To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–21

Seeding and adjoining zero-halo partitioned parallel scientific codes

P. Mohanamuraly[a*] and L. Hascoët[b] and J.-D. Müller[a]

[a] *Queen Mary University of London, London, UK*
[b] *TROPICS team, INRIA Sophia-Antipolis, France*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Algorithmic differentiation tools can automate the adjoint transformation of parallel message passing codes [23] using the AMPI library. Nevertheless, a non-trivial and manual step after the differentiation is the initialisation of the seed and retrieval of the output values from the differentiated code. **MPK:** *Ambiguities in seeding occurs in programs where the user is unable to expose the complete program flow to the AD tool (a single entry and single exit point).* **:KPM** In this work, we present the problems and ambiguities associated with seed initialisation and output retrieval for adjoint transformation of halo and zero-halo partitioned MPI programs. **JDM:** *point out relevance: - no pb with single master paradigms, when using brute-force AD - problem when using more advanced SPMD/all-worker models, and/or partial hand-assembly.* **:MDJ** Shared-node reduction is an important parallel primitive in the zero-halo context. We introduce a general framework to eliminate ambiguities in seeding and retrieval for shared-node reduction over $+$, and $*$ operators using a conceptual master-worker model. Corollaries from the model show the need for new MPI calls for retrieval and eliminate MPI calls for seed initialisation. Different possible implementations for seeding manually assembled adjoints were inferred from the model, namely, (i) partial and (ii) unique seeding. We successfully demonstrate the seeding of the manually assembled adjoint fixed-point iteration in a $3d$ zero-halo partitioned unstructured compressible flow solver. The different implementations, their merits and demerits are highlighted. Tapenade AD tool was used throughout this work.

Keywords: Algorithmic differentiation; adjoint source transformation; parallel reverse mode; Gradient-based optimisation; Fixed-point iteration; Computational fluid dynamics (CFD); message passing

1. Introduction

Gradient-based optimisation methods are essential for solving optimisation problems with many control variables. The efficient computation of the required gradients is essential for application of numerical optimisation to large-scale and industrial cases, such as aerodynamic design which is the focus of our work. Algorithmic differentiation (AD) tools differentiate computer codes by applying symbolic differentiation to individual program statements and combining the sequence of statements using the chain rule of calculus. Moreover, in the typical case of a small number of output variables, their derivatives with respect to a large number of input variables can be efficiently obtained using the 'reverse' mode of AD, which accumulates the derivative by stepping through the code statements in reverse order. There are two major approaches to AD, namely (i) operator overloading

*Corresponding author. Email: pavanakumar.mohanamuraly@qmul.ac.uk and (ii) source transformation, in the present work we restrict to source transformation AD, although the problems and solutions presented are equally applicable to (i).

Differentiating sequential computer models with AD in reverse mode to obtain adjoint codes is reasonably mature and a range of AD tools have been developed, see www.autodiff.org for various offerings. However, application to parallel codes in reverse mode remains a challenge. A major step forward has been made with the development of the AMPI library [21] which, under certain restrictions, allows to automatically derive the required MPI communications for the reverse-differentiated code. AMPI focuses on the correct differentiation of an individual piece of code, but does not automate the integration of the differentiated elements into a driver code. In particular, the user has to provide appropriate seeding for the relevant inputs in order to obtain correct gradients. As we show in the following, this is far from trivial in practical application to typical solvers for PDES. We propose a framework that conceptually transforms the parallel algorithm into a simpler structure, which then allows to consistently define seeding for the typical range of reduction operations at parallel interfaces. In this paper, we focus on the more versatile message-passing parallelism and restrict our analysis to blocking communication.

2. Background

The solution of large-scale optimal design problems in the turn-around times required by industrial application is feasible today using adjoint models and parallel computing. Application of AD to obtain differentiated code of parallel computer codes based on both shared and distributed memory has been investigated in the literature. The parallel AD forward mode was first investigated by Hovland [10, 11]. The first major effort in applying AD to obtain adjoint MPI code has been on the MITgcm code using TAMC [9]. The authors report that a substantial intervention into the original code was necessary to enable correct adjoint code generation. The AD book of Griewank and Walther [7] also has a section dedicated to AD of parallel programs. A comprehensive theory of MPI within AD for both forward and reverse mode was presented by Utke et al. [23]. This has been implemented into the AMPI library [21] and later into the Adjoinable MPI library [23]. The libraries require rewriting the MPI calls into AMPI wrapper calls, and the AD tools generate the forward/reverse MPI calls automatically. This substantially reduces the burden of generating the AD MPI code for both source-transformation (S-T) and operator-overloading (O-O) AD tools. The Adjoinable MPI with the AD tool ADOL-C was used to differentiate the Ice Sheet System Model [22]. The authors had wrapped the MPI calls in the original code to generate correct MPI adjoint code. Recently, a discrete adjoint version of the open-source CFD solvers SU2 [1] has been carried out using the CoDiPack AD tool [18] and the Adjoinable MPI library.

Differentiation using AMPI in combination with an AD tool demands certain restrictions. Firstly, the library currently supports a subset of MPI v2.0 specification. Any unsupported MPI operations (by AMPI) in the user code must be replaced with an equivalent supported MPI call. But we hope that such restrictions would not exist in a future or next releases of the library. In fact, Schanen et al. [19, 20] have covered the adjoining of MPI v2.2 one-sided calls in a recent work. Secondly, the MPI function calls must be wrapped into equivalent AMPI ones. Lastly, the source for differentiation is isolated into a routine, referred to as the *head*. A *driver* routine generates the input seed and recovers the output values from the differentiated *head*. The *driver* routine generation is a manual process and non-trivial. Hovland [10] briefly mentions seed initialisation and associated problems in forward mode AD of MPI codes. The author states, “ … *user of an AD tool should be*

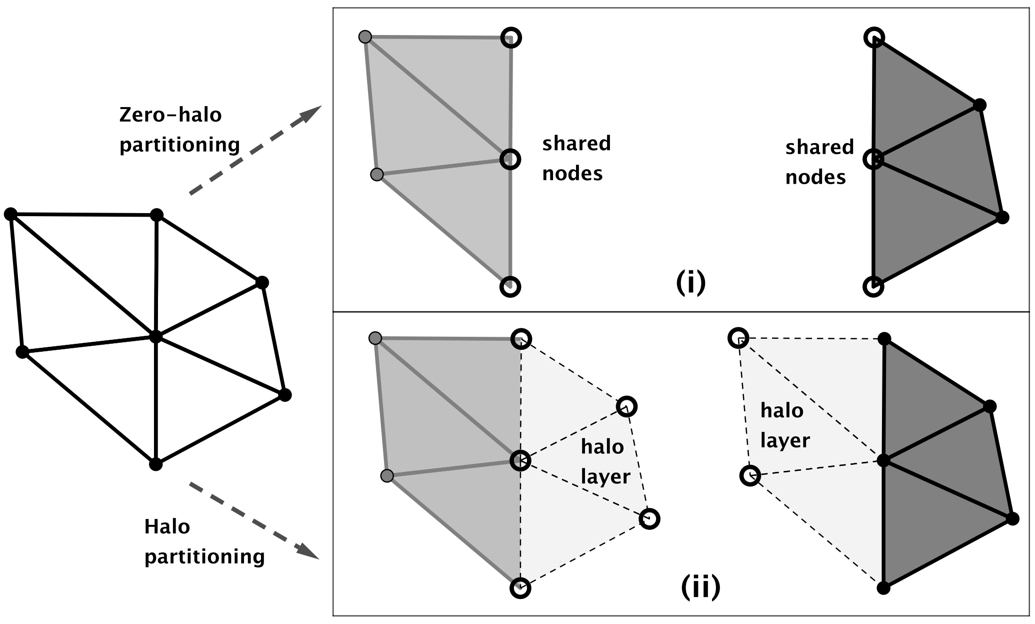


Figure 1.: The (i) Zero-halo and (ii) one-layer halo based partitioning methods illustrated for a sample mesh split into two parts (grey and black); halo/shared nodes denoted by hollow circles and internal nodes denoted by filled circles

careful to take advantage of situations where the structure of a computation allows derivatives to be computed much more efficiently". To the knowledge of the authors there are no existing works which investigate seeding for reverse-differentiated MPI code.

