First improvements toward a reproducible Telemac-2D

Rafife Nheili
and Philippe Langlois
Univ. Perpignan Via Domitia
Digits, Architectures et Logiciels Informatiques,
Perpignan. France.
Univ. Montpellier, Laboratoire d'Informatique,
Robotique et de Microélectronique de Montpellier,
UMR 5506, Montpellier. CNRS. France.
Email: {rafife.nheili, langlois}@univ-perp.fr

Christophe Denis
ENS Cachan,
CMLA Research Center for Applied Maths,
Cachan. France.
Email: Christophe.Denis@cmla.ens-cachan.fr

***Abstract*—OpenTelemac suffers from numerical reproducibility failures. In parallel simulations, the domain distribution toward units computing with floating-point arithmetic may yield different numerical results. Numerical reproducibility is a requested feature to facilitate the debug, the validation and the test of industrial or large codes. We present how to apply compensation techniques to recover reproducibility in the finite element computation of a hydrodynamics simulation. Compensation is used in both the building and the resolution phases of the linear system which are not reproducible in the current version of the software. Here the building step relies on the element-by-element storage mode and the solving step applies the conjugated gradient algorithm. We also measure that the running time extra-cost of the reproducible version is reasonable enough in practice.**

I. Introduction

The openTelemac suite is a HPC code for the simulation at the industrial scale of free surface flows in 1D-2D-3D hydrodynamics. It is an integrated set of open source Fortran 90 modules developped since 20 years of international collaboration [11]. Like most large and complex applications, openTelemac introduces parallelism. The domain is distributed toward several processors which solve simultaneously their own sub-domains. In practice, the processors are not independent of each other, but each one needs the results from the others. This technology reduces the processing time since ideally using p processors divides by p the time to solve the same problem with only one processor.

An undesirable consequence of parallelism reported in openTelemac is the non-reproducibility of the results. The ability to reproduce the simulation results becomes a crucial property to improve the confidence in large scale numerical experiments. The changes of a simulation results must depend only on the changes of the simulations inputs, and not be accidentally affected by uncontrolled floating-point calculations. The non-deterministic propagation of the rounding errors and the dynamic reductions of parallel executions may yield to different outputs. Indeed, numerical reproducibility is a requested feature to facilitate the debugging and the testing of the code: it is not obvious to fix a bug nor to test a code when the results differ from one run to another. Moreover, critical simulations should verify this reproducibility to satisfy legal agreements.

We studied a schematic test case, called gouttedo, which is a 2D-simulation of a water drop fall in a square basin. The test case specificities are the EBE storage matrix, the uses of the wave equation system and of the conjugate gradient for the solving phase. This resolution runs for a triangular element mesh (8978 elements, 4624 nodes) and simulates several time steps of 0.2 sec. Figure 1 illustrates the gouttedo test case where we display the non reproducible behavior of the water depth simulation between the sequential and the 2 processor runs. The left plot shows the water depth values returned by the sequential simulation and the right one corresponds to the $p = 2$ parallel run. White spots exhibit the mesh elements where these latter results differ from the sequential ones.

The first task is to carefully identify the sources which produce the reproducibility failure and then to apply as few as possible modifications to limit their extra-cost. The difficulty in this work was to identify these sources. Figure 2 illustrates the main aspects we need to consider. First row shows how the results of a code, in different colors, are not reproducible when varying the number of processors. A convincing modified reproducible code should satisfied two criteria:
i) bit-wise *identical* result for every p-parallel run and for every $p \geq 1$, as showed in the second row in Figure 2 where the results share the same color. Reproducibility is measured as the relative error between the modified sequential simulation and the parallel ones (in green).
ii) the reproducible results must be within a reasonable range of differences compared to the original sequential simulation ones. This measure is important for the code developers who, in practice, trust their sequential results as a reference. The measure of the accuracy is the relative error between the original sequential simulation and the modified one (in red). The paper is organized as follows. In Section II we introduce floating-point arithmetic to explain the reasons of the non-reproducibility in parallel executions. In Section III, we summarize the compensation algorithms which are used in Telemac to recover the reproducibility. The sources

Figure 1: gouttedo: white spots are non reproducible water depth values between the sequential (left) and a 2 processors run (right). Time steps: 1, 2, ..., 7, 8.

