CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Energy-efficient OpenMP Programming

Evaluating the Effects of Code Transformations and Runtime Parallelism Constructs on Energy Efficiency for OpenMP Programs

Master's thesis in Computer science and engineering

Axel Karlsson, Henrik Valter

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Energy-efficient OpenMP Programming

Evaluating the Effects of Code Transformations and Runtime Parallelism Constructs on Energy Efficiency for OpenMP Programs

Axel Karlsson, Henrik Valter

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Energy-efficient OpenMP Programming
Evaluating the Effects of Code Transformations and Runtime Parallelism Constructs on Energy Efficiency for OpenMP Programs
Axel Karlsson, Henrik Valter

Supervisor: Miquel Pericàs, Department of Computer Science and Engineering
Examiner: Johan Karlsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Energy-efficient OpenMP Programming
Evaluating the Effects of Code Transformations and Runtime Parallelism Constructs on Energy Efficiency for OpenMP Programs
Axel Karlsson, Henrik Valter
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OpenMP is the de facto API for parallel programming in HPC applications. These programs are often computed in data centers, where energy consumption is a major issue. Whereas previous work has focused almost entirely on performance, we here analyse aspects of OpenMP from an energy consumption perspective. This analysis is accomplished by executing novel microbenchmarks and common benchmark suites on data center nodes and measuring the energy consumption. Three main aspects are analysed: directive-generated loop tiling and unrolling, parallel for loops and explicit tasking, and the policy of handling blocked threads. For loop tiling and unrolling, we find that tiling can yield significant energy savings for some, mostly unoptimised programs, while directive-generated unrolling provides very minor improvement in the best case and degenerates performance majorly in the worst case. For the second aspect, we find that parallel for loops yield better results than explicit tasking loops in cases where both can be used. This becomes more prominent with more fine-grained workloads. For the third, we find that significant energy savings can be made by not descheduling waiting threads, but instead having them spin, at the cost of a higher power consumption. We also analyse how the choice of compiler affects the above questions by compiling programs with each of *ICC*, *Clang* and *GCC*, and find that while neither is strictly better than the others, they can produce very different results for the same compiled programs. As a final step, we combine the findings of all results and suggest novel compiler directives as well as general recommendations on how to reduce energy consumption in OpenMP programs.

Keywords: OpenMP, energy efficiency, compiler optimisations, loop tiling, loop unrolling

Acknowledgements

We would like to thank our supervisor Miquel Pericàs for his support throughout the entire project, from the early days of clueless information searching to the final weeks of finishing this report.

Axel Karlsson and Henrik Valter, Gothenburg, June 2022

Contents

1
Introduction

Up until approximately the 1990s, computers were rapidly made faster due to increasing clock frequencies and exploitation of instruction-level parallelism (ILP), with little concern for power consumption [25]. Methods to increase performance using ILP generally didn't scale well and used significant amount of hardware resources for relatively little gain [25]. This was however not a problem due to Moore's law and Dennard scaling being in full effect and made these costs justifiable. Moore's law states that the number of transistors in a dense integrated circuit doubles approximately every two years [10], while Dennard scaling states that power usage stays in proportion with area as chips get smaller [25]. However, in the mid to end 1990s, power became a major problem as increasing processor frequencies would become impossible to cool, and Dennard scaling was declared broken around 2005-2007 [25]. To get around these problems and keep improving performance, the industry instead moved towards putting several cores on one chip, so thread-level parallelism (TLP) also could be exploited. This allowed further performance without relaying on additional costly ILP optimisations or increased clock frequency.

However, these new performance gains did not come free, which normally was the case for ILP optimisations, but instead required explicit effort by the programmer. To properly utilise these new multicore processors, code needed to be written in a way that took into account correctness issues, such as race conditions, starvation and deadlock, but also performance issues, like false sharing and minimising parallel overheads such as data synchronisation [24]. Generally, writing parallel code required significantly more effort than serial, and automatically converting serial code to efficient parallel code also turned out to be a very challenging task [13].

The difficulty of writing parallel code lead to the creation of parallel APIs such as OpenMP. It is a widely applied framework for fork-join model parallelism, especially in computer clusters running HPC workloads. These environments are in turn becoming increasingly more interesting due to cloud computing and offloading of heavy computations in areas such as machine learning inference [5]. This motivates an analysis of OpenMP, especially in terms of performance, but also in terms of power and energy.

Power efficiency is especially important in data center contexts, where cooling makes up a significant cost and challenge [3][10]. Increased heat dissipation raises the operating temperature, which in turn causes increased aging of hardware components causing degraded performance and shortened lifetime [17]. In addition to the challenges of heat dissipation, high energy consumption is also linked with low reliability in HPC systems [18]. Beyond the context of data centers, energy consumption is also highly important for mobile devices due to the obvious issue of battery lifetime.

OpenMP is not a programming language in itself, but rather an API that is called using compiler directives in C, C++ and Fortran. The semantics of these directives is defined in OpenMP specifications. Recently, in November 2020, the specification for OpenMP 5.1 was introduced [21]. This specification introduced the *tile* and *unroll* directives, which are loop transformation techniques used for optimisation. These optimisation techniques are described in the background, see section 2.2.
The most well-known directive is probably *parallel for*, which executes iterations of a loop in parallel using a team of threads. There is also the *task* directive, which packages a section of code to be possibly computed in parallel by another thread.
Another aspect of the runtime aspect of OpenMP (and parallel programming in general) is what a thread should do when blocked by another thread (such as on a barrier). There are in general two solutions: running an idle loop (spin-locking) or descheduling the thread. In OpenMP, such behavior can be controlled with an environment variable called the waiting policy.

1.1 Our contribution

In short, we answer the following research questions:

- **RQ1**: How much impact do code transformations such as the recently introduced unrolling and tiling have on energy consumption?
- **RQ2**: How do the runtime parallelism constructs of OpenMP (tasks and parallel loops) compare in terms of energy?
- **RQ3**: How does the OpenMP waiting policy affect power and energy?
- **RQ4**: How do different implementations of OpenMP impact the research questions above?
- **RQ5**: How can the findings from the points above be combined into novel directives targeting energy efficiency?

1.2 Limitations

Although OpenMP supports Fortran, we will focus on C and C++ implementations only. Besides being a framework for parallelism for multi-core computers, OpenMP also has support for offloading work to other devices such as GPUs [22]. There has also been some work on offloading work to FPGAs (Field-Programmable Gate Arrays) [11]. These are, however, out of scope of this project, which only considers shared-memory computers with homogeneous architectures.
When performing analysis on source code optimisations, we never go lower than assembly code, for example into compiler source code. This is mainly due to a lack of time and expertise in the area.

2
Background

This chapter covers the relevant background information for the project. First, we cover the aspects of OpenMP that are relevant to the research questions. Then, we describe the concepts of loop unrolling and tiling, how they can improve programs, and how they can be applied automatically by OpenMP. Finally we present related work concerning energy optimisation of OpenMP programs.

2.1 OpenMP

As described in the introduction, OpenMP is an API for shared-memory programming. It consists of compiler directives, runtime library functions and environment variables that simplifies parallel programming on shared-memory machines [21].

2.1.1 Runtime constructs

When writing parallel programs with OpenMP there are three available directives that directly describes how the parallel parts of the program should behave, those being: *parallel for*, *tasks* and *sections*. They are used to describe parallel behaviour for slightly different scenarios. Sections are, however, in our experience, rarely used in practise, so we do not consider it in our analysis.
The *parallel for* directive is used to parallelise *for* loops, and is the most used and straight forward of the three directives. The loop iterations are split up into chunks, distributed over the assigned number of threads, and then run in parallel. It is possible to change the size of these chunks and how they are distributed with input parameters passed to the directive. There exist four scheduling options that decide the distribution: *static*, *dynamic*, *guided*, and *auto*. The *static* option distributes the chunks evenly over the threads at compile time, *dynamic* instead dynamically distributes chunks at run time as threads complete their previous work, *guided* works in a similar way as *dynamic* but utilizing a dynamic chunk size, and lastly, *auto* gives full control to the compiler to decide how to handle the scheduling (e.g. the GCC compiler just defaults to the *static* setting when *auto* is selected [14]) [21]. An example of a parallelized loop using the directive can be seen in listing 2.1.
The *task* directive is used to defines a scope of code that can be run concurrently by any thread. An example where this directive would be useful is in while loops, where the number of iterations is unknown and a *task* can instead be created for each iteration. Another common example is recursive functions, where each recursive call can be its own *task*. These two scenarios can be seen in listings 2.2 and 2.3

respectively.

Listing 2.1: Parallelized for loop using the *parallel for* directive

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
{
    foo(i); // Some work
}
```

Listing 2.2: Parallelized while loop using the *task* directive.

```
#pragma omp parallel
#pragma omp single
while(...)
{
#pragma omp task
    foo(i); // Some work
    i++;
}
```

Listing 2.3: Recursive function using the *task* directive

```
int main() {
  ...
  #pragma omp parallel
  #pragma omp single
  fib(N); // Executed in a parallel region
  ...
}

int fib(int n)
{
  int i, j;
  if (n<2) {
    return n;
  }
  else {
    #pragma omp task shared(i)
    i=fib(n-1);
    #pragma omp task shared(j)
    j=fib(n-2);
    #pragma omp taskwait
    return i+j;
  }
}
```

2.1.2 Waiting policy

The waiting policy, specified by the *OMP_WAIT_POLICY* [1] environment variable, provides a hint about the desired behaviour of waiting threads to the runtime system. It can be set to *active* or *passive*. With the *active* setting, threads should be active and consume processor cycles (such as spinning) while *passive* threads should avoid consuming cycles by, for example, yielding the processor core. If the value is not set, an implementation of OpenMP may have a third alternative as default behaviour, which is often a mix of the two.

2.2 Code transformations

Code transformations are methods of changing or rearranging source code, often with the aim of reducing different kinds of performance overheads while being semantically equivalent. This section covers the two approaches considered in this work: loop tiling and loop unrolling.

2.2.1 Loop Unrolling

Loop unrolling is a method that coalesces the logic of several loop iterations into one iteration with the goal of reducing overhead related to loop control logic such as counter updates, termination conditions and branching [12].
An example of a partially unrolled loop can be seen in listing 2.5, the unmodified version of the loop can be seen in listing 2.4. Here, the functionality of five iterations have been grouped together, effectively reducing the amount of control operations needed by the loop by 80%.

Listing 2.4: Unmodified loop.

```
int i;
int n[N];
for(i = 0; i < N; i++)
  n[i] = 10 * i;
```

Listing 2.5: Partially unrolled loop

```
int i;
int n[N];
for (i=0;i<N;i+=5) {
  n[i+0] = 10 * (i+0)
  n[i+1] = 10 * (i+1)
  n[i+2] = 10 * (i+2)
  n[i+3] = 10 * (i+3)
  n[i+4] = 10 * (i+4)
}
```

Note, however, that if the unrolling factor does not evenly divide the total number of iterations, this will lead to more iterations being performed than in the unmodified code, leading to incorrect behaviour. Additionally, when the total number of iterations is not known at compile time, assumptions about evenly divisibility can not be guaranteed. One way to solve this is seen in listing 2.6, where the first loop is unrolled and the second handles the special case.

[1] Information about the waiting policy in OpenMP can be found in https://www.openmp.org/spec-html/5.0/openmpse55.html

Listing 2.6: Partially unrolled loop

```
int n[N];
int i=0;
for (;i<N-4;i+=5) {
  n[i+0] = 10 * (i+0);
  n[i+1] = 10 * (i+1);
  n[i+2] = 10 * (i+2);
  n[i+3] = 10 * (i+3);
  n[i+4] = 10 * (i+4);
}
for(;i<N;i++) {
  n[i] = 10 * i;
}
```

One should be careful when applying loop unrolling, since blindly unrolling loops does not always lead to better performance. A problem with excessive unrolling is that the L1 instruction cache misses can go up since the number of unique instruction has increased [12], overall degrading performance due to cache eviction. Unrolling also has the undesirable effect of making the program less readable and larger in size [2].

Directive-generated unrolling was introduced in OpenMP 5.1 [21]. Listing 2.7 shows how a loop can be unrolled with similar results to listing 2.6.

Listing 2.7: Unrolled loop by OpenMP

```
int i;
int n[N];
#pragma omp unroll partial(5)
for(i = 0; i < N; i++) {
  n[i] = 10 * i
}
```

2.2.2 Loop Tiling

Loop tiling, also known as *loop blocking*, is a loop transformation that targets nested loops with the aim of improving data locality on data accesses. It does this by splitting each loop affected into two separate loops, the outer loop defines the size of blocks of data, while the inner iterates over said block. The reason for this loop rearrangement is to maximise the reuse of data before it becomes evicted from the cache, thus improving performance.

An example of loop tiling applied to matrix multiplication, where each matrix has been split into tiles with height and width of four cells, can be seen in figure 2.1. Here the result for all the cells in the tile for matrix C is calculated to completion before moving on to another tile.

[2]Figure taken from [16], with permission.

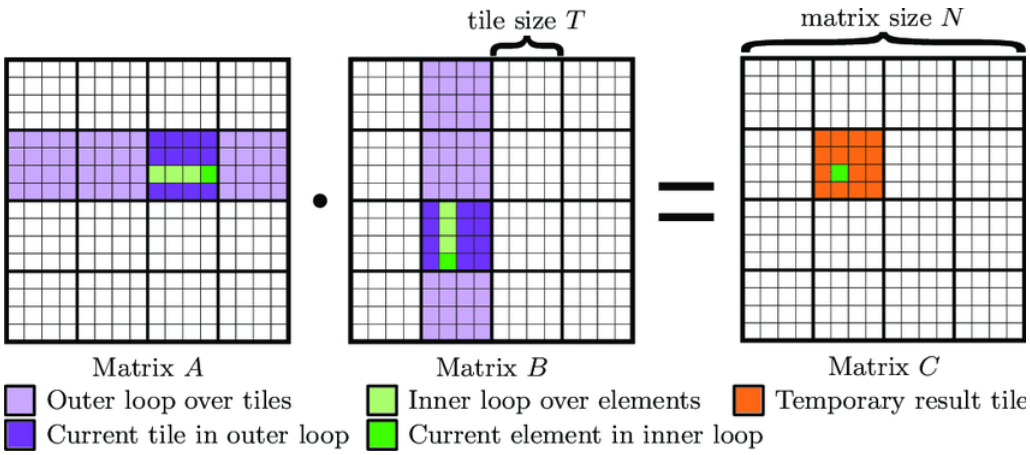


Figure 2.1: Loop tiling applied to matrix multiplication.[2]

Listing 2.8: Basic square matrix multiplication

```
// C = A * B
for (int row = 0; row < N; row++) {
  for (int col = 0; col < N; col++) {
    for (int k = 0; k < N; k++) {
      C[row*N + col] += A[row*N + k] * B[k*N + col];
    }
  }
}
```

Listing 2.9: Basic square matrix multiplication with loop tiling

```
// C = A * B
for (int r0 = 0; r0 < N; r0 += tile_size) {
  for (int c0 = 0; c0 < N; c0 += tile_size) {
    for (int k0 = 0; k0 < N; k0 += tile_size) {
      for (int r1 = r0; r1 < r0 + tile_size; r1++) {
        for (int c1 = c0; c1 < c0 + tile_size; c1++) {
          for (int k1 = k0; k1 < tile_size; k1++) {
            C[r1*N + c1] += A[r1*N + k1] * B[k1*N + c1];
          }
        }
      }
    }
  }
}
```

Listing 2.10: Basic square matrix multiplication with loop tiling with bound checks

