Analysis and Comparison of ForeC Benchmark Performances with OpenMP

Mario Blazevic

Department of Electrical and Computer Engineering
University of Auckland

Abstract

This report details the analysis and comparison of a new parallel programming language, ForeC, developed at the University of Auckland, with the standard in parallel programming, OpenMP. The two languages are described and compared in terms of strengths and ease of use. Benchmark testing is also carried out to compare runtime performance.

1. Introduction

Parallel computing has become the dominant paradigm in the last decade, with multi-core processors becoming wide spread in today's consumer desktop and mobile computers. Programming to take full advantage of these architectures is still lacking and is therefore of much interest in current research and development on a global scale.

Parallel programming has its inherent challenges. In order to make it a standard practice amongst software developers world-wide, these challenges must be overcome. The biggest challenge for programmers is to learn to find concurrency in programs and be able to "think in parallel". They must also have access to synchronisation constructs that enable them to write programs free from deadlocks and race conditions, and there must be sufficient tools and methodologies to support debugging.

The implementation of parallel programming can vary from libraries within traditional sequential languages, to language extensions, to complete execution models. The existing implementation which will be used for comparison in this paper is OpenMP. It is an API that supports shared memory multiprocessing programming in C, C++ and Fortran, and is one of the more popular implementations as it can be run on most processor architectures and operating systems.

The aim of this research is to compare a new parallel programming language, ForeC, developed by Eugene Yip at the University of Auckland, to OpenMP. Runtime performances using benchmark testing will be analysed and compared to determine the abilities of each language.

2. Language Comparison

The Matrix Multiply benchmark program is used in the following sections to describe the use of both parallel programming languages compared to its sequential original. The program simply carries out multiplication operations between two large matrices.

2.1. Sequential

The program simply initialises three matrices, matrix A, B and C using the function allocateMatrix. It then fills all three matrices with random numbers using the function fillMatrix. The function multiplyMatrix is shown below in Figure 1. It performs the matrix multiplication operation; A*B=C, in a two-tiered nested for loop. The function is called within a for loop in the main function to perform the multiplication a total of ten times.

```
void multiplyMatrix(int*** matrixA, int*** matrixB,
    int*** matrixC, const int startIndex) {

    int i;
    for (i = startIndex; i < SIZE; i++) {

        int j;
        for (j = 0; j < SIZE; j++) {
            int sum = 0;

            int k;
            for (k = 0; k < SIZE; k++) {
                sum += (*matrixA)[i][k] * (*matrixB)[k][j];
            }

            (*matrixC)[i][j] = sum;
        }
    }
}
```

Figure 1: Code snippet of multiplyMatrix function.

2.2. OpenMP

OpenMP allows a programmer to separate a program into both serial and parallel regions. It uses a fork-join

model of parallel execution where the execution starts as a single process in a “master thread” until it reaches a parallel construct where a team of threads is created to execute in parallel [1]. Once the parallel construct is completed, the threads in the team synchronise. Thereafter, only the master thread continues execution until the next parallel construct. The process is described visually in Figure 2 below.

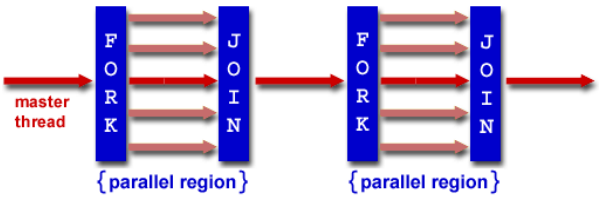

Figure 2: OpenMP fork-join execution model [2].

Figure 3 shows the OpenMP version of the multiplyMatrix function. It demonstrates the use of the OpenMP data parallel loop construct. The directive tells the compiler to calculate the loop bounds for each thread and manages the data partitioning of shared variables (matrixA, matrixB and MatrixC). The variables used within the parallel construct are shared by default, meaning they are visible to all threads. However, the variables initialised within the construct, (e.g. j, sum and k) are private, meaning each thread sees their own version of that variable. OpenMP allows the user to allocate which variables are shared and which are private. The multiplyMatrix for loop is an ideal candidate for simple parallelisation since there is no interference when accessing and updating shared variables between loop iterations. Each thread in the team computes its allocated loop iterations and once the loop is complete, they synchronise and only the master thread continues execution.

```
void multiplyMatrix(int*** matrixA, int*** matrixB,
    int*** matrixC, const int startIndex) {

    int i;

    #pragma omp parallel for
    for (i = startIndex; i < SIZE; i++) {

        int j;
        for (j = 0; j < SIZE; j++) {
            int sum = 0;

            int k;
            for (k = 0; k < SIZE; k++) {
                sum += (*matrixA)[i][k] * (*matrixB)[k][j];
            }

            (*matrixC)[i][j] = sum;
        }
    }
}
```

Figure 3: Code snippet of multiplyMatrix function with OpenMP pragma included.