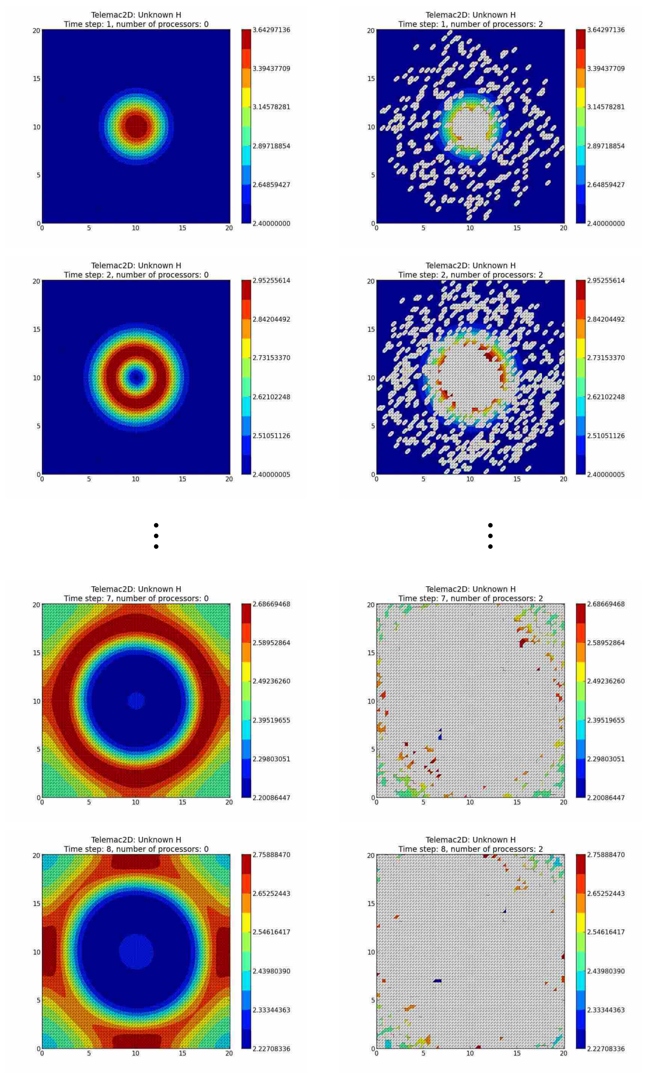
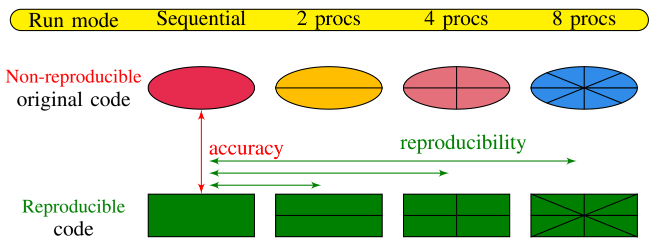


Figure 2: The two measures of a convincing modified reproducible code



identification of the non-reproducibility and how we modify them by using compensation are detailed in Section IV. Section V presents the reproducible simulation. Finally the running time extra-cost of these modifications is analyzed in Section VI.

II. The sources of numerical non-reproducibility

A. *Floating-point arithmetic features*

Computer memory is limited so it cannot store the infinite precision of real numbers. The floating-point (FP) numbers are approximate real numbers. The IEEE-754 standard [5] defines the most common floating-point representation and the behavior of their arithmetic operations. It also defines the data formats, conversion rules, some special values, the rounding modes and the accuracy of basic operations. This standard aims to obtain predictable and portable programs which produce identical results when running on different machines.

1) Floating-point representation: It is based on the scientific notation to represent a floating-point number x as:

$$x = (-1)^s.m.\beta^e, \tag{1}$$

where $s \in \{0, 1\}$. The mantissa m is a string of integer digits which depends on the radix $\beta > 1$ ($0 \leq m_i < \beta$) and e (represented by w bits) acts as a scaling factor for floating-point number x. The number of mantissa digits is the precision of the floating-point number, denoted t. Figure 3 details each component for the binary case ($\beta = 2$).

Figure 3: Binary FP representation

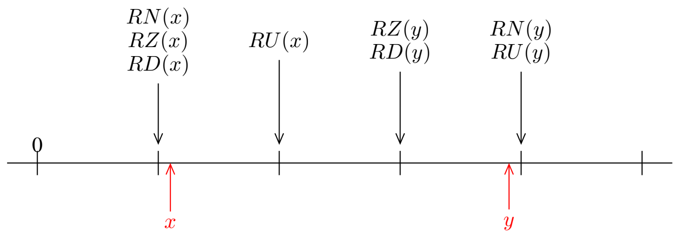
sign	exponent	mantissa
$s \in \{0, 1\}$	$e_{min} \leq e \leq e_{max} \in \mathbb{Z}$	$m = 1.m_1m_2 \dots m_{t-1}; 0 \leq m_i < 2$
1 bit	w bits	t bits

The IEEE-754 standard defines different precision where the most used ones are the binary single precision represented with 32 bits, and the binary double precision represented with 64 bits [5]. They correspond, respectively, to $(w, t) = (8, 24)$ or $(11, 53)$.

2) Rounding function: The rounding function, denoted $\circ$, applies to floating-point numbers and their operations. It is needed to represent one number or one operation result which can not be exactly represented by a floating-point number, e.g. the constants π or $1/10$. The rounded operation replaces these values by an approximating floating-point value. For a non representable number $x \in \mathbb{R}$, $\circ(x) = \hat{x}$ denotes the floating-point number $\hat{x} \in \mathbb{F}$ resulting from the rounding.

The IEEE-754 standard defines how any numerical value is rounded to a floating-point number by introducing several rounding modes. There are illustrate in Figure 4, where we distinguish the rounding to nearest (RN) which is the default rounding one, rounding toward $-\infty$ (RD), rounding toward $+\infty$ (RU) and rounding toward 0 (RZ). The standard forces the *correct* rounding for the basic operations (addition, subtraction, multiplication, division and square root). "Correct rounding" means that the returned result is computed as an infinitely precise result then it is rounded to the floating-point number, according to the chosen rounding mode. At each rounding, you lose a priori some accuracy which corresponds to the rounding error. This rounding error is bounded by the arithmetic precision. It is classic to denote $\mathbf{u}$ this working precision that verifies $\mathbf{u} = 2^{-t}$ for the RN rounding mode and $\mathbf{u} = 2^{1-t}$ for the other mode.

Figure 4: The rounding modes, $x, y > 0$.