```
// C = A * B
for (int r0 = 0; r0 < N; r0 += tile_size) {
  int rmax = r0 + tile_size > N ? N : r0 + tile_size;
  for (int c0 = 0; c0 < N; c0 += tile_size) {
    int cmax = c0 + tile_size > N ? N : c0 + tile_size;
    for (int k0 = 0; k0 < N; k0 += tile_size) {
      int kmax = k0 + tile_size > N ? N : k0 + tile_size;
      for (int r1 = r0; r1 < rmax; r1++) {
        for (int c1 = c0; c1 < cmax; c1++) {
          for (int k1 = k0; k1 < kmax; k1++) {
            C[r1*N + c1] += A[r1*N + k1] * B[k1*N + c1];
          }
        }
      }
    }
  }
}
```

The tile size needs to be chosen in such a way that all the data needed for one tile iteration can fit into the memory hierarchy that is optimised for. If the tile size chosen is too small, then the overhead from the additional loops can instead start to dominate the execution time, undoing any performance gain. Therefore, it is of utmost importance to find an appropriate value of the tile size to balance these two factors.

An example of a loop tiled version of matrix multiplication can be seen in listing 2.9, with the unmodified version in listing 2.8. However, similar to unrolling, this assumes that the number of iterations are evenly divided by the tile size. The code in listing 2.10 fixes this.

OpenMP introduced compiler-generated loop tiling in OpenMP 5.1 [21]. This makes it possible to perform loop tiling using the *tile* directive. Listing 2.11 shows how matrix multiplication can be loop tiled using OpenMP with a similar result to listing 2.10.

Listing 2.11: Basic square matrix multiplication with OpenMP loop tiling

```
// C = A * B
#pragma omp tile (tile_size, tile_size, tile_size)
for (int row = 0; row < N; row++) {
  for (int col = 0; col < N; col++) {
    for (int k = 0; k < N; k++) {
      C[row*N + col] += A[row*N + k] * B[k*N + col];
    }
  }
}
```

2.3 Related work

We begin with an example of loop unrolling being used to increase energy efficiency. In [20], the authors evaluate loop unrolling on an energy-efficient implementation of Strassen's algorithm. Strassen's algorithm is an algorithm for fast matrix multiplication, significantly faster than naive matrix multiplication for large matrices. Their method yields a performance gain of 98 % and a reduction in energy consumption of 95 %, which is mainly due to better utilisation of vector instructions.

To effectively experiment with energy consumption, we need a way to measure it. Goel and McKee [7] present a method for measuring and modelling dynamic and static power in cores and uncores. Uncore is Intel's term for components close but not part of the CPU such as last-level cache, memory controller and interconnects. They find that while more cores generally yield higher performance, the optimal number of running threads may be lower due to stalling in serial sections of the program.

Two of our research questions, those regarding tasking versus parallel loops and the waiting policy, consider the runtime resources of OpenMP. In these contexts, the number of threads with which to execute parallel regions, often referred to as DCT (Dynamic Concurrency Throttling), is a major concern. In [15], Lorenzon et al. perform DCT on OpenMP programs targeting performance, energy or energy-delay-product (EDP). The tool, called Aurora, performs a hill-climbing approach to selecting the optimal number of threads for a parallel OpenMP loop optimising performance, energy or EDP. It is made fully transparent to both programmer and end user by extending the *libgomp* library (The OpenMP implementation for the GCC compiler), which is dynamically linked at runtime. The work improves EDP by 98 % compared to the baseline (which uses as many threads as machine cores), 86 % compared to using the OMP_DYNAMIC (which selects number of threads based on previous machine workload during the last 15 minutes) and 91 % compared to feedback-driven threading ([26]). However, the work targets only parallel loops (eg. *#pragma omp parallel for [...]*) and not sections or tasks. Energy minimisation is also beneficial due to mitigating hardware ageing, as mentioned in the introduction. In [17], the authors perform DCT specifically for this purpose. Their work is very closely related to the other DCT work mentioned above.

In [9], the authors introduce a novel auto-tuning framework. The framework tunes parameters such as unrolling factors, tile sizes and loop ordering to optimise a combination of performance, energy and power. It also selects the optimal number of threads for code regions. The framework is based on the *Insieme Compiler and Runtime* infrastructure [8], which provides source-to-source transformations for parallel C, C++, OpenMP, MPI and OpenCL programs. Internally, Inseime transforms the input source code into an internal representation called INSPIRE on which the transformations and optimisations occur [27]. As a final step the internal representation is translated back into the input language. Evaluations show a 70 % performance improvement over solutions tuned for a specific number of threads [9].

Related to this work is OpenMPE, which is an extension of OpenMP for application-level optimisation of energy, power and performance [1]. Most importantly, OpenMPE introduces an *objective* clause, which specifies how a region of code should

be optimised in terms of performance, energy and power. As a simple example, *#pragma omp ... objective(E)* will optimise a code region with the only objective of minimising energy consumption. A more advanced example is *#pragma omp ... objective(0.4*E+0.6*T : P<200)* meaning that the optimisation is weighted 40 % for energy, 60 % for performance and the power must not exceed 200 W. The authors have also implemented a version of OpenMPE. In the first step of the compilation process the source code is transformed into INSPIRE representation, on which the objective directives are applied. The INSPIRE code is then translated back into C, and then normally compiled using GCC. Evaluation of OpenMPE shows energy savings of 15 % across 9 use cases.

3
Benchmark programs

This chapter covers the benchmark programs used in the project. These programs fall into one of three categories. The first is *microbenchmarks*, which are very small programs created to test very specific feature of the language, such as the overhead of starting a parallel region. These are written by us and allows well-controlled experimentation, but make no claim to represent a realistic program.
The second category is *own benchmarks*, which consists of two larger programs: matrix multiplication and a 2-dimensional stencil. These provide a slightly more realistic workload. They allow experimentation with different optimisations and algorithmic changes, providing stronger grounds for conclusions than the microbenchmarks. They should however be supplemented with standardised benchmark suites in order to be compared with other work.
The final category is the *benchmark suites*: well-known benchmarks what are commonly used in papers and therefore much easier for other researchers to compare against, with no potential source of error from biased or poorly written own benchmarks. Unfortunately, these often have the drawback of being big and complex, making it difficult to determine the impact of aspects such as loop tiling, for example.

3.1 Microbenchmarks

This section covers the three microbenchmarks introduced for this project. As mentioned above, these are very small kernels meant for experimenting with very specific program aspects. First is the *parallel constructs microbenchmark*, the purpose of which is to experiment with the different ways of introducing parallelism to a program. It is especially important for the second research question, which considers tasking versus parallel loops. The second benchmark is the so-called *inactivity microbenchmark*, which is meant for evaluation of the waiting policy. The third and final program is the *unrolling microbenchmark*, which as the name implies evaluates loop unrolling.

3.1.1 Parallel constructs microbenchmark

The purpose of this microbenchmark is to measure the overhead of the different parallel constructs, as well as the overhead of creating and managing parallel regions. The program creates a number of tasks to complete in parallel. The main kernel, in its most basic form, can be seen in listing 3.1. The tasks themselves simply stall a

random amount of time. Stalling in this case refers to running a busy-waiting loop for the required amount of time in contrast to sleeping, which yields the running thread. The stalling is implemented using the *gettimeofday* function and can be seen in listing 3.2.

Listing 3.1: Basic sleeping microbenchmark

```
for (int i = 0; i < iterations; i++) {
    // New parallel region for every iteration
    #pragma omp parallel
    {
        #pragma omp [...]
        for (int t = 0; t < num_tasks; t++) {
            perform_task(taskarray[t]);
        }
    }
}
```

Listing 3.2: Implementation of busy-waiting for a number of microseconds

```
void stall_us(double us) {
    struct timeval t0;
    gettimeofday(&t0, 0);
    struct timeval t1;
    double elapsed;
    do {
        gettimeofday(&t1, 0);
        elapsed=((t1.tv_sec-t0.tv_sec)*
            1000000+t1.tv_usec-t0.tv_usec);
    } while (elapsed < us);
}
```

The inner loop can be made parallel using one of many OpenMP directives. One is using a *for* loop, which in this case would divide the loop iterations among the threads. Another is using a *parallel for* loop, which in this case would create a nested parallel region. A third is the *taskloop* directive. Example of a taskloop can be seen in listing 3.3.

We also try a manual variant of the taskloop construct where a single or multiple thread create explicit tasks for each iteration as seen in listings 3.4 and 3.5.

Listing 3.3: Task generation with taskloop directive

```
#pragma omp single
#pragma omp taskloop
for (int t = 0; t < num_tasks; t++) {
    perform_task(waittimes[t]);
}
```

Listing 3.4: Single-threaded task generation

```
#pragma omp single
for (int t = 0; t < num_tasks; t++) {
    #pragma omp task
    perform_task(waittimes[t]);
}
```

Listing 3.5: Multi-threaded task generation

```
#pragma omp for
for (int t = 0; t < num_tasks; t++) {
    #pragma omp task
    perform_task(waittimes[t]);
}
```

The iterations parameter defines how many times to perform all tasks, and a new parallel region is created for each iteration. The granularity of each task is how many microseconds to stall. This is uniformly randomised between 0 and a maximum task size parameter. The final parameter is the number of tasks. Since the task granularity is uniformly distributed between 0 and max_task_size, the expected execution time is

$$T_{exec} = \frac{iterations \cdot num\_tasks \cdot max\_task\_size}{2 \cdot num\_threads}$$

without considering overheads. This microbenchmark has the characteristics of an embarrassingly parallel workload, since no kind of dependencies between the tasks exists.

3.1.2 Inactivity microbenchmark

The purpose of this microbenchmark is to evaluate the waiting policies of inactive threads. The kernel consists of a team of threads where only one thread performs some dummy work while the other threads wait. This is repeated for a number of iterations to obtain a reasonable sample size. The dummy work is the same as in the sleeping microbenchmark, meaning a busy-wait loop that stalls for the requested number of microseconds.

Listing 3.6: Kernel of inactivity microbenchmark

```
#pragma omp parallel
{
  for (int i = 0; i < ITERATIONS; i++) {
    #pragma omp master
    stall_us(WAITTIME_US);
    #pragma omp barrier
  }
}
```

This can be executed with different values of the waiting time. Intuitively, you would expect active scheduling to have a lower energy consumption for small task sizes due to the overheads of passive scheduling increasing execution time, while passive scheduling is better for large task sizes due to the power savings of not having threads spin.

3.1.3 Unrolling microbenchmark

The purpose of this microbenchmark is to test the effects unrolling has on some simple for loops with various performance characteristics. For each scenario three different implementations will be tested, these are 1) no explicit unrolling, 2) OpenMP unrolling and 3) manual unrolling.
Four programs have been created for this microbenchmark, these are:

- `SIMPLE_MEM`, which loops over an array and initialises it with values (listing 3.7).
- `SIMPLE_COMP`, which calculates the sum for some simple calculations (listing 3.8).
- `SIMPLE_COMP_DEPEND`, similar to the previous program but there are strict dependencies between each iteration (listing 3.9).
- `COMPLEX_NESTED`, which contains an additional nested loop that does some calculation for each element in an array (listing 3.10). Only the outermost loop is unrolled for this program.

Listing 3.7: The `SIMPLE_MEM` program

```
for(int i=0; i<len; i++)
  A[i] = i;
```

Listing 3.8: The `SIMPLE_COMP` program

```
float sum = 0;
for(int i=0; i<len; i++)
  sum += 1/(float)i;
```

Listing 3.9: The `SIMPLE_COMP_DEPEND` program

```
int sum = 0;
for(int i=0; i<len; i++)
  sum = (sum + i)%100;
```

Listing 3.10: The `COMPLEX_NESTED` program

```
for(int i=0; i<len; i++){
  A[i] = 0;
  for(int j=0; j<len; j++)
    A[i] += i*j;
}
```

The different unrolling implementation for each program have been created in the fashion described in section 2.2.1. The only exception is the manual unrolling implementations for `SIMPLE_COMP` and `COMPLEX_NESTED`, where some additional manual optimisations have been applied. The optimised versions can be seen in listings 3.11 and 3.12 respectively.
For `SIMPLE_COMP`, several different sum variables are used for the different offsets that are then added together after the loop is completed. This because it removes the implicit dependencies between the calculations, and can potentially improve performance.

For `COMPLEX_NESTED`, the inner nested loops are combined into a single nested loop (often called *loop jamming*) when the outer loop is unrolled.

Listing 3.11: The `SIMPLE_COMP` program, manually unrolled by a factor of two, with additional optimisations.

```
float sum0, sum1;
for(int i=0; i<len; i+=2)
{
  sum0 += 1/(float)(i+0);
  sum1 += 1/(float)(i+1);
}
float sum = sum0+sum1;
```

Listing 3.12: The `COMPLEX_NESTED` program, manually unrolled by a factor of two, with additional optimisations.

```
for(int i=0; i<len; i+=2){
  A[(i+0)] = 0;
  A[(i+1)] = 0;
  for(int j=0; j<len; j++)
    {
    A[(i+0)] += (i+0)*j;
    A[(i+1)] += (i+1)*j;
    }
}
```

3.2 Own benchmarks

This section covers our own benchmarks, which are more realistic than our microbenchmarks. The ones we have made are particularly useful for experimenting with unrolling and tiling.

3.2.1 Matrix multiplication

As seen in listing 2.8, basic matrix multiplication can be performed by iterating over the rows and columns of the target matrix and then for each element calculate the dot product of the row of the left matrix and column of the right matrix.
However, it is also possible to change the order of the loops and achieve a significant speedup due to locality, as seen in listing 3.13.

Listing 3.13: Optimised code for matrix multiplication where the inner loops are switched.

```
for (int row = 0; row < N; row++) {
  for (int k = 0; k < N; k++) {
    for (int col = 0; col < N; col++) {
      C[row*N + col] += A[row*N + k] * B[k*N + col];
    }
  }
}
```

We refer to the first as *naive* matrix multiplication and the second as *reordered* matrix multiplication. Due to its better memory access pattern we expect a smaller improvement from using loop tiling.
Both loop tiling and loop unrolling can be applied to matrix multiplication due to having nested loops. Tiling is applied using OpenMP directives as seen in list-

ing 2.11, but also manually as seen in listing 2.9. Example code for loop unrolling using OpenMP can be seen in listing 3.14 and the corresponding manual unrolling code can be seen in listing 3.15. Only the innermost loop is unrolled in both cases.

Listing 3.14: Loop unrolled matrix multiplication using OpenMP.

```
for (int row = 0; row < N; row++) {
  for (int col = 0; col < N; col++) {
    #pragma omp unroll partial(UNROLL_FACTOR)
    for (int k = 0; k < N; k++) {
      C[row*N + col] += A[row*N + k] * B[k*N + col];
    }
  }
}
```

Listing 3.15: Loop unrolled matrix multiplication of using manual unrolling of factor 4.

