes preclude the need for an explicit exchange of the state information. Unlike halo-layer implementation, here it is easier to avoid the duplication of operation and improve scalability. The most attractive feature of this approach is the self-adjoint nature of the MPI call sequence: figure 2(a). This simplifies the implementation in a fixed-point iteration (FPI) type primal/adjoint solver as shown in listing (1) and (2).

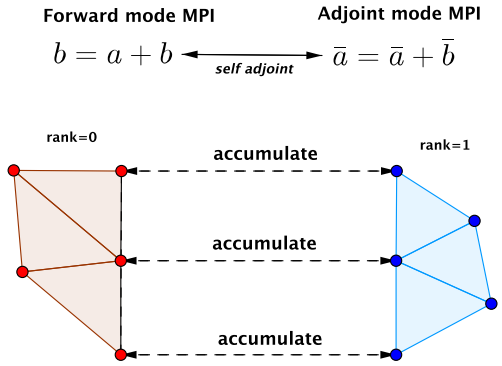
Listing 1: Primal FPI

```
...
do iter = 1, n
  call residue(u, R)
  call accumulate(R)
  call update(u, R)
end do
call cost_fun(u, J)
```

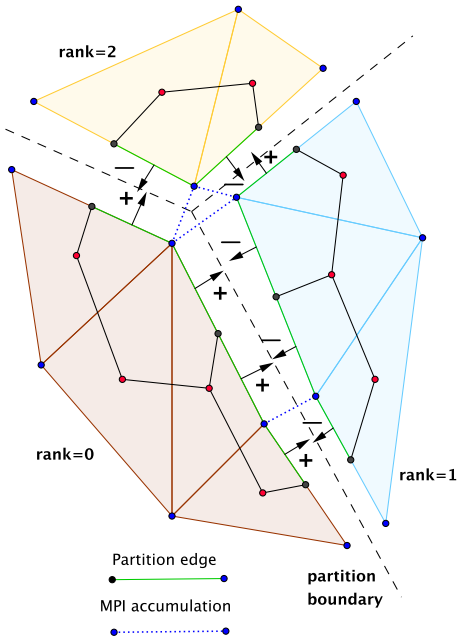
Listing 2: Hand assembled adjoint FPI [4]

```
J̄ = 1
call cost_fun_b(u, (∂J/∂u)^T J̄, J, J̄)
do iter = 1, n
  call residue_b(u, (∂R/∂u)^T v, R, v)

  call accumulate((∂R/∂u)^T v)

  R = (∂R/∂u)^T v - (∂J/∂u)^T
  call update(v, R)
end do
```

Although the adjoint of the MPI operation for shared-node accumulation is symmetric, other



(a) Zero-halo accumulation and the forward-adjoint mode MPI equivalents



(b) Exploded view of a three partition zero-halo flux exchange

Figure 2: Zero-halo implementation, flux accumulation, and self-adjoint MPI operation

MPI operations warrant careful examination. Consider evaluation of a scalar cost function (J), for example, the aerodynamic lift or drag. The pseudo code to calculate the cost function is shown in listing (3). Here the cost function subroutine receives as an input a vector quantity u and returns the scalar J.

Listing 3: Cost function (drag) evaluation (parallel)

```
! Input  : u (vector)
! Output : J (scalar)
subroutine cost_fun(u, J)
  call drag_force( u, J )
  call Allreduce( IN_PLACE, J, ..., SUM )
end subroutine cost_fun
```

The adjoint in the serial case (without Allreduce) is simply obtained by passing the $drag\_force$ subroutine to a source transformation engine like Tapenade. On the contrary, it is not straightforward to construct the cost function adjoint in parallel (with IN_PLACE Allreduce). One would be tempted to use the MPI adjoint recipe from reference [2] for the Allreduce operation as shown in listing (4). For the hand assembled adjoint shown in listing (2), verbatim use of the recipe results in an incorrect source term.

