LEJOUX Thibault

TP1 distributed computing

1 Hardware knowledge

1.1 CPU info

```
[thibault.lejoux@localhost ~]$ more /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 6
model name      : Common KVM processor
stepping        : 1
microcode       : 0x1
cpu MHz         : 2194.842
cache size      : 16384 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx lm constant_tsc nopl xtopology
 eagerfpu pni cx16 x2apic hypervisor lahf_lm
bogomips        : 4389.68
```

1.2 Cache

```
[thibault.lejoux@localhost index0]$ cd /sys/devices/system/cpu
[thibault.lejoux@localhost cpu]$ grep . cpu0/cache/index*/size
cpu0/cache/index0/size:32K
cpu0/cache/index1/size:32K
cpu0/cache/index2/size:4096K
cpu0/cache/index3/size:16384K
[thibault.lejoux@localhost cpu]$ grep . cpu0/cache/index*/type
cpu0/cache/index0/type:Data
cpu0/cache/index1/type:Instruction
cpu0/cache/index2/type:Unified
cpu0/cache/index3/type:Unified
[thibault.lejoux@localhost cpu]$
[thibault.lejoux@localhost cpu]$ grep . cpu0/cache/index*/shared_cpu_list
cpu0/cache/index0/shared_cpu_list:0
cpu0/cache/index1/shared_cpu_list:0
cpu0/cache/index2/shared_cpu_list:0
cpu0/cache/index3/shared_cpu_list:0-7
[thibault.lejoux@localhost cpu]$ grep . /sys/devices/system/cpu/cpu0/cache/i*/ways_of_associativity
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity:8
/sys/devices/system/cpu/cpu0/cache/index1/ways_of_associativity:8
/sys/devices/system/cpu/cpu0/cache/index2/ways_of_associativity:16
/sys/devices/system/cpu/cpu0/cache/index3/ways_of_associativity:16
```

1.3 Compiler optimizations

```
[thibault.lejoux@localhost cpu]$ gcc -Q -O3 --help=optimizers | grep enabled
  -faggressive-loop-optimizations       [enabled]
  -falign-functions                     [enabled]
  -falign-jumps                         [enabled]
  -falign-labels                        [enabled]
  -falign-loops                         [enabled]
  -fasynchronous-unwind-tables          [enabled]
  -fbranch-count-reg                    [enabled]
  -fcaller-saves                        [enabled]
  -fcombine-stack-adjustments           [enabled]
  -fcommon                              [enabled]
  -fcompare-elim                        [enabled]
  -fcprop-registers                     [enabled]
  -fcrossjumping                        [enabled]
  -fcse-follow-jumps                    [enabled]
  -fdce                                 [enabled]
  -fdefer-pop                           [enabled]
  -fdelete-null-pointer-checks          [enabled]
  -fdevirtualize                        [enabled]
  -fdse                                 [enabled]
  -fearly-inlining                      [enabled]
  -fexpensive-optimizations             [enabled]
  -fforward-propagate                   [enabled]
  -fgcse                                [enabled]
  -fgcse-after-reload                   [enabled]
  -fgcse-lm                             [enabled]
  -fguess-branch-probability            [enabled]
  -fhoist-adjacent-loads                [enabled]
```

```
[thibault.lejoux@localhost optimisations]$ gcc -Q -O0 --help=optimizers | grep enabled
  -faggressive-loop-optimizations       [enabled]
  -fasynchronous-unwind-tables          [enabled]
  -fbranch-count-reg                    [enabled]
  -fcommon                              [enabled]
  -fdce                                 [enabled]
  -fdelete-null-pointer-checks          [enabled]
  -fdse                                 [enabled]
  -fearly-inlining                      [enabled]
  -fgcse-lm                             [enabled]
  -finline                              [enabled]
  -finline-atomics                      [enabled]
  -fira-hoist-pressure                  [enabled]
  -fivopts                              [enabled]
  -fjump-tables                         [enabled]
  -fmath-errno                          [enabled]
  -fmove-loop-invariants                [enabled]
  -fpeephole                            [enabled]
  -fprefetch-loop-arrays                [enabled]
  -frename-registers                    [enabled]
  -frtti                                [enabled]
  -fsched-critical-path-heuristic       [enabled]
  -fsched-dep-count-heuristic           [enabled]
  -fsched-group-heuristic               [enabled]
  -fsched-interblock                    [enabled]
  -fsched-last-insn-heuristic           [enabled]
  -fsched-rank-heuristic                [enabled]
  -fsched-spec                          [enabled]
  -fsched-spec-insn-heuristic           [enabled]
  -fsched-stalled-insns-dep             [enabled]
  -fshort-enums                         [enabled]
  -fsigned-zeros                        [enabled]
  -fsplit-ivs-in-unroller               [enabled]
  -fno-threadsafe-statics               [enabled]
  -ftoplevel-reorder                    [enabled]
  -ftrapping-math                       [enabled]
  -ftree-coalesce-vars                  [enabled]
  -ftree-cselim                         [enabled]
  -ftree-forwprop                       [enabled]
  -ftree-loop-if-convert                [enabled]
  -ftree-loop-im                        [enabled]
  -ftree-loop-ivcanon                   [enabled]
  -ftree-loop-optimize                  [enabled]
  -ftree-phiprop                        [enabled]
  -ftree-pta                            [enabled]
  -ftree-reassoc                        [enabled]
  -ftree-scev-cprop                     [enabled]
  -ftree-slp-vectorize                  [enabled]
  -ftree-vect-loop-version              [enabled]
  -funit-at-a-time                      [enabled]
  -fvar-tracking                        [enabled]
  -fvar-tracking-assignments            [enabled]
  -fweb                                 [enabled]
```

