Improving the Numerical Accuracy of High Performance Computing Programs by Process Specialization

Farah Benmouhoub, Nasrine Damouche, and Matthieu Martel

LAMPS Laboratory
University of Perpignan,
52 Avenue Paul Alduy,
Perpignan, France, 66860.
{first.last}@univ-perp.fr

Abstract

In high performance computing, nearly all the implementations and published experiments use floating-point arithmetic. However, since floating-point numbers are finite approximations of real numbers, it may result in hazards because of the accumulated errors. These round-off errors may cause damages whose gravity varies depending on the critical level of the application. To deal with this issue, we have developed a tool which improves the numerical accuracy of computations by automatically transforming programs in a source-to-source manner. Our transformation, relies on static analysis by abstract interpretation and operates on pieces of code with assignments, conditionals, loops, functions and arrays. In this article, we apply our techniques to optimize a parallel program representative of the high performance computing domain. Parallelism introduces new numerical accuracy problems due to the order of operations in this kind of systems. We are also interested in studying the compromise between execution time and numerical accuracy.

1 Introduction

The IEEE754 Standard [1, 23] specifies the floating-point number arithmetic which is more and more used in many industrial domains including numerical simulations. However, floating-point arithmetic is prone to accuracy problems due to the round-off errors. The round-off errors which may be introduced by the approximation, can generate catastrophic results like the destruction of the British Petroleum plateform [27].

In this context, ensuring the correctness of the computations done in numerical simulations is extremely important. Some useful tools and techniques have been developed to validate [3, 12, 13, 16, 26] and improve [20, 24] the accuracy of arithmetic expressions in order to avoid failures. One limitation of these tools, for example, `Sardana` [20] or `Herbie` [24], is that they are applicable on a single arithmetic expression only. In other words, they cannot merge computations between the different lines of the code. For example, they can do nothing on a code written in 3-address style. To cope with this limitation, we have proposed an automatic tool, `Salsa`, which takes programs made of assignments, conditionals, loops, functions, arrays, etc., and generates another program numerically more accurate using a source-to-source transformation. This is possible thanks to the set of intraprocedural and interprocedural transformation rules defined in previous work [10, 9]. `Salsa` relies on static analysis by abstract interpretation [4] to compute variable ranges and round-off error bounds.

In this article, we are interesting in applying our techniques in order to improve the numerical accuracy of high performance computing programs. More precisely, we aim at taking advantage of the fact that each process may execute the operation in its own order. In other words, the idea is to pass from a hand-written SPMP code to a synthesized MIMD code in which each process

is specialized with respect to its data. To do this, we have taken an example of application that simulates the propagation of heat in a grid, implemented with the `MPI` library.

We present a comprehensive summary of how `Salsa` works. An extended description is given in [10]. We give an overview of the formal intraprocedural [9] and interprocedural [10] rules used in our transformation as well as on how the transformation of expressions is done [20].

The article is organized as following. Section 2 describes related work. In Section 3, we present the IEEE754 Standard and how to compute the error bounds, we give also a brief description of how transforming arithmetic expressions, intraprocedural and interprocedural programs. Section 4 details our motivation example and describe how we have proceeded in our experimentations. Section 5 describes the experimental results obtained when measuring the numerical accuracy of programs. In Section 6, we give numbers on the execution time required by programs before and after being optimized. Section 7 concludes.

2 Related Work

Several static analyses of the numerical accuracy of floating-point computations have been introduced during these last years. These methods over-approximate the worst error arising during the executions of a program. Static analyses based on abstract interpretation [4, 5] have been proposed and implemented in the `Fluctuat` tool [17, 18] which has been used in several industrial contexts. A main advantage of this method is that it enables one to bound safely all the errors arising during a computation, for large ranges of inputs. It also provides hints on the sources of errors, that is on the operations which introduce the most important precision loss. This latter information is of great interest to improve the accuracy of the implementation. More recently, Darulova and Kuncak have proposed a tool, `Rosa`, which uses a static analysis coupled to a SMT solver to compute the propagation of errors [12]. Solovyev et al. have proposed another tool, FP-Taylor based on symbolic Taylor expansions [26]. None of the techniques mentioned above generate more accurate programs.

Other approaches rely on dynamic analysis. For instance, the `Precimonious` tool tries to decrease the precision of variables and checks whether the accuracy requirements are still full filled [2]. Lam et al. instrument binary codes in order to modify their precision without modifying the source codes [21]. They also propose a dynamic search method to identify the pieces of code where the precision should be modified. Again, these techniques do not transform the codes in order to improve the accuracy.

