|epcc|

Performance Programming

B087928

March 31, 2016

1 Introduction

Computer simulation is a massive, and rapidly growing, application in science. Processing power increases exponentially with time however we can also greatly extend what we can simulate by carefully considering the efficiency, or performance, of our program. Therefore correct code is far from our only consideration when writing a program. There are many ways to perform any function and some are a lot better than others. This can be for reasons of performance readability or robustness.

We report on the optimisation of a molecular dynamic code. The code implements molecular dynamical algorithms correctly however it has been purposely written poorly in order to give a lot of scope for improvement. The aim is to achieve an optimisation of the code particularly in relation to runtime on Morar without breaking the 'correctness' of the code.

We shall discuss what aspects of the code gives bad performance and shall follow the process we took to optimise it. Each time a potential source of considerable inefficiency is identified a solution is proposed and implemented and the resulting speed compared to that of the previous version of the code. We shall also see later how we took difficulty to implement and development risk into account when deciding which path of optimisation to take. Code maintainability was an important factor.

All runs were done on Morar, where we reserved an entire socket and requested the same core for each run. This code was serial but extra cores were reserved in order to get the most consistent results. Each timing shown throughout this report is the average of five runs with the above stimulations.

2 Profiling

An initial profile was taken of the code using pgprof profiling tool. This had many purposes. It gave a convenient way of checking the time for the program to run as well as showed in which functions the program was spending most of its time.

Times taken when profiling the code will be longer than a normal run since there is an overhead associated with the profiling. This was not considered important in the optimisation

process as the focus was to see if-and why- our changes had a positive impact on performance. Thus all runs and times are taken while profiling the code and so slightly underestimate the speed of the code after each optimisation which is worth baring in mind.

Details from initial profiling may be seen in table 1.

Table 1: Initial Profile.

Function	Time (seconds)	% of runtime
evolve	468.31	77%
force	90.00	15%
add_norm	47.11	8%
Total Runtime	605.42	100%

The program spends a negligible amount if time in other functions with have been excluded; we are only interested in optimising sections of the code that take a considerable amount of time. It is not essential to know what these functions do seeing how the distribution of runtime changes is helpful in following our optimisation process.

3 Memory Access

The effect of memory access on runtime used to be insignificant. Moore's law predicts that CPU power roughly doubles every 1.5 years, however memory access improvements are much slower due to limitations on memory bandwidth and latency- limited by the speed of light which was never a consideration with 'old' processors because data could not be processed fast enough for this to be an issue.

The first performance issue considered in the optimisation of this molecular dynamic simulation was Memory access. This was done before compiler optimisation because the compiler can not perform memory location related optimisations. It does sometimes attempt loop reversal however this is not possible if more than just simply switching loops is required, as is mostly the case in this code. Furthermore, memory access can cause massive performance issues on modern computers and therefore is expected to yield the greatest initial speedup.

3.1 Dynamic Vs. Static Arrays

The first issue considered in regards to memory access was the declaration of pointer arrays which initially is how all arrays are declared in the program. The size of these pointer arrays are declared dynamically but with values that are hard coded into the program.

Static arrarys are kept on stack memory which is Last In First Out (LIFO) data structure, hence the name. New variables/arrays are pushed onto stack at compile time and then destroyed when a function exits. Stack memory is closely optimised by the CPU. Dynamically declared arrays are kept on heap memory which is slower and must be accessed by pointers. Variables are written to heap during runtime and are accessed using pointers. The main advantage of this is that our variable size is unrestricted and we may change array sizes during runtime. However since our arrays are all declared at the beginning of the program and unaltered it is favourable to use static variables/arrays. This also frees us from the responsibility of needing to manage the memory in the program as memory on stack is automatically freed when we exit the function on which theyre declared. Hence we also avoid any memory leakage[1].

Table 2: Profile after all variable/array declarations have been changed from dynamic to static.

