Improving performance and reliability of industrial simulations thanks to stochastic arithmetic

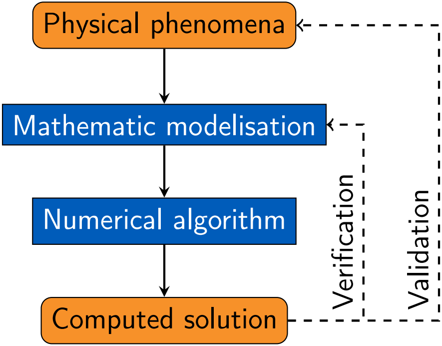
François Févotte [2]
Stef Graillat [1]
Fabienne Jézéquel [1]
Jean-Luc Lamotte [1]
Bruno Lathuilière [2]
Romain Picot [1,2]

LIP6 Equipe PEQUAN [1]
EDF R&D PERICLES I23 [2]

28 juin 2017

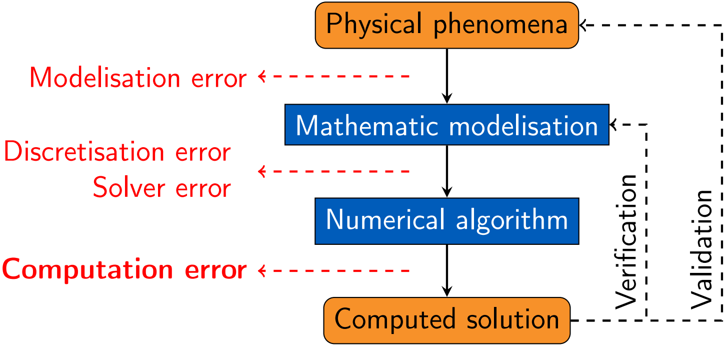
Context & issue

Verification & Validation



Context & issue

Verification & Validation

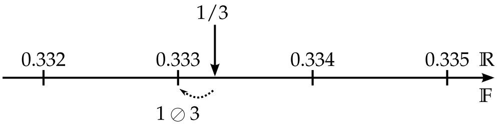


Error quantification:

- quality of the produced result
- efficiency of the means (computation/development time)

Floating-point arithmetic

- Finite precision of the floating-point representation
  - [our examples] decimal, 3 significant digits (% – ‰): 42.0, 0.123
  - [`float`] binary, 24 significant bits ($\simeq 10^{-7}$)
  - [`double`] binary, 53 significant bits ($\simeq 10^{-16}$)



- Consequences: floating-point computation $\neq$ real computation
  - rounding $a \oplus b \neq a + b$
  - no more associativity $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$

Floating-point arithmetic

Which consequences ?

Possible consequences:

- locked parallel computation (ex. TELEMAC2D)
- invalid result (ex. optimization under constraints of hydraulic production)
- results non-reproducibility (ex. ASTER, COCAGNE, ATHENA. . .)
- performance issue (ex. SATURNE, Apogene)

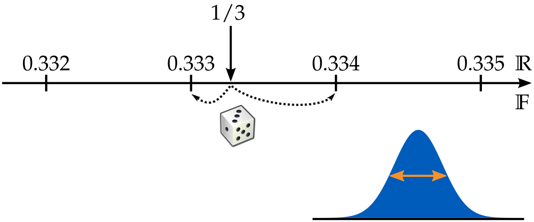
Floating-point arithmetic

Which solutions?

- A lot of solutions exist:
  - compensated / exact algorithms
  - reproducible algorithms
  - higher precision (`float` → `double` → MPFR)
  - comparisons tolerance
  - ...
- Need = problems detection
  - existence
  - quantification
  - localization

Stochastic arithmetic
The CADNA library

Stochastic arithmetic



- each operation executed 3 times with a random rounding mode
- number of correct digits in the results estimated using Student's test with the probability 95%
- operations executed synchronously
  - ⇒ detection of numerical instabilities
    Ex: `if (A>B)` with `A-B` numerical noise
  - ⇒ optimization of stopping criteria

The CADNA library

cadna.lip6.fr

- implements stochastic arithmetic for C/C++ or Fortran codes
- provides stochastic types (3 floating-point variables and an integer)
  - `float_st` in single precision
  - `double_st` in double precision
- all operators and mathematical functions overloaded
  $\Rightarrow$ few modifications in user programs
- uncertainty on data taken into account
- cost of CADNA ≈ 10
- support for MPI, OpenMP, GPU, vectorised codes