```
for (int r = 0; r < N; r++) {
  for (int c = 0; c < N; c++) {
    int k = 0;
    for (; k < N - 3; k += 4) {
      C[r*N + c] += A[r*N + k] * B[k*N + c];
      C[r*N + c] += A[r*N + (k+1)] * B[(k+1)*N + c];
      C[r*N + c] += A[r*N + (k+2)] * B[(k+2)*N + c];
      C[r*N + c] += A[r*N + (k+3)] * B[(k+3)*N + c];
    }
    for (; k < N; k++) {
      C[r*N + c] += A[r*N + k] * B[k*N + c];
    }
  }
}
```

3.2.2 2D stencil

Our own benchmark *2D stencil* is a simple program that takes a matrix as an argument and, for each cell, computes the average of all its surrounding neighbours. For simplicity, the program skips computing the average of outermost cells since they would be a special case otherwise. Listing 3.16 shows an outline of the basic code. The purpose of introducing this benchmark is to have a program less computationally and memory intensive than matrix multiplication, while still being a relatively realistic pattern. More importantly, tiling can be applied very easily.

Listing 3.16: Basic code outline for the 2D stencil program. Note that no calculation is performed for the edges.

```
for (int i = 1; i < N - 1; i++)
  for (int j = 1; j < N - 1; j++)
      matrix[i*N + j] = (
          matrix[(i-1)*N+(j-1)] +
          matrix[(i-1)*N + j] +
          matrix[(i-1)*N + (j+1)] +
          matrix[(i)*N + (j-1)] +
          matrix[(i)*N + j] +
          matrix[(i)*N + (j+1)] +
          matrix[(i+1)*N + (j-1)] +
          matrix[(i+1)*N + j] +
          matrix[(i+1)*N + (j+1)]
      ) / 9.0;
```

It should also be mentioned that there are possible race conditions in this program. As one thread operates on row i, another can operate on row $i+1$ leading to race conditions as the second thread will read from row i which the first thread writes to. This is, however, not an issue we will consider.

3.3 Common benchmark suites

This section covers the used benchmark suites: BOTS, NPB, and PARSEC. All of these are used for the analysis of the waiting policy by executing their respective programs with the different waiting policy options, compilers and numbers of threads. They are also used for a more realistic analysis of the code transformations, where we profile each program and try to apply loop unrolling and tiling in a more realistic setting.

BOTS (Barcelona OpenMP Task Suite) is a benchmark suite targeting exploitation of the irregular parallelism offered by OpenMP tasking[1]. From BOTS we use all programs: *alignment*, *fft*, *floorplan*, *nqueens*, *sparselu*, *strassen* and *uts*. Two of these, *alignment* and *sparselu*, have the option to create tasks in either a single-threaded or multi-threaded manner, similar to how the parallelism constructs microbenchmark does as seen in listings 3.4 and 3.5. We therefore use these as part of our analysis of parallelism constructs.

The NAS Parallel Benchmark Suite (NPB) was developed by NASA in 1991 to evaluate the performance of parallel supercomputers. The programs are derived from computational fluid dynamics applications [19]. Most of these programs are, however, implemented in Fortran, which is not in the scope of this thesis. Therefore, we instead use an unofficial implementation ported to C [2]. From NPB we use the following programs: *BT*, *CG*, *EP*, *FT*, *IS*, *LU* and *MG*. There is also the SP program,

[1]The BOTS benchmarks can be found in https://github.com/bsc-pm/bots

[2]NAS Parallel Benchmarks ported to C can be found in https://github.com/benchmark-subsetting/NPB3.0-omp-C

but we decided to not use it due to issues with compilation and execution. The problem sizes with which to run NPB are known as classes [3]. The smallest problem size (S) is very small, making parallel computations not worth the overhead. Instead, we use the larger A class.
The final benchmark suite we used was PARSEC, (Princeton Application Repository for Shared-Memory Computers). PARSEC is a benchmark suite designed to test multiprocessors for parallel workloads ranging from many different domains such as computer vision and financial analysis [4]. The programs each have several implementations utilising different parallelism models, such as pthreads and OpenMP. But since our main focus was OpenMP, instead an extension of the benchmark called PARSECSs [4], which includes two OpenMP versions, a *parallel for* and *tasking* implementation, was used. We use these versions in our analysis of the OpenMP parallelism constructs.
Not all of the original 12 PARSEC benchmarks was ported to use both the OpenMP implementations, and some of the benchmarks we did not get to work or had other technical issues with. In the end the two following benchmarks was used for the final evaluation: *Blackscholes* and *Fluidanimate*. PARSEC comes with many different input sizes with execution times ranging from almost instantaneous to several minutes. In our analysis we used the largest input set, called *Native*.

[3]NPB classes are described in https://www.nas.nasa.gov/software/npb_problem_sizes.html
[4]The Git repository for PARSECSs can be found here: https://pm.bsc.es/gitlab/benchmarks/parsec-ompss

4
Experimental Methodology

This chapter describes details about the experimental setup and procedure of the project, especially how measurement of energy consumption is performed. It also includes an overview of the overall workflow used, as well as potential sources of error.

4.1 Parameter search space

We systematically evaluate loop unrolling, loop tiling, tasking and parallel for loops with different compiler optimisations, number of threads and waiting policies.
Loop unrolling and tiling are evaluated both by applying them to our own benchmarks and the common benchmark suites BOTS, NAS and PARSEC. In the benchmark suites, we analyse the program and attempt to apply unrolling and tiling where it is possible and evaluate the results. These programs are heavily optimised and often have manually unrolled and tiled loops. These optimisations can be removed and replaced with both unrolled and tiled loops using OpenMP and with the optimisation removed. We expect however that the original version will be better since it has optimisations that OpenMP or the compiler might not be able to replicate.
Tasking and *parallel for* are two distinct ways of achieving parallelism with OpenMP. However, in many cases, it is possible to solve a problem using either one of them. Therefor we will run some of the applications using two different implementations, one implemented with tasks and the other with parallel for, to compare if there is an advantage of using one over the other.
Waiting policies is evaluated mainly by executing the common benchmark suites and our microbenchmarks with the different settings.

4.2 Hardware and software details

Experiments were performed on nodes on the Tetralith HPC cluster. These nodes have 16-core dual socket Intel Xeon Gold 6130 processors (32 cores in total), 96 GiB RAM and run Linux version 3.10.0. Allocated nodes can have different disk sizes, but we limit ourselves to only the smallest disk size of 240 GB (SSD). The Tetralith nodes have support to model energy consumption using RAPL, described further in section 4.3.
Three compilers were used to compile C code with OpenMP: the Intel compiler *ICC* version 18.0.1, which was available on Tetralith, the GNU C compiler *GCC* version 10.1.0, compiled from source, and the LLVM C compiler *Clang* version 13.0.0, also

compiled from source. Clang 13.0.0 is the only compiler of the three which has support for the newly introduced tiling and unrolling OpenMP directives.

4.3 Energy consumption measurements

Many modern processors provide means to obtain the power consumption of the processor through model-based power estimations [7]. In Intel processors, since the Sandy Bridge architecture, this is done using the tool Running Average Power Limit (RAPL), which provides data on the accumulated energy consumption. The tool has an power estimation accuracy of 1 %, with a standard deviation of 1.1 % [6]. Through RAPL it is also possible to constrain the maximum power consumption allowed by various subsystems, but it was decided that exploring this feature of RAPL was out of scope for this paper and will not be used in experimentation.

On Tetralith, RAPL reports energy consumption as a running total, which resets after reaching a threshold. To measure the energy consumption between two points in time is therefore done by reading this value at both points in time, and then subtracting the initial energy value from the energy value at the second point in time. It could happen that the counter reaches the threshold during this time and resets, so this overflow must also be taken into account. The energy counters are stored as files, which means that accessing them is done by reading them as any other file. The energy consumption of each 16-core socket and their main memory is also reported separately, but we do not consider this in our analysis and simply add these together.

4.4 Evaluation workflow

In order to integrate energy measurements into the benchmarks as seamlessly as possible, we implement all of the energy reading logic in a separate C file. This allows us to measure the energy consumption of only parts of a program, instead of measuring the energy for all of it. Listing 4.1 shows how this can be done with the example of matrix multiplication. The *poll_before* function creates a data structure with the readings before the program kernel. *poll_after* reads the energy counters again, compensates for potential overflow, and outputs the energy consumption to a JSON file.

Performing no readings during the execution has the benefit of not interrupting the program during execution to read the counter values. It has however a major disadvantage in that the peak power cannot be measured. Average power can however be calculated easily as the energy consumption divided by the execution time. **All reported power values in this report consider average power**.

Listing 4.1: Methodology of energy measurements with a matrix multiplication example