Listing 4: Allreduce recipe from [2]

```
! Original call
Allreduce( x, y, ,,, SUM )
! Adjoint call
Allreduce( ȳ, t̄, ,,, SUM )
ȳ = 0
x̄ = x̄ + t̄
```

The hand assembled adjoint source term becomes unambiguous when viewed in the context of solving the (adjoint) linear system shown in equation 4. The subroutine $cost\_fun\_b$ calculates the right hand side forcing term of equation 4, namely $\left(\frac{\partial J}{\partial U}\right)^T$. In order to do that one has to set the values of $\bar{J} = 1$ and $\bar{u} = 0$. The serial version of the subroutines can now be recast into the mathematical equivalent as shown in listing (5) and (6).

Listing 5: Cost function primal (serial)

```
cost_fun(u, J):

  J = J(u)
```

Listing 6: Cost function adjoint (serial)

```
cost_fun_b(u, ū, J, J̄):

  ū = ū + (∂J/∂U)^T J̄
```

In parallel execution, each rank i calculates J_i local to the partition and an Allreduce sum is performed to obtain the global cost function value $J = \sum_i J_i$. Following a similar argument it can be shown (with reference to figure 3) that the cost-function adjoint calculates a local value of $\left(\frac{\partial J}{\partial U}\right)^T \bar{J}$ in each rank. The only inconsistency appears at the shared nodes, which require accumulation of the source term from neighbouring partition: listing (8). *Note that it is possible to completely avoid the shared-node accumulation inside cost-function by deferring and combining the accumulate with the adjoint residual as shown in listing (10).* Listing (9) is shown for comparison with accumulation in cost function. This idea can be extended to find the MPI adjoints of other types of cost functions like total pressure loss, entropy loss, etc.

Listing 7: Cost function primal (parallel)

```
cost_fun(u, J):

  J_i = J(u)
  J = Σ_i J_i
```

Listing 8: Cost function adjoint (parallel)

```
cost_fun_b(u, ū, J, J̄):

  ū = ū + (∂J/∂U)^T J̄
  call accumulate(ū)
```

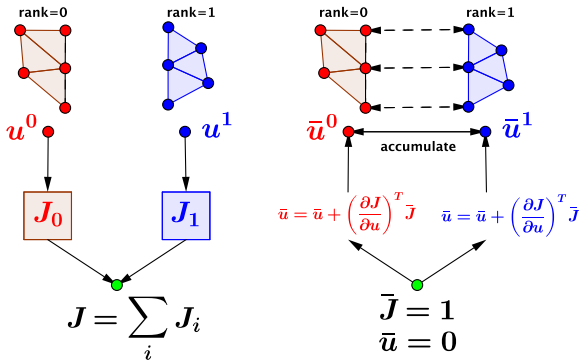


Figure 3: Schematic of cost function and its hand-assembled adjoint implementation

Listing 9: Adjoint FPI (accumulate)

```
J̄ = 1
call cost_fun_b(u, (∂J/∂u)^T J̄, J, J̄)

call accumulate((∂J/∂u)^T)
do iter = 1, n
  call residue_b(u, (∂R/∂u)^T v, R, v)

  call accumulate((∂R/∂u)^T v)

  R = (∂R/∂u)^T v - (∂J/∂u)^T
  call update(v, R)
end do
```

Listing 10: Adjoint FPI (no accumulate)

```
J̄ = 1
call cost_fun_b(u, (∂J/∂u)^T J̄, J, J̄)


do iter = 1, n
  call residue_b(u, (∂R/∂u)^T v, R, v)

  R = (∂R/∂u)^T v - (∂J/∂u)^T

  call accumulate(R)
  call update(v, R)
end do
```

3 Shared memory OpenMP parallelisation

The primal solver is using a shared-memory parallelisation strategy based on edge colouring as presented e.g. in [8, 9]. In this strategy, the edges are coloured in such a way that edges with the same colour do not share any nodes. More formally, for two edges $E_\alpha = (V_i, V_j)$ and $E_\beta = (V_k, V_l)$ we demand that

$$\text{colour}(E_\alpha) = \text{colour}(E_\beta) \Leftrightarrow i, j, k, l \text{ pairwise distinct} \tag{5}$$

We can then perform all flux updates on edges of the same colour simultaneously. This is illustrated in Figure 6. The adjoint CFD solver performs adjoint flux updates in the same fashion, i.e. as a loop over edges. The memory access pattern that describes read and write access of the primal and adjoint CFD solver are shown in Figure 4 and Figure 5.

