Energy Consumption of Parallel Patterns

Toms K. Popdjakuniks
ID: 2420042
AC40001 Honours Project
BSc (Hons) Computing Science
University of Dundee, 2024
Supervisor: Prof. V. Janjic

Abstract **-** With ever-increasing awareness of the impact computer systems have on the environment and the push for more energy efficient systems, an investigation into energy consumption is vital. Parallel patterns are implementations of common parallel programming applications designed to reduce the complexity of parallel programming. The aim of this project is to measure the energy consumption of parallel patterns by developing a library which contains two patterns - farm and pipeline - and applications that utilise them, then measuring the energy consumption based on the number of threads utilised. In addition, for a more comprehensive data set, the same applications are also implemented with the use of Intel's Thread Building Blocks library. This project finds that while increasing the thread count in a parallel pattern does increase instantaneous power draw, the decrease in execution time significantly outweighs the increase in power draw, and results in lower total energy consumption. Further research into other patterns, and additional implementations of the existing patterns is recommended.

1 Introduction

Over the last few decades, the main driving force of innovation within computing has been the relentless pursuit of performance, often at the cost of energy efficiency. Lately, however, we have become increasingly aware of the effects our systems have on the world. With companies all over the world pledging to reduce their energy consumption and carbon footprint, the impetus to reevaluate and address energy efficiency within computer systems has never been more obvious. It stands to reason that even slight reductions in energy consumption would have a significant impact, given the ever increasing computerization of our world.

Ever since the beginning of wide-spread adoption of computers in the 1980's, there has been an effort to increase processing power. The leading paradigm until the mid-2000's was frequency scaling, where the computational power of a processor was increased by simply increasing the clock frequency. This worked well at the time, but since raising the frequency also raised the power consumption, it eventually reached a point where further increasing the frequency would be detrimental to the overall system due to power usage and thermal constraints.

This meant that a new solution was needed for the ever-increasing demand for computing power. This is where parallel computing first comes into focus. Multi-core systems are a direct answer to the issues of late-stage frequency scaling processors. They provide much more computing power, with less power consumed, and so, less thermal energy to worry about.

Since their widespread commercial adoption in the late 2000's, multicore systems have been the baseline for all modern computing. Of course, having multiple cores which can execute multiple tasks at the same time is quite different from having one core which can only execute tasks in sequence. This change required programmers to adapt, and implement ways to take advantage of the possibility of concurrency.

Parallel programming is used to create programs which can be broken up into smaller tasks and each task can then be assigned to a different core. Logically, this results in increased performance. However, parallel programming is significantly more complicated than implementing an application the usual way. Parallel programming has pitfalls like deadlocks and race conditions. Programmers have to take extra care to make sure data is handled correctly, which would be much easier with a sequential application.

This is where parallel patterns are used. Parallel patterns are implementations of many standard parallel computing principles, created to be easily used by other programmers without having to worry about the aforementioned issues like deadlocks. A well implemented pattern should take care of all of the complex details. Examples of implementations include Intel's Thread Building Blocks[1] library, Microsoft's Parallel Patterns Library[2], and openMP[3]. While these patterns are good for reducing the complexity of parallel programming, they do take a level of control away from the programmer. Generally, parallel patterns are thought to provide acceptable performance, as demonstrated by their popularity within the programming community, but there are still questions about their energy efficiency.

The goal of this project is to answer those questions - to gather and analyse data on the energy consumption of applications that utilise parallel patterns.

To accomplish this, several objectives have to be fulfilled. In order of importance:

1. Implement a farm pattern library.
2. Write at least three applications which use the farm pattern.
3. Gather data on the energy profile of the farm pattern by measuring the energy consumption based on the number of threads used..
4. Implement the pipeline pattern.
5. Write an application which uses the pipeline pattern.
6. Gather data on the energy profile of the pipeline pattern.
7. Revisit the pattern applications, and implement them with the TBB library.
8. Gather data on the TBB implementations.

The findings of this project have significance both from a purely theoretical standpoint and a practical one. Theoretical, in that measuring and understanding energy consumption within a widely used component of modern computing can foster further discussion within the scientific community. Practical, in that understanding the energy consumption of specific patterns and how certain variables affect this consumption, can directly influence further research and development efforts within the field.

With proper insight into the energy consumption profiles of parallel patterns and their variables, researchers can focus their attention on developing solutions to data-proven problem points. Thus, this project serves as a stepping stone towards sustainable computing practices.

2 Background

There are some aspects of this topic which we had to research to gain greater understanding of either the wider topic, or how to approach this project. First of all, we had to understand what exactly parallel programming is, second of all, we had to understand the ways we could gather the data we would need and what exactly energy efficiency is, and finally, insight into research performed by others would help us contextualise the results we achieved. The following section is divided accordingly.

2.1 Parallel Programming

Before attempting to implement a pattern, it's important to understand the basics of multithreading and parallel programming. The approach with the most control for the developer, and, consequently, the most pitfalls, is the explicit approach. Doing all of the work yourself by directly implementing parallelisation yourself.

There are a few ways to approach this. The way we did it was by using the "pthreads" library[4]. To practise multithreading we looked at resources available online, specifically, a Linux "pthreads" tutorial from the Carnegie Mellon's School of Computer Science [5] and a PDF file from Oregon State University on the use of "pthreads"[6]. Another way to do multithreading would be by using the C++ threads library. We decided to use "pthreads" out of personal preference.

The aforementioned "explicit" approach involves using these libraries to parallelise an application yourself. For example, in the case of a for loop, this would involve creating the function to be multithreaded, creating threads through the use of a relevant function, in the case of "pthreads" - the "pthread_create" function. When creating the thread, you can assign a function to that thread, and pass all the necessary parameters along with it. In the case of a one-hundred number long array, you could write four functions which loop through a specific section of the array - 0 to 24, 25 to 49, 50 - 74, and 75 to 99, and assign each function to a thread.

Explicitly coding multithreading for a for loop is not particularly complex, as a matter of fact, the only difference between using a pattern or doing it yourself is that you don't need to create the threads yourself when using a pattern. Other than that, the process is very similar. The complexity becomes more obvious when you look at the pipeline pattern.

The biggest issue, and the biggest selling point of parallel patterns, besides the fact that it requires less coding, is the existence of deadlocks. If you wish to explicitly parallelise an application with a pipeline, you will have to deal with deadlocks. Since the pipeline requires the passing of data from one function to another, a developer has to make sure the right piece of data is accessed at the right time, by the right function. Failure to do so can result in race conditions - data being unpredictably modified, and unexpected output. In addition, improper implementation of data handling will result in a deadlock, which is notoriously hard to debug. This is one of the biggest draws of parallel patterns. You don't have to worry about any of these issues, you just have to provide the data and the functions.

