Distributed Computing

Lab1

Lucas Chenel

Table des matières

[1. Hardware knowledge 3](#_Toc68775792)

[1.1. CPU info 3](#_Toc68775793)

[1.2. Cache 4](#_Toc68775794)

[1. Information about the caches of the machine 4](#_Toc68775795)

[2. Look up “cache misses” 5](#_Toc68775796)

[1.3. Compiler optimization 5](#_Toc68775797)

[Information about the compiler optimizations 5](#_Toc68775798)

[2. Optimization 6](#_Toc68775799)

[2.1. Basic operations 6](#_Toc68775800)

[1. Measure the “atomic” runtime of basic operations 6](#_Toc68775801)

[2. Compare their runtime using different optimization flags 6](#_Toc68775802)

[2.2. Invariants 7](#_Toc68775803)

[1. Find wich flag is used to remove an invariant expression that involves a division 7](#_Toc68775804)

[2. Compare the runtime of invariants.c with and without this flag 8](#_Toc68775805)

[3. Identify expression and remove it 8](#_Toc68775806)

[4. Compare the new runtime 9](#_Toc68775807)

[2.3. Lookup tables 9](#_Toc68775808)

[1. Modification of lookup.c 9](#_Toc68775809)

[2. Comparaison of the runtime of both version 10](#_Toc68775810)

[2.4. Indirect addressing 10](#_Toc68775811)

[1. Use the function randrange () 10](#_Toc68775812)

[2. Compare runtimes between a randomized access and a sequential access of the table. 11](#_Toc68775813)

[3. Modification of the program so that the total number of operations 12](#_Toc68775814)

[2.5. Loop interchange 12](#_Toc68775815)

[1. Copy interchange.c as interchange-ok.c and modify interchange-ok.c so that the arrays are ran through the efficient way 12](#_Toc68775816)

[2. Compare the runtime of both programs without flags for ndim=200, niter=2000 and ndim=1000, niter=100 13](#_Toc68775817)

[3. Repeat with -O3 13](#_Toc68775818)

[4. Conclude 14](#_Toc68775819)

[2.6. Loop fusion 14](#_Toc68775820)

[1. Differences between fusion.c and fusion-ok.c 15](#_Toc68775821)

[2. Compare the runtime of this files 15](#_Toc68775822)

[3. With -O3 16](#_Toc68775823)

[4. Conclude 16](#_Toc68775824)

[3. Vectorization 16](#_Toc68775825)

[3.0. Vectorization 16](#_Toc68775826)

[1. Compile the program optimisations/vectorization/forward.c 16](#_Toc68775827)

[2. Loops have been vectorized : 16](#_Toc68775828)

[3. Compare the performances with and without vectorization 17](#_Toc68775829)

[3.1. Unrolling 18](#_Toc68775830)

[1. Try the flag with the file unrolling.c. 18](#_Toc68775831)

[2. With unrolling-(2,4,6).c 18](#_Toc68775832)

[3.2. Inlining 19](#_Toc68775833)

[1. Very important in complex codes, inlining reduces the overhead of the function call and can trigger other compiler optimizations. 19](#_Toc68775834)

[2. Test the flag with the code inlining.c 19](#_Toc68775835)

[3.3. Dependencies 20](#_Toc68775836)

[1. Determine in both cases D < 0 and D > 0 whether the iterations present a dependence. If so, which type of dependence is it 20](#_Toc68775837)

[2. Which values of \_\_D\_\_ allow the compilator to vectorize the loop? Why? Repeat for tables with double precisions and interpret the result. Use the flag -fopt-info-vec to check the loops were vectorized. 20](#_Toc68775838)

[3. Test the performance of the program with and without vectorization in the two cases 20](#_Toc68775839)

[4. Interpret of results 21](#_Toc68775840)

[3.4. Branching 21](#_Toc68775841)

[1. Compile if.c 21](#_Toc68775842)

[2. Create a new program if-ok.c such that (a[i] + a[i+1]) / (b[i] \* b[i+1]) are computed before the loop containing the if statement, and stored in a temporary array 21](#_Toc68775843)