Finally, another related research axis concerns the compile-time optimization of programs to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [14]. The `Sardana` tool takes arithmetic expressions and optimize them using a source-to-source transformation. Herbie optimizes the arithmetic expressions of Scala codes. While `Sardana` uses a static analysis to select the best expression, Herbie uses dynamic analysis (a set of random runs). A comparison of these tools is given in [11]. These techniques are limited to arithmetic expressions.

3 Background

In this section, we first present the floating-point arithmetic and then how round-off errors are computed. Second, we briefly describe how to transform arithmetic expressions in order to improve their numerical accuracy [20]. Finally, we give the principles of the transformation of both intraprocedural and interprocedural programs implemented in our tool, `Salsa`.

3.1 Floating-Point Arithmetic

In this section, we start by giving a brief description of the IEEE754 Standard [1] and then we present how to compute the round-off errors.

3.1.1 The IEEE754 Standard

Floating-point numbers are used to represent real numbers [1, 15]. Because of their finite representation, round-off errors arise during the computations and this may cause damages in critical contexts. The IEEE754 Standard formalizes a binary floating-point number as a triplet made of a sign, a mantissa and an exponent. We consider that a number x is written:

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot b^e = s \cdot m \cdot b^e \ , \tag{1}$$

where, s is the sign $\in \{-1, 1\}$, b is the basis ($b = 2$), m is the mantissa, $m = d_0.d_1 \dots d_{p-1}$ with digits $0 \leq d_i < b$, $0 \leq i \leq p-1$, p is the precision and e is the exponent $e \in [e_{min}, e_{max}]$. The IEEE754 Standard specifies some particular values for p, e_{min} and e_{max}.

The IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero and to the nearest respectively denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$. Let $\mathbb{R}$ be the set of real numbers and $\mathbb{F}$ be the set of floating-point numbers (we assume that only one format is used at the time, e.g. single or double precision). The semantics of the elementary operations specified by the IEEE754 Standard is given by Equation (2).

$$x \circledast_r y = \uparrow_r (x * y) \ , \text{ with } \uparrow_r : \mathbb{R} \to \mathbb{F} \ , \tag{2}$$

where a floating-point operation, denoted by $\circledast_r$, is computed using the rounding mode $r \in \{\uparrow_{+\infty}, \uparrow_{-\infty}, \uparrow_0, \uparrow_\sim\}$ and $* \in \{+, -, \times, \div\}$ is an exact operation. Obviously, the results of the computations are not exact because of the round-off errors. This is why, we use also the function $\downarrow_r : \mathbb{R} \to \mathbb{R}$ that returns the round-off error. We have

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \tag{3}$$

3.1.2 Error Bound Computation

In order to compute the errors during the evaluation of arithmetic expressions [22], we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} \equiv \mathbb{E}$ where x is the floating-point number used by the machine and μ is the exact error attached to $\mathbb{F}$, *i.e.*, the exact difference between the real and floating-point numbers as defined in Equation (3). For example, the real number $\frac{1}{3}$ is represented by the value $v = (\uparrow_\sim (\frac{1}{3}), \downarrow_\sim (\frac{1}{3})) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [22].

Our tool uses an abstract semantics [4] based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval contains the range of the floating-point values of the program and the second one contains the range of the errors obtained by subtracting the floating-point values from the exact ones. In the abstract value $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, $x^\sharp$ is the interval corresponding to the range of the values and $\mu^\sharp$ is the interval of errors on $x^\sharp$. This value abstracts a set of concrete values $\{(x, \mu) \ : \ x \in x^\sharp \text{ and } \mu \in \mu^\sharp\}$ by intervals in a component-wise way. We now introduce the semantics of arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp (x^\sharp)$. Here bounds are rounded to the nearest, see Equation (4).

$$\uparrow^\sharp ([\underline{x}, \overline{x}]) = [\uparrow_\sim (\underline{x}), \uparrow_\sim (\overline{x})] \ . \tag{4}$$

We denote by $\downarrow^\sharp$ the function that abstracts the concrete function $\downarrow_\sim$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp ([\underline{x}, \overline{x}])$. For a rounding mode to the nearest, we have

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \tag{5}$$

Formally, the *unit in the last place*, denoted by `ulp(x)`, consists of the weight of the least significant digit of the floating-point number x. Equations (6) and (7) give the semantics of the addition and multiplication over $\mathbb{E}^\sharp$, for other operations see [22]. If we sum two numbers, we must add the errors on the operands to the error produced by the round-off of the result. When multiplying two numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)\big) \ , \tag{6}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)\big) \ . \tag{7}$$