3) Relative errors: Whatever we work about accuracy or reproducibility of floating-point operation, we are interested in these rounding errors and more generally in approximation errors. Let $\hat{x} \in \mathbb{F}$ be the rounded value of $x \in \mathbb{R}$, the relative error is:

$$Err_{rel} = \frac{|\hat{x} - x|}{|x|}, \quad \text{if} \quad x \neq 0. \tag{2}$$

When a result is compared to the exact value x, Relation 2 measures the accuracy of the result $\hat{x}$. In our simulations we do not know the exact result. In this quest for reproducibility, we will compare the value $\hat{x}$ to a reference one, which is the sequential result, as explained in Figure 2.
The major problem of floating-point computation is the rounding errors propagation which occurs within a sequence of computations. Although one isolated operation returns the best possible result (the relative error is bounded by $\mathbf{u}$), a sequence of calculations may lead to significant errors due to the accumulation of every single rounding error. It is well known that a final result of several operations may be far from the exact value, see [9] for details and numerous entries on the subject.

4) Non-associativity of floating-point arithmetic: Because of the rounding errors, the floating-point addition is not associative. So change in the addition order may produce different results: $(a + b) + c \neq a + (b + c)$. For instance, $\circ((-1+1)+\mathbf{u}) = \mathbf{u}$, which differs from $\circ((1+\mathbf{u})-1) = 0$. This phenomena is a consequence of the limited precision and range of the IEEE floating-point representation. Hence, depending on the computation order, the propagation of rounding errors differs to yield a different result.

B. Parallel computation and numerical reproducibility

Parallel computation is introduced by domain decomposition where each computing unit solve its own sub-domain. The sub-domains compute their local contributions, exchange and summed them to obtain the global value of the whole domain. In this procedure two causes may lead to non-reproducible results.
Firstly when the sub-domain number varies, the computation of the local contributions differs due to the different rounding errors propagation. In this case, even if the global value is summed in a static order (as it is the case in openTelemavc's `paraco`), it will differ when the number of sub-domain changes whereas being reproducible for successive runs with a fixed number of sub-domains.
Secondly, parallel computation may introduce collective communications where the arrival order of the local contributions differs because of dynamic scheduling or resource sharing for instance. That leads to different order of the computations and due to the non-associativity of the floating-point arithmetic, results are not reproducible. In this case, there is non-reproducibility even when the number of sub-domains does not change.

III. Compensation technique

Compensation is a way to increase the accuracy of results. We apply compensation techniques to recover the reproducibility in our context. The principle of this technique is to use error-free transformation (EFT) which computes the rounding errors generated by the elementary operations in floating point arithmetic. Many compensation algorithms have been developed using these transformations to compute the sum or the dot product of floating-point vectors [10].
The principle in these transformations is that, for an elementary operation $op \in \{+, -, *\}$ of two floating-point numbers $\hat{a}$ and $\hat{b}$, there are two floating-point numbers $\hat{x}, \hat{y}$ that verify:

$$\hat{a}\ op\ \hat{b} = \hat{x} + \hat{y}. \tag{3}$$

Here $\hat{x} = \circ(\hat{a}\ op\ \hat{b})$ is the rounded part of the result and $\hat{y}$ is the generated rounding error. It can also be respectively named the high-order and low-order parts of the result.

Now we present the EFT and the compensation algorithms that are useful in our context.
Algorithm 1. 2Sum is one EFT of the addition proposed by Knuth [6] in 1969. It computes the high (x) and the low (y) parts of the sum of two floating-point numbers a and b, in $6flop$.

Algorithm 1 [x,y]=2Sum(a,b)

```
$x = RN(a + b)$
$a' = RN(x - b)$
$b' = RN(x - a')$
$\delta_a = RN(a - a')$
$\delta_b = RN(b - b')$
$y = RN(\delta_a + \delta_b)$
```

Algorithm 2. 2Product is one EFT of the multiplication introduced by Dekker in 1971 [1]. It starts by calling the Split algorithm proposed by Veltkamp [9]. This algorithm splits the inputs a and b into their high and lower parts, respectively a_h, b_h and a_l, b_l. The 2Product algorithm returns the two floating-point numbers $x = RN(a \times b)$ and the generated error y. It requires $17flop$. Another EFT for the product exists when a fused multiply and accumulate operator (FMA) is available [9].

Algorithm 2 [x,y]=2Product(a,b)

```
$[a_h, a_l] = \text{Split}(a)$
$[b_h, b_l] = \text{Split}(b)$
$x = RN(a \times b)$
$t_1 = RN(-r_1 + RN(a_h \times b_h))$
$t_2 = RN(t_1 + RN(a_h \times b_l))$
$t_3 = RN(t2 + RN(a_l \times b_h))$
$y = RN(t_3 + RN(a_l \times b_l))$
```

Algorithm 3. Sum2 is the compensation algorithm which approximates the sum of a vector using Algorithm 1. It was proposed by Rump, Ogita and Oishi in 2005 [10]. The result res of this algorithm is as accurate as if it is computed in twice the working precision and finally rounded to the working precision. It requires $7(n-1) flop$.

Algorithm 3 res = Sum2(a)

```
$s_1 = a_1, \sigma_1 = 0$
**for** i=2 **to** n **do**
    $[s_i, q_i]$ =2Sum$(s_{i-1}, a_i)$
    $\sigma_i = RN(\sigma_{i-1} + q_i)$
**end for**
$res = RN(s_n + \sigma_n)$
```

Algorithm 4. Dot2 is a compensated dot product, that uses Algorithms 1 and 2 to compute a twice more accurate result. It requires $25n - 7 flop$.