Function	Time (seconds)	% of runtime
evolve	396.30	76%
force	83.02	16%
add_norm	41.19	8%
Total Runtime	521	100%
Speedup:	1.16	

From table 2 we see a significant performance increase by changing all variables/arrays from dynamic to static declaration. Additional to the aforementioned reasons, this is also because variables saved on heap must be accessed by pointers where we encounter the issue of pointer chasing[2] which destroys Instruction Level Parallelism (ILP). Speedup is the runtime as a multiple of initial runtime.

[1]Failure to release memory when application is finished with it

[2]2Pointer Chasing refers to the multiple consecutive memory accesses via pointers.

3.2 Contiguous Vs. Non-Contiguous

Consider the declaration of a 2D array, or list of lists:

```
array[M][N];
```

Memory structures are a feature of the programming language. In C, which is the language used here, this is contained in the hardware's memory as:

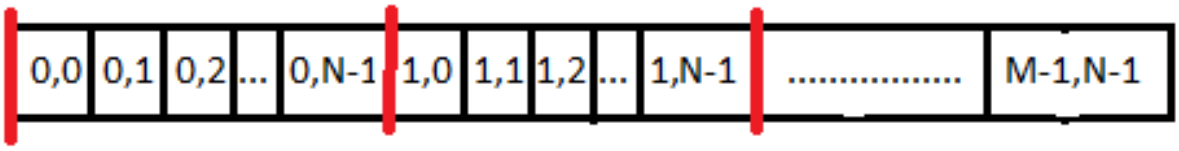

Figure 1: array[M][N] stored in memory.

Access to contiguous[3] elements always gives a performance benefit over access to non-contiguous memory. Thus if we have an operation on the elements $array\_[i][j]$ in nested for loops over i and j we have two distinct ways of accessing elements based on our ordering of the loops which makes a big different in speed to the accessed memory. If we have the i as the outer loop, and j as the inner then our access looks like this:

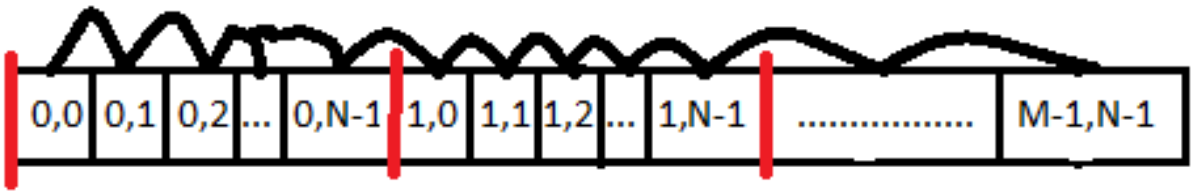

Figure 2: Order of memory access to array[i][j] with nested for loops, j on the inside loop and i on outside loop. Every element is accessed contiguously.

However if we reverse the order of these loops so that i is the inner loop then our access looks like this:

[3]Occupying consecutive addresses in memory.

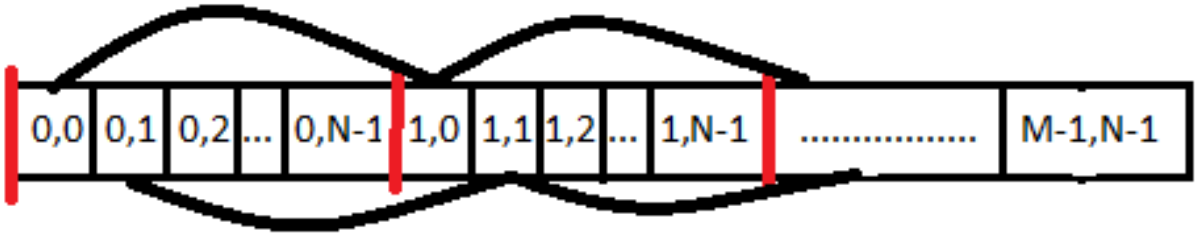

Figure 3: Order of memory access to array[i][j] with nested for loops, i on the inside loop and j on outside loop. Elements are accessed non-contiguously, every element accessed is N memory addresses apart.