2 Optimization

2.1 Basic operations

1.Measure the “atomic” runtime of basic operations (+, *, /, sqrt, ...):

```
[thibault.lejoux@localhost operations]$ gcc -lm operations-sum.c && time ./a.out
2.5

real    0m0.364s
user    0m0.364s
sys     0m0.000s
```

```
[thibault.lejoux@localhost operations]$ gcc -lm operations-mul.c && time ./a.out
0.5

real    0m0.359s
user    0m0.359s
sys     0m0.000s
```

2.Compare their runtime using different optimization flags:

```
[thibault.lejoux@localhost operations]$ gcc -O3  -lm operations-sum.c && time ./a.ou
-bash: ./a.ou: Aucun fichier ou dossier de ce type

real    0m0.001s
user    0m0.000s
sys     0m0.001s
```

With -03 compiler optimisation, the processing of the command is really faster (300* faster) than without it.

```
[thibault.lejoux@localhost operations]$ gcc -O3 -ffast-math  -lm operations-sum.c && time ./a.ou
-bash: ./a.ou: Aucun fichier ou dossier de ce type

real    0m0.001s
user    0m0.001s
sys     0m0.000s
```

Same result, really faster but with the flag -ffast-math the system time is now almost null.

2.2 Invariants

1) `-freciprocal-math`

2)

```
[thibault.lejoux@localhost operations]$ gcc -lm invariants.c && time ./a.out
4.7619
real    0m4.167s
user    0m4.166s
sys     0m0.001s
```

```
[thibault.lejoux@localhost operations]$ gcc -freciprocal-math  -lm invariants.c && time ./a.out
4.7619
real    0m4.089s
user    0m4.087s
sys     0m0.000s
```

The time does change juste a liitle bit of 0.1 second. But if we test this flag : -funsafe-math-optimizations we got an optimisation onf 0.4 second

```
[thibault.lejoux@localhost operations]$ gcc -funsafe-math-optimizations -lm invariants.c && time ./a.out
4.7619
real    0m3.826s
user    0m3.826s
sys     0m0.000s
```

3)

```c
#include <stdio.h>
#include <math.h>

int main (int argc, char * argv[])
{
  int i, j, k, niter=100000, ndim=4000;
  float a[ndim], b[ndim], c;

  /* initialisation: ne pas modifier */
  for (i=0; i<ndim; i++){
    a[i] = 1.0;
  }
  c = 0.1;
  float  cst =1/ 2.0*c + pow(c,2);
  /* la boucle sur k est la pour alourdir le calcul */
  for (k=0; k<niter; k++) {
    for (i=0; i<ndim; i++){
      b[i] = a[i]*cst;
    }
  }
  printf("%g", b[1]);
  return 0;
}
```

4)

```
[thibault.lejoux@localhost operations]$ gcc -lm invariants.c && time ./a.out
0.06
real    0m0.868s
user    0m0.868s
sys     0m0.000s
```