[3. Compile if-ok.c and compare with if.c 22](#_Toc68775844)

[4. GProf and SLURM 22](#_Toc68775845)

[3.1. Using GProf 22](#_Toc68775846)

[1. Compile and run 22](#_Toc68775847)

[2. The code outputs an additional file (.x) 22](#_Toc68775848)

[3. Run the profile on this file 22](#_Toc68775849)

# Hardware knowledge

## CPU info

Une image contenant texte

Description générée automatiquement

Figure 1 Informations about the processeur 0 of CPU of the machine

Une image contenant texte

Description générée automatiquement

Figure 2 Informations about the last processeur (31) of CPU

## Cache

### Information about the caches of the machine

Une image contenant texte

Description générée automatiquement

Figure 3 Information about file cpu

Une image contenant texte

Description générée automatiquement

Figure 4 Information about size of cpu0

Une image contenant texte

Description générée automatiquement

Figure 5 Information about type of cpu0

Une image contenant texte

Description générée automatiquement

Figure 6 Information about shared\_cpu\_listed of cpu0

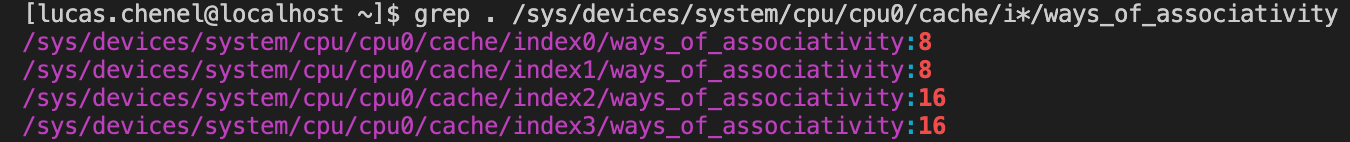


Figure 7 Information about ways of associativity of cpu0

### Look up “cache misses”

No questions

## Compiler optimization

### Information about the compiler optimizations

Une image contenant texte

Description générée automatiquement

Figure 8 Information about the compiler optimizations O3

Une image contenant texte

Description générée automatiquement

Figure 9 Information about the compiler optimizations O0

# Optimization

## Basic operations

### Measure the “atomic” runtime of basic operations

Une image contenant texte

Description générée automatiquement

Figure 10 Mesure of atomic runtime of sum

Une image contenant texte

Description générée automatiquement

Figure 11 Mesure of atomic runtime

### Compare their runtime using different optimization flags

Une image contenant texte

Description générée automatiquement

Figure 12 Runtime with flag -O3

Une image contenant texte

Description générée automatiquement

Figure 13 Runtime with using flags -O3 -ffast-math

Une image contenant texte

Description générée automatiquement

Figure 14 Runtime with previous exercise

## Invariants

### Find wich flag is used to remove an invariant expression that involves a division

The flag wich is used to remove an invariant expression that involves a division is a flag

-freciprocal-math.