```
int main() {
    matrix_t* A, B, C;
    rand_matrices(A, B);
    measurements_t* m = poll_before();
    compute_matmul(C, A, B);
    poll_after(m, "measurements_outputs.json");
    free_matrices(A, B, C);
}
```

Thus far, the approach does not take idle power into account, i.e., the power consumed while the machine is inactive. This can give misleading results, especially when the program only uses a small fraction of the machine such as only running with one thread. This is assumed to be fairly constant for a given system, and can therefore be measured once and the compensated energy given by

$$E_{compensated} = E_{total} - P_{static} \cdot T_{exec}$$

where E_{total} is the total energy consumed, P_{static} is the static energy consumption of the system, and T_{exec} is the execution time of the program kernel.
However, static power can differ between nodes on the data center. Therefore, it is measured before running a set of benchmarks on the allocated node, simply by sleeping for a number of seconds and measuring the used consumed energy.
We use Python scripts to compile and execute a range of configurations, saving the outputs to a results file *results.json*. We then use another set of scripts to read these results and plot figures. This process was developed so large batch jobs could be executed with no further input from the user, which let us easily and methodically test large amounts of benchmarks and configurations. The whole process of our workflow when testing the benchmarks can be seen in figure 4.1.
When compiling code, optimisation level *O3* is always used unless specified otherwise. Furthermore, for every configuration a program is executed between 3 and 10 times, and the measurements (execution time, energy, and power) are reported as the geometric mean of these executions.

4.5 Potential sources of error

On Tetralith nodes, processor frequency is controlled by system administrators and is not possible to adjust by us. The nodes run the intel_pstate driver on the scaling governor *powersave*, which selects p states based on current CPU utilisation [1]. While it perhaps would be better to run with the highest performance all the time, this is a more realistic setting.
C states are also controlled by system administrators and are configured to aggressively limit the power consumption. In [23], it is found that enabled C-states highly

[1]Information about intel_pstate can be found in
https://www.kernel.org/doc/html/v5.12/admin-guide/pm/intel_pstate.html

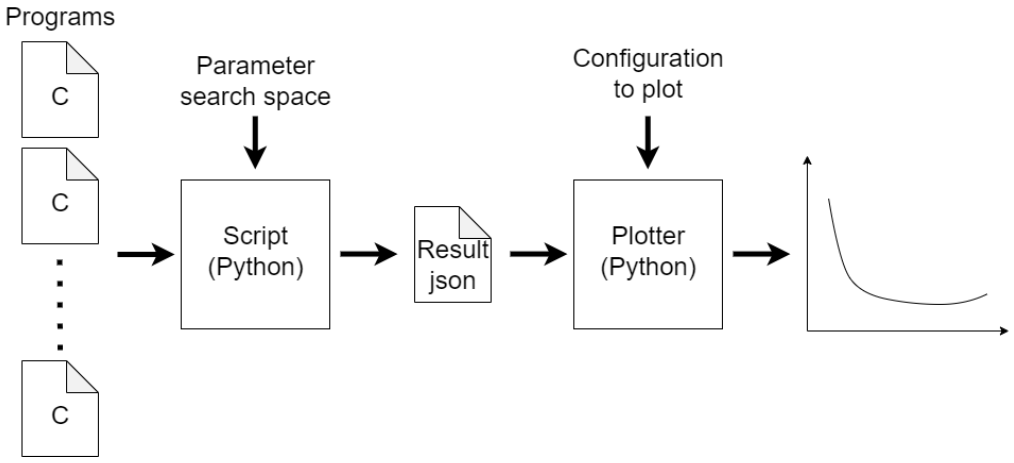


Figure 4.1: The workflow we developed and used for running benchmarks and plotting their result. A python script is used that takes some benchmarks written in C and also a set of runtime parameters, such as waiting policy and number of threads. The script then runs these benchmarks with all parameters combinations and saves the results to a json file. Another python script can then read this file and make some plot after a specified configuration.

impacts variations in energy of benchmarks, so it would have been better to disable them. Again, however, enabled C-states provides more realistic data since most data centers will have them enabled.

5
Results

This chapter presents the results of the experiments that we ran for the different benchmarks. First, we cover our own matrix multiplication and 2D stencil programs in sections 5.1 and 5.2. Next, we present the microbenchmark results in sections 5.3, 5.4 and 5.5. After that, we cover the common benchmark suites in sections 5.6, 5.7, and 5.8. Finally, we conclude the chapter with a compilation of the most interesting data in section 5.9 and answer some of the research questions based on that.
In general, we present metrics (energy, execution time and power) either in absolute numbers or relative to some baseline. Use of relative numbers is useful when the differences between configurations is very small, which is often the case for example in unrolling.
When plotting data, we often show only the energy consumption and not execution time. This is because the two quantities are almost always proportional to one another, which makes showing both redundant. As this project focuses on energy consumption, we choose to focus on that. Execution time and power are however often reported in summarising tables along with energy.

5.1 Matrix multiplication

This section evaluates loop transformations on matrix multiplication. The input matrices are 1024x1024 single-precision floating point numbers. This size is chosen because the execution time is about a second, which is short enough to run many configurations in a short time but long enough for the overheads of parallelism management to be minor. All experiments use the *active* waiting policy and the *static* loop scheduling.

5.1.1 Loop tiling

As described in section 2.2.2, loop tiling transforms for loops to improve data locality and reduce overhead. We apply three versions of tiling. The first is using the OpenMP tiling directive, as described in listing 2.11. The second is a tiling version similar to listing 2.9, where we do not assume that the number of iterations are divisible by the tile size. The final version, similar to listing 2.10, is a manually tiled version where we do make this assumption. The tested tile sizes are 1, 2, 4, 8, 16, 32 and 64. Size 8 proved to be the best, which is used in the presented results.
We first present the results for the Clang compiler only. In figure 5.1 we show the

energy consumption of the tiled versions relative to the baseline program when using the naive matrix multiplication algorithm. We see a clear reduction in energy when applying loop tiling, which seems fairly consistent for most numbers of threads but is especially effective for very low numbers of threads (1 or 2). We also see that both manual versions outperform the OpenMP version, but the second version much more so.

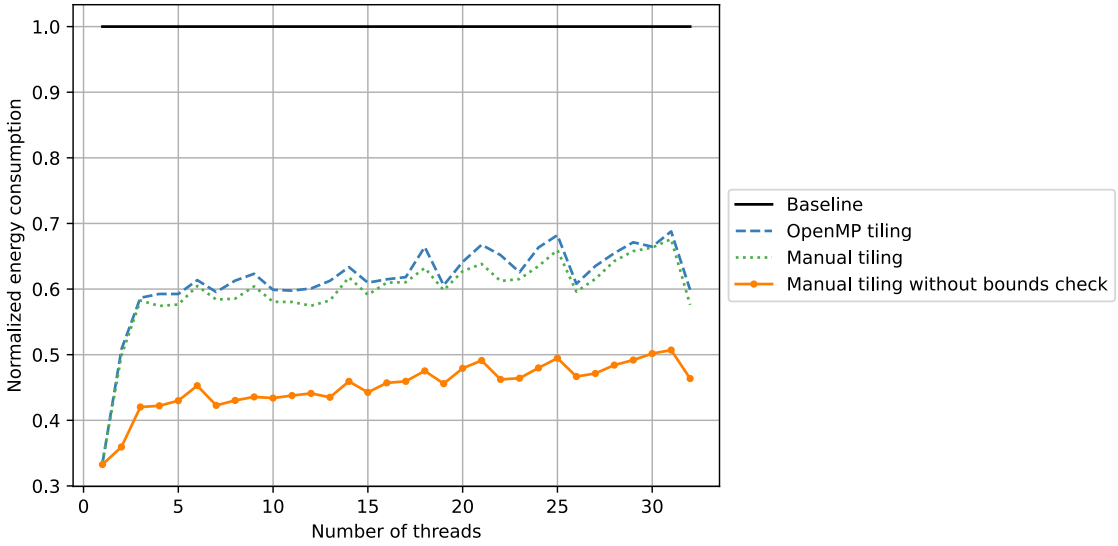


Figure 5.1: Relative energy consumption for the naive matrix multiplication with tiling.

Moving on to the reordered algorithm described in section 3.2.1, we see the energy reduction results in figure 5.2. While we do not expect to see any major speedup from tiling with the reordered algorithm, we see a major performance downgrade. This is probably not due to the tiling itself, but rather that the tiling prevents other, more significant compiler optimisations.
To investigate the cause of this performance downgrade, we use the *perf* tool to analyse L1-D cache miss rate with and without loop tiling. We use 16 threads and the *active* waiting policy. Results can be seen in table 5.1. For the naive algorithm, we see that the load misses go down significantly while the total loads increase somewhat when applying tiling, which is the expected result since tiling increases locality. With the reordered algorithm however, both the load misses and total loads are significantly higher when applying tiling. This could be due to a compiler optimisation that is applied successfully without tiling, but not with it.
Thus far we have analysed results produced by Clang only. The results for GCC are very similar to those of Clang, with tiling being highly beneficial for the naive algorithm and highly detrimental to the reordered algorithm. Plots can be found in figures A.1 and A.2 in the appendix. For ICC, however, tiling is highly detrimental in all cases except for the reordered algorithm where the loop iterations are assumed to be divisible by the tile size. In that specific case, the tiling provides a noticeable energy reduction that seems to be fairly constant across different numbers of threads. Plots can be found in figures A.3 and A.4, also in the appendix. For ICC, the naive

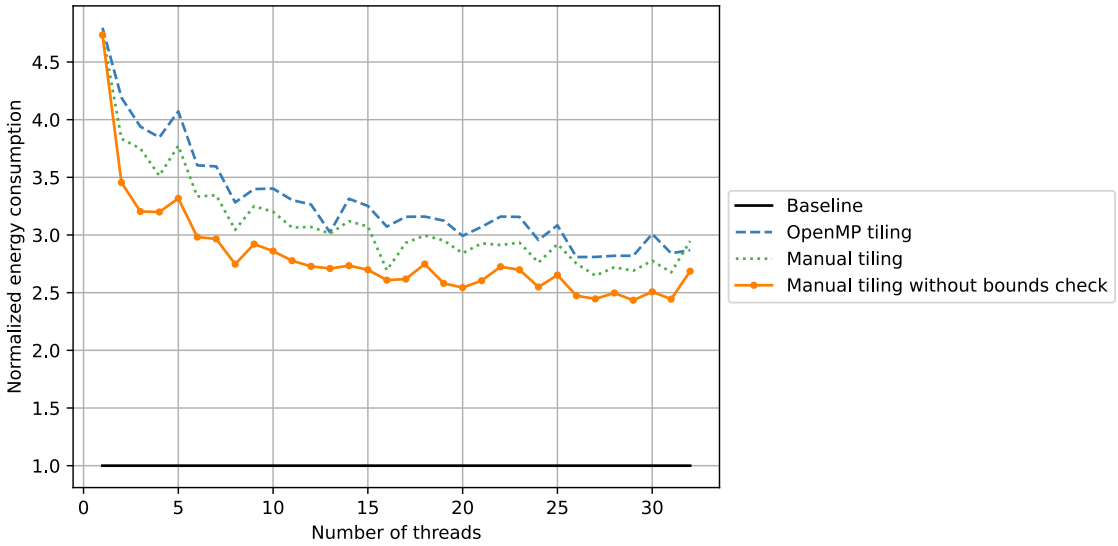


Figure 5.2: Relative energy consumption for the reordered matrix multiplication with tiling.

Table 5.1: L1-D cache behaviour for 16-threaded matrix multiplication with and without tiling.

	L1-D load misses	L1-D loads	L1-D load miss rate
Naive, baseline	1,080,290,162	2,184,350,663	49.46 %
Naive, OpenMP tiling	304,085,476	2,439,204,188	12.47 %
Naive, Manual tiling	307,412,605	2,439,204,151	12.60 %
Reordered, baseline	67,327,997	573,736,879	11.73 %
Reordered, OpenMP tiling	191,507,084	3,397,602,579	5.64 %
Reordered, Manual tiling	195,533,659	3,399,716,260	5.75 %

algorithm without tiling performs about as well as the reordered algorithms for Clang and GCC without tiling.
Table 5.2 shows the average improvements across all the number of threads for each compiler. Manual v1 refers to code such as listing 2.10, where it is not assumed that the tile size evenly divides the iterations, while manual v2 refers to code such as listing 2.10, where this is assumed.
We note that for ICC, the naive algorithm with no tiling performs about as well as the reordered algorithms for Clang and GCC without tiling. This indicates that ICC is capable of changing loop ordering and turning the naive algorithm into the reordered one by itself, which in this case is a drastic improvement. However, when we try to apply tiling, it fails to apply this optimisation and the performance deteriorates to being comparable to those of the other compilers.
To summarise, there are essentially two groups of results here: either everything gets much better with tiling, as with Clang and GCC on the naive version, or it gets much worse, as with Clang and GCC on the reordered version. ICC is the odd one out, as it gets significantly worse with all versions of tiling except the second tiling version for the reordered algorithm. While ICC seems to perform the best overall,

the single best result is actually using GCC with the baseline reordered version.

Table 5.2: Summary of the matrix multiplication tiling results across all numbers of threads. The reported execution time, power and energy are relative to the unmodified program.

Compiler	Version	Tiling	Absolute			Relative		
			T [s]	P [W]	E [J]	T	**P**	**E**
clang	Naive	Baseline	0.290	172.3	49.90	1	1	1
clang	Naive	OpenMP	0.183	166.7	30.47	0.631	0.967	**0.610**
clang	Naive	Manual v1	0.181	163.8	29.66	0.625	0.950	**0.594**
clang	Naive	Manual v2	0.138	163.2	22.47	0.475	0.947	**0.450**
gcc	Naive	Baseline	0.276	172.8	47.71	1	1	1