3.1.3 Transformation of Expressions

We briefly introduce former work [20, 28] to semantically transform arithmetic expressions using Abstract Program Expression Graph (APEG). This data structure remains in polynomial size while dealing with an exponential number of equivalent arithmetic expressions. To prevent any combinatorial problem, APEGs hold in abstraction boxes many equivalent expressions up to associativity and commutativity. A box containing n operands can represent up to $1 \times 3 \times 5 ... \times (2n-3)$ possible formulas. In order to build large APEGs, two algorithms are used (propagation and expansion algorithms). The first one searches recursively in the APEG where a symmetric binary operator is repeated and introduces abstraction boxes. Then, the second algorithm finds a homogeneous part and inserts a polynomial number of boxes. In order to add new shapes of expressions in an APEG, one propagates recursively subtractions and divisions into the concerned operands, propagate products, and factorizes common factors. Finally, an accurate formula is searched among all the equivalent formulas of the APEG using the abstract semantics of Section 3.1.2.

Example 3.1. *An example of APEG is given in Figure 1. When an equivalence class (denoted by a dotted ellipse) contains many sub-APEGs $p_1, \ldots, p_n$ then one of the p_i, $1 \leq i \leq n$, must be selected in order to build an expression. A box $\boxed{*(p_1, \ldots, p_n)}$ represents any parsing of the expression $p_1 * \ldots * p_n$. For instance, the APEG p of Figure 1 represents all the following*

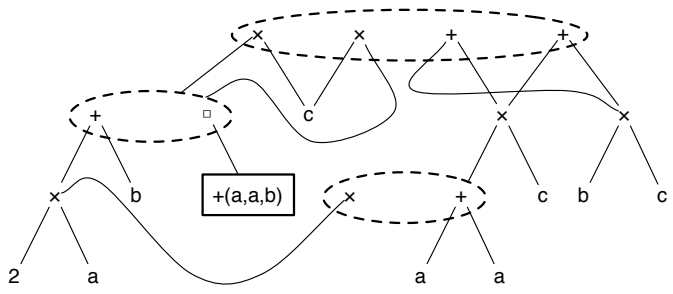

Figure 1: APEG for the expression $e = \big((a + a) + b\big) \times c$.

expressions:

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c,\ \big((b+a)+a\big)\times c,\ \big((2\times a)+b\big)\times c, \\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big),\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big), \\ (a+a)\times c+b\times c,\ (2\times a)\times c+b\times c,\ b\times c+(a+a)\times c,\ b\times c+(2\times a)\times c \end{array} \right\}.$$

In this example, the last step of the transformation consists of evaluating all the expressions in $\mathcal{A}(p)$ with the abstract semantics of Section 3.1.2 to select the most accurate one. ■

3.1.4 Intraprocedural and Interprocedural Transformation

In this section, we introduce our tool `Salsa` for numerical accuracy optimization by program transformation. `Salsa` is a tool that improves the numerical accuracy of programs based on floating-point arithmetic [6]. It reduces partly the round-off errors by automatically transforming C programs in a source-to-source manner. We have defined a set of intraprocedural transformation rules [9] like assignments, conditionals, loops, etc., and interprocedural transformation rules [10] for functions and other rules which deal with arrays. These rules have been implemented in the `Salsa` tool. `Salsa` relies on static analysis by abstract interpretation to compute variable ranges and round-off error bounds. it takes as first input ranges for the input variables of programs $id \in [a, b]$. These ranges are given by the user or coming from sensors. `Salsa` takes as second input a program to be transformed. `Salsa` applies the required transformation rules and returns as output a transformed program with better accuracy.

`Salsa` is composed of several modules. The first module is the parser that takes the original program in C language with annotations, puts it in SSA form (Static Single Assignment form) and then returns its binary syntax tree. The second module consists in a static analyzer, based on abstract interpretation [4], that infers safe ranges for the variables and computes errors bounds on them. The third module contains the intraprocedural transformation rules. The fourth module implements the interprocedural transformation rules. The last module is the `Sardana` tool, that we have integrated in our `Salsa` and call it on arithmetic expressions in order to improve their numerical accuracy.

When transforming programs we build larger arithmetic expressions that we choose to parse in a different ways to find a more accurate one. These large expressions will be sliced at a given level of the binary syntactic tree and assigned to intermediary variables named `TMP`. Note that the transformed program is semantically different from the original one but mathematically are equivalent. In [8], we have introduced a proof by induction that demonstrate the correctness of our transformation. In other words, we have proved that the original and the transformed programs are equivalent.