Generally, distributed parallel scientific codes divide a given problem domain (data) into multiple partitions, which are independently run in hardware computing units. Some level of synchronisation is necessary to ensure that the partitioned solution remains consistent across the whole domain. There are two major approaches to partitioning for parallel computation, namely, the (i) halo and (ii) zero-halo methods (or a combination thereof); illustrated in Fig. 1. OpenFOAM [12], AVBP [5], and STAMPS [15] (our in-house CFD solver) represent examples of parallel scientific codes which employ a zero-halo partitioning scheme. **MPK:** *SU2 [4], parallel frameworks like OP2 [16] library employ halo partitioning to distribute data across MPI ranks.* **:KPM** Zero-halo partitions contain shared nodes between partition boundaries (Fig. 1(i)), which are synchronised across partitions **MPK:** *but halo partitions have unique ownership of data and use halo data (overlap with neighbouring MPI rank) to access off-processor data.* **:KPM** We use a halo and zero-halo example here to demonstrate the difficulty in obtaining correct seeding values for reverse-differentiated complex parallel algorithms. **JDM:** *Since we also show halo examples, give codes that use it. SU2? Or Mike Giles' Op2 library?* **:MDJ**

The paper is organised as follows. In sec. 3 the seeding problem is described for the halo and zero-halo partitioning using a simple example. A conceptual master-worker (CMW) model is proposed in sec. 4 to guide the seeding process. We then demonstrate the use of CMW to seed the parallel adjoint FPI of an unstructured CFD solver in sec. 5. Concluding remarks are discussed in sec. 6.

3. Seeding the differentiated code

AD tools differentiate program code and can provide derivative values which are exact up to machine precision. Typically, the program code of a numerical application can be considered as a multivariate vector function F of an independent variable $\mathbf{x}$ as defined in Eq. (1), where F maps an input vector **MPK:** *$\mathbf{x} \equiv (x_i),\ i = 1, \ldots, n \in \mathcal{R}^n$ onto an output vector $\mathbf{y} \equiv (y_i),\ i = 1, \ldots, m \in \mathcal{R}^m$.* **:KPM** **JDM:** *What is this duble subscript notation? You mean $(x_i), i = 1, n$?* **:MDJ**

$$\mathbf{y} = F(\mathbf{x}) \tag{1}$$

The entries of the Jacobian $\nabla F(\mathbf{x}) \in \mathcal{R}^{m \times n}$ exist if F is continuously differentiable in the neighbourhood of all arguments. In order to compute this Jacobian using AD, we have the tangent-linear model and the adjoint (reverse-differentiated) model at our disposal. The computational complexity of the Jacobian accumulation based on the tangent-linear model depends on the dimension of the input n. In contrast, the complexity of the adjoint model depends on the output dimension m. In typical aerodynamic shape optimisation, the input dimension is much larger than the output size. Hence we focus on the adjoint model in this paper. The adjoint model of F is

$$\bar{\mathbf{x}} = \bar{\mathbf{x}} + \nabla F(\mathbf{x})^T \cdot \bar{\mathbf{y}}\,. \tag{2}$$

MPK: *Seeding is best described as the process of retrieval of the directional derivative in eq. (2) by initialising appropriate values for $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$. For example,* **:KPM** the Jacobian matrix can be constructed row-by-row in $\bar{\mathbf{x}}$ by initialising or seeding **MPK:** *$\bar{\mathbf{x}} \equiv (\bar{x}_i),\ i = 1, \ldots, n = 0$ followed by seeding $\bar{\mathbf{y}} = (\mathbf{e}_i),\ i = 1, \ldots, m$* **:KPM** over the range of Cartesian unit basis vectors $\mathbf{e}_i \in \mathcal{R}^m$. Note that m evaluations are necessary to assemble the complete Jacobian matrix. Moreover, when $m = 1$ only one evaluation is necessary to assemble all the entries of the gradient in contrast to n evaluations in the tangent-linear mode.

JDM: *You spent some introducing tan-lin and adj modes, but we don't yet know what seeding does.* **:MDJ** While the correct choice of seeding is straightforward for a serial code, it is not so intuitive for a parallel code. In the AMPI approach, the program to be differentiated has a top level routine *head* which does the numerical computation and communication. A *driver* routine calls this function in the main code. The *driver* routine would have to be manually adjusted to initiate the derivative computation, retrieve, and use the derivative values. Since the input and output data are distributed, so are the input seeds and output adjoint values. Therefore it is necessary to ensure correct distribution of the input seeds and retrieval of output values. In this work we use as an example the seeding of parallel programs based on the zero-halo data partition. To illustrate the parallel seeding issues for halo and zero-halo partitioning method, a simple summation problem is described next.

3.1 *Code example*

Let us consider a simple summation function shown in Eq. (3). The reverse differentiated form for this simple function is shown in Eq. (4).

$$y = F(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2 \tag{3}$$

$$\begin{aligned}\bar{x}_1 &= \bar{x}_1 + 2\bar{y}x_1 \\ \bar{x}_2 &= \bar{x}_2 + 2\bar{y}x_2 \\ \bar{x}_3 &= \bar{x}_3 + 2\bar{y}x_3\end{aligned} \tag{4}$$

Let us perform the computation in parallel on two MPI ranks, ranks 0 and 1. There are multiple ways one can partition this computations between the two ranks. We present two approaches (i) Halo and (ii) Zero-halo partitioning in the next section.

3.2 *Halo Approach*

In this approach $\{x_1\}$ is distributed to rank 0 and and $\{x_2,\ x_3\}$ are distributed to rank 1. Note that there is no sharing of data between the partitions. The primal and adjoint parallel code for ranks 0 and 1 are shown in Listings 1-4. The adjoint is simple to construct using MPI idioms [23]. The `read` function reads the value for the variable provided in the function argument. The shorthand notations `send` and `recv` stand for the `MPI_Send` and `MPI_Recv` functions. In Table 1 the comparison of the output from serial and parallel AD code is shown for seed values $\bar{y} = \bar{x}_1 = \bar{x}_2 = \bar{x}_3 = 1$. Note that the same serial seed value was used in the parallel adjoint code. We clearly find that the seed $\bar{y}$ is counted twice, which leads to a erroneous result. While it can be argued that the operation $y = y + t$ is a reduction and only one of the rank should take ownership of this operation, it is far from obvious for a practical user of AD. In fact, one might argue that exposing the complete program to the AD tool (including the data decomposition) will eliminate this issue. But this is impractical and impossible in many situations because source code availability cannot be assumed (commercial libraries). The AD tool might not support a given mix of programming languages or supports only a restricted subset. Therefore, in such situations one differentiates pieces of the overall program and assembles the differentiated subroutines manually.

Conceptually the AD process is clear and given a single entry and single exit for the MPI program one can quite reliable carry out the seeding and adjoining. The PGAS representation [20] of the program can be used to verify its correctness. In Listing 5-8 we provide the correct implementation using the idea of a single entry and single exit. In fact this problem has been acknowledged in the AD literature. For example Hovland [10] states that replicated operations on individual ranks are indeed a source of problem for seeding in the context of forward AD MPI codes.

Listing 1: Halo parallel code (rank 0)

```
read(x_1)

y = x_1^2
send(y, to=1)
recv(t, from=1)
y = y + t
```

Listing 2: Halo parallel code (rank 1)

```
read(x_2)
read(x_3)
y = x_2^2 + x_3^2
recv(t, from=0)
send(y, to=0)
y = y + t
```

Listing 3: Halo partitioning adjoint of $y = x_1^2 + x_2^2 + x_3^2$ (rank 0)

```
$\bar{t}$ = $\bar{y}$
send($\bar{t}$, to=1)
recv($t_0$, from=1)
$\bar{y}$ = $\bar{y}$ + $t_0$
$\bar{x_1}$ = $\bar{x_1}$ + $2\bar{y}x_1$

$\bar{y}$ = 0
```

Listing 4: Halo partitioning adjoint of $y = x_1^2 + x_2^2 + x_3^2$ (rank 1)

```
$\bar{t}$ = $\bar{y}$
recv($t_1$, from=0)
$\bar{y}$ = $\bar{y}$ + $t_1$
send($\bar{t}$, to=0)
$\bar{x_2}$ = $\bar{x_2}$ + $2\bar{y}x_2$
$\bar{x_3}$ = $\bar{x_3}$ + $2\bar{y}x_3$
$\bar{y}$ = 0
```

Table 1.: Serial seed with value 1, resulting gradients for sequential code and on each partition (halo approach).

