Tina Memo No. 2016-001
Internal

An Exercise in C Code Optimisation.

Neil Thacker

Last updated
13 / 01 / 2016

Centre for Imaging Sciences,
Medical School, University of Manchester,
Stopford Building, Oxford Road,
Manchester, M13 9PT.

An Exercise in C Code Optimisation.

N. A. Thacker 13/1/16

abstract

PhD projects are often limited by how quickly large data sets from complex experiments can be analysed by C software developed by the student. Our laboratory has many equivalent Linux machines each with multiple processing cores. One easy way of running data analysis quickly is to run up to N copies of a program, T times on M machines. This can normally be done over an evening when other researchers have gone home, using scripts. In our group a factor of $N \times T \times M = 338$ *is achievable for a 2 hour run time. We may consider moving the software to a purpose built multi-processor farm. However, the effort involved in doing this (perhaps a months work) should not be underestimated (Appendix C). Also parallel systems need specialised tools to debug and modify software (different to those used during initial development). Alternatively, we can try to make the software faster on the machines it was developed on.*

This document describes a real example of software optimisation (a pharmokinetic fitting model requiring two convolutions) and is intended as a technical reference for students. It demonstrates some standard methods and things to look out for. Code execution times were measured using

```
gprof -pg myprog
```

(see Appendix A) and all software was compiled with the **- O2** *option. The quoted speed increases are therefore obtained after any benefits of compiler optimisation. The document concludes with a timing analysis (Appendix B).*

Disclaimer

This work was undertaken prior to recent PC hardware changes which have resulted in heavy penalties on data access. Unless the currently unfavourable ratio of on chip to data bus clock speed (1:40) is redressed then the use of look-up tables, and methods such as loop expansions may not provide the benefits shown. Whilst the basic idea of using gprof to assess code changes are still valid, programmers will also need to consider, and potentially optimise for, RAM access.

Stating Point

The starting point for this software is shown below. This piece of C software (taken from **ther_conc1()**) has been identified by **gprof** (Appendix A) as that which initially takes up 98.5% of execution time. It had already been partly optimised by the student (note precomputed **speed** variables), who had achieved perhaps a **3x** increase in efficiency over the original version (including **-O2**), but wanted to know if it was possible to get more.

```
double A,B,*part,*partp;
double speedy1, speedy2;
float *conc1_ther,tscale,tt;
float sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1,tau,k,i;
.
.  .  missing code . .
.
for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         for(i=tina_int(tt), tau=0; i<= k; i++, tau++)
         {
              if (i==k)
              {
                  sumA+= 0.5*part[speedtt+tau]*exp(speedy1*(speedt-tau));
                  sumB+= 0.5*part[speedtt+tau]*exp(speedy2*(speedt-tau));
              }
              else
              {
                  sumA+= part[speedtt+tau]*exp(speedy1*(speedt-tau));
                  sumB+= part[speedtt+tau]*exp(speedy2*(speedt-tau));
              }
         }

     }
     conc1_ther[k]=(A*sumA+B*sumB)*tscale;
}
```

To the experienced eye, inspection of the code is enough to identify several problems. The inner **for** loop is the one which takes up all the processing time, so we wish to make this execute as quickly as possible. Currently it contains; array indexing (**a**[**i**]), transcendental calculations (**exp()**), use of a mixed variety of variable types and a rather inefficient **if** statement. After reading this report you should start to understand why these should be avoided, and how this can be done. Ultimately all software must be readable by others if it is is to be maintained. As we will see, optimised software can tend to be more difficult to read and numerical issues may also be introduced. Optimisation must therefore be balanced to some extent with pragmatism, but for the purposes of this document we will try to establish the ultimate limit of performance.

1st Stage Optimisation : precision and arrays

We start here with arrays and the declaration of variables. Indexing an array requires many clock cycles of the CPU, we need to compute the index value and then convert this to a pointer in order to locate the data in memory. This can be done more efficiently by stepping through an array with the **array++** mechanism, the actual balance of these costs will be estimated in Appendix B later.

In this piece of code the variable declarations **float sumA, sumB** are forcing the inner calculations to convert (cast) between the native double precision and the summation variables. This is also very slow, but worse unnecessary, and can be prevented by declaring them instead as **double**, with significant speed increase. Eliminating the **if** is a simple matter of recoding. Array indexing can be eliminated by use of an array pointer with **part[speedtt+tau]** being replaced by ***partp** in the inner loop.