Parallel loop constructs are also applied to the initialisation functions. These parallel regions allow the program to execute faster depending on the number of processors (threads) available for use.

2.3. ForeC

```
int main(int argc, char **args) {

    printf("Initialising matrices\n");
    par(initialiseMatrixA, initialiseMatrixB,
        initialiseMatrixC);

    int i;
    for (i = 0; i < REPEAT; i++) { //REPEAT = 10
        printf("Iteration %d\n", i);

        /* matrix multiplication code here */
        par(multiplyMatrix0, multiplyMatrix1,
            multiplyMatrix2, multiplyMatrix3);
    }
            .....

thread initialiseMatrixA(void) {
    allocateMatrix(matrixA_ptr);
    fillMatrix(matrixA_ptr);
}
            .....

thread multiplyMatrix0(void) {
    int RANGE = 150;
    multiplyMatrix(matrixA_ptr, matrixB_ptr,
        matrixC_ptr, 0 * RANGE);
}

thread multiplyMatrix1(void) {
    int RANGE = 150;
    multiplyMatrix(matrixA_ptr, matrixB_ptr,
        matrixC_ptr, 1 * RANGE);
}
```

Figure 4: Code snippet of Matrix Multiply ForeC program showing use of *par* statement and thread functions.

ForeC is a C-based, multi-threaded, synchronous language that enables the deterministic parallel programming of multicores with minimal extension to C [3]. As with OpenMP, fork-join parallelism is used. A fork is created using the $par(f_0, \dots, f_n)$ statement, as demonstrated in Figure 4, allowing the input functions to be executed in parallel threads. This allows for easy

implementation of task parallelism, which is discussed in the next section. The threads are executed in relation to a global clock. Threads can execute until they are complete, or until they reach a *pause* statement. Pause statements force thread synchronization as their execution is halted until the next global clock tick. Synchronisation of threads allows the user to update any shared variables which have been updated by two or more threads.

Due to the synchronous and deterministic method of parallel programming, ForeC guarantees freedom from race conditions in shared variables. This is not the case in OpenMP, where the programmer must make certain there are no race conditions, which can be extremely difficult to debug. The semantics of ForeC are explained in further detail in [3].

Figure 4 shows how the Matrix Multiply program was altered to suit the ForeC language. The first section shows two *par* statements being used. The first one is for the initialisation of the matrices. A 'thread' function e.g. initialiseMatrixA, is created for each matrix to run both allocate and fill functions. These threads are run in parallel to speed up the initialisation process.

The second *par* statement is used to parallelise the matrixMultiply function. The function is slightly altered to allow the operation to be sectioned off into four separate 'thread' functions e.g. multiplyMatrix0, multiplyMatrix1 etc. The threads are run in parallel to speed up the overall computation runtime. Figure 5 shows visually how matrixC is divided into four sections and computed concurrently.

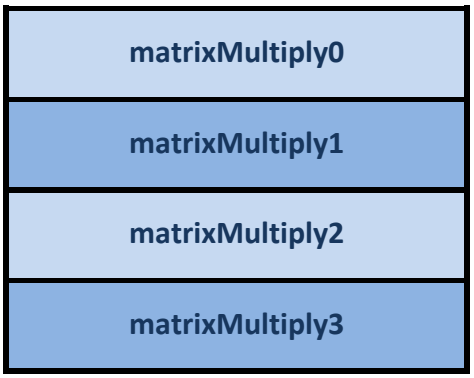

Figure 5: Illustration showing how the computation of result matrixC is sectioned into parallel thread functions.