Variable	**Seed**	**Serial**	**Rank 0**	**Rank 1**
$\bar{y}$	1	0	0	0
$\bar{x}_1$	1	$1 + 2x_1$	$1 + 4x_1$	$1 + 4x_1$
$\bar{x}_2$	1	$1 + 2x_2$	$1 + 4x_2$	$1 + 4x_2$
$\bar{x}_3$	1	$1 + 2x_3$	$1 + 4x_3$	$1 + 4x_3$

Listing 5: Corrected version of halo parallel code (rank 0)

```
read($x_1$)

$y$ = $x_1^2$
recv($t$, from=1)
$y$ = $y$ + $t$
send($y$, to=1)
```

Listing 6: Corrected version of halo parallel code (rank 1)

```
read($x_2$)
read($x_3$)
$y$ = $x_2^2$ + $x_3^2$
send($y$, to=0)
recv($t$, from=0)
$y$ = $t$
```

Listing 7: Corrected version of halo adjoint of $y = x_1^2 + x_2^2 + x_3^2$ (rank 0)

```
recv($\bar{y}$, from=1)
$\bar{t}$ = $\bar{y}$
send($\bar{t}$, to=1)
$\bar{x_1}$ = $\bar{x_1}$ + $2\bar{y}x_1$

$\bar{y}$ = 0
```

Listing 8: Corrected version of halo adjoint $y = x_1^2 + x_2^2 + x_3^2$ (rank 1)

```
$\bar{t}$ = $\bar{y}$
send($\bar{t}$, to=0)
recv($\bar{y}$, from=0)
$\bar{x_2}$ = $\bar{x_2}$ + $2\bar{y}x_2$
$\bar{x_3}$ = $\bar{x_3}$ + $2\bar{y}x_3$
$\bar{y}$ = 0
```

3.3 *Zero-halo Approach*

In this section we consider a zero-halo partitioning, where x_1 and x_3 are uniquely distributed to ranks 0 and 1, respectively, but x_2 is shared between the two ranks. In zero-halo terminology, x_2 is called a shared node variable at the zero-halo partition boundary.

In Listing 9 and 10 we have shown the pseudo code for the parallel implementation. A peculiar aspect of this zero-halo splitting is the partial computation on x_2 (halving of the value) on each rank and the final accumulation on y. The reverse-differentiated code in each rank is shown in Listings 11 and 12.

Listing 9: Zero-halo parallel code (rank 0)

```
read($x_1$)
read($x_2$)
$y$ = $x_1^2$ + $\frac{1}{2}x_2^2$
send($y$, to=1)
recv($t$, from=1)
$y$ = $y$ + $t$
```

Listing 10: Zero-halo parallel code (rank 1)

```
read($x_2$)
read($x_3$)
$y$ = $\frac{1}{2}x_2^2$ + $x_3^2$
recv($t$, from=0)
send($y$, to=0)
$y$ = $y$ + $t$
```

Listing 11: Zero-halo adjoint of $y = x_1^2 + x_2^2 + x_3^2$ (rank 0)

```
$\bar{t}$ = $\bar{y}$
send($\bar{t}$, to=1)
recv($t_0$, from=1)
$\bar{y}$ = $\bar{y}$ + $t_0$
$\bar{x_1}$ = $\bar{x_1}$ + $2\bar{y}x_1$
$\bar{x_2}$ = $\bar{x_2}$ + $\bar{y}x_2$
$\bar{y}$ = 0
```

Listing 12: Zero-halo adjoint of $y = x_1^2 + x_2^2 + x_3^2$ (rank 1)

```
$\bar{t}$ = $\bar{y}$
recv($t_1$, from=0)
$\bar{y}$ = $\bar{y}$ + $t_1$
send($\bar{t}$, to=0)
$\bar{x_2}$ = $\bar{x_2}$ + $\bar{y}x_2$
$\bar{x_3}$ = $\bar{x_3}$ + $2\bar{y}x_3$
$\bar{y}$ = 0
```

Table 2 compares the values of $\bar{x}_1, \bar{x}_2$ and $\bar{x}_3$ between the serial and parallel adjoint code. In Tab. 2 naively the same seeding approach as for the serial code was used. This results in incorrect derivatives, since the message exchange doubles the seed $\bar{y}$, but there is a missing accumulate in $\bar{x}_2$, cancelling out the error for $\bar{x}_2$. Although the doubling of $\bar{y}$ is obvious from the previous halo example one would still gets erroneous results using the proposed fix due to the shared node accumulation. As described previously the problem arises because the complete program flow is unavailable to the AD tool. The presence of shared variables x_2 and y in both MPI ranks is the source of the problem. The information that these variables should belong to a single rank and that the reduction should happen in a single rank is hidden from the AD tool. This motivates our conceptual master-worker (CMW) model, described in the next section.

4. Conceptual Master-Worker model for shared-node reduction

To derive correct choices of seed values, we consider transforming a parallel MPI program into a Conceptual Master-Worker (CMW) model. An abstract CMW model for the shared-node reduction operation is shown in Eq. (5), where the operator $\otimes$ represents a reduction operation at a shared node. Functions $\{F_0, F_1, \ldots, F_{N-1}\}$ are the rank-local operations on the shared node value x, and $\mathbf{S}_{rank}$ is the set of ranks shared by the node.

$$y = F_0(x) \otimes F_1(x) \otimes \ldots F_i(x), \;\; i \in \mathbf{S}_{rank} \tag{5}$$

In abstracting the shared node reduction using CMW we assume that a master process,

(1) controls ownership of the shared node variables x and y
(2) broadcasts the shared node variable x to worker ranks $\mathbf{S}_{rank}$
(3) delegates computation of $\{F_0, \ldots, F_i\}$ to worker ranks $\mathbf{S}_{rank}$
(4) upon completion, performs a global reduction over $\otimes$ and writes to y.

Thus the model essentially serialises the reads/writes to the shared nodes, and the workers merely perform computations using their local data. By defining input and output ends controlled by the single master process, this model eliminates the ambiguities in seeding these shared nodes. For the non-transformed original code, the broadcasts to or reductions from the workers then inform about the correct handling of seed values.

In the cases where technical limitations forbid to expose the complete program flow to the AD tool, the CMW model can be presented as an alternative way to provide the AD tool with a code that has a single entry point and a single exit point namely, the entry and the exit points of the master process. The CMW applied to the shared-node reduction is shown in the next subsection.

4.1 *CMW adjoint seeding algorithm for shared-node reduction*

The CMW adjoint model can be derived using Eq. (2) for the shared node reduction in the master process as shown in Eq. (6). Using the chain rule of calculus and grouping terms, one obtains Eqs. (7) and (8), which are specialised adjoints for two classes of binary operators, namely (i) additive ($\otimes \equiv +$) and (ii) multiplicative ($\otimes \equiv *$).

$$\bar{x} = \bar{x} + \left[\frac{d}{dx}\left(F_0(x) \otimes F_1(x) \otimes \ldots F_i(x)\right)\right]^T \bar{y} \tag{6}$$

Case (i): Additive CMW adjoint model

$$\bar{x} = \bar{x} + \sum_{i \in \mathbf{S}_{rank}} \left(\frac{dF_i(x)}{dx}\right)^T \bar{y} \tag{7}$$

Table 2.: Serial seed with value 1, resulting gradients for sequential code and on each partition (zero-halo approach).

Variable	**Seed**	**Serial**	**Rank 0**	**Rank 1**
$\bar{y}$	1	0	0	0
$\bar{x}_1$	1	$1 + 2x_1$	$1 + 4x_1$	$1 + 4x_1$
$\bar{x}_2$	1	$1 + 2x_2$	$1 + 2x_2$	$1 + 2x_2$
$\bar{x}_3$	1	$1 + 2x_3$	$1 + 4x_3$	$1 + 4x_3$

Case (ii): Multiplicative CMW adjoint model

$$\bar{x} = \bar{x} + \sum_{i \in \mathbf{S}_{rank}} \left(\frac{dF_i(x)}{dx} \prod_{j \neq i}^{\mathbf{S}_{rank}} F_j(x) \right)^T \bar{y} \tag{8}$$

The following corollaries can be inferred from the adjoint models of Eqs. (7) and (8),

a) whatever be the type of binary operator for the shared node reduction, the adjoint model is always additive
b) individual terms under the summation are rank local operations (*head* routine)
c) the seed $\bar{y}$ is the common multiplying factor for all additive terms (in *head*)
d) the seed $\bar{x}$ is accumulated in only one process (master)

The doubling of $\bar{y}$ in the adjoint of the summation example is explained by (c), and corollary (a) explains the missing accumulate in $\bar{x}$. The corollaries (b) and (d) motivate the following rearrangement of Eq. (7).

$$\bar{x} = \sum_{i \in \mathbf{S}_{rank}} \left[\alpha_i \bar{x} + \left(\frac{dF_i(x)}{dx} \right)^T \bar{y} \right], \qquad \text{where} \sum_{i \in \mathbf{S}_{rank}} \alpha_i = 1 \tag{9}$$

Note that a similar rearrangement of Eq. (8) is possible. This rearrangement paves way for many possible implementation for seeding $\bar{x}$, which are discussed in the next section on seeding strategies.