Once again, before attempting to create the pipeline pattern, we had to have an understanding of the basic principles, so we implemented an explicit version of the pipeline. The basic approach is to use queues. A pipeline stage accesses a queue and reads data from it, making sure to lock it using mutexes. Once the data is read, the lock is released. The data is then processed and written to an output queue. Once again, during the writing process, the queue has to be locked. During our research, we found a code fragment in a paper written by V. Janjic, C. Brown, and A. D. Barwell.[7] See Figure 1. Pipeline Stage below for the code snippet. While the paper itself does nearly the complete opposite of what our goal is by taking a parallelised piece of code and writing a sequential application, this code snippet was extremely helpful in conceptualising the basic logic behind the pipeline.

```
// Second stage reads an element from the input queue, adds 1 to it,
// and writes it to the output queue.
void *Stage2(void *arg) {
  int my_input, my_output;
  pipeline_stage_queues_t *myQueues = (pipeline_stage_queues_t *)arg;
  queue_t *myOutputQueue = myQueues->outputQueue;
  queue_t *myInputQueue = myQueues->inputQueue;

  for (my_input = read_from_queue(myInputQueue);
       my_input>0 || my_input == EOS;
       my_input = read_from_queue(myInputQueue)) {
    if (my_input != EOS) {
      my_output = my_input + 1;
      add_to_queue(myOutputQueue, my_output);
    } else { // EOS is a terminating token. Pass on if received.
      add_to_queue(myOutputQueue, EOS);
      break;
    }
  }
  return NULL;
}
```

Figure 1. Pipeline Stage

2.3 Energy Consumption

While the purpose of this project is not to fully understand what exactly energy consumption is and all of the factors that come into play within a system, there is value in at least having some basic insight. There have been quite a few papers written about the subject.

In “The Design and Implementation of PowerMill”[8], written in 1995, the paper discusses the eponymous PowerMill - a transistor level simulator created to simulate current and power behaviour in circuits. The paper, like the title indicates, describes the design and implementation of the simulator, describing the algorithms used and providing proof of accuracy by comparing their data with a verified source. In their conclusion, they mention the program’s active usage in commercial sites and recommend further work in developing a comprehensive set of tools to help designers with power related problems. This paper shows that even back in 1995, designers and developers had to deal with power consumption issues. As time has passed, and system power has increased, these issues have only grown.

Further, in “The Benefits of Event–Driven Energy Accounting in Power-Sensitive Systems”[9], written in 2000, the paper makes a case for event-driven energy accounting. The paper takes a moment to describe how certain system components affect a system’s energy pattern, before moving on to investigate the relation between certain readings from event counters and energy consumption. Before concluding, they propose some energy-aware scheduling strategies to be considered. The paper concludes that the more the operating system is aware of the hardware, the better it can allocate the system’s power. They contend that thread-specific and event-driven energy accounting is imperative for power-sensitive systems.

Finally, in “Variations in CPU Power Consumption”[10], the paper attempts to fill a gap in CPU power consumption research, by investigating the disparity in CPU power usage in nominally identical systems. They do this by conducting experiments on several different processor types, for each type comparing several different physical processors in otherwise identical systems. They conducted tests by running several different CPU bound tasks, like SHA256 hashing, data compression and others. Of course, data was also gathered from idle systems. After gathering and analysing data, the paper concluded that individual processors have a great, yet unexplained effect on power consumption. They found that different processors can have differences in idle power consumption of up to 29.6%. At high load, newer processors tended to have less deviation, as they have more advanced throttling mechanics which minimise differences at the power limit.

While it’s clear that, in general, there is plenty of research on CPU power consumption, less work has been done on the effect parallel patterns in specific have on energy consumption. Our goal is to do some of that work.

2.4 Measuring Energy Consumption

Another important aspect to consider is the topic of measuring energy consumption. For this project, we focused on the CPU power efficiency, as the format of the developed code doesn’t have a graphical aspect. This means that GPU power usage wouldn’t be significant. As for the way we measured this power, there are a few different methods that we could have used.

The first method is to use software to monitor power usage. This is the method we chose. Specifically, we used AMD’s Ryzen Master software, which provides useful readings from the sensors within the CPU. We specifically focused on the CPU power metric, measured in Watts. This method is easy and cheap, though it does open the possibility of human error, as all the measurements have to be noted manually. Some power supply units do offer similar applications, which would be better, as they allow for monitoring the whole system.

Another method would be to use hardware. There are commercially available power meters which can be attached to either the power socket, or the psu, depending on the meter, and measure the power usage. This method would be more precise, as it directly measures the power used, which avoids some of the inherent issues of software monitoring tools, which can be imprecise. We didn't choose this method as it would add another potential point of failure, and increase the costs of the project.

Given the fact that multicore processors are nearly ubiquitous at this point, obviously, we are not the only ones who have done research into this topic. In an effort to limit our biases, we decided against looking at other research until a lot of work had already been finished. Having said that, we recognise that there is value in comparing our conclusions to the work of others, as any disagreements could result in a valuable point of discussion.

2.4 Similar Research

A paper published in the International Journal of Advanced Computer Science and Applications by R. Isidro-Ramirez, A. M. Viveros, and E. H. Rubio[11] sets out to do something similar to our project. The paper presents an analysis of different types of multicore-based architectures. They first propose a model for energy consumption, this model being an extension of Amdahl's law. Afterwards, they confirm the model through experiments. The paper utilises the Linpack benchmark to stress the CPU. Through analysis of data gathered, the paper concludes that multi-core architecture offers a lower limit of energy consumption, though they show that cases exist where applying parallelisation does not increase energy efficiency.

Another paper in the International Conference on Green Computing by C. Tseng and S. Figueira also investigates the potential energy consumption decrease offered by multithreading. The paper evaluates the power consumption of a processor under different conditions. For testing, they used the Jacobi algorithms as benchmarks. This paper differs from ours in the variables used, in that, they tune the frequency as well as the number of threads. The paper concludes that the optimal configuration in terms of energy efficiency will also be the optimal configuration in terms of performance.

While the goal of their paper is not exactly the same, D. De Sensi[13] takes a similar approach, by utilising the PARSEC benchmark applications in their testing and taking similar measurements. They go a step further by attempting to create a model to predict the performance and power consumption based on common variables.

From these papers, it's clear that parallel computing and parallel patterns are generally accepted within the computing community to be more energy efficient than sequential programming.

3 Specification

As the goal of this project is data analysis, the code was written entirely as a means to gather said data. As such, all design decisions revolved around having a great degree of control when the time came to perform measurements.

The library needed to be universal - capable of being used for any and all applications without the need to make any drastic changes. Given that two patterns were chosen from the outset to act as the main research subject, we needed to make sure the final version contained both the pipeline and farm patterns.

Additionally, an important part of the study is not just how much energy is consumed when a pattern is used, but how that consumption changes when a variable like the thread count is changed. We needed to make sure that it was simple to change this variable.