### Compare the runtime of invariants.c with and without this flag

Une image contenant texte

Description générée automatiquement

Figure 15 Runtime of invariants.c without flag

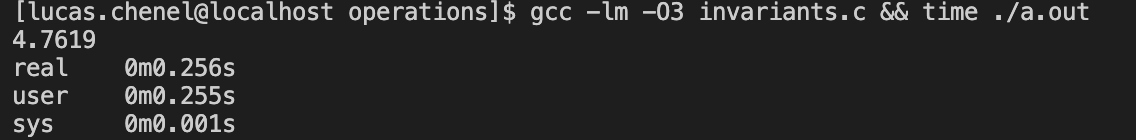


Figure 16 Runtime of invariants.c with flag O3

Une image contenant texte

Description générée automatiquement

Figure 17 Runtime of invariants.c with flag O

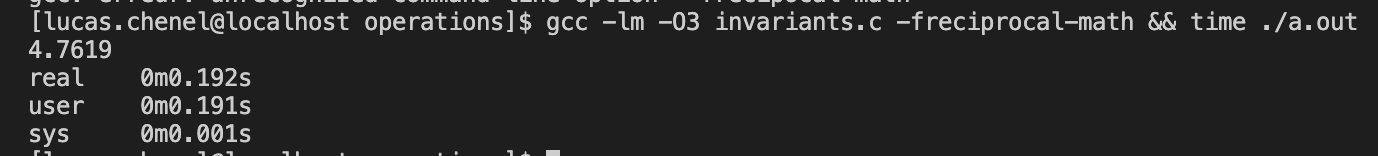


Figure 18 Runtime of invariants.c -freciprocal-math

### Identify expression and remove it

Une image contenant texte

Description générée automatiquement

Figure 19 New invariants.c with modifications

### Compare the new runtime

Une image contenant texte

Description générée automatiquement

Figure 20 New comparaisons of runtimes

The runtimes are all smaller, but we still have:

Runtime (with -O3 -freciprocal-math) < runtime (with -O3) < runtime (without)

## Lookup tables

### Modification of lookup.c

Une image contenant texte

Description générée automatiquement

Figure 21 Modification of lookup.c

### Comparaison of the runtime of both version

Une image contenant texte

Description générée automatiquement

Figure 22 Runtime before lookup.c modifications

Une image contenant texte

Description générée automatiquement

Figure 23 Runtime after lookup.c modifications

To conclude: storing the values in an array to use them afterwards allows a better optimization than when calculating the values at each turn of the loop.

## Indirect addressing

### Use the function randrange ()

Une image contenant texte

Description générée automatiquement

Figure 24 Instantiate an array with randrange ()

### Compare runtimes between a randomized access and a sequential access of the table.

Une image contenant texte

Description générée automatiquement

Figure 25 Runtime with a randomized access

Une image contenant texte

Description générée automatiquement

Figure 26 Runtime with a sequential access

### Modification of the program so that the total number of operations

Une image contenant texte

Description générée automatiquement

Figure 27 Modification of random.c program so that the total number of operations