By making the calcul of the constant outside of the loop, the program is much faster (3.5 second)

2.3 Lookup tables

1) the run by default :

```
[thibault.lejoux@localhost operations]$ gcc -lm lookup.c && time ./a.out

real    0m2.403s
user    0m2.402s
sys     0m0.001s
```

Optimisation :

```c
#include <stdio.h>
#include <math.h>
int main (int argc, char * argv[])
{
  int i, j, k, niter=100000, ndim=1000;
  float a[ndim], b[ndim],a_tmp[ndim];

  /* initialisation: ne pas modifier */
  for (i=0; i<ndim; i++) {
    a[i] = 1.0;
  }

  /* partie à optimiser */
  for (i=0; i<ndim;i++){
    a_tmp[i]=exp(-i*a[i]);
  }
  for (k=0; k<niter; k++) {
    int k_mod = k % ndim;
    for (i=0; i<ndim; i++){
      b[i] = 1.0 - a[k_mod]* a_tmp[i];
    }
  }
  return 0;
}
```

2) it is 2second faster

```
[thibault.lejoux@localhost operations]$ gcc -lm lookup.c && time ./a.out

real    0m0.296s
user    0m0.294s
sys     0m0.002s
```

2.4 Indirect addressing

1)

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* macro qui controle la taille des tableaux */
#ifndef __N__
#define __N__ 400
#endif

/* http://c-faq.com/lib/randrange.html */
/* returns a random integer from 0 to n-1 */
int randrange(int n) {
  return rand() / (RAND_MAX / n + 1);
}

int main (int argc, char * argv[])
{

  int i, j, k;
  int N = __N__, niter=100000;
  float a[N], b[N], c;
  int aran[N];

  // initialisation: ne pas modifier
  for (i=0; i<N; i++) {
    a[i] = 1.0;
  }
  c = 0.5;
  for (i=0; i<N;i++){
     aran[i]=randrange(N);
  }
  /* la boucle sur k est la pour alourdir le calcul */
  for (k=0; k<niter; k++) {
    /* boucle principale */
    for (i=0; i<N; i++){
      b[i] = exp(a[aran[i]] + c);
    }
  }
  printf("%g", b[1]);
  return 0;
}
```

2) This is the time before and after indirect addressing :

```
[thibault.lejoux@localhost cache]$ gcc -lm random.c && time ./a.out
4.48169
real    0m0.780s
user    0m0.779s
sys     0m0.000s
[thibault.lejoux@localhost cache]$ nano random.c
[thibault.lejoux@localhost cache]$ gcc -lm random.c && time ./a.out
4.48169
real    0m0.823s
user    0m0.822s
sys     0m0.001s
```

So, with indirect addressing it is slower.

3)

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* macro qui controle la taille des tableaux */
#ifndef __N__
#define __N__ 1000
#endif

/* http://c-faq.com/lib/randrange.html */
/* returns a random integer from 0 to n-1 */
int randrange(int n) {
  return rand() / (RAND_MAX / n + 1);
}

int main (int argc, char * argv[])
{

  int i, j, k;
  int N = __N__, niter;
  float a[N], b[N], c;
  int aran[N];

  // initialisation: ne pas modifier
  for (i=0; i<N; i++) {
    a[i] = 1.0;
  }
  c = 0.5;
  for (i=0; i<N;i++){
     aran[i]=randrange(N);
  }
  /* la boucle sur k est la pour alourdir le calcul */
  N=5;
  niter=100000000/N;
  for (k=0; k<niter; k++) {
    /* boucle principale */
    for (i=0; i<N; i++){
      b[i] = exp(a[aran[i]] + c);
    }
  }
  printf("%g", b[1]);
  return 0;
}
```

N=5 :

```
[thibault.lejoux@localhost cache]$ gcc -lm random.c && time ./a.out
4.48169
real    0m2.055s
user    0m2.055s
sys     0m0.000s
```

N=50 :

```
[thibault.lejoux@localhost cache]$ gcc -lm random.c && time ./a.out
4.48169
real    0m2.064s
user    0m2.064s
sys     0m0.000s
```

N=999 :

```
[thibault.lejoux@localhost cache]$ gcc -lm random.c && time ./a.out
4.48169
real    0m2.049s
user    0m2.049s
sys     0m0.000s
[thibault.lejoux@localhost cache]$
```

Le temps d'execution ne varie pas beaucoup. La variation est probablement due aux nombre random.