The CMW abstraction for the summation problem (zero-halo partitioned) described in the previous section involves substituting, $\otimes \equiv +$, $N = 2$, $\mathbf{S}_{rank} = \{0, 1\}$, $\{F_0, F_1\} \equiv \{\frac{1}{2}x_2, \frac{1}{2}x_2\}$ in Eq. (5). The summation problem and its adjoint using the CMW adjoint model on two MPI ranks is illustrated in Fig. 2. The pseudo-code for the CMW primal and adjoint is shown in Listings 13 and 14, which show the fix for the doubling of $\bar{y}$ and the missing accumulate in $\bar{x}_2$ as identified previously in (a-d). The master broadcasts the same seed $\bar{y}_{master}$ to both MPI ranks. The results $t_{0,1}$ from both ranks are accumulated in the master rank ($\bar{x}_2 = \bar{x}_{2master} + t_0 + t_1$).

Listing 13: CMW primal of summation problem (zero-halo)

```
!------- master --------
read($x_1$, $x_2$, $x_3$)
send([$x_1$,$x_2$], to=0)
send([$x_2$,$x_3$], to=1)
!------- worker 0 -------
recv([$x_1$,$x_2$], from=master)
$y$ = $x_1^2$ + $\frac{1}{2}x_2^2$
send($y$, to=master)
!------- worker 1 -------
recv([$x_2$,$x_3$], from=master)
$y$ = $x_3^2$ + $\frac{1}{2}x_2^2$
send($y$, to=master)
!------- master --------
recv($t_0$, from=0)
recv($t_1$, from=1)
$y$ = $t_0$ + $t_1$
...
...
```

Listing 14: CMW adjoint of summation problem (zero-halo)

```
!------- master --------
$\bar{y}$ = $\bar{y}_{master}$
send($\bar{y}$, to=0)
send($\bar{y}$, to=1)
!------- worker 1 -------
recv($\bar{y}$, from=master)
$\bar{x}_2$ = $\bar{y}x_2$; $\bar{x}_1$ = $2\bar{y}x_1$
send([$\bar{x}_1$,$\bar{x}_2$], to=master)
!------- worker 0 -------
recv($\bar{y}$, from=master)
$\bar{x}_2$ = $\bar{y}x_2$; $\bar{x}_3$ = $2\bar{y}x_3$
send([$\bar{x}_2$,$\bar{x}_3$], to=master)
!------- master --------
recv([$\bar{x}_1$,$t_0$], from=0)
recv([$t_1$,$\bar{x}_3$], from=1)
$\bar{x}_1$ = $\bar{x}_{1master}$ + $\bar{x}_1$
$\bar{x}_2$ = $\bar{x}_{2master}$ + $t_0$ + $t_1$
$\bar{x}_3$ = $\bar{x}_{3master}$ + $\bar{x}_3$
```

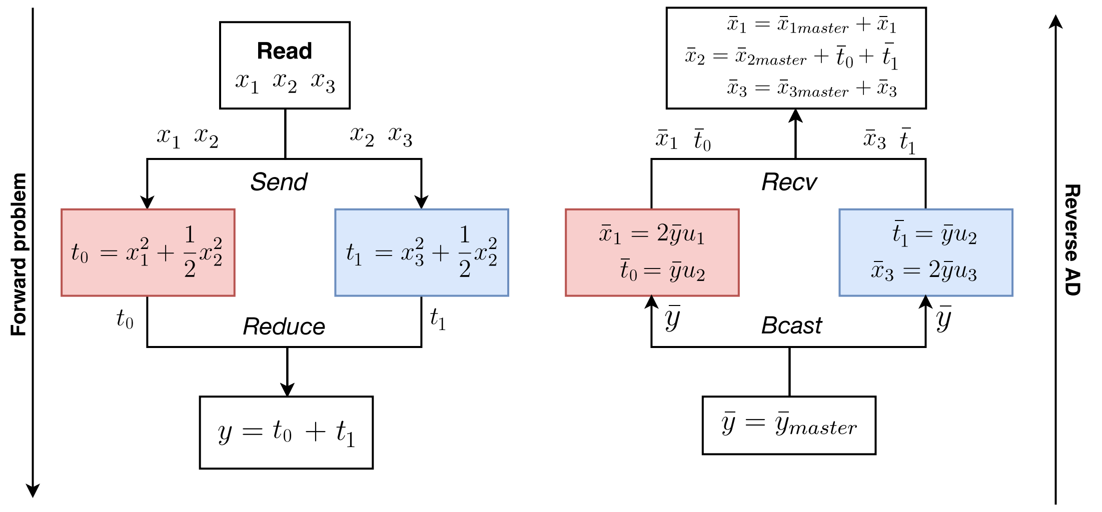

Read
x_1 x_2 x_3
x_1 x_2
x_2 x_3
Send
$t_0 = x_1^2 + \frac{1}{2}x_2^2$
$t_1 = x_3^2 + \frac{1}{2}x_2^2$
t_0
Reduce
t_1
$y = t_0 + t_1$
Forward problem
$\bar{x}_1 = \bar{x}_{1master} + \bar{x}_1$
$\bar{x}_2 = \bar{x}_{2master} + \bar{t}_0 + \bar{t}_1$
$\bar{x}_3 = \bar{x}_{3master} + \bar{x}_3$
$\bar{x}_1$ $\bar{t}_0$
$\bar{x}_3$ $\bar{t}_1$
Recv
$\bar{x}_1 = 2\bar{y}u_1$
$\bar{t}_0 = \bar{y}u_2$
$\bar{t}_1 = \bar{y}u_2$
$\bar{x}_3 = 2\bar{y}u_3$
$\bar{y}$
Bcast
$\bar{y}$
$\bar{y} = \bar{y}_{master}$
Reverse AD

Figure 2.: CMW model of parallel summation problem (left) and its adjoint (right) on two MPI ranks (red and blue boxes) and a master (white box)

4.2 *Seeding strategies*

Listings 15 and 16 show two possible strategies, unique seeding (US) and partial (PS) seeding for the summation problem using the CMW model. Note that both codes yield identical results and they are consistent with the CMW model. In both approaches the seed $\bar{y}$ is assigned the same serial value, which we denote as $\bar{y}_{master}$.

The final value of $\bar{x}_2$ is accumulated in both US and PS. The difference stems from the seeding of $\bar{x}_b$. In US only one rank amongst the shared ranks assigns the serial seed value $\bar{x}_{2master}$, while others have a zero seed value. In PS a partial seed value is assigned to $\bar{x}_2$ in each rank. For this example we have chosen to use the value of $\bar{x}_{2master}$ divided by the total number of shared ranks. In fact, using Eq. (9) one can show that any linear combination of the seed whose shared node sum is equal to the $\bar{x}_{2master}$ will produce correct adjoint values as shown in Eq. (10).

$$\bar{x}_2^{local} = \alpha^{local}\bar{x}_{2master}, \quad \sum^{\mathbf{S}_{rank}} \alpha^{local} = 1 \tag{10}$$

Zero-halo partitioning performs partial computations at a shared node. Therefore by using the PS approach no change is needed to the output value produced by reverse AD for the partition.

In the next section, we demonstrate this seeding approach applied to a parallel CFD code. We show that choosing the PS seeding strategy one can also simplify the code and reduce the number of MPI calls.