1. Measure the runtime t(N) as a function of the array size N

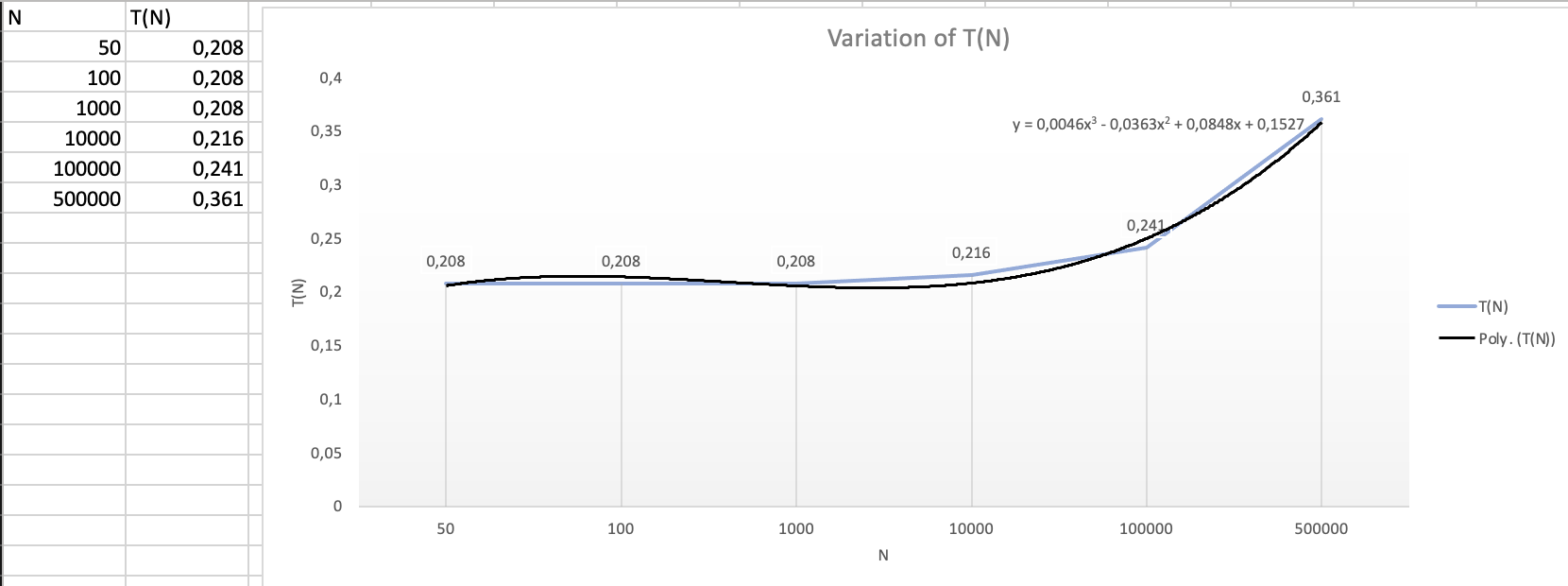


Figure 28 Measure of the runtime t(N) as a function N

## Loop interchange

### Copy interchange.c as interchange-ok.c and modify interchange-ok.c so that the arrays are ran through the efficient way

Une image contenant texte

Description générée automatiquement

Figure 29 Interchange-ok.c

### Compare the runtime of both programs without flags for ndim=200, niter=2000 and ndim=1000, niter=100

Une image contenant texte

Description générée automatiquement

Figure 30 Compare runtimes for ndim = 200 and niter = 2000

Une image contenant texte

Description générée automatiquement

Figure 31 Compare runtimes for ndim = 1000 and niter = 100

### Repeat with -O3

Une image contenant texte

Description générée automatiquement

Figure 32 Compare runtimes for ndim = 200 and niter = 2000, with -O3

Une image contenant texte

Description générée automatiquement

Figure 33 Compare runtimes for ndim = 1000 and niter = 100, with -O3

### Conclude

We notice that the interchange-ok.c runtime is almost twice as fast as interchange.c. We can assume that reading column then rows is more optimized than rows then columns.

## Loop fusion

### Differences between fusion.c and fusion-ok.c

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Figure 34 Fusion-ok.c

Figure 35 Fusion.c

We observe that:

* In fusion.c, there are two loops: one for the array b and one for the array c;
* In fusion-ok.c, we have only one loop allowing to obtain b and c.

### Compare the runtime of this files

Une image contenant texte

Description générée automatiquement

Figure 36 Compare runtimes between fusion.c and fusion-ok.c

### Une image contenant texte Description générée automatiquementWith -O3

Figure 37 Compare runtimes between fusion.c and fusion-ok.c, with -O3

### Conclude

Without optimization: fusion-ok.c is faster because it has one less loop

With optimization: same thing except that we observe that fusion.c had a better optimization than fusion-ok.c (it lost 0.140s while fusion-ok.c, 0.04s).

Conclusion: a file can have a better runtime but a worse optimization (than a file that has a good runtime point)?

# Vectorization

## Vectorization

### Compile the program optimisations/vectorization/forward.c

$ gcc -O1 -ftree-vectorize -fopt-info-vec -fopt-info- vec-missed forward.c

### Loops have been vectorized :

Une image contenant texte

Description générée automatiquement

Figure 38 Loop forward.c:20 has been vectorized



Figure 39 Other loops have not been vectorized



### Compare the performances with and without vectorization

Une image contenant texte

Description générée automatiquement

Figure 40 Performances without vectorization

Une image contenant texte

Description générée automatiquement

Figure 41 Performances with vectorization

Vectorization is therefore much better than using an array.

## Unrolling

### Try the flag with the file unrolling.c.

Une image contenant texte

Description générée automatiquement

Figure 42 Compare runtimes with and without -funroll-loops

### With unrolling-(2,4,6).c

Une image contenant texte

Description générée automatiquement

Figure 43 Compare performances with other unrolling files

Runtime (with -funroll-loops) < Runtime (without -funroll-loops)

Une image contenant texte

Description générée automatiquement

Figure 44 To see modifications

## Inlining

### Very important in complex codes, inlining reduces the overhead of the function call and can trigger other compiler optimizations.