2.5 Loop interchange

Avant :

```
[thibault.lejoux@localhost cache]$ gcc -lm interchange-ok.c && time ./a.out
0
real    0m0.292s
user    0m0.289s
sys     0m0.002s
```

Après :

```
// plain -> perf -> compile (increase niter) -> 00 / O3 -> ndim
#include <stdio.h>

int main (int argc, char * argv[])
{
  /* int i, j, m, niter=20000,ndim=200; */
  int i, j, m, niter=100,ndim=1000;
  float a[ndim][ndim], b[ndim][ndim];

   for (m=0; m<niter; m++) {
    for (i=0; i<ndim; i++){
      for (j=0; j<ndim; j++) {
        b[i][j] = a[j][i];
      }
    }
   }
   // sans ce printf, gcc v 4.8 optimise trop !
   printf("%g", b[1][1]);
   return 0;
```

```
[thibault.lejoux@localhost cache]$ gcc -lm interchange-ok.c && time ./a.out
0

real    0m0.259s
user    0m0.254s
sys     0m0.005s
```

In C it is more efficient when we have two loop i,j to parse in the order [i][j], C parse faster columns and after lines.

Whit the flag O3 it is much faster :

```
[thibault.lejoux@localhost cache]$ gcc -O3  -lm interchange-ok.c && time ./a.out
0

real    0m0.074s
user    0m0.072s
sys     0m0.002s
```

2.5 Loop fusion

Run time of fusion.c and fusion-ok.c :

```
[thibault.lejoux@localhost cache]$ gcc -lm fusion-ok.c && time ./a.out

real    0m0.871s
user    0m0.870s
sys     0m0.001s
[thibault.lejoux@localhost cache]$ gcc -lm fusion.c && time ./a.out

real    0m1.003s
user    0m1.002s
sys     0m0.001s
```

Fusion-ok.c is 13% faster than fusion.c

In fusion.c, in the inter loop we have an another same loop. It is exactly the same loop than the second, so in fusion-ok.c they delete the third loop and put the operation in it in the second loop. That is why it is faster in fusion-ok.c

Screen of fusion.c :

```c
// localité temporelle
#include <stdio.h>
#include <math.h>
int main (int argc, char * argv[])
{
  int i, j, k, niter=10000, ndim=20000;
  float a[ndim], b[ndim], c[ndim];

  for (i=0; i<ndim; i++){
    a[i] = 1.0;
  }
  for (k=0; k<niter; k++) {
    for (i=0; i<ndim; i++){
      b[i] = 2*a[i];
    }
    for (i=0; i<ndim; i++){
      c[i] = sqrt(b[i]);
    }
  }
  return 0;
}
```

Screen of fusion-ok.c :

```c
// localité temporelle //
#include <stdio.h>
#include <math.h>
int main (int argc, char * argv[])
{
  int i, j, k, niter=10000, ndim=20000;
  float a[ndim], b[ndim], c[ndim];

  for (i=0; i<ndim; i++){
    a[i] = 1.0;
  }
  for (k=0; k<niter; k++) {
    for (i=0; i<ndim; i++){
      b[i] = 2*a[i];
      c[i] = sqrt(b[i]);
    }
  }
  return 0;
}
```

3. Loop interchange

the first loop line 9 has been vectorized :

```
Vectorizing loop at forward.c:9

forward.c:9: note: === vec_transform_loop ===
```

And the loop line 20 has been vectozied too :

```
Vectorizing loop at forward.c:20

forward.c:20: note: === vec transform loop ===
```

With vectorization :

```
forward.c:28: note: not vectorized: failed to find SLP opportunities in basic block.
1

real    0m0.481s
user    0m0.480s
sys     0m0.001s
```

Without :

```
[thibault.lejoux@localhost vectorization]$ gcc -lm  forward.c && time ./a.ou
l

real    0m2.291s
user    0m2.290s
sys     0m0.001s
```

So it is much faster with vectorization.