We created applications that would utilise these patterns for testing - a simple matrix multiplication program as well as a program that performs simple addition for the farm pattern, and an array of simple functions for the pipeline pattern.

We decided to approach development in segments. The first two weeks were dedicated to researching the topic and testing the "pthreads" library which would be instrumental to development.

After that, the next month would be dedicated to implementing the farm pattern, testing it, and developing the applications which would be used in data gathering. As the farm pattern is not particularly complex, the allocated month was divided in half, where the first half was spent developing the pattern, and the second half on working on the applications.

After that, we moved on to the pipeline pattern, once again allocating a full month to work on it. Since the pipeline pattern involved the use of mutexes and was more involved than the farm pattern we recognized that issues may arise, and development may take longer than predicted, so we allocated two to three weeks to development. We could afford to take this extra time, as we didn't need to develop multiple applications for the pipeline.

4 Implementation and Testing

We have split the following section into three main sections. First of all, the library, where we describe the implementation of the pattern library. Second of all, how we created and used the benchmark applications. And finally, a section on our usage of other libraries, in our case, TBB.

4.1 Library

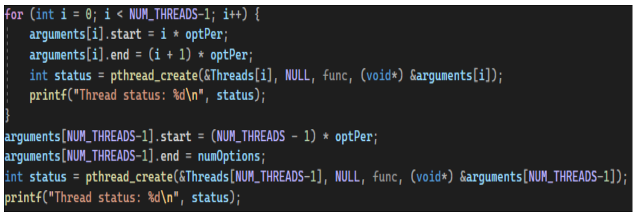
Before any applications could be created, and measurements taken, we had to implement the pattern library. We began with the farm pattern, as it is, arguably, the most simple pattern to implement and use.

We first began by implementing a regular simple application which utilised a for loop, which we would then attempt to parallelise. We choose to do a simple addition function. The loop would go through an array of numbers and simply add one to each number.
The goal was to then implement parallelisation utilising the "pthreads" C++ library. Before attempting to create a generic version of the pattern, we decided to make sure the multithreading library was working as expected, so we created four functions where each function worked on its own section of the array. These functions were then assigned to their own threads and made to run in parallel. Receiving the expected results, with the predicted speed increase we could move on to implementing the first pattern in our library.

Ideally, the implementation of the pattern should be easy to use, and automate all of the steps taken when manually parallelising an application. Our implementation receives as arguments the total length of the array, which is used to divide an equal amount of work between all available threads, and the function which will be performed on the data. The user still has to implement the function themselves. The user also has to create a struct for the purpose of storing the variables which will be used when creating the loop.

The function created by the user is then generic instead of specific values. Usually, when creating a for loop the programmer in some way sets the number of loops, either by specifying a number, or allowing a variable to be set, but in this case, the number of loops will be provided by the library calling the function. The number of loops is decided by the total size of the data to be processed, and the number of threads.

Within the library, after receiving the size of the data and the function, the library calculates the start and end point of the loop. Then, as many threads as the user chooses are created, and the user provided function, along with the required variables, is passed to said threads. See Figure 2. Thread Creation below. Afterwards, the library waits for all threads to signal that the work is finished, before stopping.



```
for (int i = 0; i < NUM_THREADS-1; i++) {
    arguments[i].start = i * optPer;
    arguments[i].end = (i + 1) * optPer;
    int status = pthread_create(&Threads[i], NULL, func, (void*) &arguments[i]);
    printf("Thread status: %d\n", status);
}
arguments[NUM_THREADS-1].start = (NUM_THREADS - 1) * optPer;
arguments[NUM_THREADS-1].end = numOptions;
int status = pthread_create(&Threads[NUM_THREADS-1], NULL, func, (void*) &arguments[NUM_THREADS-1]);
printf("Thread status: %d\n", status);
```

Figure 2. Thread Creation

Upon changing the addition function to utilise the library function and confirming that the results return exactly as expected, we could move on to creating the applications which we would use to gather the data for the final analysis. We decided upon three applications - an implementation of the Black-Scholes calculation, matrix multiplication, and a function which sums all previous numbers. These applications were picked due to their differing computational intensity and their ease of scalability. This is covered in the Benchmark section.

After enough testing had been done to ensure all of the applications were performing as intended, we could move on to implementing the second pattern - the pipeline.

The initial process was exactly the same as for the farm pattern. We first had to create a simple example application and implement pipelining directly, before attempting to add it to the library. The biggest hurdle for the pipeline is the need for mutexes. Since in the pipeline two several functions have to access the same data containers - in this case intermediary queues - deadlocks are possible. Proper utilisation of mutexes is required to make sure the application can correctly execute.

Initially, we created three simple functions which would be utilised by the pipeline. The first function would add one to a number, the second function would multiply a number by two, and the third would once again add one to a number. These functions were chosen, as the final result would be noticeably different, if they were executed out of order.

As there were three functions, two queues would need to be created, together with their corresponding mutexes and conditions. These would be used to pass data from from the first function output to the second function input and similarly between the second and third function. Only the intermediary queues require the mutexes and conditions, as the initial queue and the final queue are only accessed by the first and last function respectively, so no race condition can exist.

Then, we added a simple initialisation function to the library. This function adds the three functions to their own separate thread and waits for them to finish their work.

Once again, the most basic test was simply to run the program, and evaluate the final output. Additionally, to check if any deadlocks existed, we had to repeatedly execute the application, as a deadlock was not guaranteed, even if it was possible. We also greatly increased the size of the data, as the longer the application took to finish, the higher the chances of spotting an issue.

Having tested the initial implementation and being sure that we understood the general principle, we could attempt to implement the pattern within the library.

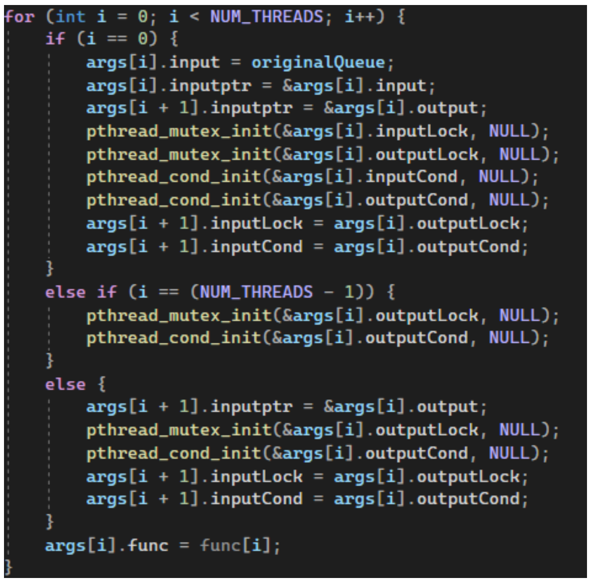
We decided to implement two different approaches to the pipeline. The first exists to give the user a greater degree of control, specifically, with the type of data and functions.