The following will generate an approximate **3x** increase in execution speed (Actual timing figures are given in table 1, Appendix B).

```
double A,B,*part,*partp;
double speedy1, speedy2;
float *conc1_ther,tscale,tt;
double sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1,tau,k,i;

.
.  .  missing code . .
.

for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         partp = &part[speedtt];

         for(i=tina_int(tt), tau=0; i<k; i++, tau++ )
         {
             sumA+= (*partp)*exp(speedy1*(speedt-tau));
             sumB+= (*partp)*exp(speedy2*(speedt-tau));
             partp++;
         }
         sumA+= 0.5*(*partp)*exp(speedy1*(speedt-tau));
         sumB+= 0.5*(*partp)*exp(speedy2*(speedt-tau));

         conc1_ther[k]=(A*sumA+B*sumB)*tscale;
     }
     else
          conc1_ther[k]= 0.0;
}
```

The execution time is now dominated by the loop size **n2-n1**, which was 180 for these tests, and we can do little about. The obvious remaining problem is the use of **exp()** and its various contents, which we will now attend to.

2nd Stage Optimisation : mathematical refactoring

By splitting the exponential into two pieces, we can replicate the changing calculation using one initial exponential (outside the main loop) and some repeated multiplications. Although the code line count has increased, this provides a further **3x** increase in speed. Faster software is not always shorter.

```
double A,B,*part,*partp;
double speedy1, speedy2;
double expfac1,expfac2;
double mult1,mult2;
float *conc1_ther,tscale,tt;
double sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1, k,i;

.
.  .  missing code . .
.

for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         partp = &part[speedtt];
         mult1 = exp(-speedy1);
         mult2 = exp(-speedy2);
         expfac1 = exp(speedy1*speedt));
         expfac2 = exp(speedy2*speedt));

         for(i=tina_int(tt); i<k; i++ )
         {
             sumA+= (*partp)*expfac1;
             sumB+= (*partp)*expfac2;
             expfac1 *= mult1;
             expfac2 *= mult2;
             partp++;
         }
         sumA+= 0.5*(*partp)*expfac1;
         sumB+= 0.5*(*partp)*expfac2;

         conc1_ther[k]=(A*sumA+B*sumB)*tscale;
     }
     else
          conc1_ther[k]= 0.0;
}

.
.
```

Notice the **tau** variable is no longer necessary and has been removed. Looking at the resulting code we can see that the factors **mult1** and **mult2** are always the same for all values of **k**. It would be nice to precompute these and so eliminate the multiplications **expfac1 *= mult1** from the inner loop.

3rd Stage Optimisation : reorganisation

```
double A,B,*part,*partp;
double speedy1, speedy2;
double expfac1,expfac2, expfac1_init,expfac2;_init;
double mult1,mult2;
float *conc1_ther,tscale,tt;
double sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1,k,i;
.
.  .  missing code . .
.
for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         partp = &part[speedtt];
         mult1 = exp(-speedy1);
         mult2 = exp(-speedy2);
         if ((expfac1_init = exp(speedy1*speedt)) > 0.0)
         {
             expfac1 = 1.0;
         }
         else
             expfac1 = 0.0;

         if ((expfac2_init = exp(speedy2*speedt)) > 0.0)
         {
             expfac2 = 1.0;
         }
         else
             expfac2 = 0.0;


         for(i=tina_int(tt); i<k; i++ )
         {
             sumA+= (*partp)*expfac1;
             sumB+= (*partp)*expfac2;
             expfac1 *= mult1;
             expfac2 *= mult2;
             partp++;
         }
         sumA+= 0.5*(*partp)*expfac1;
         sumB+= 0.5*(*partp)*expfac2;

         conc1_ther[k]=(A*sumA*expfac1_init+B*sumB*expfac2_init)*tscale;
     }
     else
          conc1_ther[k]= 0.0;
}
.
```

.

This gives no additional improvement in speed. There is actually a potential numerical problem, as one very large exponential factor must be computed and multiped by another very small one. This raises the possibility that **sumA** or **sumB** might reach numerical zero or infinity. But in this form we see that the computed **expfac1** and **expfac2** can now be replaced by two fixed arrays (as **speedy1** and **speedy2** are fixed values for all **k**).