2.4. Data vs. Task Parallelism

The OpenMP parallelisation shown above is an example of data parallelism, meaning each processor executes the same code on different sections of shared data. ForeC favours task parallelism, where each processor executes a different thread on the same or different data. The threads may execute the same or different code. Data and task parallelism can be represented as two ends of a continuum, since the difference between the two may often be ambiguous. The ForeC example above is a definite example of task parallelism. Each processor executes its own thread function, even though the functions execute virtually the same code on different sections of the shared data. This type of task parallelism is used throughout the benchmarks tested.

Both OpenMP and ForeC can achieve both types of parallelism. However, OpenMP allows for simple data parallelism, especially in loops, where ForeC excels in task parallelism. The differences lie in the structure of the code and how user friendly it is to implement either data or task parallelism. The OpenMP parallel for loop directive makes it extremely simple to apply data parallelism to loops. On the other hand, the *par* statement creates an environment where it is much simpler to implement task parallelism which is free from race conditions.

3. Experimental Procedure

A total of six benchmarks programs are used, all with sequential, OpenMP and ForeC versions. Table 1 below shows the names and descriptions of each benchmark program.

Table 1: Names and descriptions of benchmarks.

Name	Description
FlyByWire	Flight computer simulation.
FmRadio	Reads and decodes a stream of bits.
Life	Game of Life - cellular automation, zero player game.
Lzss	Text file archiver.
Mandelbrot	Creates a Mandelbrot set image for display.
Matrixmultiply	Carries out multiplication operator between two matrices.

Intel® VTune™ Amplifier XE software was used during the development and testing of the parallelised benchmarks. The software was useful in providing insight into the regions of code which are the most time consuming and therefore top candidates for parallelisation. It was also used to calculate the total runtimes and show the number of cores being used during execution of each benchmark. Table 2 shows the specifications of the desktop computer on which all testing was carried.

Table 2: Testing desktop computer specifications.

Processor:	Intel Core i5-3570
No. of physical cores:	4
No. of logical cores:	4
Processor speed:	3.4GHz
Turbo Boost:	Turned Off
Speed Step:	Turned Off
RAM:	8GB
Operating system:	Ubuntu 12.04 LTS
Compilers used:	gcc

4. Experimental Results and Discussion

Figures 6 and 8 show the speed up factors of OpenMP and ForeC benchmarks respectively. The values are calculated by dividing the sequential runtime by the parallel runtime. One, two and four core runtimes were recorded.

As expected, most of the OpenMP 1 Core runtimes are either the same as sequential or slightly higher. This is due to the unnecessary overheads added by the OpenMP directives; however, this would not be done in practice. The exception in this case is the FlyByWire benchmark as the sequential runtime is actually longer than the OpenMP 1 Core runtime. Upon further inspection using the Intel Vtune software, the main difference in runtime comes down to a 'move' instruction in assembly code. The same instruction is evident in both versions, and therefore the difference is put down to variations in stack retrieval during the instruction.

```
void computation(void) {
    double result = 1.0;

    int i;
    #pragma omp parallel for reduction(+:result)
    for (i = 1; i < 2000; i++) {
        result += (double)((int)result % i) * (result / result);
    }
}
```

Figure 6: Code snippet from FlyByWire benchmark showing parallelised loop.

The level of speed up acquired from two core and four core OpenMP results vary between benchmarks. The variation is dependent on how well the benchmark program lends itself to parallelism, and in the case of