The first implementation requires the user to handle the mutexes and deadlocks themselves. While this does allow for mistakes, this also allows for greater control. The previously implemented basic initialization function could be used for this. The only alteration was that we extracted the processes for writing to and reading from a queue into their own functions within the library. All the user has to do is pass in the queue, the mutex, and the condition, and the locking and unlocking will be handled automatically.

The second implementation is easier to use, but also more specific, as it only works if the data is integers. This implementation was chosen because it would be directly helpful to the current project, as all of the applications are working only with integers.

To create this automatic pipeline, we needed to create another struct which would hold all of the data required to execute a whole pipeline stage - the mutexes and conditions for both the input and output queues, the queues themselves, and the user provided function which would be performed on the data.

The function which the user calls would be responsible for generating an array of these structs to be used by the pipeline function. It would also populate the relevant fields, like setting the first struct's input queue as the start queue provided by the user. It would also make sure the relevant mutexes and conditions are available to the correct threads. These threads would then be initialised and the pipeline function assigned to each thread. See Figure 3. Pipeline Struct Initialisation below.



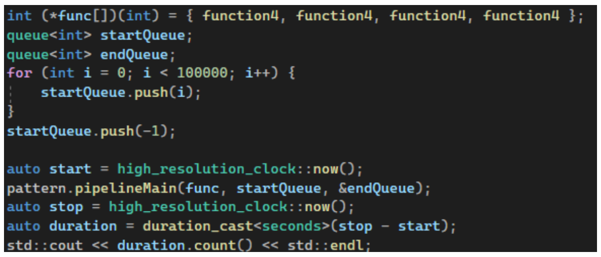
```
for (int i = 0; i < NUM_THREADS; i++) {
    if (i == 0) {
        args[i].input = originalQueue;
        args[i].inputptr = &args[i].input;
        args[i + 1].inputptr = &args[i].output;
        pthread_mutex_init(&args[i].inputLock, NULL);
        pthread_mutex_init(&args[i].outputLock, NULL);
        pthread_cond_init(&args[i].inputCond, NULL);
        pthread_cond_init(&args[i].outputCond, NULL);
        args[i + 1].inputLock = args[i].outputLock;
        args[i + 1].inputCond = args[i].outputCond;
    }
    else if (i == (NUM_THREADS - 1)) {
        pthread_mutex_init(&args[i].outputLock, NULL);
        pthread_cond_init(&args[i].outputCond, NULL);
    }
    else {
        args[i + 1].inputptr = &args[i].output;
        pthread_mutex_init(&args[i].outputLock, NULL);
        pthread_cond_init(&args[i].outputCond, NULL);
        args[i + 1].inputLock = args[i].outputLock;
        args[i + 1].inputCond = args[i].outputCond;
    }
    args[i].func = func[i];
}
```

Figure 3. Pipeline Struct Initialisation

Next, we needed to create the generalised function which would use these variables. Since the general flow of data is quite simple, the function is not particularly complex. The previously mentioned function for reading from a queue is invoked with the variables provided from the struct, the data is processed using the function provided by the user and the result is written to the output queue.

After this was implemented, we needed to do rigorous testing to make sure no deadlocks occur, and the output is as expected. Close attention needed to be paid to the assignment of the queues within the initialisation function, as pointers had to be used and misuse could result in inconsistent data, and incorrect output.

To utilise this implementation of the pipeline function, the user has to provide an array of functions to be used, the start queue and the end queue. The library will handle the rest of the pipeline implementation. See Figure 4. Pipeline Usage below.



```
int (*func[])(int) = { function4, function4, function4, function4 };
queue<int> startQueue;
queue<int> endQueue;
for (int i = 0; i < 100000; i++) {
    startQueue.push(i);
}
startQueue.push(-1);

auto start = high_resolution_clock::now();
pattern.pipelineMain(func, startQueue, &endQueue);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<seconds>(stop - start);
std::cout << duration.count() << std::endl;
```

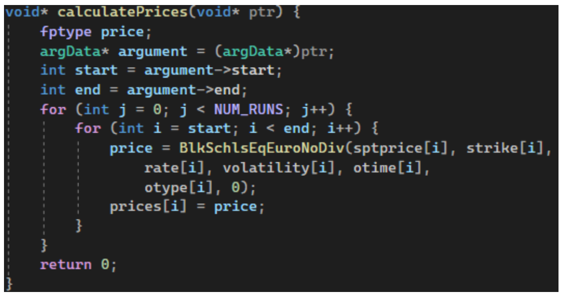
Figure 4. Pipeline Usage

Finally, we needed to create an application used for data gathering. Once again, this is covered in the Benchmark section.

Initially, we required the user to provide the length of the queue to be used by the pipeline function to know when to stop waiting for the next input, but after some deliberation, this was changed. Now, the function halts after receiving an end of stream symbol. Currently, the symbol is set to "-1" as our application uses only positive integers, but this can easily be changed to suit the user's needs. This was done as typical implementations of the pipeline expect extremely long series of data. Changing the halt condition to be an end of stream symbol allows for the input to be essentially infinite.

4.2 Benchmarks

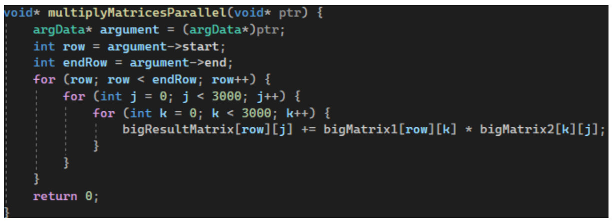
The first application to be implemented using the library was the Black-Scholes calculation. Most of the implementation was taken from the PARSEC[13] benchmark implementation and adapted to work with our pattern library. The adaptation was fairly simple, as the original implementation was done in C, which meant there was little work to be done. The Black-Scholes calculation would provide a solid baseline of data, as the calculation was reasonably complex, but also easily scalable, as part of the original implementation was the number of repeated calculations. By increasing this number, the execution time would increase dramatically. See Figure 5. Black Scholes Implementation With Project Library below.



```
void* calculatePrices(void* ptr) {
    fptype price;
    argData* argument = (argData*)ptr;
    int start = argument->start;
    int end = argument->end;
    for (int j = 0; j < NUM_RUNS; j++) {
        for (int i = start; i < end; i++) {
            price = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i],
                rate[i], volatility[i], otime[i],
                otype[i], 0);
            prices[i] = price;
        }
    }
    return 0;
}
```