Algorithm 4 res = Dot2(a,b)

```
$[r, \epsilon]$ =2Product$(a_1, b_1)$
**for** i = 2 **to** n **do**
    $[p, \pi]$ =2Product$(a_i, b_i)$
    $[r, \sigma]$ =2Sum$(r, p)$
    $\epsilon = RN(\epsilon + RN(\sigma + \pi))$
**end for**
$res = RN(r + \epsilon)$
```

Sum2 and Dot2, are almost maximally accurate while their conditioning remains smaller than $1/\mathbf{u}$. The condition numbers for the sum and the dot product are respectively, $\sum |a_i| / |\sum a_i|$ and $\sum |a_i \cdot b_i| / |\sum a_i \cdot b_i|$. When this conditioning is larger than $1/\mathbf{u}$, the computation is ill-conditioned and one needs more than twice the working precision to remain accurate. Another levels of compensation can be applied, see [10] for instance.

IV. Necessary modifications to recover the reproducibility

We recover the numerical reproducibility for the gouttedo test case of Telemac-2D using the compensated algorithms presented in Section III. OpenTelemac applies the finite element library BIEF. This one includes many subroutines in Fortran 90 which provide the data structure and the subroutines to the building and the solving phases of the simulation. Almost all our subroutine modifications have been restricted to this library. We now describe 4 types of modifications: data structure, algebraic operations, building phase and solving phase. The unknowns at each node of the domain mesh are the depth of water (H) and the two velocity components (U, V). In the Introduction, it was reported that these results are not reproducible but the sources of this issue are unknown. Finite element method leads to build and solve a general sparse linear system. The strategy is to observe each component of the linear as the computation is performed. This system mixes the three sub-systems related to H, U, V. The sub-system components are computed from physic algebraic equations by taken into account all the physical condition inputs. In the case of the pseudo wave equation, the mixed system is simplified by eliminating the velocity from the continuity equation at the discrete level, more details of these transformations are detailled in [8]. So we obtain the decoupled system :

$$\begin{pmatrix} A_1 & 0 & 0 \\ 0 & A_2 & 0 \\ 0 & 0 & A_3 \end{pmatrix} \begin{pmatrix} H \\ U \\ V \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}, \tag{4}$$

where A_2 and A_3 become diagonal matrices thanks to the lumped-mass method. The matrix A_1 and the three second members are computed by algebraic transformations defined in [3]. It is important to note here that the two velocity second members C_2 and C_3 still depend on the H unknown. Hence, this procedure mixed the building and the resolution phases. System (4) is solved in two steps: H is first computed applying the conjugate gradient method to $A_1 H = C_1$. Then C_2 and C_3 derive from H, then the diagonal systems with A_2 and A_3 are solved to yield U and V.

In the following we distinguish the modified parts of the openTelemac code highlighting it on pink. The users choose the two following keywords in the test case file. The desired number of processors is defined by the keyword 'PARALLEL PROCESSORS' in English (or 'PROCESSEURS PARALLELES' in French). This correspond to the Fortran variable `NCSIZE` that takes value 0 for a sequential execution and p for a parallel one. The original computation or the reproducible ones is defined by the keyword "FINITE ELEMENT ASSEMBLY" in English (or "ASSEMBLAGE EN ELEMENTS FINIS" in French). This correspond to the Fortran variable `MODASS` that takes the values 1,2,3 respectively for the original, integer, compensated modes.

A. *Modifications in the data structure*

The main type in the BIEF library is `BIEF_OBJ` which may be a vector, a matrix or a block. This type contains many components that define the data, the size, the name *etc.* We write `V%R` the R component of the vector V which corresponds to the data. In the compensated version, these R values will be accompanied, when necessary, with the accumulation of their generated rounding errors. These errors will be stored in a component named E and we write `V%E` to access to them. The same way applies for the diagonal D of the matrix M: we write `M%D%R` for data and `M%D%E` for errors, *etc.*

Note: the code is not optimized to these modifications because the subroutine inputs/outputs are not always a `BIEF_OBJ` type, but sometimes double precision vectors. In other words, if the parameters of the subroutines were always of the `BIEF_OBJ` type, all the structure components would be accessible as `V%R` and `V%E`. But when the subroutine parameters are double precision vectors which refer to `V%R`, we had to manually add supplementary input/output parameter that refers to `V%E`.

B. *Modifications in the building phase*

The steps of the building phase which condition the reproducibility are the finite element assembly and its complement step in parallel, the interface node assembly. The finite element assembly is the main step that builds the linear system. It recovers the finite elements values to express them on the nodal values computing:

$$V(i) = \sum_{ielem=1}^{nelem} W_{ielem}(i). \tag{5}$$

This process builds the global vector V of size $npoin$ by accumulating the elementary contributions W_{ielem} for every element $ielem$ in the mesh that contains the node i. This assembly is applied to the elementary vector, the diagonal of any matrix and in the EBE matrix-vector product process, see Section IV-D.
The domain decomposition in a parallel resolution introduces inner and interface mesh nodes. The latter ones belong to a common boundary between several sub-domains and are shared between several computing units. The interface node assembly is one of the main significant differences between the sequential and the parallel resolutions. It significantly affects the numerical reproducibility by a non-deterministic rounding errors propagation. Let V be an arbitrary vector extracted from the linear system, and let i an interface node that belongs to k sub-domains. $V^{d_k}(i)$ is one of the contribution of V in the sub-domain d_k at the interface node i (the computation of V^{d_k} only includes quantities related to d_k). Communications between the sub-domains $d_1, \ldots, d_k$ yield the final value $V(i)$ at every interface node i as the following reduction:

$$V(i) = \sum_{\text{sub-domains } d_k} V^{d_k}(i). \tag{6}$$

Accumulation (5) has the same order with respect to $ielem$ for inner nodes in the sequential and the parallel cases. Nevertheless, a given inner node i may become one interface node in another domain decomposition, *i.e.* when the number p of computing units varies. Hence this type of nodes suffers from a non-deterministic error propagation. // In practice, the computation of the vectors V or the diagonal of the matrix D is performed respectively in the subroutines `vectos` or `matrix`, which call the subroutine `assvec` to compute the finite element assembly process, *i.e.* Relation (5). In the compensated mode, the generated rounding errors of the assembly are calculated by subroutine 2Sum (Algorithm 1) which is added in BIEF. From Relation (5) we derive an assembly which now computes the rounding error of each node i:

$$[V(i), E_V(i)] = \text{ReprodAss}_{ielem=1,\ldots,nelem} W_{ielem}(i), \tag{7}$$

where:

$$(V(i), e_i) \quad = \quad \text{2Sum}(V(i), W_{el}(i)),$$

and $E_V(i)$ accumulates the errors e_i for all elements that include the node i.
In the subroutine `assvec` (Listing 1) this pair is accumulated with the errors generated by the assembly for each node $ile(ielem, idp)$.
We continue the modifications with the same idea to provide a reproducible interface point assembly. In practice this assembly is produced in the subroutine `paraco` which is called at several steps in the computation. It becomes `paraco_comp` in the compensated mode. In the original mode, the assembly of the sub-domains contributions (Relation 6) is an accumulation of the received data. In the compensated one, each sub-domain receives a pair [data, error] from every neighbor sub-domain.

Listing 1 FE assembly in `assvec`

```
DO IDP = 1 , NDP
 DO IELEM = 1 , NELEM
  IF (MODASS.EQ.1)
&   X(IKLE(IELEM,IDP)=X(IKLE(IELEM,IDP)
&   +W(IELEM,IDP)
  ELSEIF (MODASS.EQ.3) THEN
   CALL 2SUM(X(IKLE(IELEM,IDP)),
&   W(IELEM,IDP),X(IKLE(IELEM,IDP)),ERROR)
   ERRX(IKLE(IELEM,IDP))= ERRX(IKLE(IELEM,IDP))
&   +ERROR
  ENDIF
 ENDDO
ENDDO
```

Then the assembly uses 2Sum to compute the rounding errors of the data accumulation. These generated rounding errors are accumulated and represent error contributions of the sub-domain. For all sub-domains d_k that share the interface node i, we compute:

$$[V(i), E_V(i)] = \bigoplus_{\text{sub-domains } d_k} [V^{d_k}(i), E_V^{d_k}(i)], \tag{8}$$

which derives from (6) using again the 2Sum error-free transformation for every d_k, as follows:

$$\begin{aligned} (V(i), e_k) &= \text{2Sum}(V(i), V^{d_k}(i)), && (9)\\ E_V(i) &= E_V(i) + E_V^{d_k}(i) + e_k. && (10) \end{aligned}$$

Step (9) accumulates $V(i)$ and computes the generated rounding error e_k. Step (10) accumulates in $E_V(i)$ this e_k and the previous errors $E_V^{d_k}(i)$. Finally, compensation occurs after the last reduction of every interface node i to yield the whole vector V as:

$$V + E_V. \tag{11}$$

We stress that this compensation applies once to the vector of inner and interface nodes after the end of the interface node assembly (8). After this step, the compensated vector becomes reproducible.
We note that this procedure is applied to every vector and also for matrix the diagonal `M%D%R` which is a vector for the EBE storage: its accompanying error term `M%D%E` is calculated in a similar way.

C. Modifications in the algebraic operations

The rounding errors `V%E` must be updated for each algebraic operation on `V%R`. Each operation on block or vector is called by the subroutine `os` which only verifies the structure validation before calling the concerned subroutine `ov`. The latter computes the requested operation *op* for the passed vectors `X%R`, `Y%R`, `Z%R`. In the compensated mode, the new subroutine `ov_comp` is called: the data vectors are accompanied with their own error vectors `X%E`, `Y%E`, `Z%E` to also update them. Listing 2 contains some of these modified operations.

D. Modifications in the solving phase

In the gouttedo test case, the resolution phase applies the conjugate gradient (subroutine `gracjg`). The modifications impact the computations of the scalar product in function

Listing 2 Algebraic operations in `ov_comp`

```
!X,Y and Z correspond to the vector value
!!X_ERR,Y_ERR and Z_ERR correspond to the
vector errors
!For initialization
CASE('0  ')
DO I=1,NPOIN
  X(I) = 0.D0
  X_ERR(I)=0.D0
ENDDO
!For copy y to x
CASE('Y  ')
DO I=1,NPOIN
  X(I) = Y(I)
  X_ERR(I) = Y_ERR(I)
ENDDO
!For addition of two vectors
!In the original code: X(I) = Y(I) + Z(I)
DO I=1,NPOIN
  CALL TWOSUM(Y(I),Z(I),X(I),ERROR)
  X_ERR(I)=(Y_ERR(I)+Z_ERR(I))+ERROR
ENDDO
!For value by value vectors product
!In the original code: X(I) = Y(I) * Z(I)
DO I=1,NPOIN
 CALL TWOPROD(Y(I),Z(I),X(I),ERROR)
 X_ERR(I)=(Y(I) * Z_ERR(I))+(Y_ERR(I) * Z(I))
&        +(Y_ERR(I) * Z_ERR(I))
 X_ERR(I)=X_ERR(I)+ ERROR
ENDDO
```