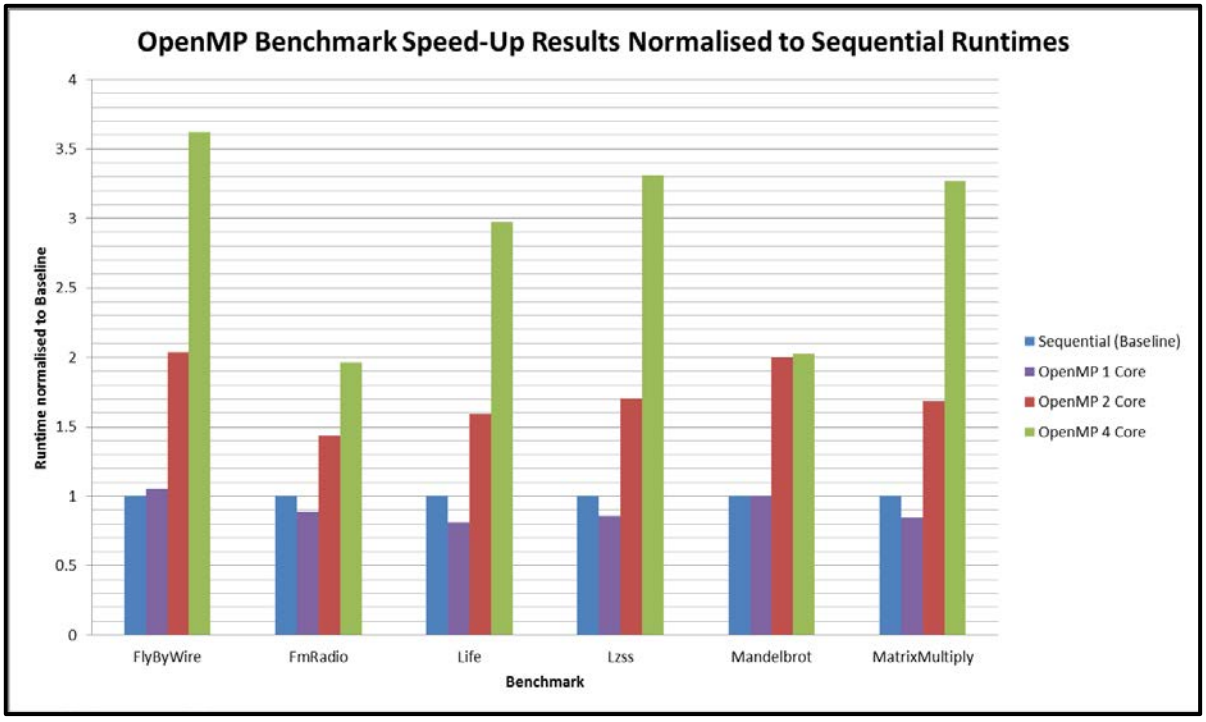

Figure 7: Graph showing OpenMP benchmark speed up results for 1, 2 and 4 Core compared to sequential.