Figure 5. Black Scholes Implementation With Project Library

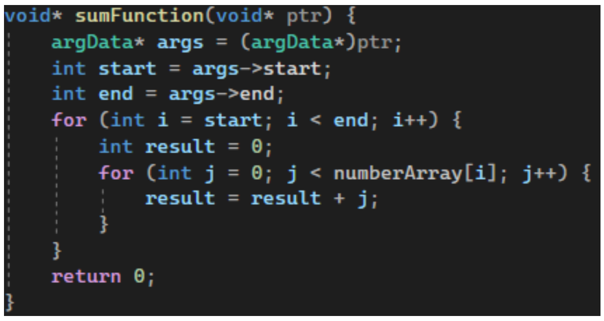
The second application to utilise the farm pattern was matrix multiplication. Within computing, matrix multiplication is a fairly simple problem that most programmers have likely implemented a solution for themselves. The benefit of this application is once again its scalability. Simply by increasing the size of the matrices, we can drastically increase the execution time, which would give us more precise measurements. As unlike the other two applications this application uses a two-dimensional array, the usage of the library pattern is slightly different. Usually, the library receives the total size of the data, but in the case of two-dimensional arrays, it instead receives the number of rows, which then get divided among the threads. See Figure 6. Matrix Multiplication Implementation With Project Library below.



```
void* multiplyMatricesParallel(void* ptr) {
    argData* argument = (argData*)ptr;
    int row = argument->start;
    int endRow = argument->end;
    for (row; row < endRow; row++) {
        for (int j = 0; j < 3000; j++) {
            for (int k = 0; k < 3000; k++) {
                bigResultMatrix[row][j] += bigMatrix1[row][k] * bigMatrix2[k][j];
            }
        }
    }
    return 0;
}
```

Figure 6. Matrix Multiplication Implementation With Project Library

The final application created for the farm pattern was a simple function which reads a number and proceeds to calculate the sum of all integers up to the number itself. Once again, this application is easily scalable, as, the larger the numbers provided, the longer it will take. See Figure 7. Sum Function Implementation With Project Library below.



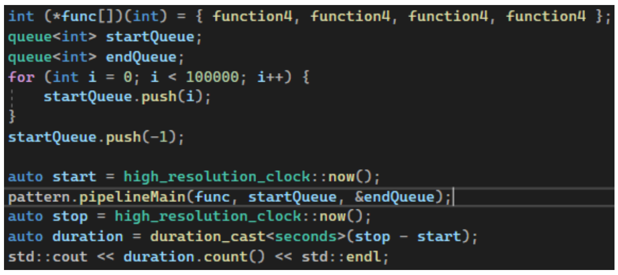
```
void* sumFunction(void* ptr) {
    argData* args = (argData*)ptr;
    int start = args->start;
    int end = args->end;
    for (int i = start; i < end; i++) {
        int result = 0;
        for (int j = 0; j < numberArray[i]; j++) {
            result = result + j;
        }
    }
    return 0;
}
```

Figure 7. Sum Function Implementation With Project Library

These three applications were created to make sure the data isn't anomalous. If we had only tested using one application, we couldn't actually be sure if the data is actually accurate or if something was simply wrong with the implementation.

The testing process for all of the implementations was straightforward, as all of the applications had to be implemented sequentially first. This meant that we already had the expected output for a certain set of data. If the output was still the same after rewriting the applications to use the farm pattern, we could be reasonably sure that the library was performing as we expected.

As for the pipeline pattern, we chose to utilise the same application we used for the farm pattern, the "sum of all numbers" function. In this case, the complexity of the function wasn't as important as the scalability. Simply by increasing the initial array to be a large enough number, we could easily monitor the data moving through the pipeline and at the same time test for any deadlocks. See Figure 8. Pipeline Call With Project Library below.



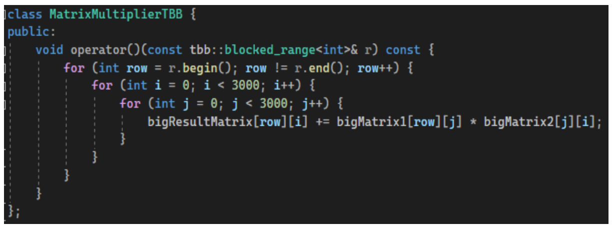
```
int (*func[])(int) = { function4, function4, function4, function4 };
queue<int> startQueue;
queue<int> endQueue;
for (int i = 0; i < 100000; i++) {
    startQueue.push(i);
}
startQueue.push(-1);

auto start = high_resolution_clock::now();
pattern.pipelineMain(func, startQueue, &endQueue);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<seconds>(stop - start);
std::cout << duration.count() << std::endl;
```

Figure 8. Pipeline Call With Project Library

4.3 TBB

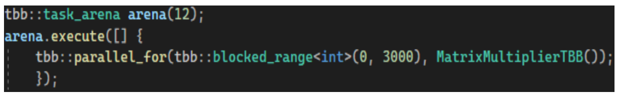
After both of the patterns were implemented and tested to a satisfactory degree for the needs of the project, all implemented applications were revisited with the intention of adding an additional implementation of parallelisation. For this we utilised Intel's TBB library.

The TBB library is a widely used and documented library, so the implementation was not difficult. We found that the farm pattern in particular was extremely similar in usage to our implementation of it. See Figure 9. Matrix Multiplication Function With TBB, Figure 10. Calling TBB Matrix Multiplication Function and Figure 11. Pipeline Call With TBB below.



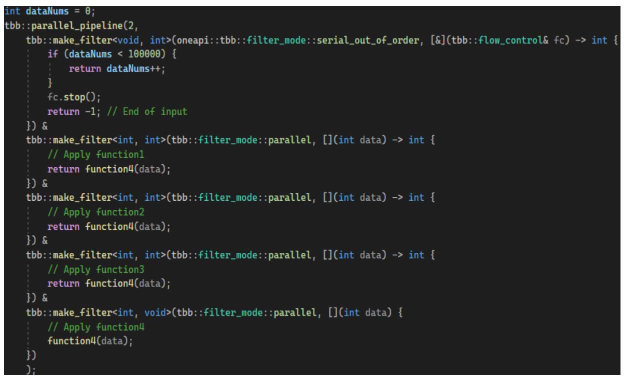
```
class MatrixMultiplierTBB {
public:
    void operator()(const tbb::blocked_range<int>& r) const {
        for (int row = r.begin(); row != r.end(); row++) {
            for (int i = 0; i < 3000; i++) {
                for (int j = 0; j < 3000; j++) {
                    bigResultMatrix[row][i] += bigMatrix1[row][j] * bigMatrix2[j][i];
                }
            }
        }
    }
};
```

Figure 9. Matrix Multiplication Function With TBB



```
tbb::task_arena arena(12);
arena.execute([] {
    tbb::parallel_for(tbb::blocked_range<int>(0, 3000), MatrixMultiplierTBB());
    });
```