gcc	Naive	Manual v1	0.129	172.3	22.31	0.469	0.998	**0.468**
gcc	Naive	Manual v2	0.059	131.2	7.764	0.214	0.760	**0.163**
icc	Naive	Baseline	0.043	119.6	5.140	1	1	1
icc	Naive	Manual v1	0.109	167.6	18.30	2.541	1.401	**3.560**
icc	Naive	Manual v2	0.067	125.9	8.44	1.560	1.052	**1.642**
clang	Reordered	Baseline	0.050	121.2	6.070	1	1	1
clang	Reordered	OpenMP	0.133	148.8	19.78	2.654	1.228	**3.260**
clang	Reordered	Manual v1	0.127	147.0	18.66	2.535	1.213	**3.075**
clang	Reordered	Manual v2	0.117	144.1	16.88	2.340	1.189	**2.782**
gcc	Reordered	Baseline	0.034	126.8	4.37	1	1	1
gcc	Reordered	Manual v1	0.064	143.3	9.13	1.846	1.130	**2.086**
gcc	Reordered	Manual v2	0.100	141.5	14.12	2.891	1.116	**3.227**
icc	Reordered	Baseline	0.043	119.1	5.15	1	1	1
icc	Reordered	Manual v1	0.081	142.7	11.56	1.871	1.199	**2.243**
icc	Reordered	Manual v2	0.041	108.2	4.44	0.949	0.909	**0.862**

5.1.2 Loop unrolling

Partial loop unrolling is applied to the innermost loop in the matrix multiplication kernel. Three versions of unrolling are used: with OpenMP, manual assuming the loop iterations are divisible by the unroll factor, and manual where this is not assumed. The tested unroll factors are 1, 2, 4, 8, 16, 32 and 64, where 32 turns out to be the best and is used for the following demonstrated results.
Once again we begin with the results for Clang only. In figure 5.3 we see the relative energy consumption for unrolling the naive matrix multiplication algorithm. We see that all versions of unrolling seem to get better results when more threads are used, while being worse than the baseline when running with a single thread. At a glance, it looks like neither unrolling method is definitely better than the others. We note that the differences between the versions are very small, only a few percent.
Figure 5.4 shows the same results using the reordered algorithm. Now, all versions of unrolling are strictly worse than not applying unrolling. We note that the increase in energy consumption is much larger than the decrease for the naive version. As

for what causes this, we believe it is the same phenomenon as with loop tiling: the unrolling prevents other, more significant optimisations from being applied.

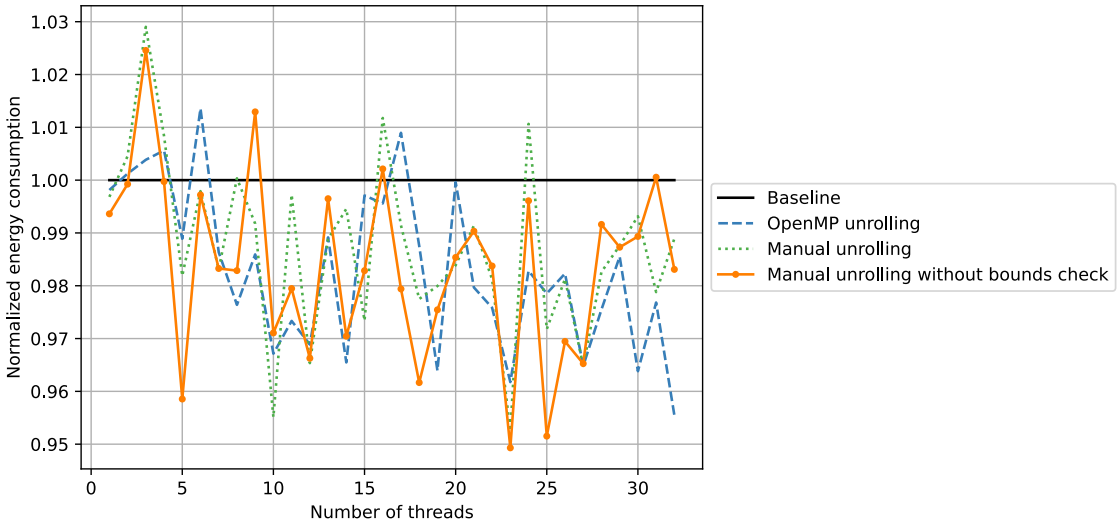


Figure 5.3: Relative energy consumption for the naive matrix multiplication with unrolling.

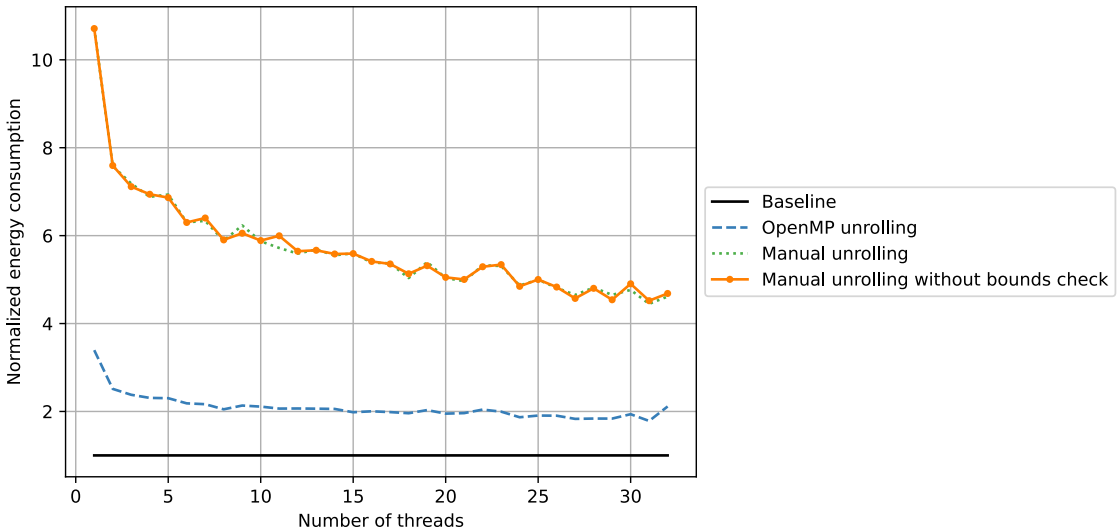


Figure 5.4: Relative energy consumption for the reordered matrix multiplication with unrolling.

For GCC, the results are very similar to those of Clang; unrolling the naive versions provides a few percent lower energy, while unrolling the reordered version roughly doubles the energy. Plots can be found in the appendix, in figures A.5 and A.6. ICC is, as usual, the odd one out. Unrolling the naive algorithm makes the results several times worse regardless of unrolling method. For the reordered algorithm we however see that the second unrolling method, where we assume that the unrolling factor evenly divides the loop iterations, the results become better than the baseline.

Table 5.3 shows the average results for all numbers of threads. As with loop tiling, there are two groups of results. In the first group, we have minor energy reductions of about 1.5 % for Clang and one GCC configuration. All other results are terrible, at least doubling energy consumption and generally increasing power about 30 %. We see that results from unrolling the naive version with ICC roughly matches the results for Clang and GCC, indicating that the same phenomenon has happened for loop unrolling as happened with loop tiling: ICC successfully performed loop reordering in the baseline version, but failed to do so when a loop transformation is applied. To conclude, when applying loop unrolling to a program such as this, the programmer can expect either a very minor improvement or a massive deterioration of execution time and energy consumption likely caused by other optimisations not being applied successfully.

Table 5.3: Summary of the matrix multiplication unrolling results across all numbers of threads. The relative numbers are relative to the baseline with no unrolling for the same compiler.

Compiler	Version	Unrolling	Absolute			Relative		
			T [s]	P [W]	E [J]	T	P	**E**
clang	Naive	Baseline	0.289	169.5	48.93	1	1	**1**
clang	Naive	OpenMP	0.288	167.0	48.13	0.998	0.985	**0.984**
clang	Naive	Manual v1	0.288	167.3	48.13	0.996	0.987	**0.984**
clang	Naive	Manual v2	0.289	167.3	48.27	0.999	0.987	**0.986**
gcc	Naive	Baseline	0.280	176.4	49.39	1	1	**1**
gcc	Naive	Manual v1	0.279	170.7	47.65	0.997	0.968	**0.965**
gcc	Naive	Manual v2	0.278	170.9	47.48	0.992	0.969	**0.961**
icc	Naive	Baseline	0.043	119.1	5.170	1	1	**1**
icc	Naive	Manual v1	0.284	168.6	47.82	6.537	1.416	**9.253**
icc	Naive	Manual v2	0.284	168.9	47.96	6.545	1.418	**9.279**
clang	Reordered	Baseline	0.050	119.5	5.960	1	1	**1**
clang	Reordered	OpenMP	0.089	135.7	12.12	1.791	1.135	**2.034**
clang	Reordered	Manual v1	0.089	136.0	12.13	1.788	1.139	**2.036**
clang	Reordered	Manual v2	0.090	135.8	12.18	1.799	1.136	**2.044**
gcc	Reordered	Baseline	0.034	131.8	4.520	1	1	**1**
gcc	Reordered	Manual v1	0.073	143.7	10.50	2.133	1.090	**2.325**
gcc	Reordered	Manual v2	0.072	144.4	10.40	2.101	1.095	**2.302**
icc	Reordered	Baseline	0.043	118.8	5.130	1	1	**1**
icc	Reordered	Manual v1	0.085	139.3	11.81	1.964	1.172	**2.303**
icc	Reordered	Manual v2	0.039	115.8	4.470	0.894	0.975	**0.871**

5.2 2D stencil

This section evaluates loop transformations on the 2D stencil described in section 3.2.2. The input matrix consists of 2050x2050 single-precision floating-point

numbers. This size is chosen for the same reason as the matrix multiplication input size; it is short enough to run many configurations in a reasonable time but long enough to not be majorly affected by external events such as the OS scheduling background processes.

5.2.1 Loop tiling

Like with matrix multiplication, we apply three versions of tiling: OpenMP, manual with bounds checks, and manual without. We test sizes 1, 2, 4, 8, 16, 32 and 64, and proceed with size 8 which yields the best results.
Like before, we begin by focusing on the results for Clang. In figure 5.5 we see the energy results relative to the baseline program without tiling. Like with matrix multiplication, we observe that OpenMP-induced tiling performs the worst, followed by both versions of manual tiling. Likewise, tiling becomes relatively less impactful as more threads are used. The corresponding plots for GCC and ICC are seen in the appendix, in figures A.9 and A.10. The results for GCC are very similar to those of Clang, with both versions of manual tiling performing about the same. The first version of tiling with ICC performs about the same as well. The odd one out is ICC with the version of tiling where the loop iterations are assumed to be divisible by the tile size, which performs significantly better than anything else. We are unsure what causes this, but it is most likely due to allowing for other compiler optimisations.

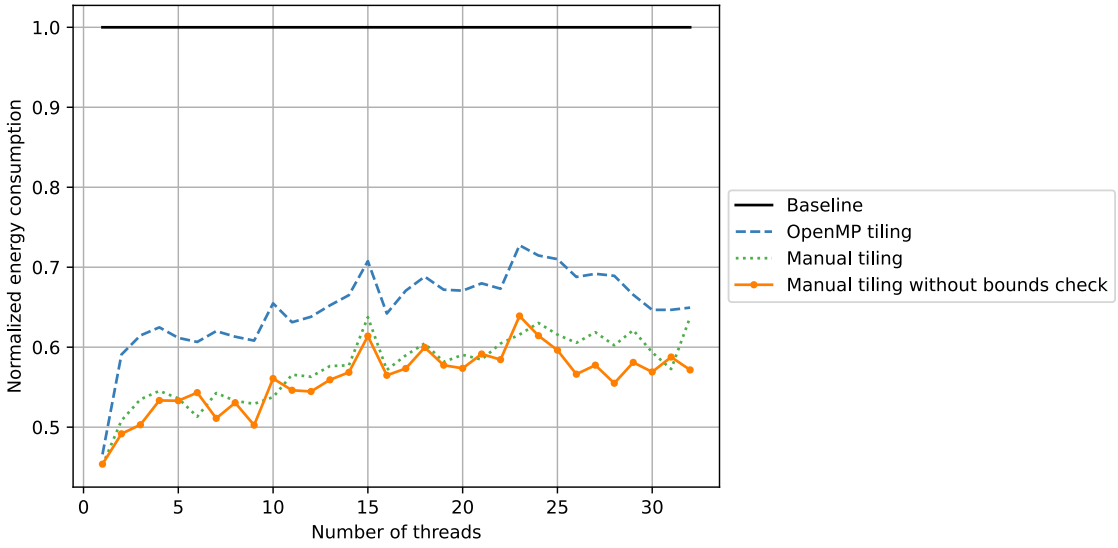


Figure 5.5: Energy consumption for the 2D stencil program with different versions of tiling, compiled with Clang.

To further analyze the impact of tiling, we use the tool *perf* to perform an analysis on the cache behavior, the results of which can be seen in table 5.4. Interestingly, the amount of misses actually increase with tiling, but the total amount of accesses decrease significantly. We are unsure what causes this.
We conclude this section with table 5.5, where we show the average result across all numbers of threads. In contrast to matrix multiplication, Clang outperforms GCC

Table 5.4: L1-D cache behavior for 16-threaded 2d stencil with and without tiling.

	L1-D load misses	L1-D loads	L1-D load miss rate
No tiling	17,861,341	1,754,756,062	1.02 %
OpenMP	20,809,110	1,228,713,710	1.69 %
Manual v1	20,886,749	1,123,857,242	1.86 %
Manual v2	21,028,672	1,123,852,658	1.87 %

here. The table also confirms that manual tiling is better than that of OpenMP in this case. We also note that ICC has the worst baseline results, but has also the best single result with the second version of unrolling.

Table 5.5: Summary of the 2d stencil tiling results across all numbers of threads. The relative numbers are relative to the baseline with no tiling for the same compiler.

Compiler	Tiling	Absolute			Relative		
		T [s]	P [W]	E [J]	T	P	**E**
clang	Baseline	0.222	126.1	27.98	1	1	**1**
clang	OpenMP	0.143	126.9	18.16	0.645	1.006	**0.649**
clang	Manual v1	0.128	125.4	16.04	0.576	0.994	**0.573**
clang	Manual v2	0.124	125.8	15.63	0.560	0.997	**0.559**
gcc	Baseline	0.238	123.9	29.45	1	1	**1**
gcc	Manual v1	0.144	125.9	18.09	0.604	1.017	**0.614**
gcc	Manual v2	0.144	125.9	18.08	0.604	1.017	**0.614**
icc	Baseline	0.259	132.2	34.29	1	1	**1**
icc	Manual v1	0.148	134.0	19.89	0.572	1.014	**0.580**
icc	Manual v2	0.087	125.3	10.94	0.337	0.948	**0.319**

5.2.2 Loop unrolling

As with unrolling in matrix multiplication, we unroll the innermost loop with three variants: OpenMP-generated unrolling, manual unrolling that does assume that the total number of iterations is divisible by the unrolling factor, and manual unrolling that does not. Similar to loop tiling, we determine the best unroll factor empirically by testing factors 1, 2, 4, 8, 16, 32 and 64. 32 turns out to be the best, which is the factor we use for the following presented data.

For Clang, there is almost no difference with either of the unrolling methods. The results can be seen in figure A.11 in the appendix. However, for both GCC and ICC there are some interesting results, as can be seen in figures 5.6 and 5.7. For GCC, both variants of unrolling perform around 10 % better no matter the number of threads, with some variance around 25 threads.

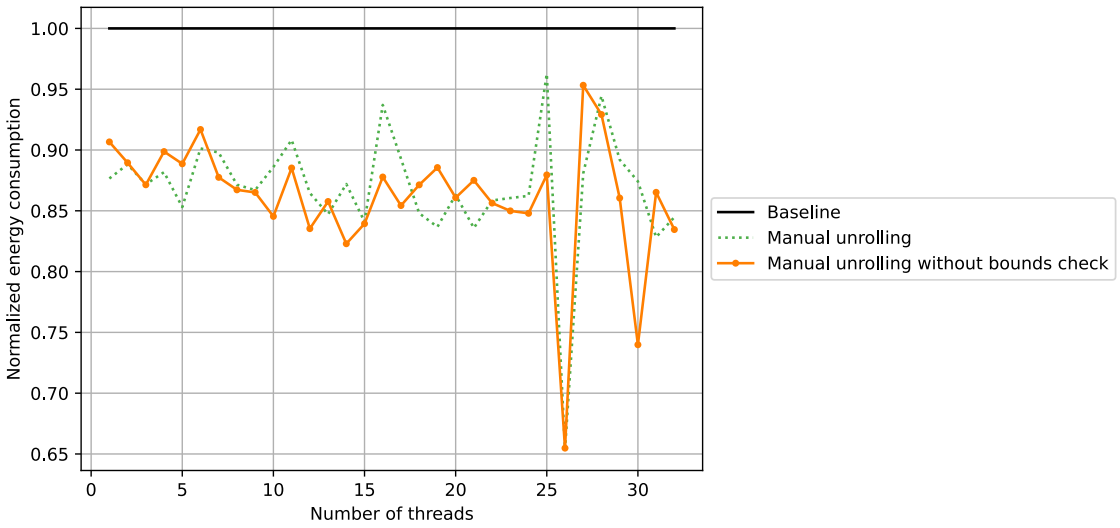


Figure 5.6: Relative energy consumption from applying unrolling to the 2D stencil program, compiled with GCC.

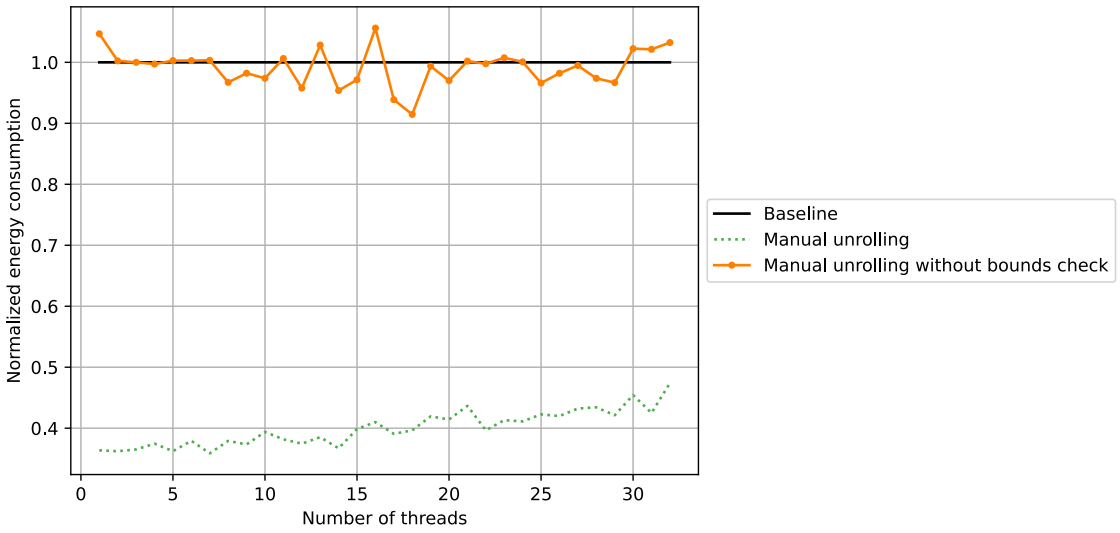


Figure 5.7: Relative energy consumption from applying unrolling to the 2D stencil program, compiled with ICC.

For ICC, however, we see a huge improvement of 60 % in one of the unrolling cases. This is a strange result, you would expect the unrolling variant without the bounds check to be faster, but it seems fairly equal to the baseline. Note that a similar result was observed for ICC with tiling: the second tiling variant performed much better than the first one.