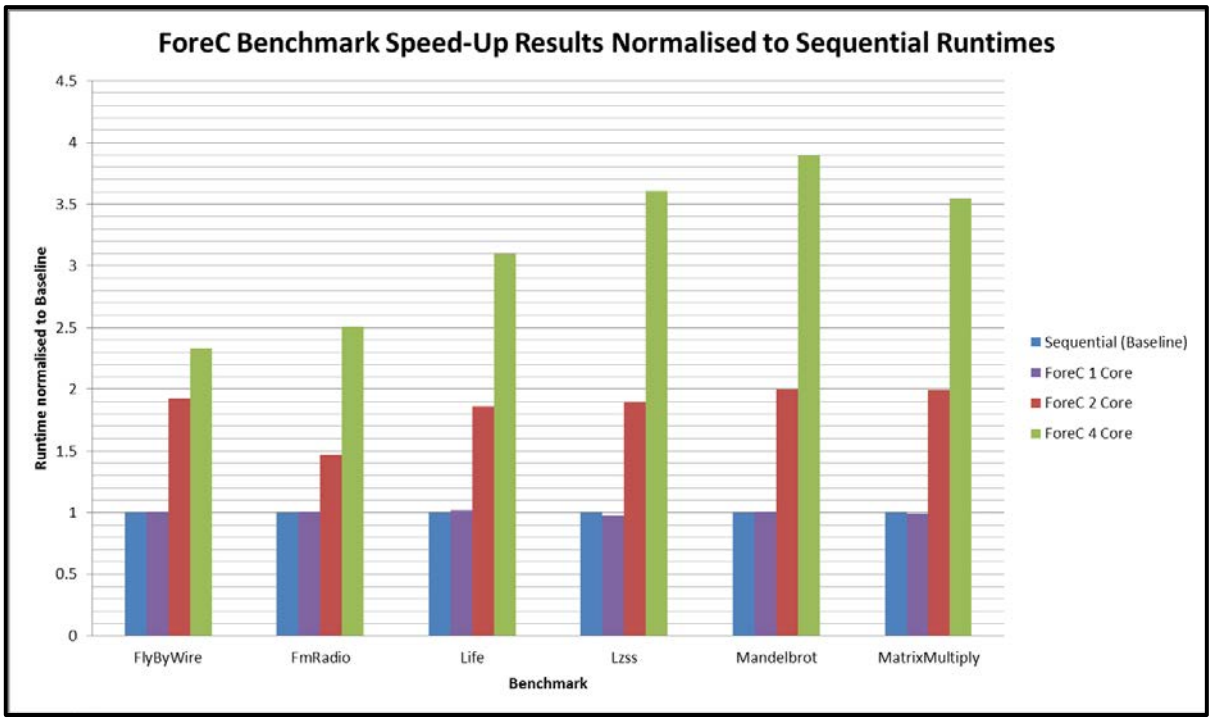

Figure 8: Graph showing ForeC benchmark speed up results for 1, 2 and 4 Core compared to sequential.

OpenMP, how well data loop parallelism can be applied to the program. FlyByWire is an excellent example of successful data loop parallelism. This is due to the fact that the most computationally intensive and time consuming loop in the program is at the lowest level of function calls and is called by almost all other higher level functions. This allows the parallelisation of that single for loop to greatly decrease the runtime of the overall program. Figure 6 shows parallelised loop using reduction. FlyByWire is able to achieve a speed up factor of 2.04 with two cores and 3.62 with four cores. On the other hand, the FmRadio benchmark program does not lend itself to simple data loop parallelisation and therefore only achieves a speed up factor of 1.44 with two cores and 1.96 with four cores. Unfortunately the Mandelbrot benchmark program was unable to utilise the total number of cores and therefore the result is much lower than expected. Intel Vtune shows that only two cores are utilised. A possible reason for this could be that OpenMP sometimes does not use all available cores as it could overload the system. As Mandelbrot is the largest benchmark with a great number of loop iterations, this reasoning seems feasible.

The ForeC benchmark results, shown in Figure 8, depict more consistent speed up factors compared to OpenMP. The ForeC 1 Core runtimes are effectively equal to the sequential runtimes, meaning there is insignificant overhead added to the computation of the program. The Mandelbrot benchmark program is the most successful in being parallelised with a speed up factor of 2.00 using two cores and 3.90 using four cores.

Contrary to the OpenMP results, the FlyByWire ForeC benchmark does not provide a high speed up factor. This is due to structure of the program suiting the data loop parallelisation rather than the high level task parallelism that is implemented using ForeC. The task parallelism requires the programmer to balance the tasks and loads on each core in order to have each process complete at approximately the same time. However, there will always be discrepancy, and therefore downtime whilst processors wait for the most time consuming process to complete so the forks can join back to the master thread. This is the case in FlyByWire, where threads are required to wait for synchronization, hence, data loop parallelism is far more optimal. FmRadio benefits from the task parallelism as a larger portion of the computation is able to run concurrently resulting in an increased speed factor of 2.51.

Figure 9 shows a direct comparison between OpenMP and ForeC 4 Core speed up results. The runtime results