Clearly the prior is far more desirable. In this case it is easy for the program to predict what element is to be accessed yet and hence write sections of the array to cache. Modern processors rely very heavily on caches so this form of optimisation is expected to have a massive effect on program efficiency. Dependency of the operation we wish to perform in the for loop may limit the possibility of changing order of the loops as we shall see in 3.3.

Following the above discussion we shall discuss the two ways we considered for improving memory access within the molecular dynamics simulation. In this program we deal with N bodies, each with an associated position, velocity and force which have three vector components contained in memory as:

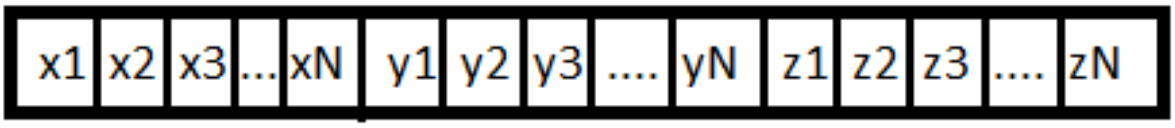

Figure 4: Order in which vector components are stored in memory. Note: x1, y1, z1 is the vector pertaining to body 1 etcetera.

We hence considered how this memory was accessed in the code. Consider the following code snippet. We iterate over l in the inner most for loop. This corresponds to the form of access as shown in figure 3 ie. we access x1 then y1 then z1 then x2 then y2 etcetera. It was noticed that several of the sections (or algorithms) in the code had this flaw. This is not as simple as the example given above, we can not simply hoist the loop over Ndim to the outside because this would change the algorithm. Here, as in several other places throughout the code

care needed to be taken to ensure the result remained unmodified and, as we shall see in section 3.3 this was not possible in all cases.

```
k = 0;
for(i=0;i<Nbody;i++){
  for(j=i+1;j<Nbody;j++){
    for(l=0;l<Ndim;l++){
      delta_pos[l][k] = pos[l][i] - pos[l][j];
    }
    k = k + 1;
  }
}
```

The following code snippet has been improved from the one above for more efficient memory access and is an example of how memory access was optimised elsewhere in the code also.

```
for(l = 0; l < Ndim; l++){
    k = 0;
    for(i = 0; i < Nbody; i++){
      for(j = i + 1; j < Nbody; j++){
        delta_pos[l][k] = pos[l][i] - pos[l][j];
        k = k + 1;
      }
    }
}
```

Table 3: Profile after memory access has been improved by changing modifying loop ordering.

Function	Time (seconds)	% of runtime
evolve	276.94	72%
force	53.99	14%
add_norm	52.26	14%
Total Runtime	384	100%
Speedup:	1.58	

Further loop reordering was later utilised when we inlined the add_norm and force routines. Though these shall be commented on later as we follow the optimisation process.

3.3 Alternative Memory Locations

We mentioned that we considered two options for optimising memory access in the code. We have discussed the one we have implemented- changing the structure and order of loops to make memory access more contiguous. We saw massive benefits from this method however it is limited for some of the algorithms within the code.

To calculate the position vector of the N particles we need to make the following calculation for each body i:

$$r = \sqrt{xi * xi + yi * yi + zi * zi} \tag{1}$$

There is no way to reorder loops to improve memory access here as all of xi, yi and zi are needed in each calculation. To overcome this problem we considered saving data to memory in an alternative way, this can be seen in figure 5.

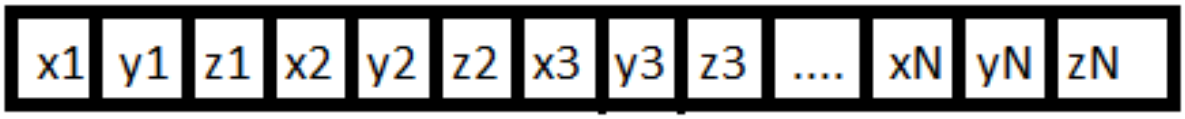

Figure 5: Alternative order to store vector components in memory. Note: x1, y1, z1 is the vector pertaining to body 1 etcetera.