3.1 Unrolling

With unrolling.c :

```
[thibault.lejoux@localhost other]$ gcc -O1 -funroll-loops unrolling.c && time ./a.out

real    0m0.318s
user    0m0.317s
sys     0m0.001s
```

With the other unrolling, we don’t have the same performance , the others are much better:

```
[thibault.lejoux@localhost other]$ gcc -O1 -funroll-loops unrolling-2.c && time ./a.out

real    0m0.168s
user    0m0.168s
sys     0m0.000s
[thibault.lejoux@localhost other]$ gcc -O1 -funroll-loops unrolling-4.c && time ./a.out

real    0m0.100s
user    0m0.099s
sys     0m0.001s
[thibault.lejoux@localhost other]$ gcc -O1 -funroll-loops unrolling-6.c && time ./a.out

real    0m0.068s
user    0m0.067s
sys     0m0.000s
```

The construction is totally different :

```
---
>                 (compare:CCZ (plus:SI (reg:SI 0 ax [orig:70 D.3380 ] [70])
26,27c25,26
<             (set (reg:SI 1 dx [80])
<                 (plus:SI (reg:SI 1 dx [orig:70 D.3380 ] [70])
---
>             (set (reg:SI 0 ax [orig:70 D.3380 ] [70])
>                 (plus:SI (reg:SI 0 ax [orig:70 D.3380 ] [70])
29,47c28
<         ]) unrolling-ok.c:10# {*addsi_2}
<      (expr_list:REG_UNUSED (reg:CCZ 17 flags)
<         (nil)))
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (note# 0 0 NOTE_INSN_DELETED)
< (insn# 0 0 3 (parallel [
<             (set (reg:CCZ 17 flags)
<                 (compare:CCZ (plus:SI (reg:SI 1 dx [80])
<                         (const_int -7 [0xfffffffffffffff9]))
<                     (const_int 0 [0])))
<             (set (reg:SI 1 dx [orig:70 D.3380 ] [70])
<                 (plus:SI (reg:SI 1 dx [80])
<                     (const_int -7 [0xfffffffffffffff9])))
<         ]) unrolling-ok.c:10# {*addsi_2}
---
>         ]) unrolling.c:10# {*addsi_2}
53c34
<             (pc))) unrolling-ok.c:10# {*jcc_1}
---
>             (pc))) unrolling.c:10# {*jcc_1}
62c43
<                 (compare:CCZ (plus:SI (reg:SI 0 ax [orig:72 D.3380 ] [72])
---
>                 (compare:CCZ (plus:SI (reg:SI 1 dx [orig:72 D.3380 ] [72])
65,66c46,47
<             (set (reg:SI 0 ax [orig:72 D.3380 ] [72])
<                 (plus:SI (reg:SI 0 ax [orig:72 D.3380 ] [72])
---
>             (set (reg:SI 1 dx [orig:72 D.3380 ] [72])
>                 (plus:SI (reg:SI 1 dx [orig:72 D.3380 ] [72])
68c49
<         ]) unrolling-ok.c:9# {*addsi_2}
---
>         ]) unrolling.c:9# {*addsi_2}
74c55
<             (pc))) unrolling-ok.c:9# {*jcc_1}
---
>             (pc))) unrolling.c:9# {*jcc_1}
81c62
<         (const_int 5000 [0x1388])) unrolling-ok.c:9# {*movsi_internal}
---
>         (const_int 5000 [0x1388])) unrolling.c:9# {*movsi_internal}
91,92c72,73
< (insn# 0 0 6 (set (reg:SI 1 dx [orig:70 D.3380 ] [70])
<         (const_int 2048 [0x800])) unrolling-ok.c:5# {*movsi_internal}
---
> (insn# 0 0 6 (set (reg:SI 0 ax [orig:70 D.3380 ] [70])
>         (const_int 2048 [0x800])) unrolling.c:5# {*movsi_internal}
102d82
```

3.2 Inlining

No it does not inline, it says that we have a problem with the function « some_function » which is in red. But after executing all the command one more time after replacing the void function by a static inline function it work :

```
[thibault.lejoux@localhost other]$ gcc -lm -O1 -finline-functions -fdump-final-insns=dump -finline-limit=10000000  inlining.c
[thibault.lejoux@localhost other]$ grep some_function dump
[thibault.lejoux@localhost other]$
```

There is no error.