In practice, our transformation performs states of the form $\langle c, \delta, C, \nu, \beta\rangle$ where:

- c is a command,
- δ is a formal environment which maps variables to expressions. Intuitively, this environment records the expressions assigned to variables in order to inline them later on in larger expressions,
- C is a single hole context [19] that records the program enclosing the current command to be transformed,
- ν denotes the *target variable* that we aim at optimizing,
- β consists in a black list that contains the list of the variables that we must not remove from the program. Initially, β contains ν, *i.e.*, the *target variable* ν must not be removed.

$\delta = \emptyset$
$C = []$
$\beta = \{z\}$

```
x = a ;
if (x > 3.) then
  z = x + 2. + 1. ;
else
  z = x - 2. - 1. ;
```

$\delta' = \delta[x \mapsto a]$
$C = x = a; []$
$\beta = \{z\}$

```
nop ;
if (x > 3.) then
  z = x + 2. + 1. ;
else
  z = x - 2. - 1. ;
```

$\delta' = \emptyset$
$C = []$
$\beta = \{x, z\}$

```
x = a ;
if (x > 3.) then
  z = x + 2. + 1. ;
else
  z = x - 2. - 1. ;
```

$\delta' = \emptyset$
$C = x = a;\ if(x > 3)\ then\ []\ else\ []$
$\beta = \{x, z\}$

```
x = a ;
if (x > 3.) then
  z = x + 3. ;
else
  z = x - 3. ;
```

Figure 2: Initial (top left) program. First intermediary transformed (top right) program. Second intermediary transformed (bottom left) program. Final transformed (bottom right) program.

Example 3.2. *For example, let us consider the program in Figure 2 with a conditional. We assume that $\nu = z$ is the variable that we aim at optimizing. Initially, in the top left of Figure 2, the environment δ and the context C are empty and the black list β contains the* target variable ***z****. The first line of the program is an assignment, so we have rules that remove the assignments from the program and saving them into the memory δ when some conditions are verified. So, the first step, consists in removing the variable a and memorizing it in δ. Consequently, the line corresponding to the variable discarded is replaced by* nop*, the new environment is $\delta = [x \mapsto a]$ and the context contains the assignment discard, i.e., $C = x = a$. The new intermediary program is given in the top right of Figure 2. Next, we apply rules for sequences of commands and then we discard the* ***nop*** *lines from the program. The next step consists in analyzing this new program. We remark that the variable x is undefined in our program, this is why, we reinsert it into the program as given in the bottom left of Figure 2. Consequently, we remove it from the memory δ and we add it to the black list β. Now, to transform the conditional block of program, we dispose of several rules that transform only the executed branch if the evaluation of the condition is statically known, otherwise, we transform both branches of the conditional. In our case, we transform the* ***then*** *and the* ***else*** *branches using partial evaluation rules. The final program is shown in the bottom right of Figure 2.*

4 Case Study

In this section, we describe our case study to demonstrate that `Salsa` improves the numerical accuracy of the floating-point computations performed by a parallel code written in MPI [25].

The considered application models the temperature spread of a rectangular plate initially heated at two corners. For the simulations, we consider a grid of size [X×Y], which temperature is initialized to the value of 0.1 and heated at the top left and bottom right corners. In our experiments, we aim at performing a fast and accurate computation of the propagation of the heat on the grid. We use a parallel code in which we split the grid into bands, so that these bands can be assigned to different computational nodes, which can then work in parallel. Then, the parallelization is done by sharing the array representing the grid between working nodes as shown in the Figure 3. For this, we have to count our working nodes and distribute the data among them. Since, the nodes need neighbor line data to calculate their own line data, the values on the borders have to be communicated between the nodes. This is done with MPI_Send() and MPI_Rec(), that are included in the MPI environment.