Table 5.6 summarises the results of unrolling the 2D stencil program. We can first of all confirm that unrolling has a small but positive effect on energy of about a percent, due to lower power consumption. GCC sees an improvement of 13-14 % due to both shorter execution time and lower power. With this improvement, it becomes better than any version using Clang despite having a worse baseline. ICC has the worst

baseline, but the unrolling version without assuming the factor evenly divides the iterations provides a huge improvement that outclasses all other configurations. We are unsure what causes this.

Table 5.6: Summary of the 2D stencil unrolling results across all numbers of threads. The relative numbers are relative to the baseline, without unrolling for the same compiler.

Compiler	Unrolling	Absolute			Relative		
		T [s]	P [W]	E [J]	T	P	**E**
clang	Baseline	0.221	120.7	26.66	1	1	**1**
clang	OpenMP	0.221	120.2	26.61	1.002	0.996	**0.998**
clang	Manual v1	0.222	118.9	26.40	1.005	0.986	**0.990**
clang	Manual v2	0.222	118.6	26.35	1.005	0.983	**0.988**
gcc	Baseline	0.242	123.7	29.98	1	1	**1**
gcc	Manual v1	0.214	121.5	26.00	0.883	0.982	**0.867**
gcc	Manual v2	0.213	121.1	25.77	0.878	0.979	**0.860**
icc	Baseline	0.259	132.2	34.29	1	1	**1**
icc	Manual v1	0.109	125.1	13.67	0.421	0.947	**0.399**
icc	Manual v2	0.260	130.5	33.99	1.004	0.987	**0.991**

5.3 Parallelism constructs microbenchmark

As described in section 3.1.1, the purpose of this microbenchmark is to experiment with the different ways of applying parallelism to a loop, and to measure and compare the overheads of doing so. It can also be used to evaluate the different waiting policies in this context as well. The program is based around creating a number of tasks, with said tasks simply consisting of busy-waiting for a number of microseconds. The granularity of these is uniformly randomised from 0 to max_task_size to make the workload somewhat uneven. During our experiments, we use 20, 40, 60, 80, 100 and 200 μs for max_task_size.

We begin the analysis by focusing on the ways of using tasking, of which we use three methods: using a single thread to create all tasks, using multiple threads to create tasks, and finally by using the taskloop construct. The single-threaded variant is a fairly common construct, where one thread creates all tasks and other threads execute the tasks, similarly to listing 2.2. The idea behind the multi-threaded variant is to have most threads create and execute mostly their own tasks, and then perform work-stealing to efficiently balance the workload. The final variant is to use the *taskloop* clause in OpenMP, which is essentially syntactic sugar for a single-threaded task generation with optional parameters, which we do not consider.

Looking at the results for Clang only, table 5.7 shows the results for some different values of task granularity. There are several interesting observations to make here. To start, we see that the multi-threaded variant actually consumes quite little power compared to both the other variants, which is not very intuitive. This makes it perform better in terms of energy for all levels of task granularity, even the larger ones

where execution time is almost equal. We also see that active waiting outperforms passive waiting in most of the cases, except for the smallest granularity for the single-threaded variant. The reason for active waiting being better is probably due to the low amount of time threads spend idle in this microbenchmark.

Table 5.7: Characteristics of the tasking methods for the parallelism constructs microbenchmark using Clang. Measurements are reported as the geometric mean of said measurement for all numbers of threads.

		Single-threaded			Multi-threaded			Taskloop		
Granularity	Waiting	T [s]	P [W]	**E [J]**	T [s]	P [W]	**E [J]**	T [s]	P [W]	**E [J]**
20	Active	0.31	156	**48.54**	0.1	120	**12.44**	0.11	132	**14.09**
20	Passive	0.27	154	**42.48**	0.12	122	**15.11**	0.11	129	**14.7**
60	Active	0.12	138	**17.15**	0.1	114	**11.69**	0.1	127	**13.08**
60	Passive	0.12	132	**15.84**	0.11	113	**12.44**	0.11	122	**13.15**
100	Active	0.1	126	**13.11**	0.1	112	**11.52**	0.1	124	**12.82**
100	Passive	0.11	119	**12.97**	0.11	109	**11.71**	0.11	117	**12.66**
200	Active	0.1	121	**12.36**	0.1	114	**11.89**	0.1	128	**13.33**
200	Passive	0.11	112	**11.82**	0.11	106	**11.53**	0.11	121	**12.98**

Indeed, we see that using single-threaded task generation seems to be very inefficient for very small workloads. Figure 5.8 shows the energy consumption of this method for different levels of task granularity, with both active and passive waiting. This confirms that this method of task generation is inefficient for small tasks, and that it is worse the smaller tasks are. Furthermore, we see that passive waiting performs better than active waiting in this case, which is likely due to passive waiting being better when many threads are mostly inactive. It should be mention that this method scales fairly well up to about 8 threads even for the tiniest tasks, so it would likely be reasonably effective in any realistic setting where tasks are typically of a larger of a larger scale than tens of microseconds.

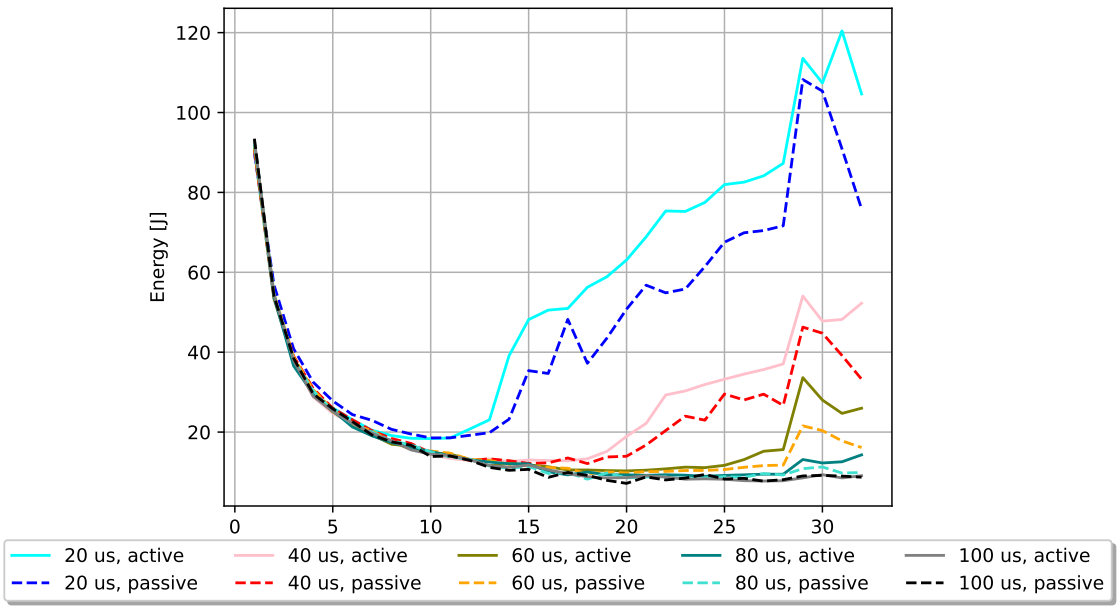


Figure 5.8: Energy consumption for the single-threaded task generation in the parallel constructs microbenchmark for Clang. The method scales well up to some number of threads, based on the granularity of tasks.

We select the multi-threaded task generation to be the best for now, and proceed to compare it to using a *parallel for* loop. Since the size of each task is uneven, the time to compute each loop iteration can vary. This means that it makes sense to experiment with both the default *static* scheduling, which statically divide the loop iterations among the team of threads, and the *dynamic* scheduling, which dynamically schedules loop iterations among the threads. The optional chunk size parameter is unused, making it default to 1.

Table 5.8 shows an overview of the results for the *parallel for* method using both static and dynamic scheduling, as well as the tasking method for reference. Once again there are several interesting observations to make. We see that the tasking construct consistently outperforms the loop method by a factor of 9.7 % in terms of energy as long as the active waiting policy is used. If passive waiting is used, however, the parallel loop variant performs on 13.8 % better in terms of energy due to the lower power consumption. Conversely, the execution time is actually around 15 % higher. Furthermore, if dynamic scheduling is used, the parallel loop variant consistently outperforms tasking regardless of the waiting policy. With active waiting, the power consumption is roughly the same but the execution time is better, while with passive waiting both execution time and power consumption is better for the parallel loop than the tasking method. Comparing the best configurations, those being tasking with active waiting and parallel loops with passive waiting and dynamic scheduling, for each task granularity gives 9.7 % better energy consumption in favor of parallel loops.

Table 5.8: Results for the parallel for versus tasking methods, with Clang. Measurements are reported as the geometric mean of said measurement for all numbers of threads.

		Tasking			ParFor, static			ParFor, dynamic		
Granularity	Waiting	T [s]	P [W]	**E [J]**	T [s]	P [W]	**E [J]**	T [s]	P [W]	**E [J]**
20	Active	0.1	120	**12.44**	0.11	123	12.97	0.1	121	11.79
20	Passive	0.12	122	15.11	0.13	91	**11.55**	0.11	104	**11.55**
60	Active	0.1	114	**11.69**	0.11	125	13.27	0.1	115	11.32
60	Passive	0.11	113	12.44	0.12	91	**10.58**	0.1	101	**10.51**
100	Active	0.1	112	**11.52**	0.11	124	13.4	0.1	119	11.81
100	Passive	0.11	109	11.71	0.12	91	**10.72**	0.1	100	**10.4**
200	Active	0.1	114	11.89	0.11	122	13.27	0.1	118	11.9
200	Passive	0.11	106	**11.53**	0.12	88	**10.46**	0.1	100	**10.41**

The corresponding results for ICC are found in the appendix, in tables A.1 and A.2, due to their similarity to Clang. To summarize for ICC, we again see that single-threaded task generation with small task granularity performs very poorly, but that this quickly becomes less of a concern for tasks around 200 μs. Comparing tasking versus parallel looping, we see that tasking consistently performs the worst provided that the passive waiting policy is used. We conclude that for ICC, an active waiting policy should be used when using tasking methods. However, when there is a choice between tasking and parallel loops, the latter should be used and the waiting policy should be passive.

The results for GCC are seen in tables 5.9 and 5.10. These results show some key differences compared to Clang and ICC. To start, we see that passive waiting consistently performs better than active waiting. Furthermore, comparing the tasking variants, we see that the taskloop is the best. This is therefore what is compared to the parallel for loop variants. In that table, we observe that the parallel for with static scheduling and the taskloop both contend for being the best, with the dynamically scheduled parallel for consistently performing the worst in terms of energy. It does however perform the best in terms of execution time. We conclude that taskloops and parallel for loops are about equal for GCC, but that manually creating tasks performs badly, be it single-threaded or multi-threaded. Figure 5.9 shows the energy results for the single-threaded version. Compared to the corresponding result for Clang, the active waiting performs significantly worse than passive. It also seems to scale linearly with the amount of threads, while the plots for passive waiting flattens and becomes constant around 8 threads. This further suggests that passive waiting should be preferred for GCC.

Table 5.9: Characteristics of the tasking methods for the parallelism constructs microbenchmark using GCC.

		Single-threaded			Multi-threaded			Taskloop		
Granularity	Waiting	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]
20	Active	0.65	138	90.27	0.3	138	41.26	0.09	126	11.7
20	Passive	0.61	58	**35.54**	0.22	135	**29.52**	0.12	79	**9.33**