Listing 15: Unique seeding approach (US)

```
!------ rank 0 ---------
$\bar{y}$ = $\bar{y}_{master}$
$\bar{x}_2$ = $\bar{x}_{2master}$
$\bar{x}_1$ = $\bar{x}_1$ + $2\bar{y}x_2$
$\bar{x}_2$ = $\bar{x}_2$ + $\bar{y}x_2$
recv($t$, from=1)
$\bar{x}_2$ = $\bar{x}_2$ + $t$
send($\bar{x}_2$, to=1)
$\bar{y}$ = 0
!------ rank 1 ---------
$\bar{y}$ = $\bar{y}_{master}$
$\bar{x}_2$ = 0
$\bar{x}_2$ = $\bar{x}_2$ + $\bar{y}x_2$
$\bar{x}_3$ = $\bar{x}_3$ + $2\bar{y}\bar{x}_3$
send($\bar{x}_2$, to=0)
recv($t$, from=0)
$\bar{x}_2$ = $\bar{x}_2$ + $t$
$\bar{y}_b$ = 0
```

Listing 16: Partial seeding approach (PS)

```
!------ rank 0 ---------
$\bar{y}$ = $\bar{y}_{master}$
$\bar{x}_2$ = $\frac{1}{2}\bar{x}_{2master}$
$\bar{x}_1$ = $\bar{x}_1$ + $2\bar{y}x_1$
$\bar{x}_2$ = $\bar{x}_2$ + $\bar{y}x_2$
recv($t$, from=1)
$\bar{x}_2$ = $\bar{x}_2$ + $t$
send($\bar{x}_2$, to=1)
$\bar{y}$ = 0
!------ rank 1 ---------
$\bar{y}$ = $\bar{y}_{master}$
$\bar{x}_2$ = $\frac{1}{2}\bar{x}_{2master}$
$\bar{x}_2$ = $\bar{x}_2$ + $\bar{y}x_2$
$\bar{x}_3$ = $\bar{x}_3$ + $2\bar{y}_b x_3$
send($\bar{x}_2$, to=0)
recv($t$, from=0)
$\bar{x}_2$ = $\bar{x}_2$ + $t$
$\bar{y}$ = 0
```

5. Case study

Let us consider an aerodynamic shape optimisation problem as formulated in Eqs. (11),(12).

$$\min_{\alpha} J(U,\alpha), \tag{11}$$

$$s.t. \quad R(U,\alpha) = 0, \tag{12}$$

where J is a scalar cost-function such as drag, lift, etc., which is minimised subject to the PDE constraint R. In this work, the constraint R are the discrete Reynolds Averaged Navier-Stokes equations (RANS) solved numerically using the edge-based finite-volume solver STAMPS [15]. The field U is the fluid-state and α is the vector of shape design variables. Gradient-based optimisation methods require the derivatives of the cost or objective function (output) with respect to the design variables (input), $\frac{dJ}{d\alpha}$. The computational cost of computing sensitivities with the reverse or adjoint mode is proportional to the dimension of the output J and independent of the dimension of the input α. In this case there is a single scalar cost-function but many design variables, it is hence computationally efficient to obtain $\left(\frac{dJ}{d\alpha}\right)^T$ using the adjoint method as shown in Eq. (13). Algorithmic differentiation (AD) in the reverse or adjoint mode is used to obtain the transposed values. The AD tool Tapenade[8] is used in this work to obtain the terms of the adjoint Eq. (14), which solves the adjoint equation for $\bar{U}$,

$$\left(\frac{dJ}{d\alpha}\right)^T = \left(\frac{\partial J}{\partial \alpha}\right)^T + \left(\frac{\partial R}{d\alpha}\right)^T \bar{U}, \tag{13}$$

$$\left(\frac{\partial R}{\partial U}\right)^T \bar{U} - \left(\frac{\partial J}{\partial U}\right)^T = 0. \tag{14}$$

The steady-state primal in Eq. (12) and the adjoint in Eq. (14) are solved using a pseudo-time marching (pseudo-time τ) formulation as shown in Eqs. (15),(16). The discretisation of STAMPS is based on primitive variables Q, hence the code differentiated by the S-T AD tool Tapenade is also with respect to Q. However the constraint PDEs (12) are based on the the conservative variables U, therefore, it is necessary to apply a transformation using the matrix $\mathbf{T}^{-1} = \frac{\partial Q}{\partial U}$.

$$\frac{\partial U}{\partial \tau} = -R(U) \tag{15}$$

$$\frac{\partial \bar{U}}{\partial \tau} = -\left[\left(\frac{\partial R}{\partial U}\right)^T \bar{U} - \left(\frac{\partial J}{\partial U}\right)^T\right] = -\mathbf{T}^{-T}\left[\left(\frac{\partial R}{\partial Q}\right)^T \bar{U} - \left(\frac{\partial J}{\partial Q}\right)^T\right] \tag{16}$$

The distributed parallelisation of STAMPS [14, 15] has been implemented using the Message Passing Interface (MPI) [3]. The computational mesh is partitioned using METIS [13], shared node information using the zero-halo approach is set up in a pre-processing step. A schematic of the flux computation at a shared-node is shown in Fig. 3. The flux $F(Q_L, Q_R)$ across an edge is obtained using an approximate Riemann solver based on the left (Q_L) and right (Q_R) reconstructed states either side of the interface associated with each edge. The residual $R(Q)$ is defined as the flux summation of all edges connected to a node. Note that in the zero-halo approach there is no need to compute any flux for an interface on the partition boundary because both matching faces either side mutually cancel each other. At the shared nodes in each rank the zero-halo approach computes only a partial flux residual. Therefore, the residuals need to be accumulated to obtain the full shared-node residual. Using the CMW model introduced in Sec. 4 and the following definitions for the shared-node residual operation, $\otimes \equiv +$, $\{F_0, ..., F_i\} \equiv \{R_0, ..., R_i\}$, and $x \equiv u$ we obtain

$$R(Q) = \quad R_0(Q) + \ldots + R_i(Q), \quad i \in \mathbf{S}_{rank} \tag{17}$$

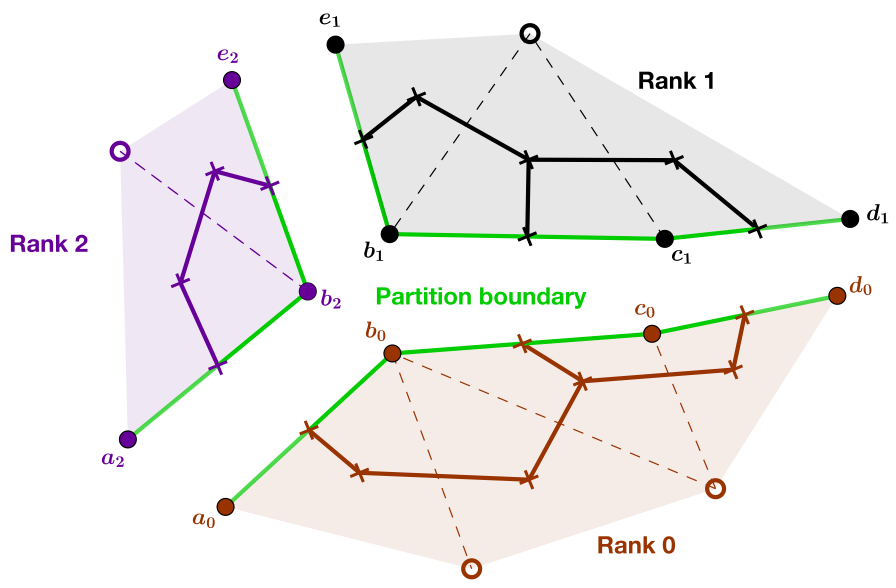


Figure 3.: Exploded view of zero-halo partitioning (three partitions); ● : shared nodes, ○ : internal nodes, × : dual nodes, — : partition boundary, — : dual element, -- : primal element.

The operator in Eq. (17) is implemented using `MPI_send` and `MPI_recv` communication of shared node values in STAMPS. Since this operator occurs quite frequently in the analysis, it is convenient to abstract this as an operator called `accumulate` which accumulates the shared node values within a rank using an off-processor shared node MPI schedule.

5.1 *MPI Fixed point loop*

The steady-state solution to Eq. (12) is obtained by solving the pseudo-time fixed-point iteration (FPI) in Eq. (18) using the multi-grid accelerated, pre-conditioned Runge-Kutta time stepping scheme, JT-KIRK [25]. The Runge-Kutta time stepping is preconditioned using an ILU(0) preconditioned inexact Newton-Krylov solver. For simplicity, we represent the complete time-stepping scheme as the preconditioning matrix M in Eq. (18). The FPI loop of the primal is re-used to solve the adjoint system by transposing the pre-conditioner, but retaining the primal multi-grid operators.

$$U^{k+1} = U^k - \mathbf{M}^{-1} R(U^k, \alpha) \tag{18}$$

$$\bar{U}^{k+1} = \bar{U}^k - \mathbf{M}^{-T}\mathbf{T}^{-T}\left[\left(\frac{\partial R}{\partial Q}\right)^T \bar{U}^k - \left(\frac{\partial J}{\partial Q}\right)^T\right] \tag{19}$$

The right hand side term in Eq. (19) containing the partial derivative is obtained by reverse differentiation of the flux residual and the cost-function subroutines using Tapenade. Eqs. (21) and (20) show the mathematical equivalent of the reverse differentiated code for the cost-function and flux residual subroutines. Firstly, we obtain $\bar{Q} = (\partial J/\partial Q)^T$ from Eq. (20) using the seed values $\bar{J} = 1$ and $\bar{Q} = 0$. Then we obtain $\bar{Q} = (\partial R/\partial Q)^T \bar{U}^k - (\partial J/\partial Q)^T$ from Eq. (21) using the seed values $\bar{Q} = -(\partial J/\partial Q)^T$ and

$\bar{R} = \bar{U}$.

$$AD: \quad J(Q) \to \bar{Q} = \bar{Q} + \left(\frac{\partial J}{\partial Q}\right)^T \bar{J} \tag{20}$$

$$AD: \quad R(Q) \to \bar{Q} = \bar{Q} + \left(\frac{\partial R}{\partial Q}\right)^T \bar{R} \tag{21}$$

5.2 *Parallel FPI seeding*

The pseudo-code for the parallel functions `residual` and `costFun` are shown in Listings 17 and 18. The program flow for the CMW model is illustrated alongside the listings in Figs. 4 and 5. The `accumulate` operator is the shared-node accumulation operator described previously. The `allreduce` is the reduction operation `MPI_Allreduce` on the scalar cost-function value J using summation (+).