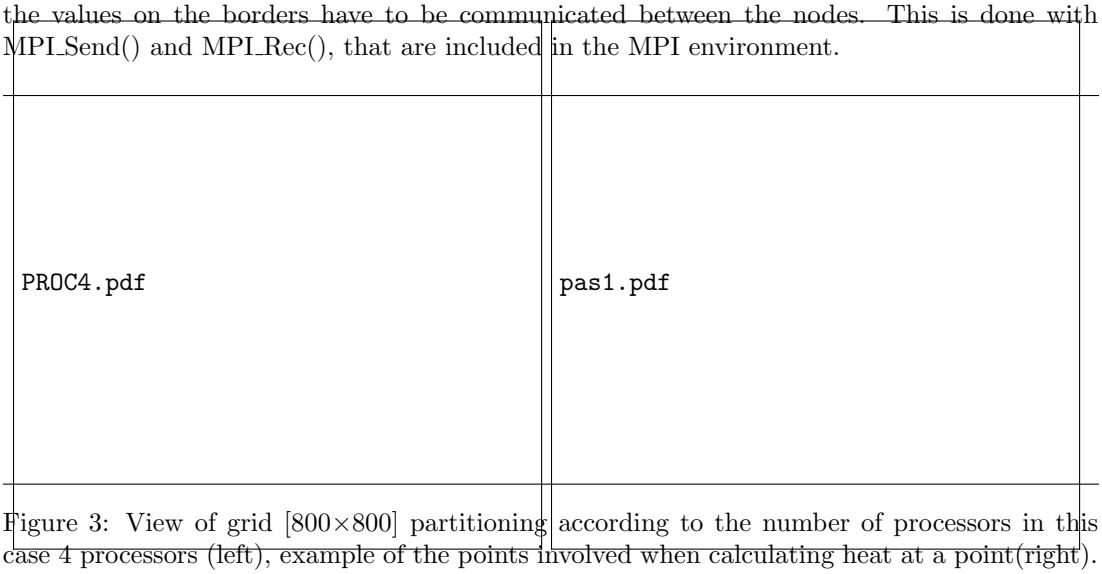

Figure 3: View of grid [800×800] partitioning according to the number of processors in this case 4 processors (left), example of the points involved when calculating heat at a point(right).

Independently of the analytical solution, we can give a numerical solution to this problem as well. This solution consists in an iterative computation, where current heat values are used for calculating the new heat values in the grid. The propagation of the heat at every point of the grid is calculated by the average of the heat of points that surround it. This average will be the new heat value of the current point, for example the case of a point in the center of the grid with stencil equal to one, see Equation (8).

$$new[i][j] = \frac{old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1]}{4}. \tag{8}$$

In our study, we have considered several parameters.

- The first one consists in increasing the stencil and observing the impact of the stencil on the spread of the heat in our grid. Equation (9) defines the computations of the heat at the center when the stencil is equal to two.

$$new[i][j] = \frac{old[i-1][j]+old[i-2][j]+old[i+1][j]+old[i+2][j]+old[i][j-1]+old[i][j-2]+old[i][j+1]+old[i][j+2]}{8}. \tag{9}$$

- The second parameter concerns the variation of the initial heat value at the two corners (top left and bottom right) according to significant melting temperatures of the chemical elements like Iron which is equal to 2800.4°F, Platinum equal to 3214.76°F, Copper equal to 1984.316°F and Calcium 1547.6°F.

- The last parameter is to change the number of iterations in order to find out if we run the simulation long enough, so that the temperature of the heated corners reaches all points of the grid.

5 Numerical Accuracy

In this section, we present the experimental results of our study concerning the numerical accuracy of computations. First, we study the impact of the propagation of the heat when increasing the stencil on the grid. To be done, we have to consider a parallel program with MPI [25]. The result of this simulation is given in Figure 4.

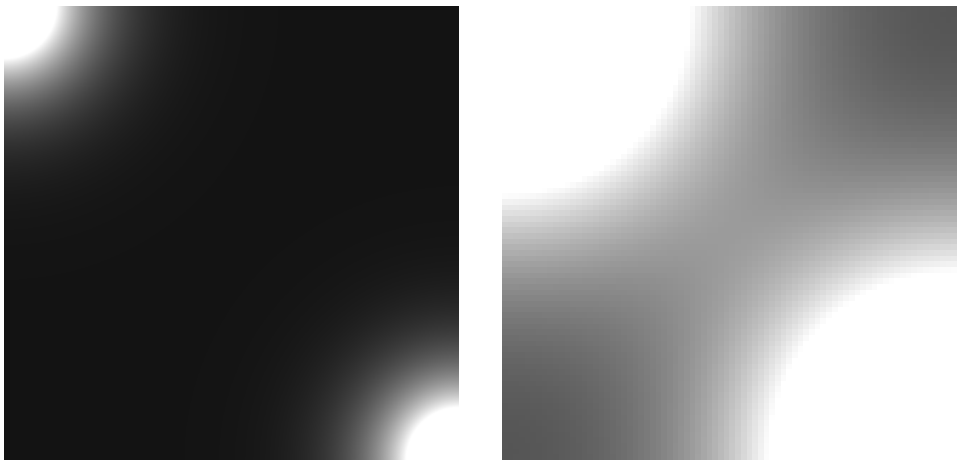

Figure 4: View of the heated grid at the two corners, number of iterations: 3600, stencil: 1 (left), stencil: 2 (right).