60	Active	0.23	137	31.85	0.14	135	18.98	0.09	127	11.73
60	Passive	0.23	68	**15.6**	0.12	127	**15.37**	0.1	87	**8.62**
100	Active	0.15	135	20.78	0.12	132	15.53	0.09	126	11.54
100	Passive	0.15	80	**11.96**	0.11	125	**13.32**	0.1	88	**8.82**
200	Active	0.1	130	12.74	0.1	128	13.15	0.09	124	11.66
200	Passive	0.1	107	**10.2**	0.1	119	**11.63**	0.1	89	**8.95**

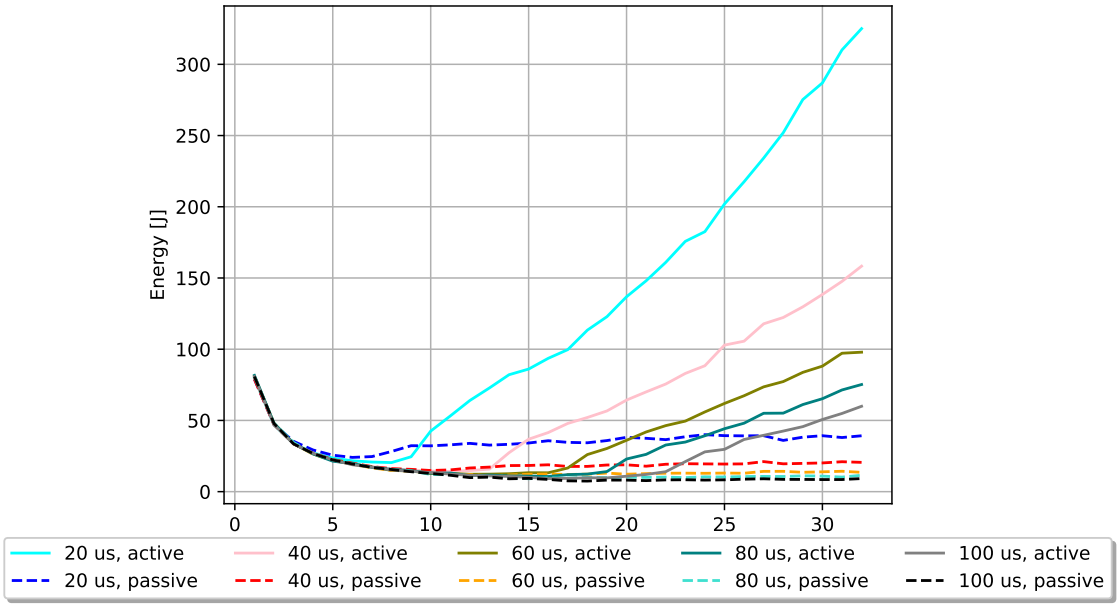


Figure 5.9: Energy consumption for the single-threaded task generation in the parallel constructs microbenchmark for GCC. The method scales well up to some number of threads, based on the granularity of tasks.

Table 5.10: Results for the parallel for versus tasking methods, with GCC. In contrast to Clang and ICC, where the multi-threaded variant is the best tasking method, here we use the taskloop variant to compare with the parallel for variants.

		Tasking			ParFor, static			ParFor, dynamic		
Granularity	Waiting	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]	T [s]	P [W]	E [J]
20	Active	0.09	126	11.7	0.09	120	10.55	0.08	117	9.38
20	Passive	0.12	79	**9.33**	0.12	79	**9.9**	0.09	106	**9.9**
60	Active	0.09	127	11.73	0.09	125	11.21	0.08	116	9.28
60	Passive	0.1	87	**8.62**	0.1	89	**8.89**	0.08	108	**9.07**
100	Active	0.09	126	11.54	0.09	123	11.15	0.08	120	9.91
100	Passive	0.1	88	**8.82**	0.1	85	**8.76**	0.08	113	**9.59**
200	Active	0.09	124	11.66	0.09	123	11.34	0.08	121	9.99
200	Passive	0.1	89	**8.95**	0.1	89	**9.23**	0.09	110	**9.43**

5.4 Inactivity microbenchmark

As mentioned previously, the purpose of this microbenchmark is to evaluate the effect of waiting policies of inactive threads. This is done by creating a parallel section where only one thread performs work and all other threads wait. The work consists of simply busy-waiting for a number of microseconds. By varying how long this waiting time is we can change the granularity of the benchmark as a whole. In our experiments we vary it between 10 and 100 μs, and set the number of iterations to $\frac{1000000}{waiting\ time}$. This means that the optimal execution time is about 1 second. Upon analysis we notice that default and active waiting perform nearly identical for this microbenchmark regardless of compiler. Therefore, we present results for active and passive waiting only.

Table 5.11: Overview of the inactivity microbenchmark results across all numbers of threads. The smallest and largest task granularity of 10 and 100 are included, as well as the granularity where passive waiting becomes better than active for each compiler. These energy values are marked in bold.

Compiler	Granularity [us]	Active			Passive		
		T[s]	P[W]	E[J]	T[s]	P[W]	E[J]
Clang	10	1.11	300	334	4.42	199	878
Clang	30	1.05	303	**317**	1.72	209	**360**
Clang	40	1.04	303	**314**	1.49	206	**306**
Clang	100	1.02	304	311	1.19	199	235
GCC	10	1.06	295	312	2.89	206	595
GCC	40	1.02	293	**298**	1.58	202	**320**
GCC	60	1.01	292	**295**	1.4	201	**282**
GCC	70	1.01	292	295	1.35	200	271
GCC	100	1.01	292	294	1.25	200	250
ICC	10	1.08	296	321	4.34	204	884
ICC	40	1.03	295	**303**	1.49	206	**307**
ICC	50	1.02	295	**302**	1.37	205	**281**
ICC	100	1.02	295	300	1.18	201	237

Some prominent results are seen in table 5.11. We see that for the lowest granularity, active waiting outperforms passive in terms of both execution time and energy. For the highest granularity however, passive waiting outperforms active due to the lower consumption. The table also includes the cutoff points, where passive waiting starts to outperform active in terms of energy. For Clang, we see that it is between 30-40 μs, while it is around 40-70 μs for GCC and 40-50 μs for ICC.

In figure 5.10 we see the energy graph for Clang for these granularities. We see that for 30 μs, the active waiting consistently has lower energy consumption than the passive. Meanwhile, at 40 μs, passive waiting has a lower energy consumption for all but the highest number of threads. The corresponding plot for GCC can be seen in figure 5.11 where we see a similar behavior but the cutoff point is at a higher granularity, as seen in the table before. For GCC, we see that the energy increase with passive waiting is a lot more linear than for Clang and ICC. Up to about 16 threads, the cutoff is about 40 μs, but for the highest numbers of threads it is around 60-70 μs. The plot for ICC can be found in figure A.12 of the appendix due to its similarity to Clang.

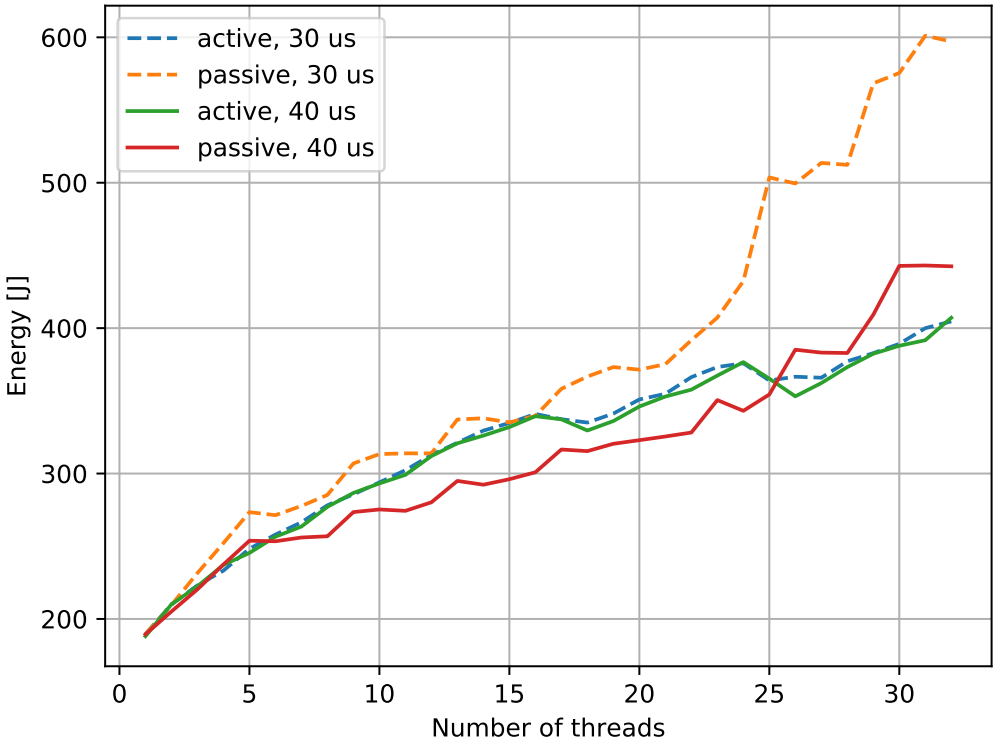


Figure 5.10: Energy consumption at the granularity where passive waiting overtakes active waiting for Clang.

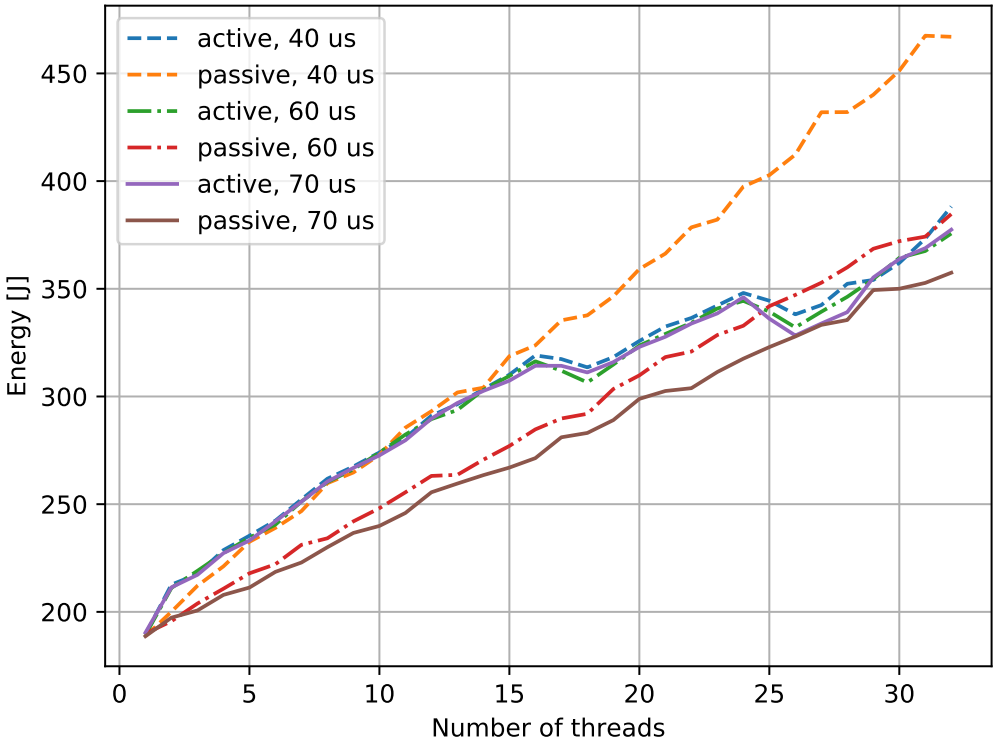


Figure 5.11: Energy consumption at the granularity where passive waiting overtakes active waiting for GCC.

To conclude, the results show that using the active waiting policy is better than the passive for most task sizes, but especially small ones. The cutoff point where passive scheduling is better is higher for GCC than for Clang and ICC. With very small task granularity passive waiting becomes significantly worse, very likely due to the runtime overheads of thread scheduling and descheduling. In these cases, while neither compiler performs well, GCC is better than both Clang and ICC. As a final note, it is worth mentioning that this microbenchmark favors passive waiting due to more threads giving no performance benefits at all, which is not reasonable for real programs. This means that active waiting will likely perform even better in practice due to reducing the execution time.

5.5 Loop unrolling microbenchmark

This section evaluates the effect loop unrolling has on the four simple programs described in section 3.1.3. All results were obtained using *Clang*, since it is the only compiler that supports the new OpenMP `unroll` directive, without any parallelisation and with the 03 optimisation level enabled. An unrolling factor of eight was chosen for both the OpenMP and manual unrolling methods.
Measures were taken so the execution time for each program would be roughly five seconds for the default implementation. This was done by measuring the execution time and energy after running the program 4096 times with some appropriate value for the `len` variable.

The results for the tests can be seen in figure 5.12. The figure shows the relative energy used by each implementation compared to the default implementation without any explicit unrolling for the four programs. What is immediate apparent from the results is that unrolling seems to either have no impact at all, or a very significant one, when looking at these four programs. The results from each program will be discussed in further detail below.
The `SIMPLE_MEM` program got significantly worse for both unrolling implementations, using about 70 % more energy. While all three implementations utilise vector instructions to do the calculations and memory assignment, the original implementation, which was automatically unrolled by a factor of two by the compiler, seem to make use of the instructions more efficiently. Interestingly, when running the OpenMP unrolling version again and matching the generated unrolling factor of two, it still performed worse than the original, but now by only 17 %.
The `SIMPLE_COMP` program didn't see any performance change for the OpenMP unrolling, but saw a significant improvement for the manual unrolling implementation, using only 21 % of the energy compared to the original implementation. The reason for this big difference is because in the manually unrolled version the calculations has been separated into eight independent parts, making it so they can be run in parallel using SIMD instructions. When naively unrolling the program manually without this optimisation, it performs on par with the original implementation.
The `SIMPLE_COMP_DEPEND` program has no meaningful difference in performance between the implementations. None of the implementations make us of vector instructions, unlike the two previous programs discussed, and is probably the reason why there has been no dramatic performance difference.
The implementations of the `COMPLEX_NESTED` program performs very similarly to the `SIMPLE_COMP` program, with OpenMP unrolling having no difference and manual unrolling significantly reducing energy, down to only 25 %. The reason here is similarly also because more efficient usage of vector instructions. When doing the *loop jamming* optimisation, the work in the nested loops gets clumped together into one loop, which the compiler then can parallelise with vector instructions.
In summary, unrolling for these very simple programs tends to either have an insignificant impact on the overall performance, or alternatively a huge impact that can be either positive or negative. What determines if there is a big difference is how well vector instructions can be utilized. If the unrolling disturbs highly efficient automatic optimisations made by the compiler, like for the `SIMPLE_MEM` program, then performance can drop significantly. If instead the unrolling exposes opportunities for ILP, like for the `SIMPLE_COMP` and `COMPLEX_NESTED` programs, then huge performance gains can be achieved.

5.6 Barcelona OpenMP Task Suite (BOTS)

This section considers the BOTS (Barcelona OpenMP Task Suite) benchmark suite, presented previously in section 3.3. We first analyse the choice of waiting policy, which is covered in section 5.6.1. Two of the benchmark programs also have the option to use either single-threaded and multi-threaded task generation, which we have previously analysed with the parallel constructs microbenchmark. This is treated

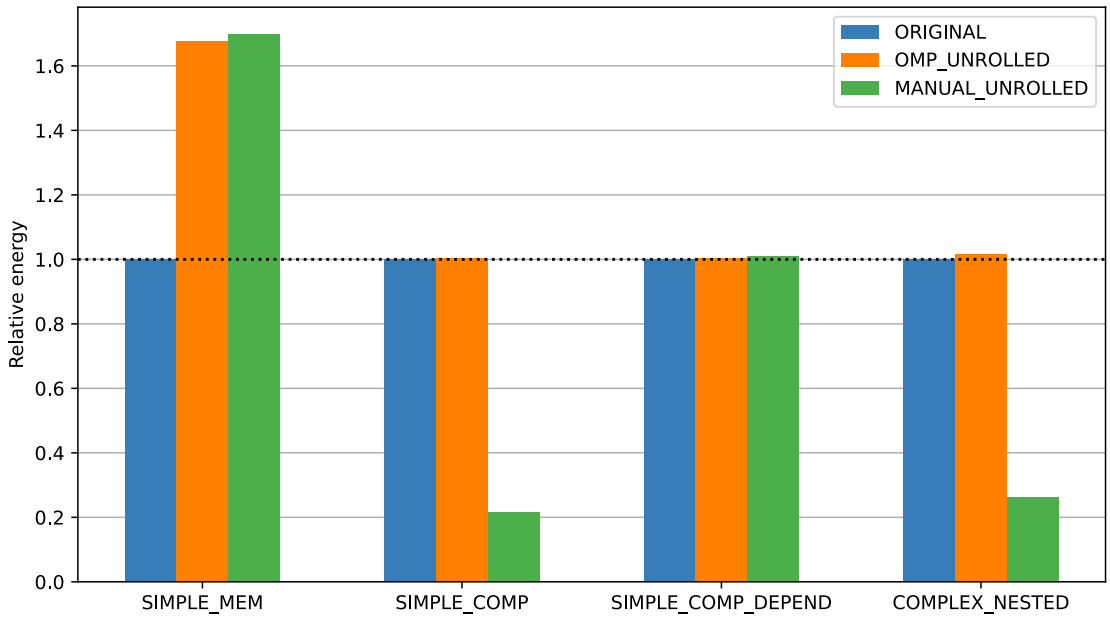


Figure 5.12: Relative energy used for the different unrolling implementations compared to no explicit unrolling for the unrolling microbenchmark programs.

in section 5.6.2. Finally, we examine the source code for each program and apply loop tiling and unrolling in performance-critical areas to analyse the effects of these transformations in a realistic setting. Unfortunately, we find no programs in which we can apply loop tiling, so therefore only loop unrolling is considered, and is covered in section 5.6.3.

5.6.1 Waiting policy

