CFD Homework 1: Optimization Exercises

Part 1

The code hw1.f90 is divided into sections involving different structures of simple algebraic operations computed across a square domain, and designed to demonstrate that faster processing times can be achieved if potential for optimization is recognised and exploited. The program is run using four different compiler options associated with different methods of optimization. The high optimization setting (-O3) is then re-run with shared-memory parallelisation enabled for one and two CPUs. Five trials are performed for each compilation setup and the arithmetic average of the CPU times are calculated for each section of the code. All runs were performed within a thirty minute period, over which it is assumed the machine load was constant. The results are summarised in table 1 below.

CODE SECTION	MEAN CPU TIME (s) USED BY COMPILER:					
	ifort -O0	ifort -O1	ifort -02	ifort -O3	ifort -O3 SMP mode, 1 CPU	ifort -O3 SMP mode, 2 CPUs
1a	8.1520	7.6220	7.5520	7.5680	7.3460	7.6480
1b	1.4660	0.9640	0.9640	0.9600	0.9540	1.0616
1c	1.6700	1.1520	1.1180	1.1160	1.1040	1.2320
1d	1.4060	0.9640	0.9660	0.9580	0.9540	1.0496
1e	1.7060	1.4160	1.4180	1.4200	1.4040	1.4728
2a	1.5900	1.2260	0.9640	0.9580	0.9520	1.1380
2b	1.6060	0.9580	0.9580	0.9600	0.9520	1.0868
CUMULATIVE CPU TIME	**17.596**	**14.302**	**13.940**	**13.940**	**13.666**	**15.470**

***Table 1**: CPU times used by sections of hw1.f90, averaged over 5 runs for each of the 4 different compiler options. The last two columns use the –O3 configuration with shared memory parallelization (SMP) for 1 or 2 CPUs.*

Compiler option –O0 disables optimizations so that all sections of the code take longer to execute. For the remaining 3 setups (without parallelisation), the CPU times used are notably faster for all portions of the code, with relative performance of the compilers varying between sections, depending on how well the setup is suited to optimizing the operation at hand. Compiler level –O1 produces minimal speed optimizations without the capacity to pipeline. Level -O2 is the standard (default) option, enabling optimizations for speed and a variety of intra-file intraprocedural optimizations including pipelining, subroutine inlining and loop unrolling. Level -O3 optimizes most aggressively, enabling all -O2 optimizations, a larger variety of loop transformations and data prefetch.

Sections 1a-e of the code deal with the computation of

$$a(i,j) = (a(i,j)*b(i,j) + c(i,j)*d(i,j))**0.315 \quad , \qquad (1)$$

using slightly different methods, where a,b,c,d are scalar constants defined over the square domain. Section 1a defines a(i,j) across the nxn matrix using a nested do-loop structure that acts to fill the matrix along its rows. Within the inner do-loop, nine operations are performed before the nested index (j) is increased by 1:

```
STEP 1:   load a into R0
STEP 2:   load b into R1
STEP 3:   R2 = R0 * R1
STEP 4:  load c into R3
STEP 5:  load d into R4
STEP 6:  R5 = R3 * R4
STEP 7:  R6 = R2 + R5
STEP 8:  R7 = R6 ** 0.315
STEP 9:  assign the value of R7 to a on the LHS
```

Compiler setting –O2 executes this section of the code fastest. The decrease in the CPU time spent on section 1a when –O2 is used could be due to the possibility of constant propagation with this option, so that the known values of the constants a,b,c and d are substituted as soon as they are encountered in the code. The most aggressive optimization level (–O3) is not best suited for this section and completed the task over a longer time period, relative to the –O2 setup. It will be seen throughout the remaining parts of the code that the use of more forceful optimization settings does not guarantee higher performance unless the various loop transformations enabled with it can be fully exploited.

Section 1b performs an identical set of operations to 1a, but nests the loop on the (leftmost) i index instead of the j. Section 1b is thus executed much faster since it acts to fill the matrix in natural ascending storage order, which is column-major order for Fortran. Traversing a column-major array in row order is inefficient since data points accessed consecutively are not located in the same part of the computer memory (Thiyagalingam *et al.* 2003). Accessing the array in a manner consistent with storage method ensures the data visited is still resident in the recent memory of the computer. Access to this cache memory reserve is much faster than accessing the RAM.