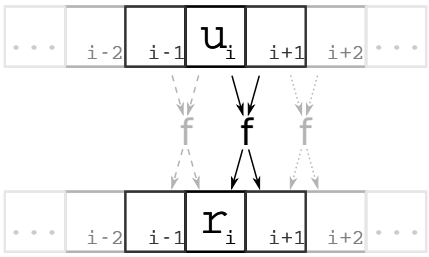


Figure 4: Primal communication pattern

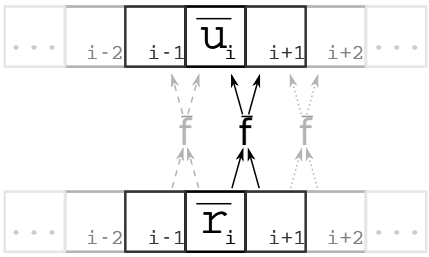


Figure 5: Adjoint communication pattern

Since the memory access of the adjoint and primal solver are identical, the mesh colouring that is used during the primal computation can be reused for the adjoint solver and helps to avoid write conflicts during the adjoint flux update in the same way as in the primal, see Figure 7 for an illustration.

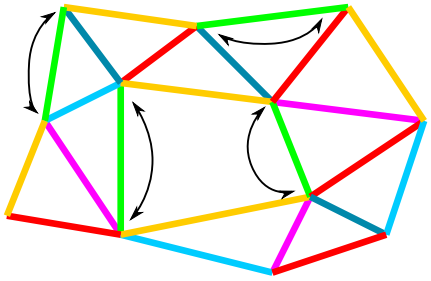


Figure 6: Edge colouring for primal shared-memory parallelisation

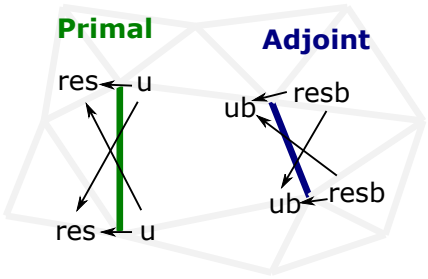


Figure 7: Equivalence of primal and adjoint communication

The AD tool Tapenade does not support OpenMP pragmas at the time of writing. We therefore employ *subroutine outlining* [10], a technique that encapsulates all private variables inside a subroutine and uses all shared variables as subroutine arguments. When this technique is used, the AD tool generates adjoint code in which only an OpenMP parallel loop with the `DEFAULT SHARED` clause has to be inserted in a postprocessing step.

Listing 11: Forward and reverse parallel flux computation

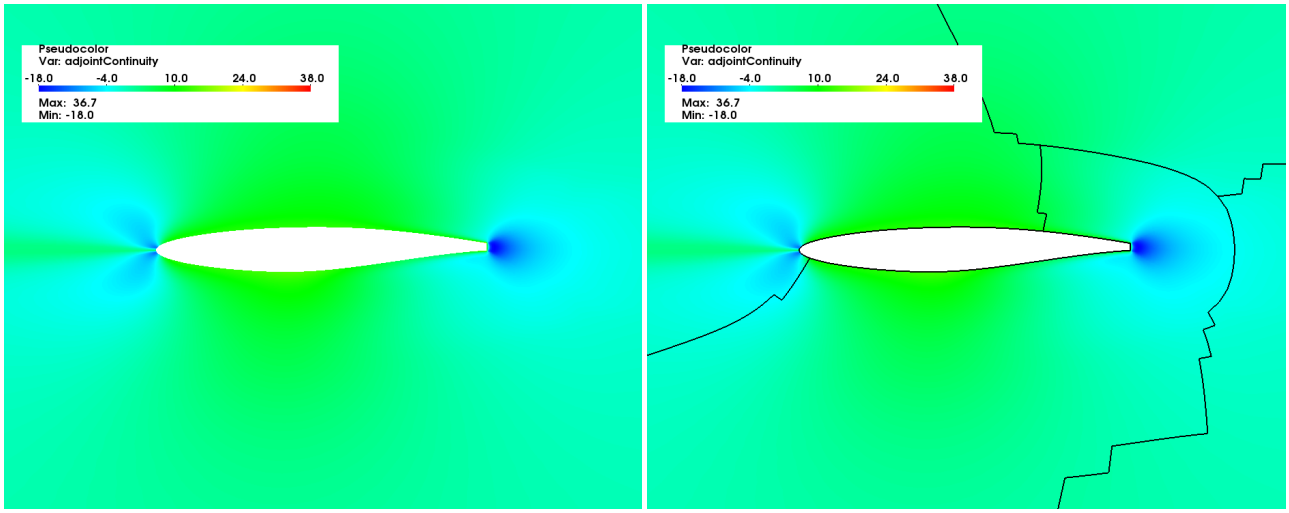
```
do colour=1,nColours !forward
  !$OMP PARALLEL DO DEFAULT SHARED
  do edge=firstEdge(colour),lastEdge(colour)
    call flux_loopbody(edge,res,u)
  end do
  !$OMP END PARALLEL DO
end do
do colour=nColours,1 !reverse
 !$OMP PARALLEL DO DEFAULT SHARED
 do edge=lastEdge(colour),firstEdge(colour)
    call flux_loopbody_b(edge,res,resb,u,ub)
 end do
 !$OMP END PARALLEL DO
end do
```