Figure 10. Calling TBB Matrix Multiplication Function



```
int dataNums = 0;
tbb::parallel_pipeline(2,
    tbb::make_filter<void, int>(oneapi::tbb::filter_mode::serial_out_of_order, [&](tbb::flow_control& fc) -> int {
        if (dataNums < 100000) {
            return dataNums++;
        }
        fc.stop();
        return -1; // End of input
    }) &
    tbb::make_filter<int, int>(tbb::filter_mode::parallel, [](int data) -> int {
        // Apply function1
        return function4(data);
    }) &
    tbb::make_filter<int, int>(tbb::filter_mode::parallel, [](int data) -> int {
        // Apply function2
        return function4(data);
    }) &
    tbb::make_filter<int, int>(tbb::filter_mode::parallel, [](int data) -> int {
        // Apply function3
        return function4(data);
    }) &
    tbb::make_filter<int, void>(tbb::filter_mode::parallel, [](int data) {
        // Apply function4
        function4(data);
    })
    );
```

Figure 11. Pipeline Call With TBB

Having data gathered from applications implemented using both our library and the TBB library would help to strengthen our analysis. If the measurements from both implementations were similar, it would increase confidence in both our implementation of the patterns, and the data gathered from it.

5 Evaluation / Testing

Having finished development and testing, it was time to move on to data gathering and analysis.

For this project, the data gathering had to be handled manually. This decision was made, as setting up an environment to run the applications and log the power usage would be an added layer of complexity. Instead, as the application would do its work, the relevant readings would be noted by hand. All of the data was gathered on a system containing a Ryzen 5 5600 6 core processor, running Windows 11.

For the readings, AMD's "AMD Ryzen Master" application was used, as it provides the relevant CPU telemetry. For the purposes of this project, all that was required was the CPU power usage reading, which the AMD application provides.

Considering the drawback of potential human error when gathering the data, the decision was made to repeat each and every measurement for a set of variables at least ten times. For example, the Black-Scholes application would be executed with one thousand lines, and two hundred and fifty thousand runs, using four threads to do the calculation, measurements would be taken and noted in a spreadsheet. This would then be repeated ten more times.

For the farm pattern, measurements were taken for all three applications with the same exact basic variables, i.e., matrix size for the matrix multiplication application, number of runs for the Black-Scholes application, and the size of the array for the "sum of all previous numbers" application. The only difference would be the number of threads utilised. All applications would be tested on a range of threads from two to twelve as this would provide a reasonable data set for typical multithreading tasks.

As for what we can extrapolate from the data gathered. First of all, for the farm pattern, there is a clear increase in the momentary power draw when increasing the number of threads utilised. See Figure 12. Power Usage Based On Number Of Threads With Project Library below. The given graph is based on data gathered from the Black-Scholes application. The same correlation can be seen for the TBB implementation of the Black-Scholes calculation. See Figure 13. Power Usage Based On Number of Threads With TBB below.

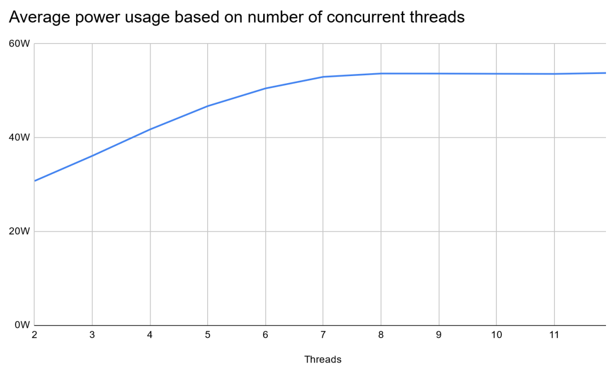


Figure 12. Power Usage Based On Number Of Threads With Project Library

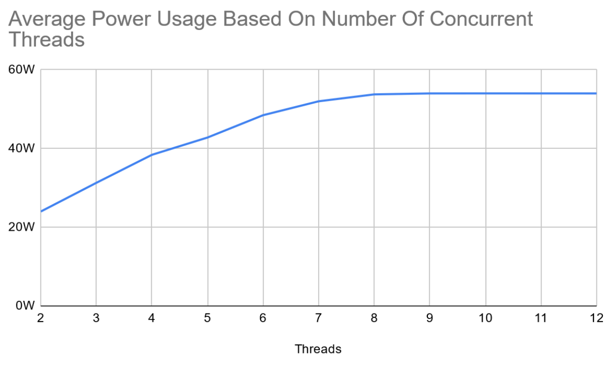


Figure 13. Power Usage Based On Number Of Threads With TBB

Something to note in the previous figures is the plateau of power usage when reaching around seven threads used. When investigating this, we noticed that at seven threads used was when our system was reaching maximum total socket power. This value will be different for other systems, and something like overclocking would have likely allowed us to push the system to greater limits, but we decided against it, as we wanted to have the data be representative of more typical systems.

While, in a vacuum it may seem that, if your goal is to be energy efficient, you would want to use as few threads as possible, the reality of it can be seen by looking at the graph showing the total power consumption. As the number of threads increases, the total energy consumption drastically decreases. See Figure 14. Total Energy Consumption Based On Number Of Threads With Project Library below. Once again, this same effect can be seen with the TBB implementation. See Figure 15. Total Energy Consumption Based On Number Of Threads With TBB below.

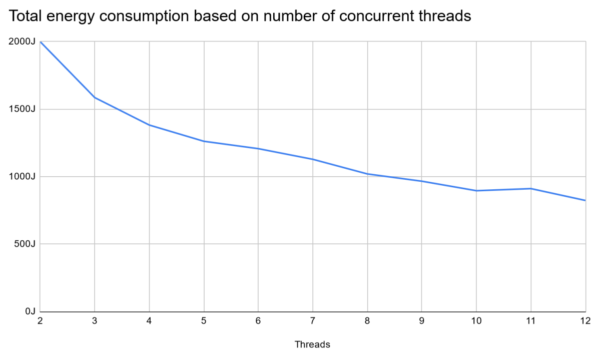


Figure 14. Total Energy Consumption Based On Number Of Threads With Project Library

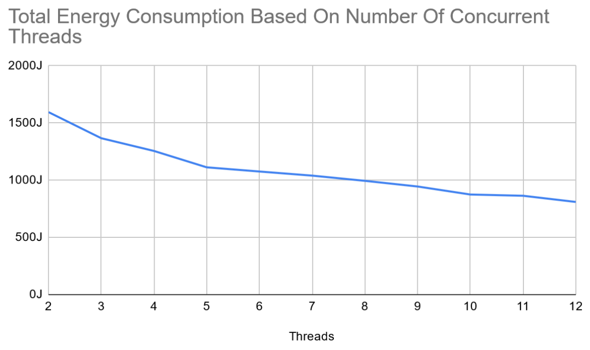


Figure 15. Total Energy Consumption Based On Number Of Threads With TBB

The reason behind this decrease of energy consumption is simple. Despite the fact that the momentary power usage increases, the efficiency improvement from using more threads is large enough to offset the increase. This efficiency improvement can be seen by looking at the time taken to complete the same task based on the number of threads. See Figure 16. Data On Black-Scholes Calculation With Project Library below. The third column clearly shows the drastic decrease in time taken as the number of threads increases. The same decrease can be seen with the data for the TBB implementation. See Figure 17. Data On Black-Scholes Calculation With TBB below.