An example without/with CADNA

Computation of $P(x,y) = 9x^4 - y^4 + 2y^2$ [S.M. Rump, 1983]

```
#include <stdio.h>
double rump(double x, double y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  double x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n", rump(x, y));
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n", rump(x, y));
  return 0;
}
```

An example without/with CADNA

Computation of $P(x,y) = 9x^4 - y^4 + 2y^2$ [S.M. Rump, 1983]

```
#include <stdio.h>
double rump(double x, double y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  double x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n", rump(x, y));
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n", rump(x, y));
  return 0;
}
```

P1=2.00000000000000e+00

P2=8.02469135802469e–01

```
#include <stdio.h>

double    rump(double    x, double    y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {

  double    x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double    rump(double    x, double    y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {

  double    x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double    rump(double    x, double    y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double    x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"

  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double    rump(double    x, double    y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double    x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double    rump(double    x, double    y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double    x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%.14e\n",      rump(x, y) );"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%.14e\n",      rump(x, y) );"
  cadna_end();
  return 0;
}
```

```
#include <stdio.h>
#include <cadna.h>
double_st rump(double_st x, double_st y) {
  return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main(int argc, char **argv) {
  cadna_init(-1);
  double_st x, y;
  x=10864.0; y=18817.0;
  printf("P1=%s\n", strp(rump(x, y)));"
  x=1.0/3.0; y=2.0/3.0;
  printf("P2=%s\n", strp(rump(x, y)));"
  cadna_end();
  return 0;
}
```

Results with CADNA

only correct digits are displayed

CADNA_C 2.0.0 software — University P. et M. Curie — LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON

P1= @.0 (no more correct digits)
P2= 0.802469135802469E+000

There are 2 numerical instabilities
2 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)

Tools related to CADNA

available on `cadna.lip6.fr`

- CADNAIZER
  - automatically transforms C codes to be used with CADNA
- CADTRACE
  - identifies the instructions responsible for numerical instabilities

  Example:

  There are 12 numerical instabilities.

  10 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S).
  5 in <ex> file "ex.f90" line 58
  5 in <ex> file "ex.f90" line 59

  1 INSTABILITY IN ABS FUNCTION.
  1 in <ex> file "ex.f90" line 37

  1 UNSTABLE BRANCHING.
  1 in <ex> file "ex.f90" line 37

Precision optimization

The PROMISE tool

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC [Lam & al., 2013]
    - binary modifications on the operations
  - Precimonious [Rubio-Gonzàlez & al., 2013]
    - source modification with LLVM

They rely on comparisons with the highest precision result.

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC [Lam & al., 2013]
    - binary modifications on the operations
  - Precimonious [Rubio-Gonzàlez & al., 2013]
    - source modification with LLVM

They rely on comparisons with the highest precision result.

⚠ [S.M. Rump, 1988]
$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC [Lam & al., 2013]
    - binary modifications on the operations
  - Precimonious [Rubio-Gonzàlez & al., 2013]
    - source modification with LLVM

They rely on comparisons with the highest precision result.

⚠ [S.M. Rump, 1988]

$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$

float: P =2.571784e+29

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC [Lam & al., 2013]
    - binary modifications on the operations
  - Precimonious [Rubio-Gonzàlez & al., 2013]
    - source modification with LLVM

They rely on comparisons with the highest precision result.

⚠ [S.M. Rump, 1988]

$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$

float: P =2.571784e+29

double: P =1.17260394005318

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC `[Lam & al., 2013]`
    - binary modifications on the operations
  - Precimonious `[Rubio-Gonzàlez & al., 2013]`
    - source modification with LLVM

They rely on comparisons with the highest precision result.

⚠ [S.M. Rump, 1988]

$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$

float:	P =2.571784e+29
double:	P =1.17260394005318
quad:	P =1.17260394005317863185883490452018

Precision optimization

- mixed precision often leads to better performance
- existing tools:
  - CRAFT HPC [Lam & al., 2013]
    - binary modifications on the operations
  - Precimonious [Rubio-Gonzàlez & al., 2013]
    - source modification with LLVM

They rely on comparisons with the highest precision result.