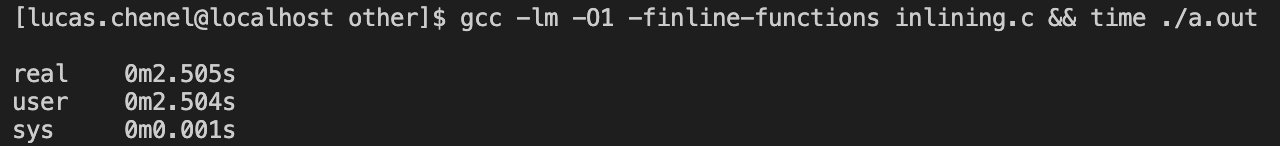


Figure 45 Runtime inlining

### Test the flag with the code inlining.c

Une image contenant texte

Description générée automatiquement

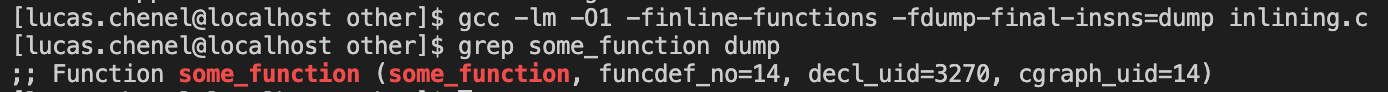


Figure 46 Does it indeed inline the function some\_function() ?

Une image contenant texte

Description générée automatiquement

Figure 47 Runtimes with function, static function, inline function and static inline function (top to bottom)

## Dependencies

### Determine in both cases D < 0 and D > 0 whether the iterations present a dependence. If so, which type of dependence is it

For D < 0, the iterations present a backward dependence of the data while for D > 0, the iterations will present a forward dependence of the data

### Which values of \_\_D\_\_ allow the compilator to vectorize the loop? Why? Repeat for tables with double precisions and interpret the result. Use the flag -fopt-info-vec to check the loops were vectorized.

We use: gcc -O1 -ftree-vectorize -fopt-info-vec -fopt-info-vec-missed forward.c

Vectorization is possible with a forward dependency, so we need D > 0.

We can see, with gcc -O1 -ftree-vectorize -fopt-info-vec -fopt -info-vec-missed dependance.c, that only the loop at the 19th line has been vectorized.

### Test the performance of the program with and without vectorization in the two cases

Une image contenant texte

Description générée automatiquement

Figure 48 Performance of the program with 'float'

Une image contenant texte

Description générée automatiquement

Figure 49 Performance of the program with 'double'

### Interpret of results

It takes more time with double because it is a float with more precisision.

So, the vectorization has a bigger impact on a float than on a double.

## Branching

### Compile if.c

Une image contenant texte

Description générée automatiquement

Figure 50 Compilation of if.c

### Create a new program if-ok.c such that (a[i] + a[i+1]) / (b[i] \* b[i+1]) are computed before the loop containing the if statement, and stored in a temporary array

Une image contenant texte

Description générée automatiquement

Figure 51 Creation of if-ok.c

### Compile if-ok.c and compare with if.c

Une image contenant texte

Description générée automatiquement

Figure 52 Compare if.c with if-ok.c

# GProf and SLURM

## Using GProf

### Compile and run



Figure 53 Compile and run forward.x

### The code outputs an additional file (.x)

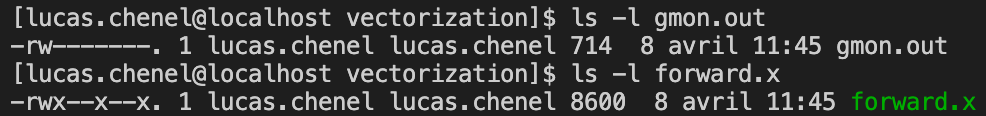


Figure 54 Use ls -l gmon.out

### Run the profile on this file

Une image contenant texte

Description générée automatiquement

Figure 55 Use gprof ./forward.x | less