Threads	Average, W	Average, s	Total Spent, J
2	30.76	65	1999.4
3	36.115	43.9	1585.4485
4	41.755	33.1	1382.0905
5	46.735	27	1261.845
6	50.505	23.9	1207.0695
7	52.965	21.3	1128.1545
8	53.67	19	1019.73
9	53.66	18	965.88
10	53.62	16.7	895.454
11	53.59	17	911.03
12	53.805	15.3	823.2165

Figure 16. Data On Black-Scholes Calculation With Project Library

Threads	Average, W	Average, s	Total Spent, J
2	23.987	66.5	1595.1355
3	31.285	43.7	1367.1545
4	38.4025	32.7	1255.76175
5	42.795	26	1112.67
6	48.47	22.2	1076.034
7	52.005	20	1040.1
8	53.76	18.5	994.56
9	54	17.5	945
10	54.01	16.2	874.962
11	54.005	16	864.08
12	54	15	810

Figure 17. Data On Black-Scholes Calculation With TBB

While this data, in tandem with the fact that the momentary energy usage plateaus, may make it seem that it is a good idea to push the number of threads as high as possible, due to the fact that the average time taken kept going lower even after thread count reached eight, this is not exactly true. It’s clear that the difference between threads used becomes much less drastic when reaching the plateau. This is because there is the same amount of power, but it keeps being divided between more and more threads.

As for the pipeline pattern, we have to approach it slightly differently, since the number of threads is dictated by the number of pipeline stages. The way we approached it was by measuring the relevant data by running all of the pipeline stages in sequence, like you would program it regularly, and comparing those results with the pipeline stages running in parallel.

From this we can see a similar energy profile to the farm pattern. There is a clear increase in momentary power usage between the pipeline stages being executed in sequence and them being executed in parallel, though the increase is much clearer with the project library than it is with the TBB library. See Figure 18. Power Usage Of Pipeline below. The total energy consumption is also similar to the farm pattern, with a clear decrease when using the parallel pattern, instead of the sequential execution. See Figure 19. Total Energy Consumption Of Pipeline below.

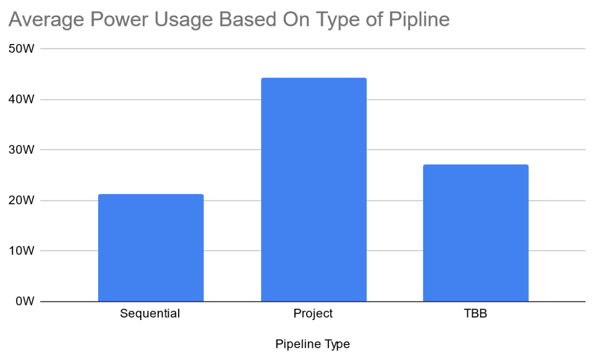


Figure 18. Power Usage Of Pipeline

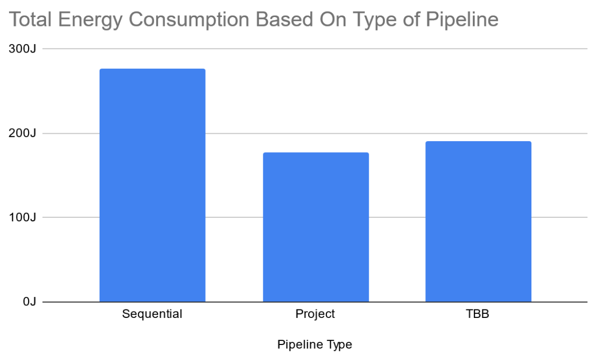


Figure 19. Total Energy Consumption Of Pipeline

Once again, this decrease in total energy consumption is explained by looking at the performance of the application with and without parallelisation. Clearly, using multithreading decreases the time taken. See Figure 20. Data On The Pipeline Pattern below. The third column shows the time taken by each implementation of the pipeline.

Type	Avg, W	Avg, s	Total Consumption
Sequential	21.335	13	277.355
Project	44.37	4	177.48
TBB	27.205	7	190.435

Figure 20. Data On The Pipeline Pattern

Considering all of the data gathered, there are few conclusions we can draw. First of all, increasing the number of threads clearly increases the momentary power draw. Since increasing the power usage increases the temperature of the CPU, this is notable in systems where thermals are important. Mobile phones are a notable example, as cooling is much more difficult, compared to computers, where a user can choose to upgrade their thermal solution. If thermals are of critical importance, a developer will have to carefully consider how many threads to use, to keep thermal readings within acceptable ranges.

Second of all, there is a clear decrease in total energy consumption when increasing the number of threads. If a developer only cares about energy efficiency, we can safely say that increasing the number of threads is a reliable way to be as energy efficient as possible.

In conclusion, we have accomplished what this project set out to do - gather and analyse data on the energy efficiency of parallel patterns and we have found that, for the purposes of conserving energy, there is clear merit to using parallel patterns.

6 Description of the final product

The final product of this project is a Visual Studio Project, which contains a parallel pattern library that was created for this project, and a few basic applications that were either created or adapted to be used with the pattern library.

The library contains implementations of the farm and pipeline patterns. If a user wishes to use these patterns, they have to add the library to their coding project as they would any other library.

To use the patterns, the user will have to create the functions they wish to be used and also provide a data set upon which these functions will be executed. For example, for the farm pattern, the user can create an array ranging from one to ten and a function which takes a value and adds one to it. Implementing both of these components according to specification and choosing the number of threads will allow the user to use the farm pattern. The library will take the length of the array, and split the array into even pieces to be assigned to different threads. Each thread will then execute the function on its section of the provided array.

To actually use the farm pattern, after the user writes the function, all they have to do is call the farm function from the library, while passing the length of the array and function itself as the parameters.

As for the pipeline pattern, there are two ways to approach it. First of all, if the data consists of integer numbers, the user can call the pipelineMain function - the user has to provide an array of functions which will be executed one after the other, an input queue, and an output queue. The library will automatically construct the pipeline, and execute the functions in the provided order.