The compiler setups -O1 and -O2 improve on the time taken by the unoptimized setting (–O0), but the performance of the highest optimization level is superior. Traversing the matrix in natural ascending storage order allows the prefetching capacity of the –O3 compiler setup to be utilised. The speed of calculations exceeds the speed of data transfer between the RAM and the CPU. Loading the data before it is summoned by the CPU reduces wait states, enabling more operations can be performed in a shorter time. The –O3 setup also enables superscalar loops to be constructed, so that independent operations can be executed simultaneously. By superscaling, only six operations are performed within the inner do-loop (j) before the nested index (i) is increased by 1:

```
STEP 1:  load a into R0 AND load b into R1
STEP 2:  R2 = R0 * R1 AND load c into R3
                      AND load d into R4
STEP 3:  R5 = R3 * R4
STEP 4:  R6 = R2 + R5
STEP 5:  R7 = R6 ** 0.315
STEP 6:  assign the value of R7 to a on the LHS
```

Alternatively, for a given column (j), the –O3 setup enables n simultaneous eq.(1)s could be computed for i=1,n. For example, to compute a(i,j) in the first column of the matrix, the relevant set is:

$$a(1,1) = (a(1,1)*b(1,1) + c(1,1)*d(1,1))**0.315 \qquad (2a)$$
$$a(2,1) = (a(2,1)*b(2,1) + c(2,1)*d(2,1))**0.315 \qquad (2b)$$
$$\vdots$$
$$a(n,1) = (a(n,1)*b(n,1) + c(n,1)*d(n,1))**0.315 \qquad (2n)$$

Since eqs.(2a)-(2n) are independent of one another, they can be performed in parallel, significantly reducing the total time taken to find a(i,j) across the entire domain.

Section 1c computes eq.(1) using 3 consecutive do-loops. First, steps 1-3 above are computed for all i,j so that a*b is specified across the entire nxn matrix. The second loop performs steps 4-6, specifying c*d across the entire domain. The final loop sums the two products across the domain, computing the (0.315) power as the sum is found. All three loops are matched to Fortran's storage layout, maximising the cache hit rate and consequently, taking less time to solve eq.(1) across the array than the method in section 1a. However, the entire array must be accessed three separate times in this section. Each loop starts by calculating a(1,1) first, filling the (i,1) column of the matrix and finishing with the calculation of a(5000,5000) at the base of the (i,5000) column. By the start of the third loop, a*b at (1,1) is required to add to c*d(1,1), but over 49 million products have been calculated since it was found. The cache space is limited so it is likely that the RAM will have to be accessed. This entails a slower transfer of data, increasing the CPU time spent on section 1c (relative to 1b) for all compiler settings.

Compiler performance for this section is improved as the optimisation level increases due to the increased capacity for parallelisation. This section of code can be further optimized by the higher level compiler using a combination of superscalar loops - as described above – and pipelining, overlapping the execution of successive instructions. An alternative approach is transforming the loops to reduce data transfer between the main memory and the CPU. When present, loops are often the most time consuming elements in numerical codes. Multiple (distinct) loops over the entire array create a large overhead, requesting repeated access to the same part of the array at different times. Compiler setup –O3 allows the three different loops to be fused, decreasing the number of branches to the top of the loop considerably and increasing the likelihood that consecutive operations within the single loop load from the same cache lines, instead of from different regions of the RAM. The average time taken to complete section 1c using the –O3 setup is therefore lowered closer to the time taken to complete section 1b.

Section 1d computes eq.(1) using array operations. The code is already vectorized so that vectors of values of a,b,c and d will be fetched from memory by all four compilers. Instead of calculating one element of the 5000x5000 matrix at a time, many data elements are operated on simultaneously, producing a highly efficient method of solving eq.(1). This approach achieves the fastest execution of eq.(1) for all compiler setups in section 1 except –O2, which solves eq.(1) faster using the method in 1b. The compiler set to the highest optimization level performs best since it can handle the vector array operations in superpipelines.