`p_dots` and the matrix-vector product in subroutine `matrbl`, which are called by `gracjg`. We now describe these two modifications.

i) The scalar product $X \cdot Y$
According to the computation mode, the corresponding scalar product is called. In the original mode, the dot product of the whole domain is computed partially by each sub-domain, then the partial contributions are summed over all the sub-domains to compute the global scalar product. This reduction is computed by the MPI dynamic reduction which proceeds with an non-deterministic order. So for a given input results may differ. In the compensated mode, a twice more accurate scalar product is computed. In sequential, Dot2 (Algorithm 4) computes a such accurate sequential dot product. It accumulates both the dot product and the generated rounding errors (addition and multiplication) and finally compensates them together. In the parallel implementation, each sub-domain computes its local scalar product and the corresponding generated rounding errors. Hence a pair [data, error] is returned by `pdot2`. These local pairs are exchanged by the processors via MPI_ALLGATHER and are accurately accumulated by Sum2 (Algorithm 3) in every computing unit, see Listing 3.

ii) The matrix-vector product $M \times V$
The EBE storage and the EBE matrix-vector product are detailed in [3]. Matrix M is stored as `M%D` for its diagonal terms and `M%X` for its extra-diagonal ones. The result RES of the product $M \times V$, of size $npoin$, satisfies the following

Listing 3 The final sum on all the sub-domains

```
!In original version, MYPART is a scalar
!CALL MPI_ALLREDUCE(MYPART,P_DSUM,1,
MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD
,IER)
!In compensated version, MYPART is a pair
of scalar
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NUM_PROCS,IER)
ALLOCATE(ALL_PARTIAL_SUM(1:2*NUM_PROCS))
ALL_PARTIAL_SUM=0.D0
CALL MPI_ALLGATHER (MYPART,2,
MPI_DOUBLE_PRECISION,ALL_PARTIAL_SUM,2,
MPI_DOUBLE_PRECISION,MPI_COMM_WORLD,IER)
CALL SUM2(2*NUM_PROCS,ALL_PARTIAL_SUM,P_DSUMERR)
DEALLOCATE(ALL_PARTIAL_SUM)
```

equality:

$$RES = D.V + \sum_{ielem=1}^{nelem} X_{ielem}.V_{ielem}. \qquad (12)$$

The first operand is a vector R_1 (of size $npoin$) resulting from the Hadamard product $R_1(i) = D(i) \times V(i)$. The second operand is calculated in two steps:

1) firstly the multiplication of the extra-diagonal terms and the vector V_{ielem} via a mapping with the connectivity table from the global numbering i and the local ones $i_1, i_2, i_3 \in ielem$.
2) Secondly each term of the previous operation has to be assembled to a global vector R_2 of size $npoin$, in the corresponding node i, by the FE assembly step, *i.e.* applying Relation (5).

Finally, the two vectors are added to obtain the final result of the matrix-vector product: $RES = R_1 + R_2$.

In the compensated mode, this procedure is modified to also take care of the accompanied errors `M%D%E` and `V%E`, previously computed. The following modifications of the product (12) produce a reproducible $M \times V$ product because it ends with one interface point assembly step of the result. The diagonal part D is now associated with its errors E_D, so we compute the pair $[DV, E_D] \times V$. The finite element assembly in Relation (12) is now computed with the modified one (8). This updates the couple $[MV, E_{MV}]$ which is, finally, assembled on the interface nodes with Relation (8). The final step is the compensation $MV + E_{MV}$. So all the $M \times V$ products in the conjugate gradient iterations are now reproducible.

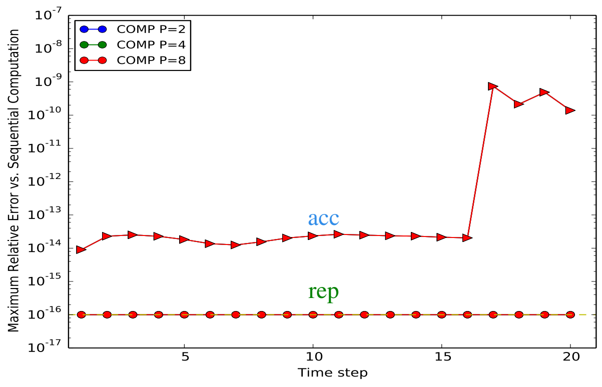
By achieving a reproducible EBE matrix-vector product and a reproducible dot product, the output H of the conjugate gradient becomes reproducible. It is important to note that there are still rounding errors in the conjugate gradient (generated by the divisions and the other operations) but they remain similar in both the sequential and the parallel executions.
Recovering the reproducibility of the last two sub-systems U and V is now straightforward. The U and V diagonal subsystems depend on H. The second members C_2 and C_3 are built (from H) and are assembled at the interface nodes before the resolution. Reproducible members A_2, C_2, A_3 and C_3 lead

to the reproducible diagonal resolution of the U and V subsystems.