We notice that the figure has a blur effect at the corners which is due to the applied heating value. Thus, the initial temperature propagates inside the grid. If we run the program long enough (increasing the number of iterations), the value of the heated corners reaches all points of the grid.

Secondly, the main goal of the parallelization is to give a fast computation but we should pay attention to the numerical accuracy of this computation. We aim at evaluating the performances of `Salsa` on the improvement of the accuracy of the computations modeling the propagation of the heat on the grid, and we establish a comparison of the results before and after transforming the program. We measure `Salsa` performances by calculating relative and absolute errors between the initial program and the one transformed. Seen that, we heat at different positions in the grid (top left and bottom right corners), the distribution of the heat in our grid is different. The computations order in each processor depends on its data, so `Salsa` transforms the program of each processor according to its data. In the case of processor 0, Equation (8) becomes Equation (10):

$$new[i][j] = \frac{(((old[i][j+1] + old[i+1][j]) + old[i-1][j]) + old[i][j-1])}{4}. \tag{10}$$

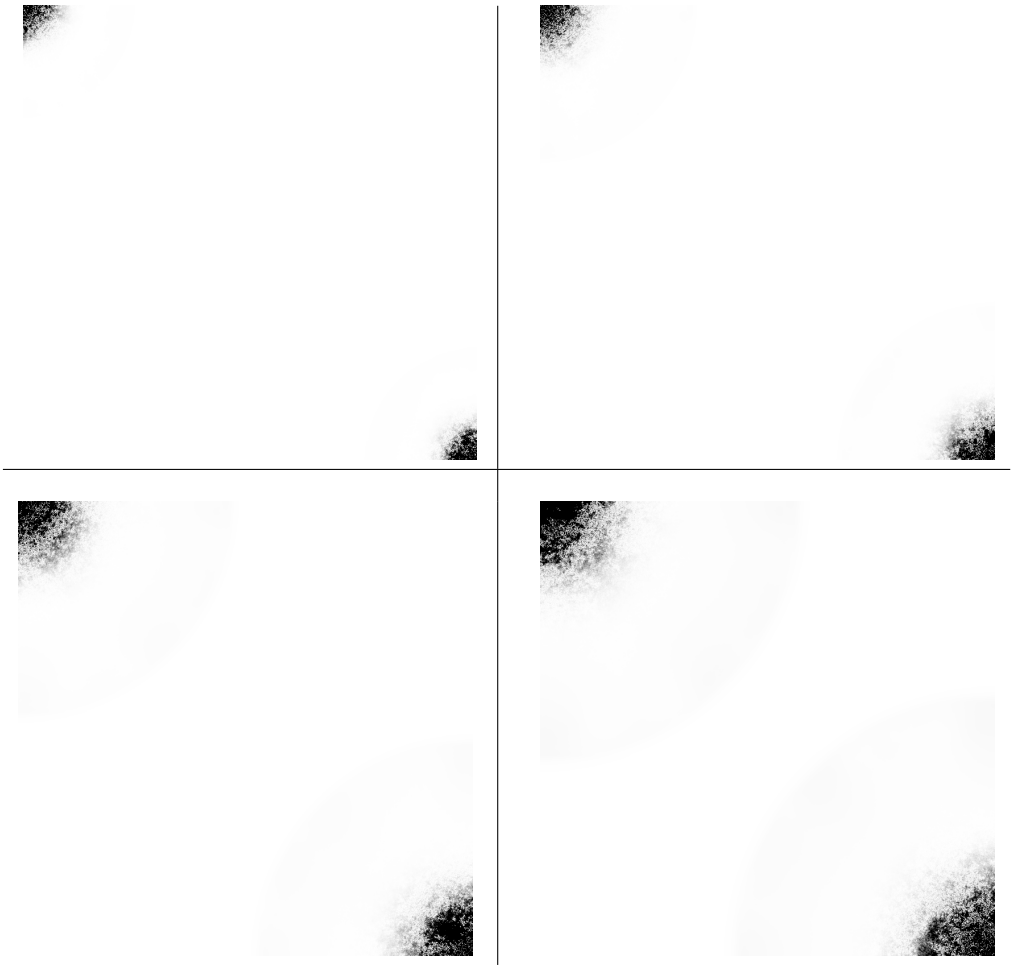

Figure 5: Absolute error between original program and optimized program, initial temperature: 10106.6°F, number of iterations: 1800 (top left), 3600 (top right), 7200 (bottom left), 10800 (bottom right).