This structure of memory makes the calculation of r in equation (1) much faster to calculate as the 3 elements required for each calculation are contiguous in memory. This is thought to be the case for one other loop in the code also as this other loop contained a similar calculation. Therefore the code performance may have benefited greatly from this alternative memory structure, especially since it appears that it would then be possible to have all major loops accessing contiguous addresses in memory. However, as we mentioned in the introduction, difficulty to implement and development risk must be taken into consideration. It would no longer be any sense to use 2 dimensional arrays and format of indices must be changed everywhere in

the program. For example instead of `pos[1][i]` representing the y position of particle i we would have `pos[3*i+1]`. Similar changes are required when reading writing data and in array declaration. This was attempted but the final version was giving a different result for bugs that could not be detected and so we returned to a version that worked.

Further optimisation of this code should involve reconsidering the way data is stored in memory as we described above.

4 Compiler Optimisation

Many optimisations are attempted by the compiler. We may choose the level of optimisation by selecting compiler flags that each have their own specific meaning to the compiler. We included these in our Makefile under compiler flags. The first we included was Minfo.

-Minfo Minfo instructs the compiler to produce information on the compiler‘s actions on the code. This was introduced first so that we could see what changes the compiler made to the code with the introduction of each new compiler flag as well as just the effect on performance. From this information it was possible to determine the shortcomings of the compiler‘s optimisations and could be used during further optimisation.

-fastsse This option creates a generally optimal set of flags for targets that support SIMD capability. They incorporate optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization. It is the ‘fast‘ way to implement compiler optimisation. This was used first to see the effect of a naive but effective all purpose compiler flag. The effect on performance may be seen in table 4. Some more time than before has been used in other functions but not significant enough to show here. We can see that this flag is likely to increase performance though we must also take into consideration that the code has spent longer in the force function. So it is possible for this flag to have negative effects on performance if used naively. We shall now also consider flags with more specific and narrow implications.

-Mpia=fast,inline This also allows inter-procedural optimisation analysis and optimisation. Further it allows routine inlining. Inlining the functions we use reduces the overhead in function calls.

Table 4: Profile after after including -fastsse compiler flag.

Function	Time (seconds)	% of runtime
evolve	191.99	61%
force	63.01	20%
add_norm	51.42	14%
Total Runtime	307	100%
Speedup:	1.98	

-Munroll Invokes the loop unroller to unroll loops, executing multiple instances of the loop during each iteration. We do some manual unrolling later so we shall show the mechanics of this more explicitly then.

Table 5: Profile after including -Mpia=fast,inline and -Munroll compiler flags.

Function	Time (seconds)	% of runtime
evolve	262.46	99%
Total Runtime	263	100%
Speedup:	2.30	

We notice that the `force` and `add_norm` functions have dropped off too negligible runtime values since the functions have been inlined at compile time.

-Mvect Vectorisation is the functionality of the compiler recognising parallelism within codes as modern CPUs can perform multiple operations per cycle in their functional units. To utilise vectorisation within loops they must not contain subroutine calls. Thus we inlined all functions manually at this point so that we could achieve maximum benefit from compiler vectorisation. This should not always be done but all subroutines in this program were very short and so did not make the code significantly more difficult to read and maintain.

-Mprefetch This enables the compiler to selectively emit instructions to explicitly prefetch data into the data cache prior to first use. Some of these prefetch instructions are already embedded in -Mvect.

The result from profiling may be seen in table 6.

Table 6: Profile after including -Mvect and -Mprefetch compiler flags.