The other method can be adapted to work with any data types, but requires much more work from the user. For this method, the user would have to create the pipeline themselves, which involves creating the intermediate queues, creating the relevant mutexes and conditions, and arranging them in a way that doesn’t result in a deadlock. The library does provide the functions ReadFromQueue and WriteToQueue, which take a queue, a lock, and a condition as parameters, with the WriteToQueue function also taking the data to be written as a parameter. Having written the pipeline stages, the user can call the pipelineInit function, which will parallelise the pipeline stages.
As for the applications created for the project, there are four in total, the Black-Scholes calculation, Matrix multiplication, and “sum of all previous numbers” for the farm pattern, and “sum of all previous numbers” within a pipeline format for the pipeline.

7 Appraisal

With the benefit of hindsight, as with any project, there are quite a few things we would have done differently, if we could start again.

A not insignificant amount of time was spent attempting to get the PARSEC Benchmark Suite to work on our computers. If we could do the project again, from the beginning, we would use a Linux OS instead of Windows, as PARSEC was developed with Linux as the intended OS. It does offer support for Windows, but, having been developed for Linux, it would have likely been easier to get it working, and get to testing earlier.

The data set is quite constrained. Ideally, we would have gathered the data from several different machines, but as it is, the data was gathered from one computer. To make data gathering as accurate as possible, every configuration was executed repeatedly and the average values were used for analysis, but data gathered from various sources would have better represented reality. Specifically, data from mobile devices like mobile phones and laptops could have been gathered, as energy efficiency is important for battery life.

Another aspect that impacted the data was the improper load balancing implementation of the library. This was especially noticeable in the “sum of all previous numbers” implementation. In this application, with the way the farm pattern is set up, the threads that handle the beginning parts of the array finish their assigned work much faster than the latter parts. This means that by the end of the process only the final thread is doing any work. This results in skewed measurements, as the CPU usage is significantly lower by the end than it was in the beginning. The best way to handle this issue would have been to implement proper load balancing, attempting to spread the amount of work evenly between all threads.

In our opinion, we spent too much time in the beginning stages practising the basics of multithreading. Our time would have been better spent immediately attempting to implement the farm pattern. The initial research into multithreading did not end up impacting the speed of the development, so I would advise spending less time on research in the beginning, and more time getting practical experience.

Another important aspect of the project was the data gathering period. While measures were taken to make the results as reliable as possible, i.e., doing at least ten measurements of the same application with the exact same settings, and working with the average values from all tests, this is still not an ideal solution, as the measurements relied on manual note taking. Ideally, an environment should have been set up with the capability of measuring power usage at any given moment, and logged as the applications were executed time and time again.

Another way to improve data gathering would have been by using a hardware approach to measuring the power consumption. Ideally, a power meter would have been used for the measurements, ensuring the greatest accuracy, and accounting for power usage changes in parts of the system other than the CPU.

8 Summary and Conclusions

Throughout the course of this project, we developed a parallel pattern library which contained implementations of two patterns, those patterns being the farm pattern and the pipeline pattern. This library was developed in C++, using Visual Studio.

In addition, four applications were either developed or adapted to utilise this parallel pattern library - three for the farm pattern and one for the pipeline. These applications were then extended to also use the TBB library, in an effort to gather a more comprehensive and reliable data set.

After the development period had finished, data was gathered using the developed applications. Efforts were made to ensure the data was reliable and robust, like doing several repeat measurements, and doing measurements with the same variables, but using a different parallel pattern library, in this case, TBB.

All of the objectives we set out to complete were completed, though future explorations of other patterns and other applications which use these patterns are recommended.

9 Future Work

The nature of this project is almost inherently expandable. Any continuation could easily start by expanding the list of parallel patterns handled by the pattern library. Then, reasonable applications for each pattern could be created and energy consumption measurements taken and subsequently analysed.

Another approach for expansion would be to mix patterns, for example, creating an application which utilises both of the already implemented patterns, i.e., making use of both the pipeline pattern and the parallel for loop.

On a purely code-based approach, there could be more improvements made in terms of ease of use. Functions could be optimised to use fewer arguments, and make better use of the arguments that are provided. This is especially true if one was to implement more complex patterns, as the ones currently implemented are relatively simple.

More applications could also be adapted to use the implemented pattern library. This would give some variety to the gathered data set, as the current data has been gathered from just the Black-Scholes program. Gathering the data from more applications and measuring similar values would increase confidence that the data is accurate and usable. One example that could be implemented would be the Frequency Mining calculation, also from the PARSEC Benchmark Suite.

For an even wider data set, one could do the measurements on a variety of systems. A data set gathered on a variety of different CPU’s would more accurately represent the energy consumption values you would encounter in the real world.

As mentioned within the appraisal section, another way to improve the library would be by implementing load balancing. Not only would this improve the accuracy of further measurements for applications with uneven workloads, it would also improve the speed of such applications, as threads that finish early could be used to do more work, instead of being idle until the application terminates.

References

[1] Intel openAPI Threading Library Manual https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.79pqjh

[2] Microsoft's Parallel Patterns Library learning suite https://learn.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170

[3] OpenMP Specification https://www.openmp.org/

[4] Official Pthreads linux manual page exported to html, created by Michael Kerrisk https://man7.org/linux/man-pages/man7/pthreads.7.html

[5] A tutorial on the use of Pthreads by Gregg Ippolito https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

[6] A tutorial on parallel programming with the use of Pthreads by the Oregon State University https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/pthreads.1pp

[7] Vladimir Janjic, Christopher Brown, and Adam D. Barwell. 2021. Restoration of Legacy Parallelism: Transforming Pthreads into Farm and Pipeline Patterns. Int. J. Parallel Program. 49, 6 (Dec 2021), 886–910. https://doi.org/10.1007/s10766-021-00716-z

[8]Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. 1995. The design and implementation of PowerMill. In Proceedings of the 1995 international symposium on Low power design (ISLPED '95). Association for Computing Machinery, New York, NY, USA, 105–110. https://doi.org/10.1145/224081.224100

[9]Frank Bellosa. 2000. The benefits of event: driven energy accounting in power-sensitive systems. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system (EW 9). Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/566726.566736

[10]Jóakim von Kistowski, Hansfried Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, and Samuel Kounev. 2016. Variations in CPU Power Consumption. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/2851553.2851567

[11] Isidro-Ramirez, R., Viveros, A. M., & Rubio, E. H. (2015). Energy consumption model over parallel programs implemented on multicore architectures. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *6*(6)

[12]Chengling Tseng and S. Figueira, "An analysis of the energy efficiency of multi-threading on multi-core machines," *International Conference on Green Computing*, Chicago, IL, USA, 2010, pp. 283-290, doi: 10.1109/GREENCOMP.2010.5598301.

[13]"PARSEC Benchmark Suite 3.0" created by B. Amos https://github.com/bamos/parsec-benchmark

Appendices

The appendices included are:

- Source code
- Spreadsheets containing data in PDF and XLSX format
- User manual
- Ethics Declaration Form
- Midterm Report
- Risk Assessment
- Project Poster in PDF and PPTX