⚠ [S.M. Rump, 1988]

$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$

float:	P =2.571784e+29
double:	P =1.17260394005318
quad:	P =1.17260394005317863185883490452018
exact:	P ≈-0.827396059946821368141165095479816292

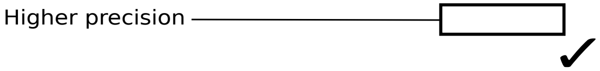
PROMISE (PRecision OptiMISE)

promise.lip6.fr

- Taking into account a required accuracy, PROMISE provides a mixed precision configuration (float, double, quad)
- 2 ways to validate a configuration:
  - validation of every execution using CADNA
  - validation of a reference using CADNA and comparison to this reference

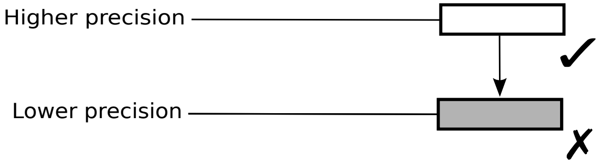
Searching for a configuration

Method based on Delta Debugging algorithm [Zeller, 2009]



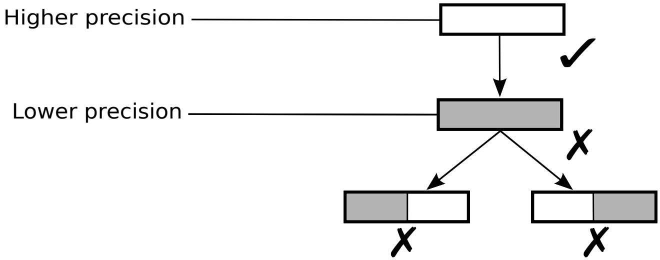
Searching for a configuration

Method based on Delta Debugging algorithm [Zeller, 2009]



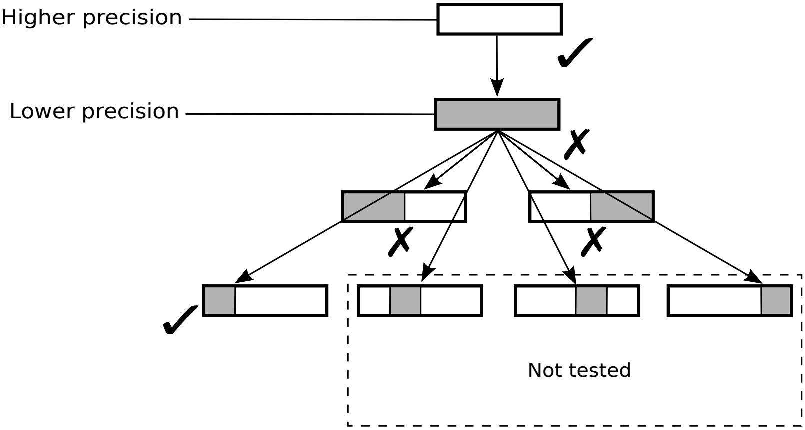
Searching for a configuration

Method based on Delta Debugging algorithm [Zeller, 2009]



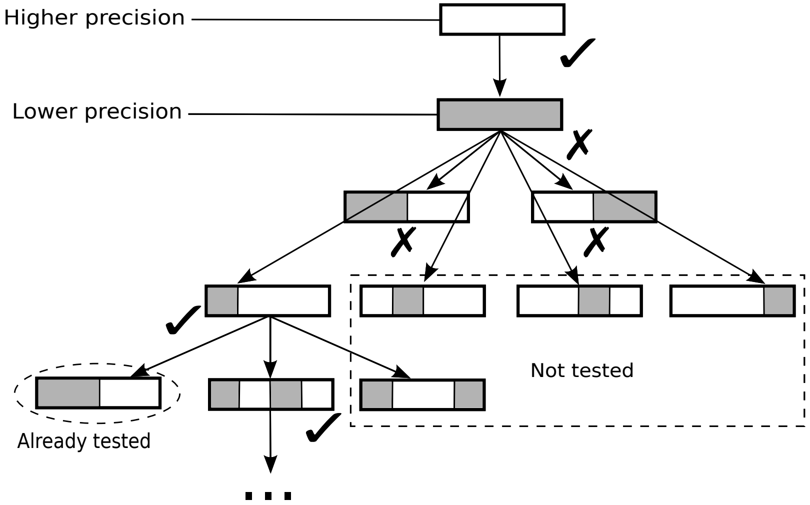
Searching for a configuration

Method based on Delta Debugging algorithm [Zeller, 2009]