Function	Time (seconds)	% of runtime
evolve	223.66	99%
Total Runtime	224	100%
Speedup:	2.70	

-O4 We also include the -O4 flag which includes a lot of the instructions already present from previous flags and so gave only a small further performance benefit.

5 Loop Fusion and Loop Fission

The aim of loop fusion is to increase the program‘s speed by reducing the total number of instructions required to control loops in the code- like ending the loop and iterating a counter. To do this we amalgamate loops that ran over the same indicies.

The use of functions in the original code made it more difficult to detect potential optimisation. Subroutines provide a useful way to modularise code but in this case the subroutines were one line or one loop. As we discussed in section 4 we inlined all these functions. When we did this it was noticed that several more loops could be reordered and then fused with other loops. As well as this there were several sections of code that contained nested for loops that could be fused with a a large nested for loop section.

The following code snippet was originally 6 different loops this is an example of of fusion operations that as carried out throughout the program.

```
for(i = 0; i < Nbody; i++){
      r[i] = ((pos[0][i] * pos[0][i]) + (pos[1][i] *
         pos[1][i])+(pos[2][i] * pos[2][i]));
      r[i] = sqrt(r[i]);
      for(j = 0; j < Ndim; j++){
            f[j][i] = -visc[i]*(vel[j][i] + wind[j]) -
               (G*mass[i]*M_central*pos[j][i])/(pow(r[i],3));
       }
}
```

Table 7: Profile after all fusion.

Function	Time (seconds)	% of runtime
evolve	171.83	99%
Total Runtime	172	100%
Speedup:	3.52	

Loop fusion will always decrease the number of loop control instruction but does not necessarily improve performance. We may wish to split one loop into two because there is increase in data locality within each loop- this is called loop fission.

In the above code snippet we can see that we have a case of non-contiguous memory access as in figure 3 for both the f[j][i] update and the r[i] update operations. However we can not fix this in the r[i] case as this is the problem discussed at length in section 3.3. Since we can't switch the order of this nested loop, fission is the best option. Splitting this into two loops allows us to, at least, switch the order of the loops for one of the operations. See the following snippet for this implementation. Notice that we have hoisted the inner loop to the outside in the latter for loop improving memory access. This improved the performance by decreasing runtime another 10 seconds approximately.

```
for(i = 0; i < Nbody; i++){
      r[i] = ((pos[0][i] * pos[0][i]) + (pos[1][i] *
         pos[1][i])+(pos[2][i] * pos[2][i]));
```

```
        r[i] = sqrt(r[i]);
    }
    for(j = 0; j < Ndim; j++){
        for(i = 0; i < Nbody; i++){
            f[j][i] = -visc[i]*(vel[j][i] + wind[j]) -
                (G*mass[i]*M_central*pos[j][i])/(pow(r[i],3));
        }
    }
```

6 Array Padding

Array padding is adding additional, unused, space between arrays, or dimensions of arrays, in memory. This is done it is easier to transform loops than arrays (since loop transforms are always local in the program). Thus we can reduce cache conflict misses. We padded the space between dimensions of arrays by $\sqrt{Nbody} = 64$.

7 Loop Unrolling

The aim of loop unrolling is similar to that of loop fusion: we want to reduce the overhead involved in controlling the loop. It involves writing instructions such that multiple tasks, which would have each taken a loop iteration previously, may be completed in one loop operation. We discussed in section 4 that the compiler shall attempt to do this, however we helped by unrolling some loops manually. The compiler could then further unroll further out loops- if the loops were originally nested.