4th Stage Optimisation : more pointer increments

Both the data and the exponential factors can now be accessed via pointer indexing **(*mult1p++)*(*partp)**. A further **1.35x** increase in speed was measured.

```
double A,B,*part,*partp;
double speedy1, speedy2;
double expfac1,expfac2;
double expfac1_init,expfac2;_init
double mult1,mult2,nullmul;
double mult1p,mult2p;
float *conc1_ther,tscale,tt;
double sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1,k,i;
.
.
.  .  missing code . .
.
for (k=0;k<n2-n1;k++)
{
    mult1[k] = exp(-speedy1*k);
    mult2[k] = exp(-speedy2*k);
    nullmult[k] = 0.0;
}

for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         partp = &part[speedtt];
         mult1p = &nullmult[0];
         mult2p = &nullmult[0];
         if ((expfac1_init = exp(speedy1*speedt)) > 0.0)
         {
             mult1p = &mult1[0];
         }

         if ((expfac2_init = exp(speedy2*speedt)) > 0.0)
         {
             mult2p = &mult2[0];
         }

         for(i=tina_int(tt); i<k; i++ )
```

```
          {
              sumA+= (*mult1p++)*(*partp);
              sumB+= (*mult2p++)*(*partp++);
          }

          sumA+= 0.5*(*partp)*(*mult1p);
          sumB+= 0.5*(*partp)*(*mult2p);

          conc1_ther[k]=(A*sumA*expfac1_init+B*sumB*expfac2_init)*tscale;
      }
      else
           conc1_ther[k]= 0.0;
}
.
.
```

5th Stage Optimisation: loop expansion

The resulting inner loop is now at such a low computational cost that a significant proportion of the execution time (5/13) is actually taken up by the **for** construct (see Appendix B). We can halve this, giving a measured **1.18x** speed increase, by expanding the loop. A little book keeping is required to deal with the **+=2** step. We could expand the loop further but as the loop construct only accounts for 38% of the execution time, the increase in loop speed will be limited to a maximum of **1.625x**.

```
double A,B,*part,*partp;
double speedy1, speedy2;
double expfac1,expfac2,expfac1_init,expfac2;_init;
double mult1,mult2,nullmul;
double mult1p,mult2p;
float *conc1_ther,tscale,tt;
double sumA, sumB, a ,b;
int    speedtt, speedt;
int   n1,n1,k,i;
.
.  .  missing code  .  .
.
for (k=0;k<n2-n1;k++)
{
    mult1[k] = exp(-speedy1*k);
    mult2[k] = exp(-speedy2*k);
    nullmult[k] = 0.0;
}


for(k=n1; k<n2; k++)
{
    t = k-tt;
    speedt = tina_int(t);
    sumA=0.0;
    sumB=0.0;
    if (t>=0.0 )
    {
         partp = &part[speedtt];
         mult1p = &nullmult[0];
         mult2p = &nullmult[0];
```

```
        if ((expfac1_init = exp(speedy1*speedt)) > 0.0)
        {
            mult1p = &mult1[0];
        }

        if ((expfac2_init = exp(speedy2*speedt)) > 0.0)
        {
            mult2p = &mult2[0];
        }

        for(i=tina_int(tt); i<k-1; i+=2 )
        {
            sumA+= (*mult1p++)*(*partp);
            sumB+= (*mult2p++)*(*partp++);
            sumA+= (*mult1p++)*(*partp);
            sumB+= (*mult2p++)*(*partp++);
        }
        if (i!=k)
        {
            sumA+= (*mult1p++)*(*partp);
            sumB+= (*mult2p++)*(*partp++);
        }

        sumA+= 0.5*(*mult1p)*(*partp);
        sumB+= 0.5*(*mult2p)*(*partp);

        conc1_ther[k]=(A*sumA*expfac1_init+B*sumB*expfac2_init)*tscale;
    }
    else
         conc1_ther[k]= 0.0;
}
.
.
```

Although this is a modest improvement in this case, in other software the **for** loop can be as much as half the number of machine clock cycles. So loop expansions are worth knowing about. At the same time however, small loops are often expanded automatically by the compiler, making this process unnecessary.

Once we have reached the limit of the number of clock cyles for the computation increasing the speed of this software further by simply modifying individual terms is not possible. To go any faster we would need something more radical, like removing the inner loop entirely. We will do this next.