The method of solution of eq.(1) implemented in section 1e is the most time consuming approach after the approach used in 1a (which was characterised by a severely inefficient loop structuring). In this case, the increase in CPU time required for execution is caused by the recursive data dependency within the inner loop:

$$a(i,j) = (a(i-1,j)*b(i,j) + \ldots \qquad (3)$$

which is one of the most severe vectorization inhibitors. The higher optimization compiler performed slowest due to its inability to vectorize and pipeline this equation.

Section 2 solves variations of eq.(1) using subroutines, which are often used in codes to reduce duplication, minimising the length of a script, and decompose the program into simpler, more readable components. In 2a, the subroutine is called within the do-loop, introducing significant overhead associated with summoning the sub-routine and waiting for the return. The nested subroutine call also inhibits vectorization. The more aggressive optimization setups can inline intrinsics, replacing the subroutine call within the loop with explicit code from the subroutine. As a result, section 2a is executed quickly by the –O2 and –O3 compiler configurations. However, the lower level optimization setups –O0 and –O1 take much longer to complete section 2a since neither has the capacity to inline.

For the three compiler setups that can vectorize, the fastest solution to eq.(1) is obtained by the method used in section 2b. The computation of eq.(1) is performed within a do-loop contained within the subroutine. The subroutine is called within the main body of the code and the nxn solution matrix is returned, taking less time to compute than 2a since no initial inlining is required.

The optimizations implemented by the higher level compiler setups listed above, confined the operations to a single CPU. For shared memory systems, there is potential for further speed up as the workload can be distributed across all processors sharing access to the same data arrays. To further this investigation into code optimization, the hw1.f90 script was recompiled using the most aggressive optimization setting (-O3) with shared-memory parallelisation (SMP) turned on. The average results for the runs using 1 thread and 2 threads are shown in the two last columns of table 1.

Parallelisation using one CPU yields the smallest (cumulative) CPU time for a completion of a full model run. The performance of this configuration for specific sections of the code is similar to that described for the –O3 level optimization with SMP turned off, however it is consistently faster since parallelisation is not restricted to the loop level but can occur at the subroutine and task level also.

When two threads are used, sharing the same memory space and variables between routines, all sections of the code are executed much slower than when run with ifort –O3 with the SMP switched off. This could be caused by the CPUs competing for critical sections.

Part 2

Optimization methods and parallelization issues are now investigated in a large atmospheric prediction model by running identical batch scripts for various jobs in The Advanced Regional Prediction System (ARPS), using executables that have been compiled with different optimizations/parallelizations on a superscalar DSM parallel system with shared memory nodes. The different configurations of the eight executables used is described in table 2 below, where the total CPU time and wall clock time for all batch runs is also listed.

Compiler Setting	**Description**	Total CPU Time (s)	Wall Clock Time (s)
1	**No automatic SMP** **Minimum optimization (-O0)** **1 CPU**	5200.2	5203.0
2	**No automatic SMP** **High optimization (-O3)** **1 CPU**	1160.4	1166.5
3	**Automatic SMP** **High optimization (-O3)** **1 CPU**	5356.1	5400.5
4	**Automatic SMP** **High optimization (-O3)** **2 CPUs**	5544.4	5602.3
5	**Distributed MPI (1x2)** **High optimization (-03)** **4CPUs**	1565.0	783.8
6	**Distributed MPI (2x2)** **High optimization (-O3)** **4CPUs**	1571.9	393.7
7	**Distributed MPI (1x4)** **High optimization (-O3)** **4CPUs**	1642.4	411.3
8	**Distributed MPI (4x1)** **High optimization (-O3)** **4CPUs**	1610.5	403.3

Table 2: Total CPU and wall clock times for batch jobs executed using the compiler settings listed in the left-most column, where the negative numbers in brackets correspond to the optimization level assumed by the ifort compiler, and the products in brackets indicate the method by which the domain is horizontally decomposed (e.g (1x4) implies the domain is divided into subdomains consisting of one row and four columns).

The total CPU time is the time taken to process the batch script on the CPU. The wall clock time is the real time elapsed as the CPU processes a function and executes all other routine jobs at periods inbetween. As seen in table 2, and emphasized in figure 1 below, the wall clock time exceeds the total CPU time for all runs except those involving horizontal domain decompositions.