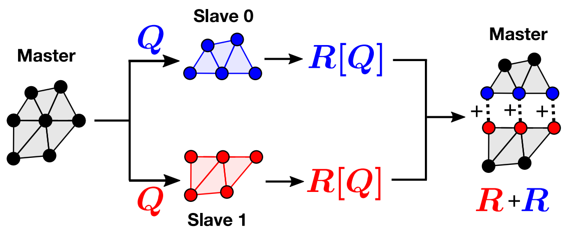


Figure 4.: Residual CMW model

```
residual(u, R):
  ! Local residual
  residualLocal(u, R)
  accumulate(R)
end
```

Listing 17: Residual function

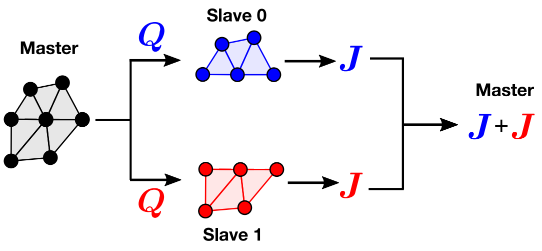


Figure 5.: Cost-function CMW model

```
costFun(u, J):
  ! Local cost-function
  costFunLocal(u, J)
  allreduce(J, +)
end
```

Listing 18: Cost-function scalar

The US and PS adjoint seeding for the flux residual function are shown in Figs. 6 and 7. In the US approach, the seed values at shared nodes are non-zero only for a uniquely chosen worker rank, say Worker 1. The shared nodes for other worker ranks are assigned a zero seed value, denoted by hollow circles (Worker 0). In the PS approach, partial seed values are assigned to shared nodes, denoted by triangles. The cost-function adjoint is agnostic to the type of seeding approach because $\bar{Q} = 0$. as shown in Fig. 8. Note that the cost-function adjoint produces only partial values of $\bar{Q}$ at the shared nodes, which is later accumulated by the master. The two adjoints can now be combined to yield the RHS term of Eq. (19).

When seeding the assembled FPI we also can use either the US or the PS approach, as illustrated in Figs. 9 and 10. The corresponding pseudo code is shown in Listings 19 and 20. The PS approach avoids two extra operations present in the the US approach, (i) accumulation of $\bar{Q}$ after the end of the adjoint cost-function and (ii) unique seeding for the adjoint residual. Thus, the PS approach simplifies the adjoint code. Note that in the PS approach the MPI call elimination occurs outside the fixed-point loop. Therefore, the general scalability remains the same for both methods. If the cost-function is not expensive

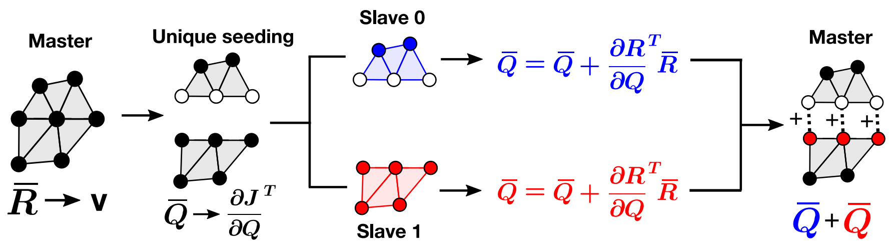

Master
Unique seeding
Slave 0
Slave 1
Master
$\bar{R} \rightarrow v$
$\bar{Q} \rightarrow \frac{\partial J}{\partial Q}^T$
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
$\bar{Q} + \bar{Q}$

Figure 6.: CMW adjoint residual using US approach; the hollow '∘' shows the zeroing of seed in non-master ranks and '+' is the reduction at shared-nodes on the master rank

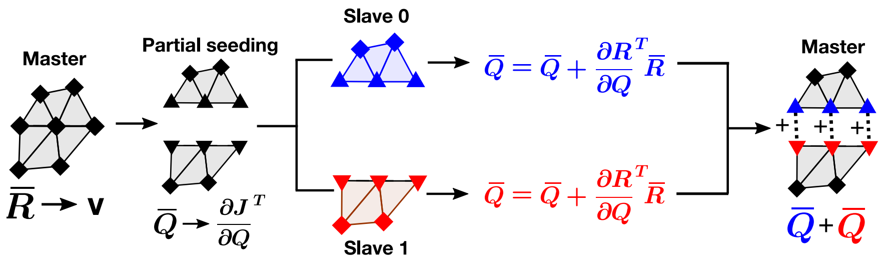

Master
Partial seeding
Slave 0
Slave 1
Master
$\bar{R} \rightarrow v$
$\bar{Q} \rightarrow \frac{\partial J}{\partial Q}^T$
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
$\bar{Q} + \bar{Q}$

Figure 7.: CMW adjoint residual using PS approach; the '▲/▼' shows the partial seed values distributed to the shared nodes and '+' is the reduction at shared-nodes on the master rank

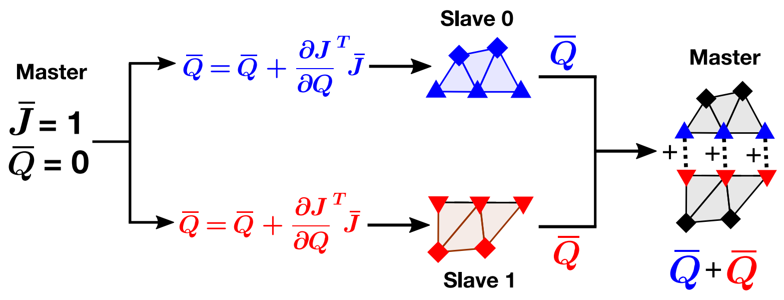

Master
$\bar{J} = 1$
$\bar{Q} = 0$
$\bar{Q} = \bar{Q} + \frac{\partial J}{\partial Q}^T \bar{J}$
$\bar{Q} = \bar{Q} + \frac{\partial J}{\partial Q}^T \bar{J}$
Slave 0
Slave 1
$\bar{Q}$
$\bar{Q}$
Master
$\bar{Q} + \bar{Q}$

Figure 8.: CMW adjoint of cost-function; the '▲/▼' shows the partial seed value computed at **MPK:** *shared nodes in each partition* **:KPM** **JDM:** *you mean 'shared' node? What is a partition share node?* **:MDJ** and '+' is the reduction at shared-nodes on the master rank

to evaluate then one can recalculate the cost-function and avoid the extra storage for dJ/dQ.

The low storage adjoint version where the cost-function is re-computed each iteration is shown in Listing 21. Note that in STAMPS only the PS algorithm of Listing 20 has been implemented for the adjoint FPI. We make use of the proof of Christianson [2], who showed that for a non-linear primal FPI the adjoint gradients can be computed without storing intermediate stages of the primal loop. The adjoint pseudo-code in Listings 19-21 follows closely the work of Giering [6].