The new parsing is due to the fact that we heat up at the top left corner. For instance, if we take the example of a point in the center with the coordinates (i, j), we will have the two points $(i + 1, j)$ and $(j - 1, i)$ with large heated values higher than two other points $(i, j + 1)$ and $(i + 1, j)$, so we sum up the small values first, then the big ones. Unlike Processor 3 while heating at the bottom right corner, and for the same point with the coordinates (i, j), the two points that will have the highest values will be $(i, j + 1)$ and $(i - 1, j)$, so Equation (8) will be of the form of Equation (11):

$$new[i][j] = \frac{(old[i][j + 1] + (old[i + 1][j] + (old[i][j - 1] + old[i - 1][j])))}{4}. \tag{11}$$

Figures 5 and 6 respectively, represent the relative and absolute error on the computations of the heat propagation on a grid of dimension [800×800], heated on the top left and bottom right corners at a temperature equal to that of platinum fusion.

On the one hand, if we are interested in the absolute errors on Figure 5, we notice that they are negligible in the middle of the grid, since we have not iterated enough on the computations of the heat propagation, so that the errors reach these points of the grid. We also notice that the absolute errors around the corners corresponding to the heating points appear small at the beginning for a number of iterations equal to 1800 top left on Figure 5. These errors will be accumulated and propagated while increasing the number of iterations such that it is presented in Figure 5 (3600 iterations_top right, 7200 iterations_bottom left, 10800 iterations_bottom right). In addition, when we measure the execution time corresponding to each case of the iterations number previously presented in Figure 5, we note that it takes only 0.52 seconds to do 1800 iterations and it needs 1.14 seconds to obtain 10800 iterations.

On the other hand, if we are interested in the relative errors of Figure 6, we notice the presence of errors around the heating corners. These errors depend on the number of iterations as well as the initial heating value of the two corners.

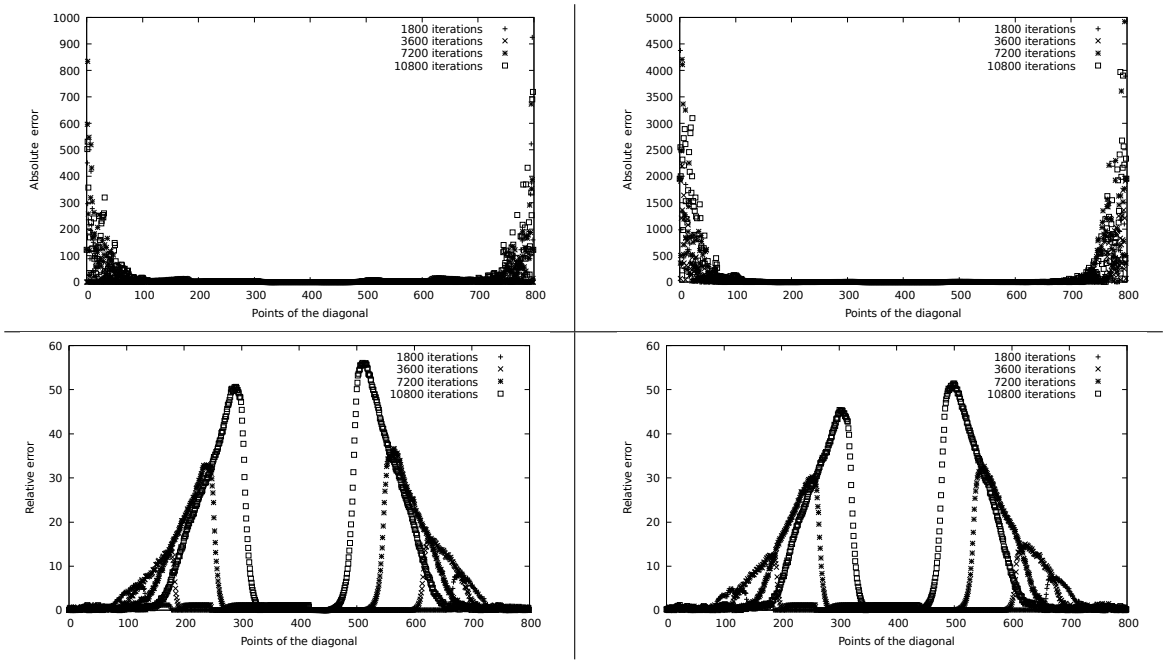

Figure 6: Absolute error and Relative error respectively between original program and optimized program, initial temperature: 1547.6°F (top left, bottom left), for different iterations, initial temperature: 10106.6°F (top right, bottom right).