Consider the following code snippet that was identified to be unrolled manually.

```
for(j = 0; j < Ndim; j++){
    for(i = 0; i < Nbody; i++){
        f[j][i] = -visc[i]*(vel[j][i] + wind[j]) -
            (G*mass[i]*M_central*pos[j][i])/(pow(r[i],3));
    }
}
```

Originally we had changed the loop ordering here in order to get more efficient memory access which is why the loop over Ndim (which is of size 3) is the outermost loop. It was a concern that unrolling this loop in the way we have would destroy the memory access benefits we gained previously as now the order of memory access is f[0][0], f[1][0] etcetera. But there is a distinct difference in this case: the computer is writing data starting at address of &f[0][0], &f[1][0], &f[2][0] etcetera to different cache lines so that we have contiguous memory accesses for each of these lines of execution.

```
for(i = 0; i < Nbody; i++){
   f[0][i] = -visc[i]*(vel[0][i] + wind[0]) -
      (G*mass[i]*M_central*pos[0][i])/(pow(r[i],3));
   f[1][i] = -visc[i]*(vel[1][i] + wind[1]) -
      (G*mass[i]*M_central*pos[1][i])/(pow(r[i],3));
   f[2][i] = -visc[i]*(vel[2][i] + wind[2]) -
      (G*mass[i]*M_central*pos[2][i])/(pow(r[i],3));
}
```

This implementation decreased the total runtime of the code by about 10 seconds, which was considerable given the speedup so far. We identified more similar loops to unroll. One in particular is worth discussing as since it dominates most of the runtime. This code section involved a lot more work than the previous code snippet; it consisted of three nested for loops, 2 over Nbody and a 3rd, outermost one (placed there during the first stage of optimisation for purpose of more efficient memory access) over Ndim. Hence the resulting optimisation benefit from unrolling this outermost loop was very significant. A few new variables need to be declared to make the unrolling work however this effect to performance was negligiable compared to the performance gain due to unrolling. In table 8 we can see the profile of the code after these optimisations. It is a massive improvement over the previous version.

Table 8: Profile after all manual loop unrolling.

Function	Time (seconds)	% of runtime
evolve	99.58	99%
Total Runtime	100	100%
Speedup:	6.07	

Optimisation by unrolling was very successful in several loops. However, one loop, with the difference that this loop contained an array that was of size 3*Nbody*Nbody (while the other two loops contained arrays at most length 3*Nbody), was found decreased performance when unrolled This is believed to be because there was more cache difficulties due to the length of this array and so it was important to not unroll loops without after ensuring that it has benefited the performance of the code.

Conclusions

If the size of arrays required in the program is known before runtime then it is usually favourable to declare static arrays as they are kept on the smaller, fast ‘stack‘ memory. This is also safer as the CPU manages the freeing of this memory when it has fallen out of scope in the program.

Memory access is one of the most significant factors in the performance of a code on a modern computer. It is important to consider the operations that we are expecting to be involved in our program when deciding how our values are stored in memory. Recall that having data in memory in the order x1, x2, ... xN, y1, y2 ... etcetera precluded the arrangement of contiguous memory access for some operations such as equation (1). Whatever way we finally decide to save data to memory will always have advantages and disadvantages. It is usually better to stick with the way you have chosen and try to make the code work to its advantages than making a global change to the code that requires a lot of programmer effort and development risk and may, in the end, result in discovering as large a problem as the one that motivated the change.

Compiler flags can be very useful tools to quickly and relatively safely improve code performance. If we wish to get the possible performance it‘s important to know what optimisations the compiler is attempting, or has achieved. This will give an insight to section of the codes that should be manually optimised or even tweeked slightly so that the compiler may have a better chance to optimise it- or optimise it further. As well as a method of optimisation in itself, subroutine inling can uncover further potential optimisations at low cost of maintainability if the subroutine is short as they all were in this code.

Loop fusion is a useful method of optimisation and can have a big impact on performance if the operations within the loop all have similar data access locality. Otherwise it may be best to

leave the loops separate or separate existing loops.

Effective loop unrolling can be achieved even when it seems that we are breaking contiguous memory access if the data structures involved are of appropriate sizes such that we can exploit different lines of cache or registers. This can very quickly have a negative effect however if the data stucture is not of appropriate size.

The final runtime, without profiling was 95.67 seconds. This was a speedup of 6.33 over the original code.