Searching for a configuration

Method based on Delta Debugging algorithm [Zeller, 2009]



Searching for a configuration

- We will not have the *best* configuration.
- But the mean complexity is $O(n\log(n))$ and in the worst case $O(n^2)$

Searching for a configuration

- We will not have the *best* configuration.
- But the mean complexity is $O(n\log(n))$ and in the worst case $O(n^2)$

Efficient way of finding a local maximum configuration

Experimental results

Benchmarks

- **Short programs:**
  - arclength computation
  - rectangle method for the computation of integrals
  - Babylonian method for square root
  - matrix multiplication
- **GNU Scientific Library:**
  - Fast Fourier Transform
  - sum of Taylor series terms
  - polynomial evaluation/solver
- **SNU NPB Suite:**
  - Conjugate Gradient method
  - Scalar Penta-diagonal solver

Requested accuracy: 4, 6, 8 and 10 digits

⇒ PROMISE has found a new configuration each time.

Benchmark results

Program	#Digits	#exec	#double - #float	Time (mm:ss)	Result
arclength	exact				5.79577632241285
	10 8	21	8-1	0:13	**5.79577632241**303
	6	26	7-2	0:15	**5.79577**686259398
	4	16	2-7	0:09	**5.796**19547341572
rectangle	exact				0.100000000000000
	10 8	15	4-3	0:06	**0.10000000000000**2
	6	16	3-4	0:06	**0.10000000**1490116
	4	3	0-7	0:01	**0.10000**3123283386
squareRoot	exact				1.41421356237309
	10 8	21	6-2	0:07	**1.41421356237309**
	6 4	3	0-8	0:01	**1.4142135**3816986

MICADO: simulation of nuclear cores (EDF)

- neutron transport iterative solver
- 11,000 C++ code lines

# Digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed up	memory gain
10	83-51	19-32	88:56	1.01	1.00
8	80-48	18-33	85:10	1.01	1.01
6	69-37	13-38	71:32	1.20	1.44
5 4	3-3	0-51	9:58	1.32	1.62

MICADO: simulation of nuclear cores (EDF)

- neutron transport iterative solver
- 11,000 C++ code lines

# Digits	# comp - # exec	# double - # float	Time (mm:ss)	Speed up	memory gain
10	83-51	19-32	88:56	1.01	1.00
8	80-48	18-33	85:10	1.01	1.01
6	69-37	13-38	71:32	1.20	1.44
5 4	3-3	0-51	9:58	1.32	1.62

- Speed-up up to 1.32 and memory gain 1.62
- Mixed precision approach successful: speed-up 1.20 and memory gain 1.44

Conclusions – Perspectives (CADNA/PROMISE)

Conclusions

- CADNA has been successfully used for the numerical validation of academic and industrial simulation codes in astrophysics, atomic physics, chemistry, climate science, fluid dynamics, geophysics,...
- PROMISE optimises the precision of variables (float, double, quad) taking into account a target accuracy
  - execution time may ↘
  - required memory ↘
  - improve SIMD vectorization

Perspectives

- half precision (in CADNA & PROMISE)
- improve performance of PROMISE with parallelization

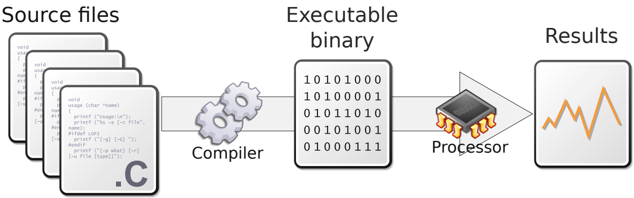
Stochastic arithmetic without modifying source codes

The Verrou tool

Why Verrou?