6 Execution Time

In this section, we extend the concept proved by `Salsa` on accelerating the convergence speed [7] of serial programs to parallel programs. In practice, we have applied this study on the program of heat propagation previously described in Section 4. The results obtained when transforming the program with `Salsa` show that the numerical accuracy and the execution time of the program have been improved and the convergence speed has been accelerated, more precisely, the number of iterations required to converge to a given value has been reduced (this value

Initial temperature	Nbre of Iterations before transfo.	Nbre of Iterations after transfo.	Exec. Time before transfo.	Exec. Time after transfo.
1010.6°	10945	10912	1.110s	1.080s
1547.6°	6712	6693	0.916s	0.901s
1984.316°	5426	5411	0.876s	0.867s
2800.4°	3652	3643	0.797s	0.814s
3214.6°	3202	3190	0.789s	0.778s
10106.6°	1303	1290	0.716s	0.705s

Figure 7: Measurement of number of iterations and execution time of program of Section 4.

consists in the middle point of the grid). Figure 7 summarizes the different results in terms of number of iterations and execution time obtained when varying the initial temperature of the grid before and after the transformation.

If we take the example of a temperature that equal to 1010.6°, we want to calculate the number of iterations necessary to reach the value of 357°in the middle of the grid. Our results show that we are reducing the number of iterations of 33 iterations.

7 Conclusion

In this article, we have experimented the principle of improving accuracy of a parallel code concerning floating-point computations. We have applied `Salsa` on a code modeling the propagation of thermal energy in function of time, when heating at two corners of the grid. The results obtained show that `Salsa` improves the numerical accuracy of codes performing parallel floating point computations by rewriting the codes of each processor according to its data. A key research direction is to study the impact of accuracy optimization on the convergence time of distributed numerical algorithms like the ones used usually for high performance computing. In addition, still about distributed systems, an important issue concerns the reproducibility of the results: Different runs of the same application yield different results on different machines due to the variations in the order of evaluation of the mathematical expression. We would like to study how our technique could improve reproducibility. We would like also to modify the order of communications to see their impact on numerical accuracy. Code transformation techniques for numerical precision have proved their efficiency to accelerate the convergence of iterative methods (Newton, Jacobi, etc,.) in the sequential case. These results will be extended to parallel algorithms to improve the convergence of iterative methods.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.

[3] J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[5] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.

[6] N. Damouche and M. Martel. Salsa : An automatic tool improve the accuracy of programs. In *6th International Workshop on Automated Formal Methods, AFM*, 2017.

[7] N. Damouche, M. Martel, and A. Chapoutot. Impact of accuracy optimization on the convergence of numerical iterative methods. In M. Falaschi, editor, *LOPSTR 2015*, volume 9527 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2015.

[8] N. Damouche, M. Martel, and A. Chapoutot. Data-types optimization for floating-point formats by program transformation. In *International Conference on Control, Decision and Information Technologies, CoDIT 2016, Saint Julian's, Malta, April 6-8, 2016*, pages 576–581, 2016.

[9] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. In *International Journal on Software Tools for Technology Transfer*, volume 19, pages 427–448. Springer, 2016.

[10] N. Damouche, M. Martel, and A. Chapoutot. Numerical accuracy improvement by interprocedural program transformation. In Sander Stuijk, editor, *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2017, Sankt Goar, Germany, June 12-13, 2017*, pages 1–10. ACM, 2017.

[11] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In P. Prabhakar S. Bogomolov, M. Martel, editor, *NSV'16*, Lecture Notes in Computer Science. Springer, 2016.

[12] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[13] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[14] P-L. Garoche, F. Howar, T. Kahsai, and X. Thirioux. Testing-based compiler validation for synchronous languages. In J. M. Badger and K. Yvonne Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 246–251. Springer, 2014.

[15] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[16] E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2013.

[17] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In D. Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[18] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real Time Software, ERTS 2006, Proceedings*, 2006.

[19] E. Hankin. *Lambda Calculi A Guide For Computer Scientists*. Clarendon Press, Oxford, 1994.

[20] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[21] M. O. Lam, J. K. Hollingsworth, R. Bronis, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Supercomputing, ICS'13*, pages 369–378. ACM, 2013.

[22] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006.

[23] 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, 2010.

[24] J. R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.

[25] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[26] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.

[27] G. Suzanne and M. Terry. Bp suspended from new us federal contracts over deepwater disaster, 2012.

[28] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.