The flux computation at boundaries could be parallelised in a similar fashion. The primal solver uses a loop over boundary nodes and performs a computation using data from ghost nodes. Each ghost node is exclusively connected with one boundary node, and there is no communication with other nodes during the boundary flux computation. The parallel boundary treatment has not been used for this work.

4 Results

The primal and adjoint solvers in our in-house code $mgopt$ have been parallelised using both MPI and OpenMP paradigm. As an initial attempt, scaling of the inviscid first order primal and adjoint solver on subsonic flow ($M_\infty = 0.3, \alpha_{AOA} = 0^o$) over an $2d\ rae2822$ airfoil is shown. The cost function for the adjoint system is the aerodynamic drag on the airfoil. The mesh has $11,510$ cells and it is run on a four core Intel i7 processor. Pure MPI and OpenMP scaling is shown in figure 9(a) and a hybrid run of two MPI ranks each running two OMP threads within the ranks is compared against four MPI ranks and four OMP thread runs in figure 9(b). The baseline serial timing was obtained by running the solver with MPI code removed and using a single OMP thread. The relative error in the serial and parallel solution matches to machine precision for both primal and adjoint. The adjoint continuity for the serial and parallel case on

four partitions is shown in figure 8. The scaling results are not conclusive due to the small size of the test case. The scalability study for larger $3d$ meshes using the second order solver with the viscous terms is planned for future work.



(a) Serial run (b) Parallel run (on 4 partitions)

Figure 8: Drag adjoint continuity solution contour for subsonic flow over rae2822 airfoil ($M_\infty = 0.3, \alpha_{AOA} = 0^o$)

5 Conclusion

We have demonstrated that naive application of adjoint differentiation for MPI calls can fail for hand-assembled adjoint CFD solvers with zero-halo partitioning. We presented a correct reverse-differentation of MPI `allgather` calls in such scenarios in the context of an unstructured Finite Volume solver for compressible flow. In addition, we exploited the fact that memory access patterns for the primal and adjoint solvers are identical to implement shared-memory parallelism using OpenMP based on edge colouring. The hybrid parallelisation strategy was used in this work to obtain adjoint results based on a drag cost function for inviscid flow around a RAE2822 airfoil with truncated trailing edge. The results from serial and parallel execution runs were validated to be identical to machine precision, and scalability was presented for shared memory, distributed memory and hybrid parallelisation.