6th Stage Optimisation: recursive filtering

If we go back to the 4th stage and inspect the inner loop, something interesting becomes apparent (you may need to step through the program with a debugger such as **ddd**). For each successive **k** each inner loop calculation is identical to the previous one, except for having one extra increment of the **sumA** and **sumB** variables. This suggest a recursive way to compute these factors.

```
  double A,B,*part,*partp;
  double speedy1, speedy2;
  double expfac1,expfac2,expfac1_init,expfac2;_init;
  double mult1,mult2;
  double mult1p,mult2p;
  float *conc1_ther,tscale,tt;
  double sumA, sumB, a ,b;
  int    speedtt, speedt;
  int   n1,n1,k,i;
  .
  .  .  missing code  .  .
  .
  for (k=0;k<n2-n1;k++)
  {
      mult1[k] = exp(-speedy1*k);
      mult2[k] = exp(-speedy2*k);
  }

  partp = &part[speedtt];
  mult1p = &mult1[0];
  mult2p = &mult2[0];
  sumA = 0.5*(*mult1p)*(*partp);
  sumB = 0.5*(*mult2p)*(*partp);

  for(k=n1; k<n2; k++)
  {
      t = k-tt;
      speedt = tina_int(t);
/* dont reset variables again, continue from previous NAT 2/1/2016 */

      if (t>=0.0 )
      {
/* new code here NAT 1/1/2016
*/
           expfac1_init = exp(speedy1*speedt);
           expfac2_init = exp(speedy2*speedt);

/* what is left of original inner loop NAT 2/1/2016 */
           sumA+= 0.5*(*mult1p++)*(*partp);
           sumB+= 0.5*(*mult2p++)*(*partp++);

           sumA+= 0.5*(*mult1p)*(*partp);
           sumB+= 0.5*(*mult2p)*(*partp);

           conc1_ther[k]=(A*sumA*expfac1_init+B*sumB*expfac2_init)*tscale;
       }
       else
            conc1_ther[k]= 0.0;
  .
  .
```

The software above passes the previous summed quantities on to the next loop, so that all but the final iteration of the inner loop are removed. We find here that the original implementation is consistent with a linear interpolation between original data points (***partp**), which introduces unwanted smoothing. Otherwise, what we have just discovered is a modified version of the linear recursive filter implementation for an exponential convolution ($o_i = exp(-\alpha)o_{i-1} + d_i$). As a consequence the resulting speed increase over the 4th Stage is a factor of **3.6x**. Also, the execution time for this piece of software is now only linearly (rather than quadratically) dependant upon the array length (**n2 - n1**).

The final step is now to eliminate the linear interpolation and recover the numerical accuracy lost earlier. We can do this in a variety of ways (including the use of long double variables if necessary), but the best way is to replace the **sumA** and **sumB** calculations with the recursive filter equations

```
sumA = (sumA*exp(speedy1)) + *partp;
sumB = (sumB*exp(speedy2)) + *partp++;
conc1_ther[k] =(A*sumA+ B*sumB)*tscale;
```

Several unnecesary variables can now be removed entirely. If we wish, we could now go on to further optimise the program and achieve perhaps another factor of two increase of efficiency, but this will involve other parts of the software and not attempted here.

Conclusions

Much of software optimisation is common sense. However, at the beginning of this document I mentioned that this example started from a previous attempt by a student to optimise the software, which had already led to a marginal improvement. We must therefore assume that the methods used are not generally known by novice programmers. So what have we learned?

- Use a program such as **gprof** to understand your software.
- Concentrate your effort on the innermost loops of the slowest functions.
- Don't mix **float** and **double** variables. Just use **double**, it will give better numerical precision and avoids unnecessary casting.
- Don't use conventional array indexing in computationally intensive parts of code, use pointer increments.
- Use the logic of the program to eliminate unnecessary calculations.
- Use mathematics to simplify and eliminate transcendental functions where possible.
- Sometimes, rewrites and reorganisation do not increase speed but make further optimisation possible.

Undoubtedly it helps a little if you know how a micro-processor works. But in all cases the consequences of even the simplest change can be tested using **gprof**, and in this way anyone can learn to optimise software.

In the test case provided, combining all of these processes and some general tidying gave a measured speed increase of **42.5x**. This factor increases to **50x** if we did not use the **-O2** compiler flag (**120x** including the students original work). Although this is typical of my experience of C code optimisation, I have also seen examples of factors of **1000x** improvement when the original code was written without giving much thought to efficiency. Optimisation is therefore worth knowing about if you wish to get your PhD finished on time.