CADNA: dynamic sources analysis

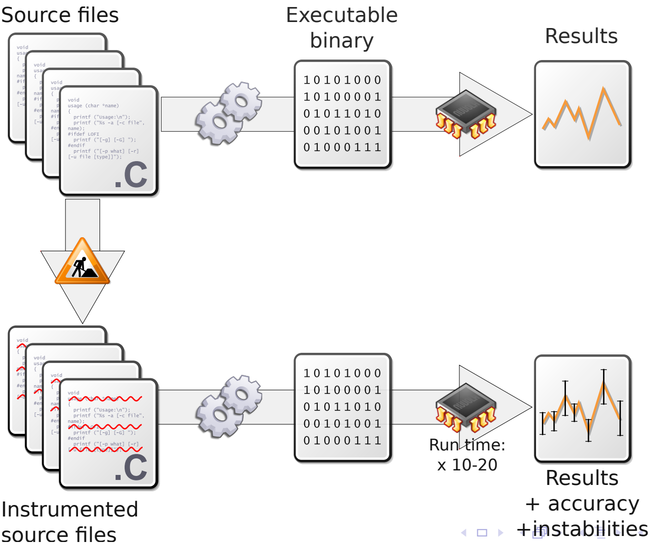
```
$ myProg in out
```



Why Verrou?

CADNA: dynamic sources analysis

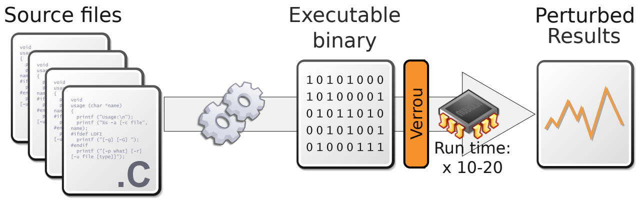
$ **myProg-cadna** `in out`



Why Verrou?

Verrou: dynamic binaries analysis

$ **valgrind --tool=verrou --rounding-mode=random** `myProg in out1`

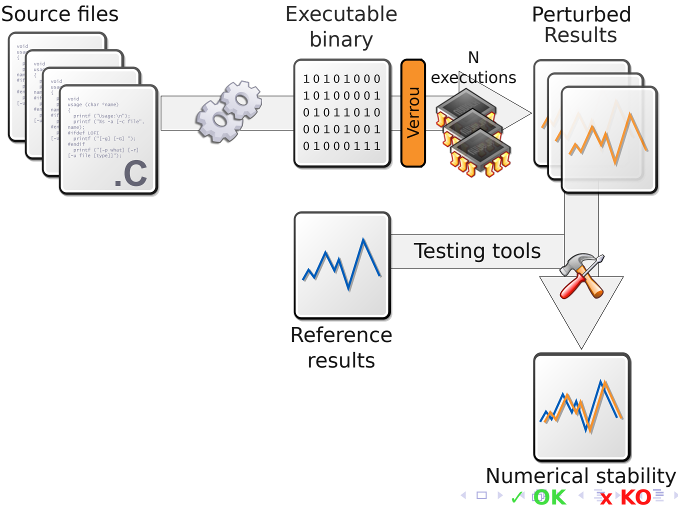


Why Verrou?

Verrou: dynamic binaries analysis

$ **valgrind --tool=verrou --rounding-mode=random** `myProg in out1`

$ **valgrind --tool=verrou --rounding-mode=random** `myProg in out2`



Comparison CADNA / VERROU

	CADNA	VERROU
Method	sync.	async.
# Samples	3	$N \geq 3$
Instrumentation	sources	binary
Localization	fine	coarse

Applications

Large number of applications

- EDF : Athena2D/3D, Micado, ApogeneV1, Morgane, Code_Aster, Moderato, Telemac, Code_Saturne, SoPlex
- EDF/EPRI : MAAP
- CEA : MFront, Alcyone

Applications

Large number of applications

- EDF : Athena2D/3D, Micado, ApogeneV1, Morgane, Code_Aster, Moderato, Telemac, Code_Saturne, SoPlex
- EDF/EPRI : MAAP
- CEA : MFront, Alcyone

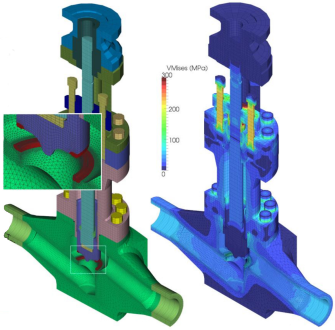
Focus on Code_Aster

1. Accuracy quantification
2. Instabilities localization
3. Accuracy improvement
4. Accuracy quantification

Application: Code_Aster

Mechanics

- Seismic
- Acoustic
- Thermo-mechanics



Code_Aster

- 1.2M code lines
- Fortran 90, C, Python
- Large number of dependencies :
  - Linear solvers (MUMPS. . .)
  - Mesh generator and partitioning tools (Metis, Scotch. . .)