3.3 Inlining

1)

With __D__ < 0 there are no data dependancy (IN(A) inter IN(B) is null, there are input dependancy (IN(A) inter out(B) is not null) and there is no output dependancy (out(A) inter out(B) is null)

With __D__>0 there are data dependancy, there are no input dependancy and no out dependancy.

2)

With __D__> 0 :

```
dependance.c:9: note: LOOP VECTORIZED.
dependance.c:7: note: vectorized 1 loops in function.
```

```
dependance.c:19: note: LOOP VECTORIZED.
dependance.c:14: note: vectorized 2 loops in function.
0
```

With a data dependancy we can still vectorize because we write after we read.

With __D__< 0 :

```
dependance.c:9: note: === vect_analyze_dependences ===
dependance.c:9: note: dependence distance  = 0.
dependance.c:9: note: dependence distance == 0 between *_8 and *_8
dependance.c:9: note: dependence distance  = 1.
dependance.c:9: note: not vectorized, possible dependence between data-refs *_11 and *_8
dependance.c:9: note: bad data dependence.
dependance.c:7: note: vectorized 0 loops in function.
```

With a input dependancy we can not vectorized beacause a variable is read after its value has been modified by a previous loop iteration.

When |__D__|>4 the loop is always vectorizable :

```
dependance.c:9: note: LOOP VECTORIZED.
dependance.c:7: note: vectorized 1 loops in function.

Analyzing loop at dependance.c:24
```

3)

	float	double
vectorization	real 0m0.389s user 0m0.388s sys 0m0.000s	dependance.c:27: n 0 real 0m0.522s user 0m0.522s sys 0m0.000s
No vectorization	real 0m2.393s user 0m2.393s sys 0m0.000s	[thibault.lejoux@lo 1.38524e-309 real 0m44.049s user 0m44.047s sys 0m0.000s

A double variable have more precision than a float but are heavier. It result that without vectorization the operation is very long compare to the float without vectorization. But with vectorization it is still longer but not as much as without vectorization.

3.4 Branching

Before :

```
[thibault.lejoux@localhost vectorization]$  gcc if.c && time ./a.out
2
real    0m4.337s
user    0m4.335s
sys     0m0.001s
```

After modifcation :

```
[thibault.lejoux@localhost vectorization]$  gcc if-ok.c && time ./a.out
2
real    0m2.648s
user    0m2.646s
sys     0m0.002s
```

```c
#include <stdio.h>

void f(int ndim,float* d, float* a, float* b, float* c) {
  int i;
  for (i=0; i<ndim-1; i++){
    if (d[i]>0.) {
      c[i] = a[i] / d[i];
    } else {
      c[i] = 0.0;
    }
  }
}

int main (int argc, char * argv[])
{
  int i, j, k, niter=200000, ndim=4000;
  float a[ndim], b[ndim], c[ndim], d[ndim], exponent;

  for (i=0; i<ndim; i++) {
    a[i] = 1.0;
    b[i] = 2.0;
    d[i]=(a[i] + a[i+1]) / (b[i] * b[i+1]);
  }

  for (k=0; k<niter; k++) {
    f(ndim,d, a, b, c);
  }
  printf("%g", b[0]);
  return 0;
}
```

With the previous code, d was replaced ndim times, instead of that it's better to put it in an other loop, make it an array. (with the vectorization we got better result for both but the second code is still better)

4 Gprof and SLURM

4.1 Using Gprof

```
[thibault.lejoux@localhost vectorization]$ gcc -pg -o dependance.x dependance.c
[thibault.lejoux@localhost vectorization]$ ls
a.out  backward-data.c  compile.sh  dependance.c  dependance_double.c  dependance.x
```

```
[thibault.lejoux@localhost vectorization]$ ./dependance.x
2
```

```
[thibault.lejoux@localhost vectorization]$ ls -l gmon.out
-rw-------. 1 thibault.lejoux thibault.lejoux 602  4 avril 16:35 gmon.out
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.78      2.34     2.34   200000    11.69    11.69  f

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
^L
Copyright (C) 2012-2016 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
^L
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.43% of 2.34 seconds

index % time    self  children    called     name
                2.34    0.00  200000/200000      main [2]
[1]    100.0    2.34    0.00  200000         f [1]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00    2.34                 main [2]
                2.34    0.00  200000/200000      f [1]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
:
```