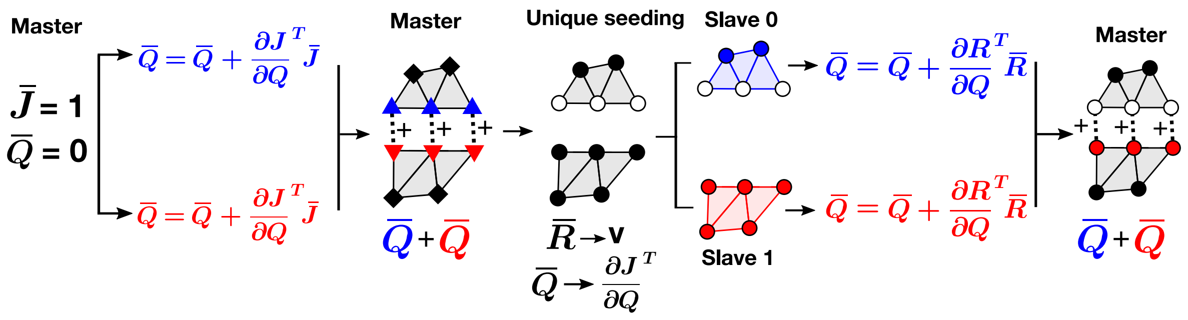

Master
$\bar{J} = 1$
$\bar{Q} = 0$
$\bar{Q} = \bar{Q} + \frac{\partial J}{\partial Q}^T \bar{J}$
Master
$\bar{Q}+\bar{Q}$
Unique seeding
$\bar{R} \rightarrow v$
$\bar{Q} \rightarrow \frac{\partial J}{\partial Q}^T$
Slave 0
Slave 1
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
Master
$\bar{Q}+\bar{Q}$

Figure 9.: CMW adjoint combining US residual and cost-function; the '▲/▼' shows the partial seed value computed at **MPK:** *share nodes in each partition* **:KPM** , **JDM:** *same comment as above* **:MDJ** the hollow '○' shows the zeroing of seed in non-master ranks, and '+' is the reduction at shared-nodes on the master rank

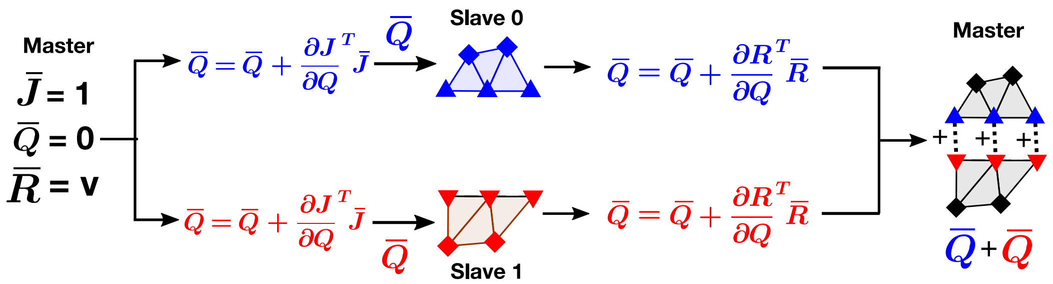

Master
$\bar{J} = 1$
$\bar{Q} = 0$
$\bar{R} = v$
$\bar{Q} = \bar{Q} + \frac{\partial J}{\partial Q}^T \bar{J}$
$\bar{Q}$
Slave 0
Slave 1
$\bar{Q} = \bar{Q} + \frac{\partial R}{\partial Q}^T \bar{R}$
Master
$\bar{Q}+\bar{Q}$

Figure 10.: CMW adjoint combining PS residual and cost-function; the '▲/▼' shows the partial seed value computed at each partition share node and '+' is the reduction at shared-nodes on the master rank

Listing 19: Adjoint FPI (US approach)

```
J̄ = 1
Q̄ = 0
costFunLocal(Q, Q̄, J, J̄)
accumulate(Q̄) !Extra MPI
dJdQ = Q̄
do iter = 1, nIter
   R̄ = v
   Q̄ = 0
   residualLocal(Q, Q̄, R, R̄)
   accumulate(Q̄)
   Q̄ = Q̄ - dJdQ
   update(v, Q̄)
end do
```

Listing 20: Adjoint FPI (PS approach)

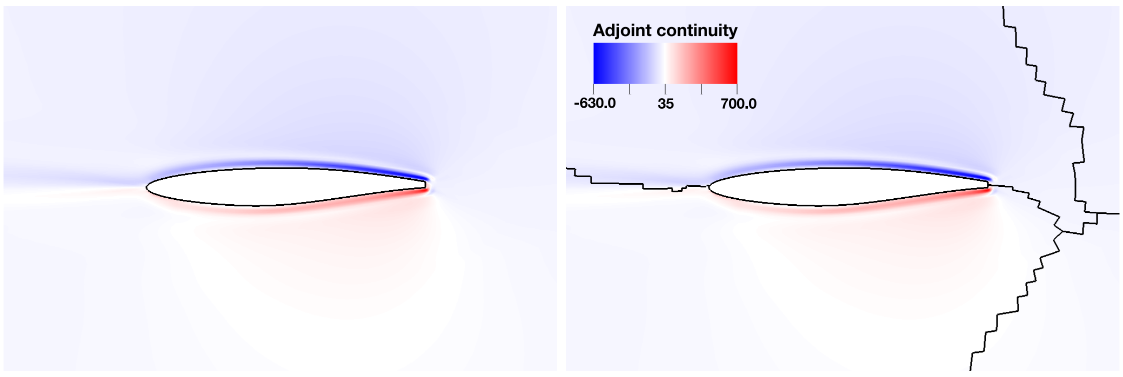
```
J̄ = 1
Q̄ = dJdQ
costFunLocal(Q, Q̄, J, J̄)

dJdQ = Q̄
do iter = 1, nIter
   R̄ = v
   Q̄ = 0
   residualLocal(Q, Q̄, R, R̄)
   accumulate(Q̄)

   update(v, Q̄)
end do
```

Listing 21: Adjoint FPI (re-computation)

```
do iter = 1, nIter
   $\bar{J}$ = 1
   $\bar{Q}$ = 0
   $\overline{\texttt{costFunLocal}}$($Q$, $\bar{Q}$, $J$, $\bar{J}$)
   $\bar{R}$ = $v$
   $\overline{\texttt{residualLocal}}$($Q$, $\bar{Q}$, $R$, $\bar{R}$)
   accumulate($\bar{Q}$)
   update($v$, $\bar{Q}$)
end do
```

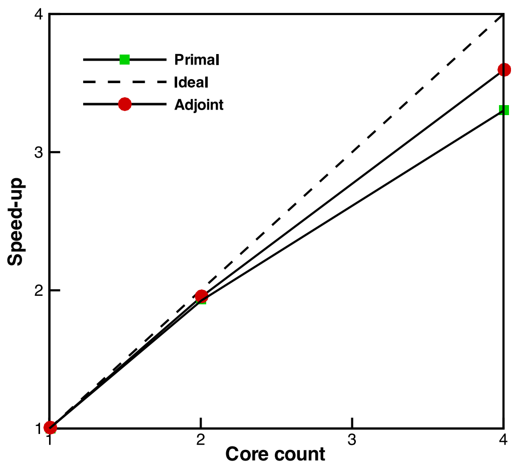
To demonstrate the MPI zero-halo adjoint in STAMPS, we have simulated external flow over a truncated RAE2822 airfoil. The simulation was carried out for a compressible viscous fluid in laminar flow at a Mach number of 0.2 and zero angle of attack. The domain was discretised using 23458 mesh nodes. The spatial discretisation uses Roe'd flux difference splitting [17] and 2nd-order MUSCL extrapolation [24] without slope limiter. Time integration is performed with explicit local time stepping using 4-stage Runge-Kutta integration. The parallel simulations were carried out using four partitions of the airfoil mesh. The cost-function used for the adjoint solver was the aerodynamic drag. The solution for the adjoint continuity is shown in Fig. 11 with the partition boundaries drawn in solid lines. Table 3 shows the L_∞ difference between the converged solutions of the serial and parallel adjoint codes. We see that the solutions match to machine round-off which validates the manually-derived MPI-parallel adjoint solver against the serial adjoint solver.

The primal and adjoint solver scaling on a dual socket 16 Core Intel Xeon compute node is shown in for the RAE2822 test case in fig. 11(a), and for a larger DLRF6 wing body case with 3.5 M hexahedral cells in fig. 11(b). In both test cases the adjoint cost-function was the aerodynamic drag. The MPI communication cost between the adjoint and primal FPI is the same, but the adjoint performs more computation than the primal. Hence we find that the explicit adjoint solver has better strong scaling than the primal. It is to be noted that blocking MPI calls were used for communication, which reduces overall scalability.