V. Reproducible results

Thanks to the previous modifications that rely on compensated techniques the results of the `gouttedo` test case are now reproducible.
Figure 5 displays two measures corresponding to Figure 2. The rep plot is the maximum relative error over the whole domain between the compensated parallel simulations and the compensated sequential one. The number of processors varies (p = 2, 4, 8) but all plots are superposed and constant at the precision level. This exhibits the reproducibility of the compensated simulations. The second plot acc displays the maximum relative difference between the original sequential Telemac-2D simulation and the compensated ones. Relative differences varies from 10^{-14} to 10^{-10}. This validates the compensated simulations that are very similar to the original sequential Telemac-2D simulation. As already mentioned, this latter is considered by the openTelemac developers as the reference simulation. Nevertheless since compensation provides more accuracy than the working precision computation, this curve certainly displays the increase of accuracy produced by the compensation.

Figure 5: Reproducibility (rep) of the compensated simulation and accuracy (rep) compared to the original Telemac-2D for the water depth in gouttedo. X-axis: time steps ($1 \ldots 20 \times 0.2$ sec). Y-axis: maximum relative difference. Number of processors: p = 2, 4, 8.

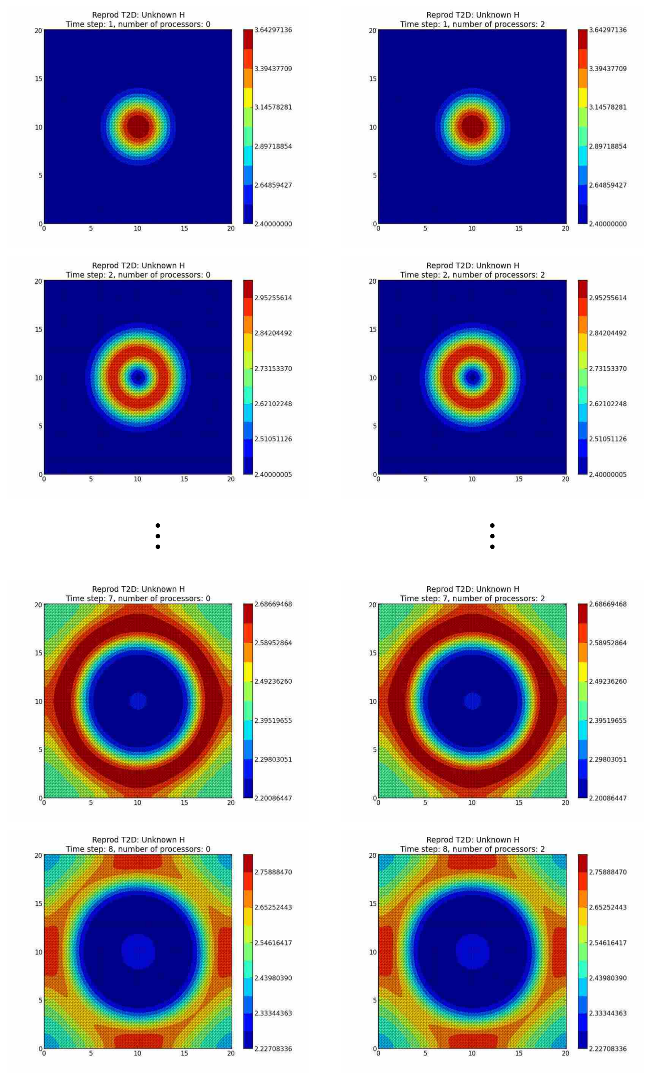


VI. Extra-cost reproducibility

In our context, we measure the extra-cost of the modifications that provide reproducible results compared to the original code. An appropriated method is to repeat the timing measurement several times and then to report the minimal.

We measure the running time extra-cost in cycles, on the version v7.2 of openTelemac. This measure are performed with the hardware counter, the Read-Time-Stamp Counter (RDTSC) instruction. For the parallel measurement, we synchronize the processors to be sure that we measure the minimal time between the processors, see Listing 4. In openTelemac and

Figure 6: gouttedo: Numerical reproducibility, no more white spots for the water depth values between the sequential (left) and a 2 processors run (right).
Time steps: 1, 2, . . . , 7, 8.



Listing 4 Synchronization of the processors in the performance test

```
! Beginning of the measure
  MPI_BARRIER()
  CALL RDTSC(Begin_Timer)
  !Time loop calculation...
  MPI_BARRIER()
  CALL RDTSC(End_Timer)
! End of the measure
  Total_Time = End_Timer - Begin_Timer
```

generally in any simulation, there are a lot of read/write data at both the pre- and post- simulation steps. It is not significant to measure the whole simulation: the extra-cost of the modifications is negligible beside the complete simulation cost. We only measure the modified parts compared to the original ones, which are both the building and the solving

phases. So we measure from the begin to the end of the time loop that includes all the modifications and avoid to take into account the large amount of read/write process.
We compare 3 different meshes with 4624, 18225 and 72361 nodes to exhibit how the extra-cost depends on the magnitude of the problem. Table I presents the significant increasing number of interface nodes in these 3 meshes that also introduces a more important communication cost.

Table I: Number of interface nodes in 3 meshes when the number of computing units varies.