Goals

- understand the non-reproducibility between test computers

Analysis of numerical instabilities with Random Rounding

Test case	nearest	Status rnd$_1$	rnd$_2$	rnd$_3$	# common digits $C(\text{rnd}_1, \text{rnd}_2, \text{rnd}_3)$
ssls108i	OK	OK	OK	OK	11 10
ssls108j	OK	OK	OK	OK	10 10
ssls108k	OK	OK	OK	OK	11 10
ssls108l	OK	OK	OK	OK	10 9
sdnl112a	OK	KO	KO	KO	6 6 6 * 3 0
ssnp130a	OK	OK	OK	OK	* * 10 10 10 10 9 * * * 9 9 9 9 * * 10
ssnp130b	OK	OK	OK	OK	* * 11 11 * 12 9 * * * 9 9 9 9 9 9 * *
ssnp130c	OK	OK	OK	OK	* 11 11 11 11 10 9 11 11 10 10 10 * 11
ssnp130d	OK	OK	OK	OK	* 9 * * * 10 9 9 9 9 9 9 9 9 * 9 * * *

Analysis of numerical instabilities with Random Rounding

Test case	Status nearest	rnd_1	rnd_2	rnd_3	# common digits $C(rnd_1, rnd_2, rnd_3)$
`ssls108i`	OK	OK	OK	OK	11 10
`ssls108j`	OK	OK	OK	OK	10 10
`ssls108k`	OK	OK	OK	OK	11 10
`ssls108l`	OK	OK	OK	OK	10 9
`sdnl112a`	OK	KO	KO	KO	6 6 6 * 3 0 (0 expected)
`ssnp130a`	OK	OK	OK	OK	* * 10 10 10 10 9 * * * 9 9 9 9 * * 10
`ssnp130b`	OK	OK	OK	OK	* * 11 11 * 12 9 * * * 9 9 9 9 9 9 * *
`ssnp130c`	OK	OK	OK	OK	* 11 11 11 11 10 9 11 11 10 10 10 * 11
`ssnp130d`	OK	OK	OK	OK	* 9 * * * 10 9 9 9 9 9 9 9 9 * 9 * * *

Unstable branchings localization by code covering

```
$ make CFLAGS="-fprofile-arcs -ftest-coverage"
$ make check
$ gcov *.c *.f
```

"standard" cover

```
120:subroutine fun1(area, a1, a2, n)
  -:    implicit none
  -:    integer :: n
  -:    real(kind=8) :: area, a1, a2
120:    if (a1 .eq. a2) then
 13:        area = a1
  -:    else
107:        if (n .lt. 2) then
107:           area = (a2-a1) / (log(a2)-log(a1))
###:        else if (n .eq.2) then
###:           area = sqrt (a1*a2)
  -:        else
###:           ! ...
  -:        endif
  -:    endif
120:end subroutine
```

"Verrou" cover

```
120:subroutine fun1(area, a1,...
  -:    implicit none
  -:    integer :: n
  -:    real(kind=8) :: area,...
120:    if (a1 .eq. a2) then
  4:        area = a1
  -:    else
116:        if (n .lt. 2) then
116:           area = (a2-a1)...
###:        else if (n .eq.2)...
###:           area = sqrt (a...
  -:        else
###:           ! ...
  -:        endif
  -:    endif
120:end subroutine
```

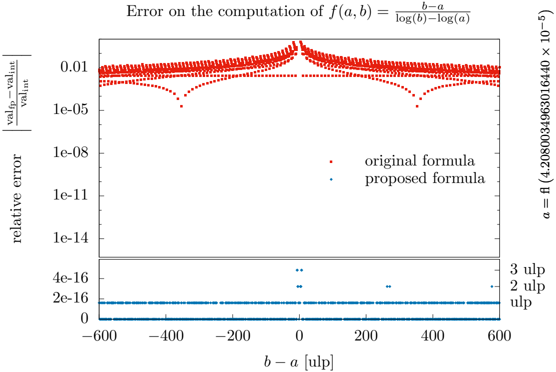
Application: Code_Aster

Formula correction

$$f(a,b) = \begin{cases} a & \text{if } a = b \\ \dfrac{b-a}{\log(b)-\log(a)} & \text{otherwise} \end{cases} \longrightarrow f(a,b) = \begin{cases} a & \text{if } a = b \\ a\,\dfrac{\frac{b}{a}-1}{\log(\frac{b}{a})} & \text{otherwise} \end{cases}$$