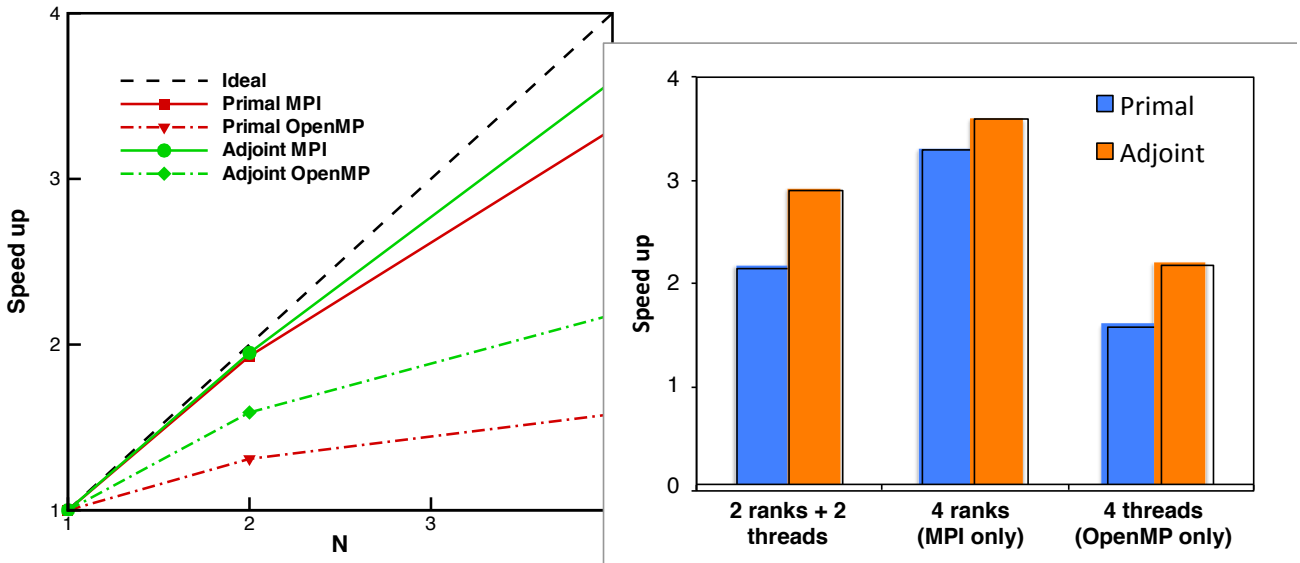
6 Acknowledgement

This research has been supported by the European Commission under the HORIZON 2020 Marie Curie fellowship (grant no. 642959).

REFERENCES

[1] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, and U. Naumann, *Toward Adjoinable MPI*, IEEE International Symposium on Parallel & Distributed Processing, 2009.

[2] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, U. Naumann, M. Schanen, and C. Hill, *Gradient of MPI-parallel codes*, slides from talk, June, 2012.



(a) Pure MPI and OpenMP

(b) Hybrid MPI and OpenMP

Figure 9: Strong scaling for the rae2822 on Intel i7 processor (four core)

[3] Y. Liu, *Hybrid Parallel Computation of OpenFOAM Solver on Multi-Core Cluster Systems*, Master of Science Thesis, KTH Sweden, 2011.

[4] C. Faidon, *Aerodynamic shape optimisation using adjoint sensitivities*, PhD Thesis, Queen Mary Univ. London, 2012.

[5] Hohenwarter, M. and Borcherds, M. and Ancsin, G. and Bencze, B. and Blossier, M. and Delobelle, A. and Denizet, C. and Éliás, J. and Fekete, Á and Gál, L. and Konečný, Z. and Kovács, Z. and Lizelfelner, S. and Parisse, B. and Sturr, G., *GeoGebra 4.4*, Dec 2013 (`http://www.geogebra.org`).

[6] P. Mohanamuraly, and K. Nagarajan, *Revisiting the space-filling curves for storage, reordering and partitioning mesh based data in scientific computing*, IEEE HiPC conference, Dec 2013.

[7] A. Griewank, and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd edition, SIAM, 2008.

[8] Guo X, Gorman G, Sunderland A et al, *Developing hybrid openmp-mpi parallelism for fluidity-next generation geophysical fluid modelling technology*.

[9] Giles MB, Mudalige GR, Sharif Z et al, *Performance analysis and optimization of the op2 framework on many-core architectures*, The Computer Journal, 2011

[10] Liao C, Hernandez O, Chapman B et al, *Openuh: An optimizing, portable openmp compiler*, Concurrency and Computation: Practice and Experience, 2007, 19(18): 2317–2332.

[11] H. Laurent, and P. Valérie, *The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification*, ACM Trans. Math. Softw., vol. 39, no. 3, pp. 20:1-20:43, 2013.