Although ultimately the initial optimisation work is found to be unnecessary, it led on to the development of the recursive calculation, and provided information on the way which can be useful to us in future. The biggest surprise here was probably the change of **float sumA, sumB** to **double sumA, sumB**, which was needed to get any real improvement in speed.

Compare the initial code

```
sumA+= part[speedtt+tau]*exp(speedy1*(speedt-tau));
sumB+= part[speedtt+tau]*exp(speedy2*(speedt-tau));
```

with the 4th Stage version

```
sumA+= (*mult1p++)*(*partp);
sumB+= (*mult2p++)*(*partp++);
```

and it should be clear where most of the factor of **5x** difference between them comes from. Both **array[i]** and **exp()** require many CPU clock cycles, not to mention the 4 extra additions. Removing the **if** and expanding the **for** loop (and removing **tau**) accounts for the rest. All of the other software changes are present solely to allow the inner loop to be written in this new form.

In principle, there is enough information from **gprof** to compare the computation efficiency seen with the specified clock frequency of the processor. In my experience the observed execution time will not be very close to the value obtained by simply dividing the number of executed instructions by this value. Micro-processor manufacturers tend to specify the frequency of the fastest part of the microprocessor circuit, which may differ by factors of between 4 and 8 from the execution rate of single machine code instructions. Under these circumstances it is far better to calibrate the expected frequency to a known quantity, such as a **for** loop, as in Appendix B.

It should be noted that **gprof** is reproducible to around a few percent, so not only is it more practical to start on the slowest routines, but also it might be difficult to see small improvements in any others.

Additional execution speed can also be obtained relatively simply by using parallelism (using methods such as **OpenMP**), but we can generally achieve factors of **x5** just by running multiple program copies (under Linux). Even **OpenMP** will not speed up data processing by more than the number of processor cores. Parallel processing farms normally use batch systems, so you also have to wait for code to start running. Given the choice of parallelisation or code optimisation I know which I choose. The current example was completed within just 1 full day of work. If processing is still too slow following optimisation then we might need to consider strategies for additional paralellisation see Appendix C.

One final caution. In the process of rewriting the mathematical form of calculations there are inevitable changes in numerical precision. In the case provided the numerical range of the calculations is reduced at stage 3. Optimised software should always be **tested** to ensure that the result is still usable.

Appendix A: Typical "gprof -pg myprog" Output

gprof is a general software profiler, which must be used by compiling the relevant C file using the "-pg" option. This is done by adding "-pg" to the list of C flags found in the "Makefile". The final output of **gprof**, following all code optimisation, is as follows. Note the initial **98.5 %** percentage of **ther_conc1()** has been reduced to **60.1 %**.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 60.13      5.40     5.40   820297     0.00     0.00  ther_conc1
 27.62      7.88     2.48      528     0.00     0.00  covariance
  4.34      8.27     0.39                             tina_int
  3.45      8.58     0.31   945617     0.00     0.00  perm_chi2
  2.45      8.80     0.22                             cmplx_cis
  1.11      8.90     0.10                             amotry
  0.45      8.94     0.04                             amoeba
  0.11      8.95     0.01      528     0.00     0.00  my_weight
  0.11      8.96     0.01                             nvector_alloc
  0.11      8.97     0.01                             ralloc_free_blocked
  0.11      8.98     0.01                             tv_image2
  0.00      8.98     0.00      528     0.00     0.00  Enf
  0.00      8.98     0.00      528     0.00     0.00  aif_est
  0.00      8.98     0.00      528     0.00     0.00  conv_st_to_conc1
  0.00      8.98     0.00      528     0.00     0.00  perm_fit_pixel
  0.00      8.98     0.00      528     0.00     0.00  perm_simplex
  0.00      8.98     0.00        2     0.00     0.00  pfit_get_image
```

```
  0.00      8.98     0.00        1     0.00     0.00  AIFtscale_get
  0.00      8.98     0.00        1     0.00     0.00  alpha_get
  0.00      8.98     0.00        1     0.00     0.00  pfit_get_SEimage
  0.00      8.98     0.00        1     0.00     0.00  pfit_get_corr_image
  0.00      8.98     0.00        1     0.00     2.50  pfit_measure_image
  0.00      8.98     0.00        1     0.00     0.00  prebolus_get
  0.00      8.98     0.00        1     0.00     0.00  r1cont_get
  0.00      8.98     0.00        1     0.00     0.00  set_perm_con
  0.00      8.98     0.00        1     0.00     0.00  sigma_get
  0.00      8.98     0.00        1     0.00     0.00  te_get
  0.00      8.98     0.00        1     0.00     0.00  tr_get