Empirical study

- outside the code (proxy app)
- around the problematic point
- reference = interval arithmetic



Proof

- error bounded by 10 ulps

Localization with Delta-Debugging

Delta-Debugging Basis

Verrou can perturb floating-point operations only in :

- selected symbols
- selected lines (if code compiled with -g)

Delta-Debugging for the users

- Inputs:
  - run script
  - comparison script
- Output:
  - DDmax: maximal set of function (or line) leading to error

Localization with Delta-Debugging

```
      do 60 jvec = 1, nbvect
         do 30 k = 1, neq
            vectmp(k)=vect(k,jvec)
30       continue
         if (prepos) call mrconl('DIVI', lmat, 0, 'R', vectmp,1)
         xsol(1,jvec)=xsol(1,jvec)+zr(jvalms-1+1)*vectmp(1)
         do 50 ilig = 2, neq
            kdeb=smdi(ilig-1)+1
            kfin=smdi(ilig)-1
            do 40 ki = kdeb, kfin
               jcol=smhc(ki)
               xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalmi-1+ki) * vectmp(jcol)
               xsol(jcol,jvec)=xsol(jcol,jvec) + zr(jvalms-1+ki) * vectmp(ilig)
40          continue
            xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalms+kfin) * vectmp(ilig)
50       continue
         if (prepos) call mrconl('DIVI', lmat, 0, 'R', xsol(1, jvec),1)
60    continue
```

Localization with Delta-Debugging

```
      do 60 jvec = 1, nbvect
         do 30 k = 1, neq
            vectmp(k)=vect(k,jvec)
30       continue
         if (prepos) call mrconl('DIVI', lmat, 0, 'R', vectmp,1)
         xsol(1,jvec)=xsol(1,jvec)+zr(jvalms-1+1)*vectmp(1)
         do 50 ilig = 2, neq
            kdeb=smdi(ilig-1)+1
            kfin=smdi(ilig)-1
```

- correction : compensated algorithm.

```
               xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalmi-1+ki) * vectmp(jcol)
               xsol(jcol,jvec)=xsol(jcol,jvec) + zr(jvalms-1+ki) * vectmp(ilig)
40          continue
            xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalms+kfin) * vectmp(ilig)
50       continue
         if (prepos) call mrconl('DIVI', lmat, 0, 'R', xsol(1, jvec),1)
60    continue
```

Accuracy quantification after correction

`sdnl112a`

Version	Status				# common digits
	nearest	rnd_1	rnd_2	rnd_3	$C(rnd_1, rnd_2, rnd_3)$
Before correction	OK	KO	KO	KO	6 6 6 * 3 0
After correction	OK	OK	OK	OK	10 10 9 * 6 0

Accuracy quantification after correction

`sdnl112a`

Version	Status				# common digits
	nearest	rnd_1	rnd_2	rnd_3	$C(rnd_1, rnd_2, rnd_3)$
Before correction	OK	KO	KO	KO	6 6 6 * 3 0
After correction	OK	OK	OK	OK	10 10 9 * 6 0

Computation time for one sample

- Reference : 2.1s
- Verrou : 14.4s (x 7)
- Memcheck : 76.9s (x37)

Conclusions – Perspectives (Verrou)

Conclusions

Verrou seems to fulfill our needs:

- low entry cost;
- floating-point accuracy quantification;
- semi-automatic instabilities localization (coarse grain).

Perspectives

- use the interface Interflop;
- take into account all instructions;
  - AVX instructions;
  - x87 scalar instructions;
- reinforce the Delta-Debugging features;
- reinforce the code covering localization features.

Conclusions – Perspectives

Conclusions

- stochastic arithmetic is well fitted to industrial applications;
- the tools CADNA and Verrou are complementary.

Perspectives

- generalize the theory from synchronous to asynchronous;
- use a libmath with random rounding.

Thank you !
Questions ?

Get verrou on github:
http://github.com/edf-hpc/verrou
Documentation:
http://edf-hpc.github.io/verrou/vr-manual.html

Get CADNA:
http://cadna.lip6.fr/
Get PROMISE:
http://promise.lip6.fr/