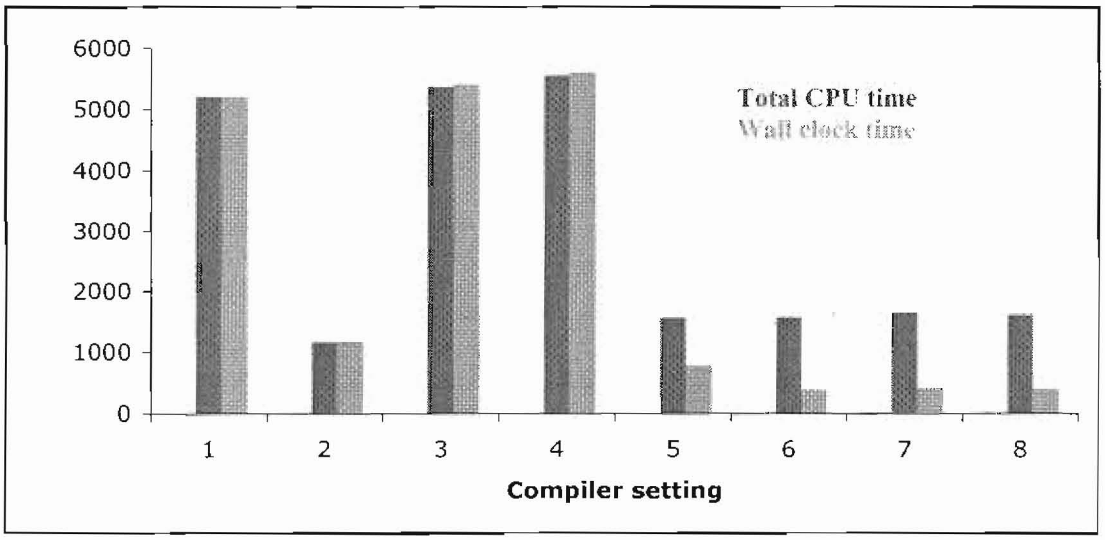

Figure 1. *Total CPU (red) and wall clock times (blue) for the execution of the same batch script, using different compiler settings as detailed in table 1.*

By allocating separate parts of the array (and the computations required in these regions) to the 4 different CPUs available in configurations 5-8, the processors are more fully and efficiently utilised. A decrease in idle time causes the CPU to drop, and by working on different parts of the domain simultaneously, the CPUs execute the batch job faster, causing a drop in the total CPU time (which remains above the wall clock time since it is the sum of CPU time used by all processors). The decomposition of the domain into 2 parts resulted in a significant decrease in CPU and clock time, which was improved upon by the decomposition of the domain into 4 parts in configurations 6,7 and 8. The greatest speed up was attained by configuration 6, which involved splitting the domain into four equally sized arrays allowing similar workloads to be given to each of the four processors so that the load was balanced..

A surprising result is that configuration 2 – compiled without automatic shared-memory parallelization – processed the batch job in significantly less time than configurations 3 and 4 which include SMPs. Since all three configurations compile using aggressive optimisation (compiler ifort –O3), it would seem that the forced SMP parallelization in 3 and 4 reduced the computation efficiency, making the execution of the batch job even slower than that conducted by compiler setup 1 which disabled all optimizations. It is important to recall that ARPS is designed to run on massively parallel computers, but is freely available for download, so must be capable of running efficiently with a default high optimization compiler (such as that used in configuration 2). Further investigation into the structure of the code in ARPS is required to confirm that this – combined with the high optimization setting – is the reason for the high-level of single CPU optimizations.

The batch job takes the longest time to process with configuration 4. This could be due to the aforementioned ineffectiveness of the forced SMP combined with the increased overhead due to more frequent inter-processor communications (the availability of 2 CPUs). However, since two CPUs share the workload in configuration 4, we would expect that – if the load was equally balanced between the two – the wall clock time would be half the total CPU time. The fact that the wall clock time exceeds the total CPU time suggests that the program does not have exclusive use of the node.

REFERENCES

Thiyagalingam, J., O. Beckmann, and P. H. J. Kelly, 2003: An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays, *Performance Engineering: 19th Annual UK Performance Engineering Workshop*, 340-351.