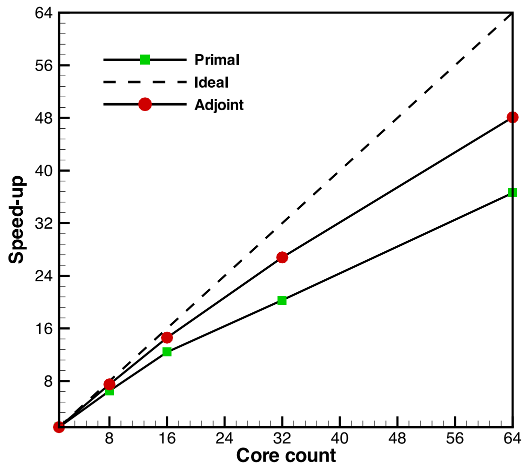


(a) Serial run (b) Parallel run (on 4 partitions)

Figure 11.: Drag adjoint continuity solution contour for subsonic flow over rae2822 airfoil ($M_\infty = 0.2, \alpha_{AOA} = 0^o$); partition boundary shown by solid black lines



(a) RAE2822 airfoil test case



(b) DLRF6 test case

Figure 12.: Adjoint and primal solver scalability for the indicated test cases (PS adjoint)

Table 3.: L_∞ error between the serial and parallel converged adjoint solution

Quantity	L_∞ error
Adjoint continuity	1.3435e-13
Adjoint X-momentum	2.9346e-13
Adjoint Y-momentum	9.52978e-14
Adjoint energy	7.3585e-14

6. Summary and Future Work

Automatic differentiation (AD) tools can apply reverse-differentiation to produce adjoints for codes with message passing parallelisation. To obtain correct gradients, context information about the communication pattern is necessary. When applying AD, the numerical computation that is to be differentiated, including its parallel communication, is isolated into a top-level routine called *head*. A manually generated *driver* which initialises the seed values, calls the differentiated head and retrieves its output. Both seeding and retrieval are manually coded non-trivial operations. **MPK:** *Except for a brief note in the thesis of Hovland [10] on seeding forward mode MPI codes, to the authors knowledge there are no known references on this topic.* **:KPM** **JDM:** *You cited Hovland for tan-lin, Schanen?* **:MDJ**

Correctness of parallel seed initialisation has been analysed for the reverse-differentiated summation reduction using the two major communication paradigms, namely halo and zero-halo partitioning. It was shown that the naïve approach of using the serial seed values and distributing them using reversed communication fails to produce the correct adjoint output. **MPK:** *The incorrectness is caused by not exposing the complete program flow to the AD tool due to practical limitations.* **:KPM** **JDM:** *And in the case of one-halo?* **:MDJ**

The paper introduced the Conceptual Master-Worker (CMW) abstraction for the shared-node reduction operation, which conceptually serialises the parallel program into a single master process. Applying either of two proposed seeding strategies between the master and her workers then removes the seeding ambiguity and defines consistent seed values which result in correct gradients. As a direct consequence of this transformation, the CMW adjoint model allows to rigorously construct the correct seed values. The abstraction has been generalised to a class of commonly used binary operators, the corollaries inferred from the CMW adjoint serve as a users' manual to seed initialisation. CMW provides an abstract framework to resolve ambiguities in seeding by conceptual serialisation and hence extends beyond the shared-node reduction discussed in this work.

The relevance of the analysis has been demonstrated on the problem of seed initialisation as arising from the manual assembly of fixed-point iterations (FPI) for parallel adjoint AD transformations. During this assembly, ambiguities in seeding occur because the output of one transformation becomes the input seed of another, linked through the manually produced driver code. The proposed abstraction was used to derive correct seeding and demonstrated with the manual assembly of the adjoint FPI in the in-house CFD solver STAMPS. The CMW abstraction motivated different possibilities to seeding, namely, partial and unique seeding. Partial seeding was found to be more efficient since it avoided MPI communication during seed initialisation. The correctness and performance of adjoint FPI using message passing parallelism. was demonstrated on the flow over an aerofoil with drag as an objective. Excellent agreement was obtained between the gradients computed with serial and parallel adjoint algorithms.

CMW modelling is a general technique to derive and guide parallel seed initialisation for adjoint transformed message passing programs. CMW simplifies reasoning for correctness because of its serial nature. Its application to two ambiguous seeding scenarios was demonstrated successfully. **MPK:** *We plan to extend the application of this method to problems involving n-halo partitions and partitions with periodic boundaries.* **:KPM** **JDM:** *I thought we included halo partitioning here already?* **:MDJ** .

7. Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642959.

References

[1] T.A. Albring, M. Sagebaum, and N.R. Gauger, *Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework*, in *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Jun., American Institute of Aeronautics and Astronautics, 2015. 2

[2] B. Christianson, *Reverse Accumulation and Attractive Fixed Points*, Optimization Methods and Software 3 (1994), pp. 311–326. 15

[3] L. Clarke, I. Glendinning, and R. Hempel, *The MPI Message Passing Interface Standard*, in *Programming environments for massively parallel distributed systems*, Springer, 1994, pp. 213–218. 12

[4] T.D. Economon, F. Palacios, S.R. Copeland, T.W. Lukaczyk, and J.J. Alonso, *SU2: An Open-Source Suite for Multiphysics Simulation and Design*, AIAA Journal 54 (2016), pp. 828–846. 3

[5] L.Y. Gicquel, N. Gourdain, J.F. Boussuge, H. Deniau, G. Staffelbach, P. Wolf, and T. Poinsot, *High Performance Parallel Computing of Flows in Complex Geometries*, Comptes Rendus Mécanique 339 (2011), pp. 104–124. 3

[6] R. Giering and T. Kaminski, *Recipes for Adjoint Code Construction*, ACM Transactions on Mathematical Software 24 (1998), pp. 437–474. 15

[7] A. Griewank and A. Walther, *Evaluating Derivatives*, Society for Industrial and Applied Mathematics, 2008 Jan. 2

[8] L. Hascoët and V. Pascual, *The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification*, ACM Transactions on Mathematical Software 39 (2013), pp. 1–43. 12

[9] P. Heimbach, C. Hill, and R. Giering, *Automatic Generation of Efficient Adjoint Code for a Parallel Navier-Stokes Solver*, in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2002, pp. 1019–1028. 2

[10] P. Hovland, *Automatic Differentiation of Parallel Programs*, Ph.D. diss., University of Illinois at Urbana-Champaign, 1997. 2, 5, 18

[11] P. Hovland and C. Bischof, *Automatic Differentiation for Message-passing Parallel Programs*, in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, IEEE Comput. Soc, 1998, pp. 98–104. 2

[12] H. Jasak, *Handling Parallelisation in OpenFOAM*, in *Cyprus Advanced HPC Workshop*, Vol. 101, 2012. 3

[13] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on scientific Computing 20 (1998), pp. 359–392. 12

[14] P. Mohanamuraly, J.C. Hückelheim, and J.D. Müller, *Hybrid parallelisation of an algorithmically differentiated adjoint solver*, in *ECCOMAS Congress 2016*, 2016. 12

[15] P. Mohanamuraly, J.C. Hückelheim, and J.D. Müller, *STAMPS: An Efficient Hybrid-parallel Discrete-Adjoint CFD Solver for Aerodynamic Design*, in *EUROGEN 2017, ECCOMACS thematic conference*, Sep., Madrid, Spain, 2017. 3, 12

[16] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, *OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures*, in *Innovative Parallel Computing (InPar 2012)*, May, IEEE, 2012, pp. 1–12. 3

[17] P. Roe, *Approximate Riemann solvers, parameter vectors, and difference schemes*, Journal of Computational Physics 43 (1981), pp. 357–372. 17

[18] M. Sagebaum, T.A. Albring, and N.R. Gauger, *High-Performance Derivative Computations using CoDiPack* . 2

[19] M. Schanen, *Semantics Driven Adjoints of the Message Passing Interface*, Ph.D. diss., RWTH Aachen University, Germany, 2016. 2

[20] M. Schanen and U. Naumann, *A Wish List for Efficient Adjoints of One-Sided MPI Communication*, in *Recent Advances in the Message Passing Interface. EuroMPI 2012. Lecture Notes in Computer Science, vol 7490*, T. J.L., B. S., and D. J.J., eds., Springer Berlin Heidelberg, 2012, pp. 248–257. 2, 5

[21] M. Schanen, U. Naumann, L. Hascoët, and J. Utke, *Interpretative Adjoints for Numerical Simulation Codes using MPI*, Procedia Computer Science 1 (2010), pp. 1825–1833. 2

[22] J. Utke and E. Larour, *What I did to Adol-C and ISSM?*, in *15th EuroAD workshop, Oxford, UK*, 2013. 2

[23] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann, *Toward Adjoinable MPI*, in *2009 IEEE International Symposium on Parallel & Distributed Processing*, May, IEEE, 2009, pp. 1–8. 1, 2, 5

[24] B. van Leer, *Towards the Ultimate Conservative Difference Scheme*, Journal of Computational Physics 135 (1997), pp. 229–248. 17

[25] S. Xu, D. Radford, M. Meyer, and J.D. Müller, *Stabilisation of Discrete Steady Adjoint Solvers*, Journal of Computational Physics 299 (2015), pp. 175–195. 13