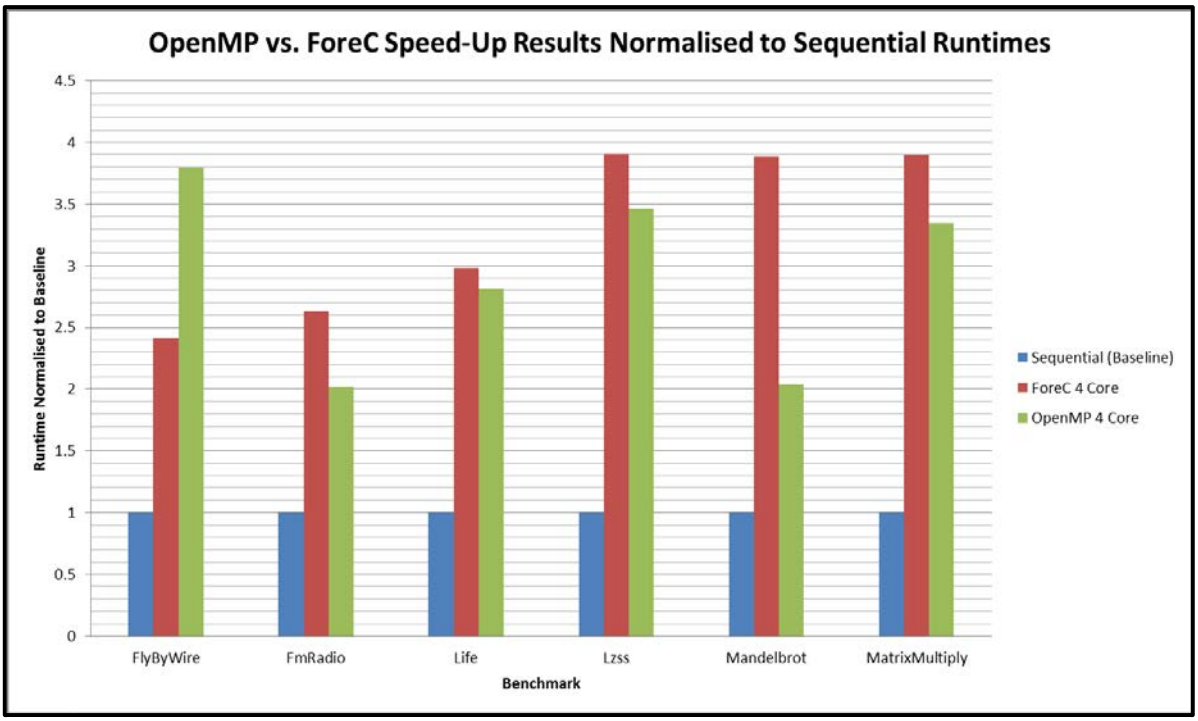

Figure 9: Graph showing OpenMP vs. ForeC benchmark results for 4 Core compared to sequential.

for this comparison were averaged over 200 iterations of each program. This was done to take into account any potential effects from long term use, e.g. filling and flushing of the cache. Both parallel programming languages were able to produce a speed up factor of between 2 and 4 when utilising a four core processor for each given benchmark. However, from the results it is clear that in most cases ForeC produces a greater speed up factor, barring two exceptions. Firstly, OpenMP was more suited and delivered a much higher speed up factor in FlyByWire. Secondly, the Mandelbrot benchmark is an unfair comparison as the OpenMP version did not utilise all four available cores.

Although there was room to improve the performance of the actual code, it was kept constant between the three versions to maintain a level playing field. However, it is difficult to accurately compare the performance of the two parallel programming languages using these benchmarks as there is much discrepancy involved. There exist many different ways the benchmarks could have been created which could have resulted in higher performance. Also, the nature of ForeC requires the structure of the original code to be changed to suit the different design paradigms, e.g. Divide and Conquer. OpenMP benchmarks, on the other hand, were all developed in keeping the original structure of the code and only adding directives. These structural differences may introduce variances in the performance to begin with.

Overall, it is clear that the results demonstrate that ForeC is comparable to OpenMP as a legitimate parallel programming language in terms of performance.

5. Conclusions

ForeC provides some benefits to the user when writing parallel programs compared to OpenMP. It makes it easier for the user to write code free from race conditions, which is very important. Although both parallel programming languages are able to achieve data and task parallelism, they both excel in different areas. OpenMP allows for easy data loop parallelisation of simple for loops; however the user must make sure it is free of race conditions and is functionally correct. ForeC semantics make for easier task parallelisation. Despite the differences in the structure of benchmarks and the types of parallelisation used between the two, the results show that ForeC is comparable, if not superior, in terms of performance compared to OpenMP.

6. Acknowledgements

I would like to thank Professor Partha Roop for supervising this research project and Dr. Avinash Malik and Eugene Yip for their help and guidance throughout.

7. References

1. *Data Parallelism in OpenMP*. *Introduction to Data Parallel Algorithms*. Mary Hall, September 9, 2010. The University of Utah. [Online]. Available. http://www.cs.utah.edu/~mhall/cs4961f10/CS4961-L6.pdf
2. *TotalView Part 3: Debugging Parallel Programs*. Blaise Barney, June 18, 2013. Lawrence Livermore National Laboratory. [Online]. Available. https://computing.llnl.gov/tutorials/totalview/part3.html
3. E. Yip, P.S. Roop and M. Abhari, "Programming and Timing Analysis of Parallel Programs on Multicores," in *International Conference on Application of Concurrency to System Design*, Grenoble, France, 2013, pp. 167 – 176.