 %         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.
```

Appendix B: Timing Analysis

It is instructive to try to interpret our timing results in terms of costs of individual expressions. This way we can develop some understanding of where improvements have been gained and better optimise future software. Although we may have some general expectations, the results will vary depending upon the processor and compiler, which will inevitably continue to change with new technology.

By taking the various timings of these experiments and a characterisation of each software version in terms of C construct counts, we can attempt to estimate the timings of the various expressions within the inner **for** loop (Table 1).

While the results are only approximate, the cost of array indexing and exponential functions (far greater than multiplication and addition) is evident.

Although there is not a lot of freedom in the numbers to allow this, the **exp()** timing may contain a hidden cast, as the initial input variables were mixtures of **float** and **int**. I have assumed that the compiler optimisation has done something sensible to remove this. Casting upwards is always something a compiler can do during optimisation, but it

expression	original	1st Stage	2nd Stage	4th Stage	4th Stage'	5th Stage	6th Stage	time(s)
a+b	6	4	2	2	2	2	0	4.57
a++	2	3	2	4	4	3.5	0	-1.42
a*b	4	4	4	2	2	2	0	4.57
exp()	2	2	0	0	0	0	0	40.14
for()	1	1	1	1	1	0.5	0	11.41
if()	1	0	0	0	0	0	0	11.41
a[i]	2	0	0	0	0	0	0	105.4*
(float)a	2	0	0	0	2	0	0	24.5
time(s)	383	133	45	33	82	28	9	-

Table 1: *These timings are for total execution time of the software as listed above 4t Stage' has the original "float" definition for the* **sumA** *and* **sumB** *variables. The* **for()** *timing excludes its ++ increment, which is counted separately. For this analysis* **if()** *is assumed to take the same time as* **for()**, *and* **a+b** *is assumed to take the same time as* **a*b**. *The column of estimated time costs is generated by solving 6 linear equations for the 6 unknowns. Given the limitations of* **gprof**, *the numbers provided are not expected to be better than 5% accurate.*

cannot cast down without compromising the programmers original intentions. If software is written to force a **double** to **float** conversion (either explicity in the line of code or implicitly via variable definition), then the compiler must honour this, with commensurate processing cost [1].

The increment construct (**a++**), which cannot have negative cost, must be interpreted here as cost free. Indeed we cannot explain the large time reduction between the 2nd and 4th stages any other way! While this sounds like magic, it is possible that the CPU has been designed to increment addresses in parallel with the main function (e.g. add or multiply). Logically, this would not be possible for **++a** as data cannot be processed before it has been loaded into the CPU.

The time for an array increment (**a[i]** marked *) is probably an overestimate, due to un-modelled changes between the original code and the 1st Stage. Luckily the other values are unaffected. I would have expected this to be much less than 20 clock cycles (but we now live in a world with 64 bit addressing!). One thing can be concluded for certain; **Don't use array indexing in a time critical part of the software!**[2]

On a conventional CPU the actual execution times must be multiples of some common factor, due to the clocking of the digital circuitry. We can expect that an integer based **for()** loop requires a minimum of 5 clock cycles on the CPU (2 for subtraction, 1 for sign test and two jumps, one into the code block and one back to the test). This allows us to estimate the number of clock cycles for the other constructs. At the 4th stage of optimisation we see that the **for()** cost is consistent with being 38 % of the execution time. The predicted 4th to 5th Stage timing reduction, taken from the ratios of nominal clock values ($13/10.5 = 1.23$), is very close ($(33\text{-}9)/(28\text{-}9) = 1.26$) and within measurement error.

Appendix C: Arguments for Paralellisation

If we have written software for a single machine, at execution times of several hours, it becomes necessary to consider ways of increasing analysis turn around. We can of course look to move to a dedicated parallel system, but if this is just a one off occurence (often towards the end of a PhD) we can consider instead running programs on local resources over-night. This requires a little co-ordination and planning, but once we have decided to take this route we can often get access to multiple machines (e.g. 8) in a typical computing laboratory for an uninterupted 14 hours.

My own laboratory of 8 machines has been assembled with this possbility in mind. With sensibly machine procurement we can have compatible OS and multiple processors on each machine. The homogeneity of machines, Linux OS and 6 multi-cores also gives a flexible working platform for code development during the day. In other words, this system can

[1] We have hit here on the reason that the optimisation methods described in this document are not already built into the compilers. Compiler optimisation cannot make trade-offs between numerical accuracy and speed. Any compiler optimisation has to guarantee the user specified calculation precisions, however silly they may be.