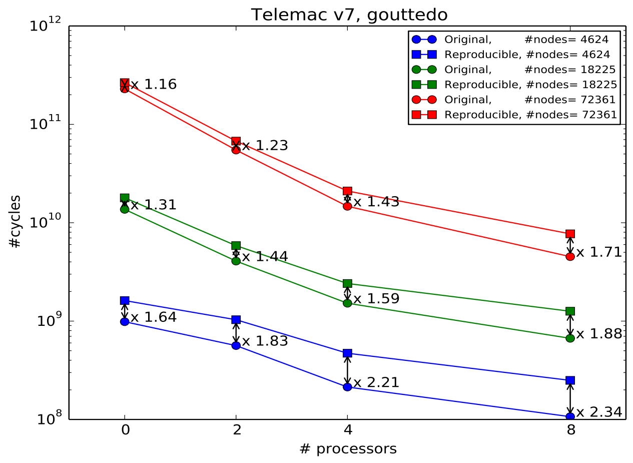
#IP		#nodes		
		4624 nodes	18225 nodes	72361 nodes
#procs	2 procs	72	143	280
	4 procs	304	674	1368
	8 procs	501	1152	2020

Figure 7 presents the running time (y-axis) and their ratios of the compensated version running time *vs.* the original version for a given time step and when varying the number of processors (x-axis). We remark that compensated algorithms double (more or less) the time of the core calculations. Cycles of the original code are represented as circles and the compensated ones as squares.
The simulation time increases as the number of mesh nodes because of the extra-computations of the construction and resolution of a larger system. In addition, the number of iterations of the conjugate gradient significantly increases depending on the number of the system unknowns. It is interesting to wrote that the extra-cost for reproducibility benefits from this time increase since our modifications impact the performance of the core simulation.
At the contrary, for a given mesh size, the ratio is larger when increasing the number of processors. This is due to the augmentation of the interface node number and to their extra-cost treatment in the compensated version, (Relation 8).

VII. Conclusion

We have presented how to recover the numerical reproducibility of the Telemac-2D simulation for the gouttedo test case using compensation techniques. The difficulties were to identify the sources of this non-reproducibility, *i.e.* where the rounding errors differ between the sequential and the parallel simulations, and to distinguish their implementations in this huge code. It was inevitable to manipulate three openTelemac components: the Bief library, the parallel library and Telemac-2D module which include respectively 493, 46 and 192 subroutines. The modifications to obtain reproducibility were restricted to about 30 subroutines, mostly in BIEF. The first source is the non-deterministic error propagation at the interfaces nodes. We recall that this step is implicitly present in several parts of the computation (building and solving phases). It is sufficient to store and propagate these errors and finally compensate them into the computed value after every step of interface node assembly. These corrections are applied for both the parallel and the sequential simulations to yield the expected reproducibility. The second source is the dynamic reduction of the parallel implementation for the dot product in the conjugate gradient iterations. It is corrected by implementing a dot product that computes in about twice the working precision.

Figure 7: Extra-cost running time and ratios of the compensated computation compared to the floating-point one, for the test case gouttedo in Telemac-2D. (Mesh size increases from bottom to top.)



This approach is reasonable in term of running time extra-cost. We measured no significant extra-cost of the whole reproducible simulation compared to the original one (when the read/write data process are considered). Of course, as the computation core part takes about twice more time in the reproducible version, the extra-cost could be of the same order for larger cases.
The feasibility and the efficiency of the compensation have been compared to other solutions like integer conversion and reproducible sums [2]. These three techniques were applied to the *Nice* test case of the Tomawac module where the non-reproducibility source is only the finite element assembly step. The compensated solution appeared to be the more efficient one [7].

In this work, we track and modify the computation sequence of the test case gouttedo to make it reproducible. Modifications were only necessary to vector and EBE matrix operations. According to their experience, openTelemac developers are optimistic that no other source of non-reproducibility remains in the code [4]. The future development should integrate the same kind of modifications to other considered solving options, *e.g.* additional physical terms or other linear system solvers. At last, other structure operations should be corrected to obtain a whole reproducible code, *e.g.* operations on block structure, segment storage of matrices, *etc.* Up to our knowledge, these modifications seem easy to be integrated in future versions of a reproducible openTelemac.

Acknowledgment

We thank J.-M. Hervouet, EDR R&D (Chatou, France) for his valuable comments and his strong support to this work.

References

[1] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, pp. 224–242, 1971.

[2] J. W. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Proc. 21th IEEE Symposium on Computer Arithmetic*. Austin, Texas, USA, 2013.

[3] J.-M. Hervouet, *Hydrodynamics of free surface flows: Modelling with the finite element method.* John Wiley & Sons, 2007.

[4] J.-M. Hervouet and al., "Private communication," Dec. 2015.

[5] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, IEEE Computer Society, New York, 1985, reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.

[7] P. Langlois, R. Nheili, and C. Denis, "Numerical Reproducibility: Feasibility Issues," in *NTMS'2015: 7th IFIP International Conference on New Technologies, Mobility and Security*. Paris, France: IEEE, IEEE COMSOC & IFIP TC6.5 WG, Jul. 2015, pp. 1–5.

[8] ——, "Recovering numerical reproducibility in hydrodynamic simulations," in *23rd IEEE International Symposium on Computer Arithmetic*, J. H. P. Montuschi, M. Schulte, S. Oberman, and N. Revol, Eds., no. ISBN 978-1-5090-1615-0. IEEE Computer Society, Jul. 2016, pp. 63–70, (Silicon Valley, USA. July 10-13 2016).

[9] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[10] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, 2005.

[11] "Open TELEMAC-MASCARET. v.7.0, Release notes," www.opentelemac.org, 2014.