[2] Again, we can ask why the compiler optimisation doesn't do this for us. Historically, strange as this may sound now, it was because arrays were not guaranteed to be contiguous blocks of memory, precluding use of the ++ operation. Memory space was small and free space often fragmented. Contiguity is only guaranteed if we allocate our own block of memory for the task. **malloc()** can actually refuse to give you any! As a consequence, passing a pointer to a compiler defined array (**float a[10]; &a[0]**) into a function using increments used to be a common way of breaking software. 2D arrays were especially bad for this.

be supported out of standard desk machine purchases. Even our local MSc cluster has 8 machines running Ubuntu, and most of the year these are lying idle.

The specific numbers will vary but the approach can be applied for any software which runs for between 2 and 14 hours. As only beyond 14 hours is research efficiency impacted by cutting into the next day. In my experience all research analyses can be adjusted to fit within an over-night process window.

In terms of raw processing for example, a C program requiring 2 hours of execution time can be run repeatedly over-night on multiple laboratory machines without any real infrastructure (beyond a server and some shell scripts), no specialist software harness (grid, VM or Docker), and no need to redevelop code (beyond basic profiling with gprof). The scripts can be prepared by anyone in an afternoon. In addition to the optimisations described above, this can deliver another processing boost of 6x8x7 = 338!

This is an impressive figure and one that a multi-user production system may well struggle to compete with. If processing power alone does not really swing the argument, what justifies the large scale approaches? The factors for and against which seem relevant to me are the following;

- Firstly, if you have no previous experience, preparing software for use on a batch system, paralellising code and uploading any data ready for analysis, can take a month to set up and debug. This is particularly true if the original software development included a graphical user interface which must be abandoned. What matters is how long it takes to complete the analysis from the day we start the port.
- The Grid is useful for supporting multiple researchers having access to vast centralised data stores. It also provides a coordinated approach to processing and software management. The machine on your desk need have no capability other than editing a file and submitiing a batch job. Resources are needed to maintain and run such a system so expect to have to pay. If the expertise and software can be maintained between projects this is an effective approach, it is not so effective for a one off analysis.
- We also have to bear in mind that in a shared multi-user system we must wait for a program to rise to the top of a run list before it is selected for execution. As a consequence, and as with our simple system, jobs are often submitted in the evening to run over-night. The effective processing power of a processing farm should take such factors into account, by dividing the raw processing capability by the average number of users.
- Virtual Machines, and systems such as Docker, reduce the dependancy of the production processing systems on the specific operating system and libraries with which software is compiled and linked. By elminating the conventional OS it is also expected to minimise down time, for example due to cyber attacks. These arguments loose relevance if you have several isolated software development machines and server all running the same OS.
- Something useful to know here is that, in terms of memory and software execution, the grid may well be efficient but VM systems take an immediate hit (estimated by some to be a factor of 10) due to the overhead of passing a copy of the OS around. Systems such as Docker are intended to help avoid most of this.
- Paralell processing (eg. Open MP) is useful for increasing the execution speed of an individual program, such as might be useful for real-time software execution. So too is specialist acceleration hardware. But both come at the price of needing to learn how to write and debug parallel code.
- If you are interested in maintaining the resulting software and making continued developments in future, we should be aware that a break from the original development platform represents a fork in code management. In my experience the new software is not maintainable in the long term. In the same way, highly optimised C code is not a good starting point for continued modification, although in this case it should never represent more than a few days of lost work.
- All such multi-user systems are more suited to large scale production than inital code development, due to limitations of user interfaces and restrictions on real time interactive execution. Certainly they are more suited to software written from scratch to run on these systems from the outset.

In summary, if we can profile our software and have access to a Linux laboratory cluster, quite substatial processing capacity is obtainable for in-house research software with minimal (a few days) effort. Therefore, if you have no particular need for any other listed benefit the overhead of adding an extra layer of software to run on the grid or in the cloud, may not add anything to help advance your work.