In these experiments we investigate the impact of the waiting policy on the BOTS benchmarks. We run the *alignment*, *fft* *fib*, *health*, *sort*, *sparselu*, *strassen* and *uts* benchmarks with the waiting policy set to active, passive or default.

The mean energy consumption can be seen in figure 5.13. The most obvious result is that GCC performs poorly with high numbers of threads. Since neither of the other compilers do so, this indicates some kind of inefficiency of tasking in GCC. The best results are produced by ICC and Clang, both with active waiting. Conversely, for GCC, active waiting performs the worst.

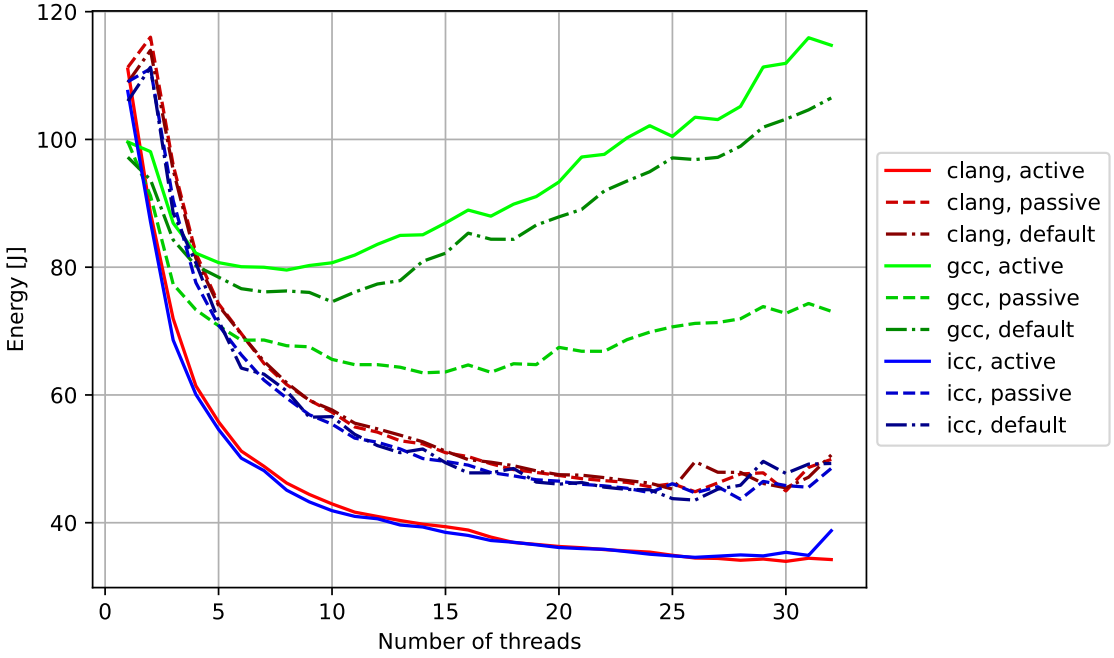


Figure 5.13: Average energy consumption of all BOTS programs for the three compilers and waiting policies.

The corresponding graphs for execution time and power consumption can be found in the appendix, in figures A.13 and A.14. The curve for execution time is very similar to that of energy consumption with some minor differences. Looking only at the two best configurations that were best in terms of energy, which were ICC and Clang with active waiting, we see that they are again best in terms of execution time. For high numbers of threads we do however see that Clang is slightly faster, but that ICC consumes less power, leading them to be roughly equal in terms of energy as seen before.
We conclude that for a task-heavy workload such as the BOTS programs, Clang or ICC are preferred and should use active waiting.

5.6.2 Single-threaded and multi-threaded task generation

The *alignment* and *sparselu* programs include both single- and multi-threaded task generation, working similarly to the parallel construct benchmark described in section 3.1.1. In figures 5.14 we compare the two methods for these programs for each compiler, with the waiting policy set to default. While there is a major difference between the compilers for these programs, GCC being much faster, there is almost no difference between the tasking methods. Single-threaded generation consumes 2.4 %, 1.6 %, and 0.7 % less energy for Clang, GCC and ICC, respectively. We draw the conclusion that for tasks of this granularity (which should represent most realistic workloads), there is no reason to use multi-threaded task generation when single-threaded is more energy efficient and easier to implement. The results are almost exactly the same regardless of waiting policy as well.

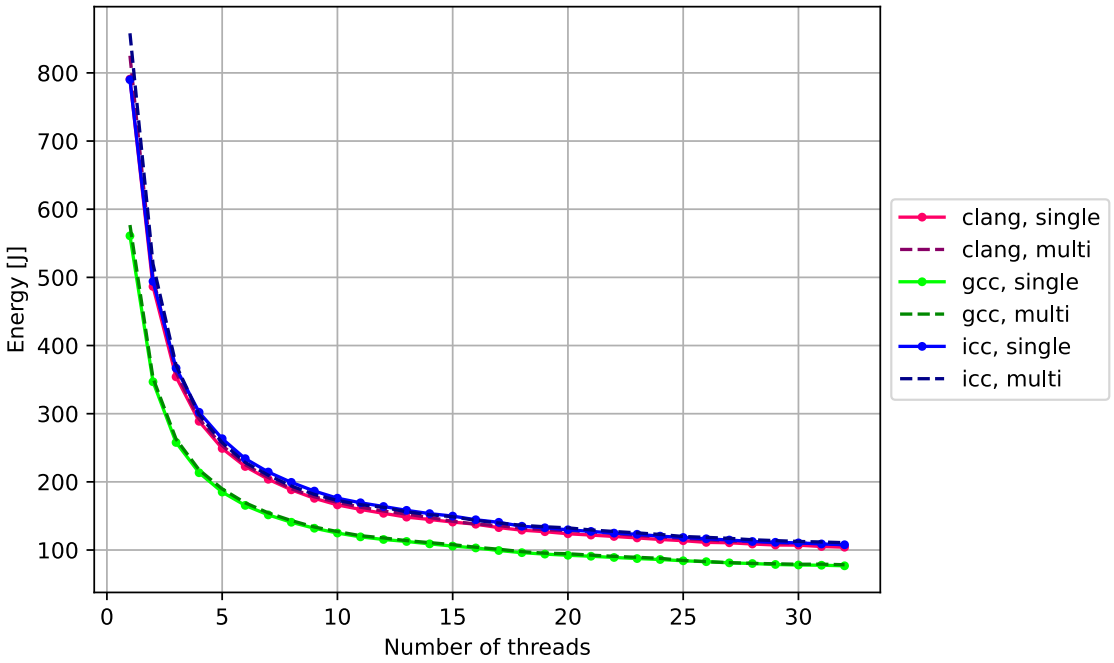


Figure 5.14: Energy consumption comparison between single-threaded and multi-threaded task generation, for the two programs supporting both in BOTS. We see that there is very little difference between the task creation variants.

5.6.3 Loop Unrolling

Two benchmarks from BOTS were chosen to be analysed with unrolling, *strassen* and *alignment*. These were chosen due to most of their execution time (from profiling) was spent in functions where unrolling could be applied.
Both programs were tested with manual unrolling and the OpenMP `unroll` directive. The results for these tests can be seen in in figure 5.15. The figure shows the relative amount of energy used by each configuration compared to the implementation without explicit unrolling. The results shown are the median relative energy values when running the program with 1 to 32 threads and compiled with Clang. Several unrolling factors were tested and only the best performing is shown for each program and implementation. It should also be mentioned that the execution time followed the energy usage very closely for the following tests, so any gain in energy is a result of a decrees directly proportional to the execution time.
For *alignment* one very simple for loop which initialises two arrays with values was analysed. For this program neither unrolling version had any positive impact on the performance, where manual and OpenMP unrolling had with their best implementation no impact and an increase of 2.2 % on energy respectively. These implementation had an unrolling factor of 32 for the manual and 8 for the OpenMP version. For some of the tested unrolling factors energy could increase by as much as 15 %.
In the case of the *strassens* program the original implementation was already manually unrolled, so the *no unrolling* baseline for figure 5.15 is instead a modified version we have created where this unrolling is removed. The code section that is being un-

rolled for this program is the outer loop of two nested loops, inside of which some calculations are done. This code section is almost identical to the `COMPLEX_NESTED` program, described in section 3.1.3, and was the inspiration for the microbenchmark. For the manual version the *loop jamming* optimisation has also here been similarly applied.
In contrast to the case of *alignment*, unrolling for *strassen* always lowered the total energy usage. With the best performing implementation, the original manual unrolling with unrolling factor 8, using only about 55 % of the energy of having no unrolling, and the OpenMP version, with the same unrolling factor, saving around 8 %. The reason why the manual method performed significantly better than both the default and OpenMP version is because the way the code is restructured allows for auto-vectorisation by the compiler.

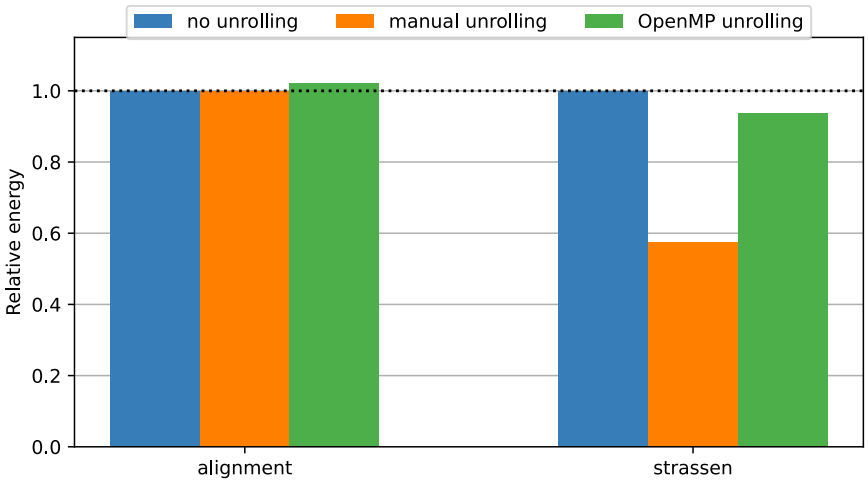


Figure 5.15: Relative energy used for manual and OpenMP unrolling compared to no unrolling for the alignment and strassen programs. Several unrolling factors were tested and only the best for each version is shown.

5.7 NAS Parallel Benchmarks

In this section we present the results of experiments on the NAS Parallel benchmarks. The impact of the waiting policy is explored in subsection 5.7.1 by executing all (unmodified) programs with the three waiting policy options. Loop transformations are explored in subsection 5.7.2 by applying transformations to loops in functions where the most execution time is spent for each program. This is mainly done with loop unrolling, since it is significantly more difficult to apply loop tiling to programs.

5.7.1 Waiting policies

To evaluate the impact of the OpenMP waiting policy we execute the otherwise unmodified programs with active, passive and default waiting with all three compilers. In figure 5.16 we see the average energy results of executing the NPB programs with the different compilers and waiting policies. We see that, while the difference

is small, active waiting consumes the least energy and passive consumes the most. ICC with active waiting consistently performs the best. Clang with active waiting also performs well. For BOTS, GCC scaled poorly beyond about 10 threads, which is clearly not the case here. This could further indicate that GCC performs fairly well with parallel loops, as used here, but not so well with tasking, as used in BOTS. We also note that GCC performs the best with default waiting in this case.

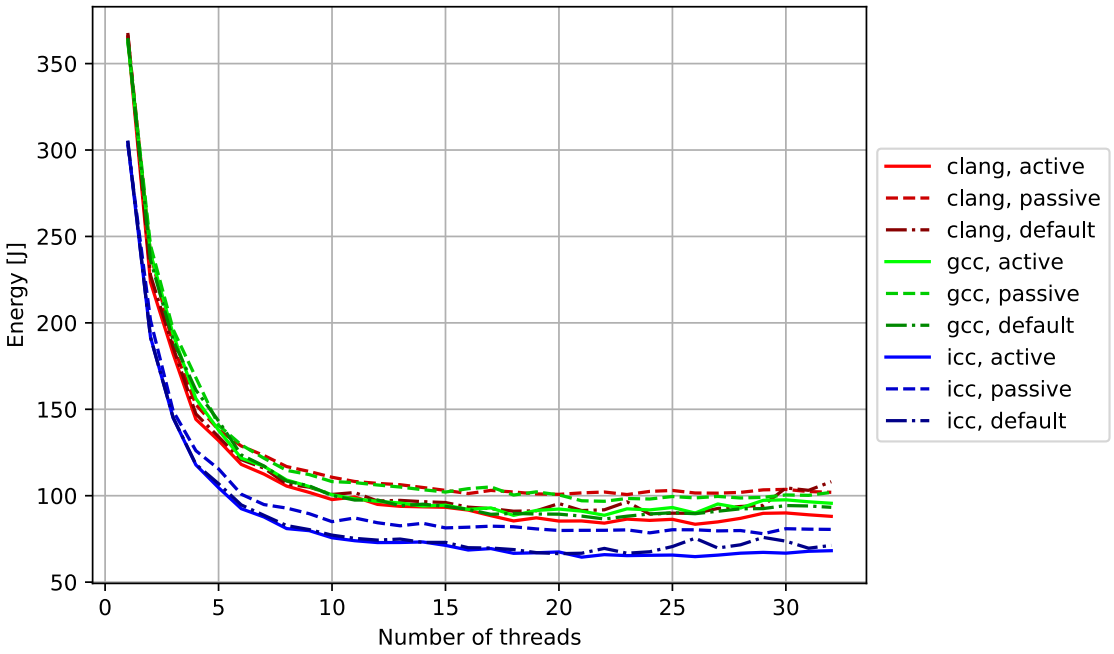


Figure 5.16: Average energy consumption of all NPB programs for different numbers of threads, compilers and waiting policies.

The corresponding plot for the execution time is very similar to the energy and can be found in the appendix, in figure A.15. The main difference is that passive waiting performs worse for all compilers. The power can be seen in figure 5.17, where we see that active and default waiting follows roughly the same curve for all compilers, while passive waiting consumes significantly less power.

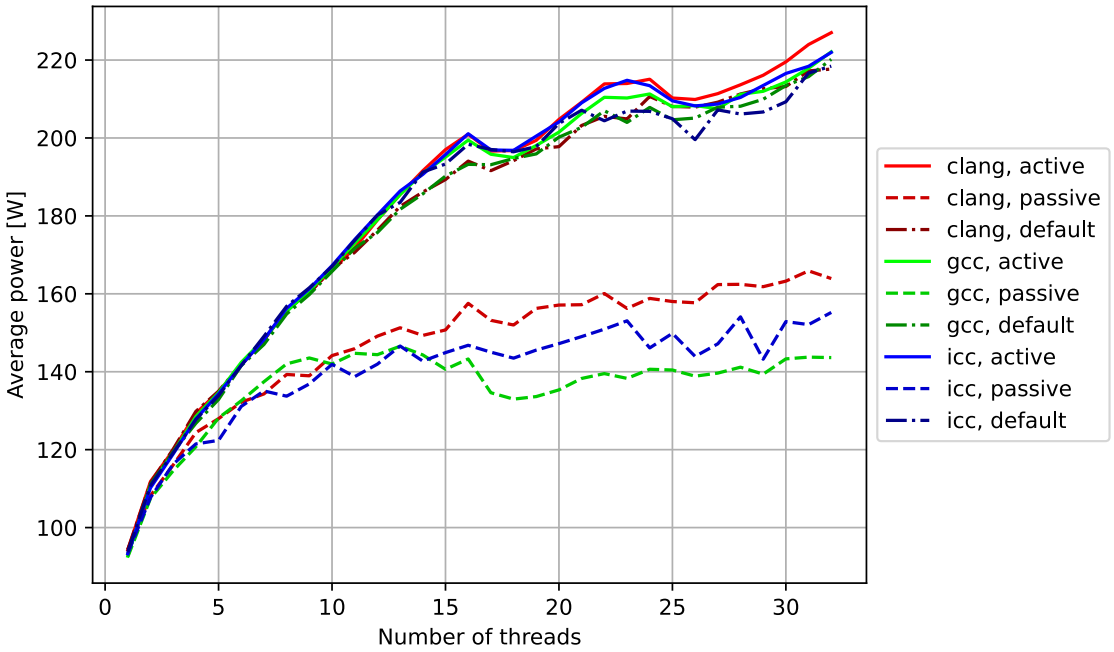


Figure 5.17: Average power consumption of all NPB programs for different numbers of threads, compilers and waiting policies.

Table 5.12 summarizes the results and shows the execution time, power and energy averaged across all numbers of threads. For Clang and ICC we confirm that active waiting consumes about 5 % less energy than default waiting, while default waiting is (barely) the best for GCC. We also see that passive waiting consumes around 20 % less power.

ICC is by far the best compiler in this case due to its comparable power consumption but much lower execution time. Clang and GCC are about equal.

Table 5.12: Summary of waiting policy results for NPB.

Compiler	Policy	Absolute			Relative		
		T [s]	P [W]	E [J]	T	P	**E**
clang	Default	99.55	160.92	218.94	1	1	**1**
clang	Active	94.04	163.24	206.6	0.945	1.014	**0.944**
clang	Passive	105.9	132.2	237.39	1.064	0.822	**1.084**
gcc	Default	97.55	159.84	214.69	1	1	**1**
gcc	Active	98.83	161.8	217.28	1.013	1.012	**1.012**
gcc	Passive	104.71	122.16	237.17	1.073	0.764	**1.105**
icc	Default	75.2	160.41	165.49	1	1	**1**
icc	Active	72.74	161.92	159.95	0.9673	1.009	**0.967**
icc	Passive	83.84	125.86	188.97	1.115	0.785	**1.142**

5.7.2 Loop Transformations

To explore the impact of loop transformations we profile each program using the *perf* tool. Then, we apply transformations to loops in these functions both manually and with OpenMP. In some cases the loops are already unrolled, in which case we also experiment with rolling the loop again. It is worth mentioning that the implemented transformations are to the best of the authors abilities.
In table 5.13 we show an overview of the implemented transformations. As shown, programs where both OpenMP and manual unrolling have been applied are BT, CG, EP and MG. In IS and LU we could not find any suitable loops. Tiling is significantly more difficult to apply to existing programs, especially those already optimised to such a level as there. The only program we applied it to was FT. This however only lead to performance detriments, so the results are not presented further.

Table 5.13: Implemented optimisations for each of the NPB programs.

Program	Rolled	OpenMP Unrolling	Manual Unrolling	OpenMP Tiling
BT	Yes	Yes	Yes	
CG	Yes	Yes	Yes	
EP		Yes	Yes	
FT		Yes		Yes
IS				
LU				
MG		Yes	Yes	

First, we consider the programs where both manual and OpenMP unrolling was applied, namely BT, CG, EP and MG. Results for these are seen in figures 5.18, 5.19, 5.20 and 5.21.

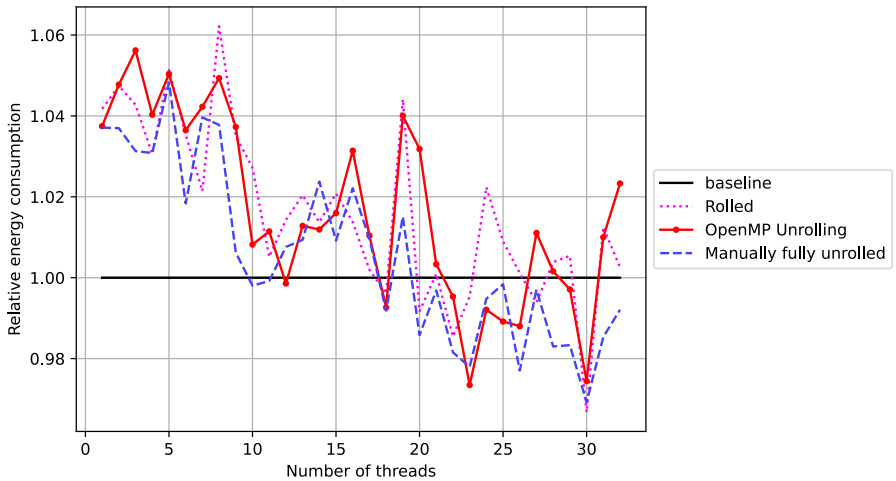


Figure 5.18: Relative energy consumption of unrolling the BT program.

In the BT program, most time is spent in the *lhs_x*, *lhs_y* and *lhs_z* functions. They contain innermost loops that are fully unrolled by default, which we roll into

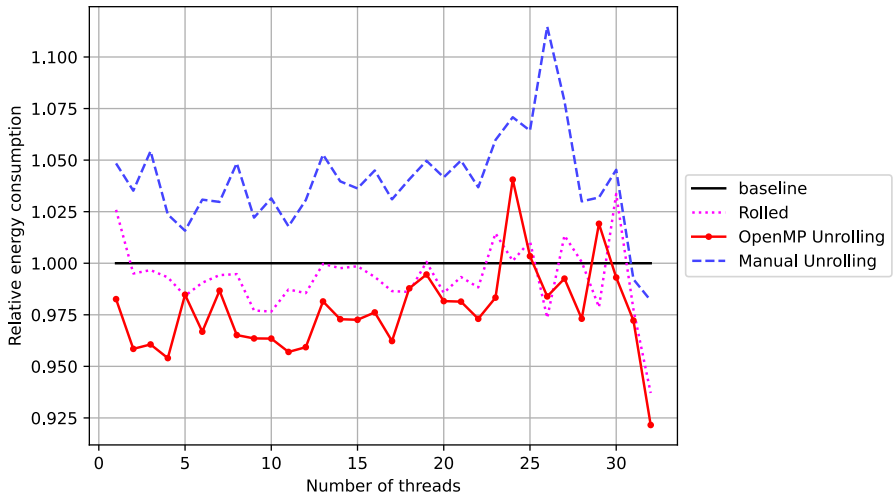


Figure 5.19: Relative energy consumption of unrolling the CG program.

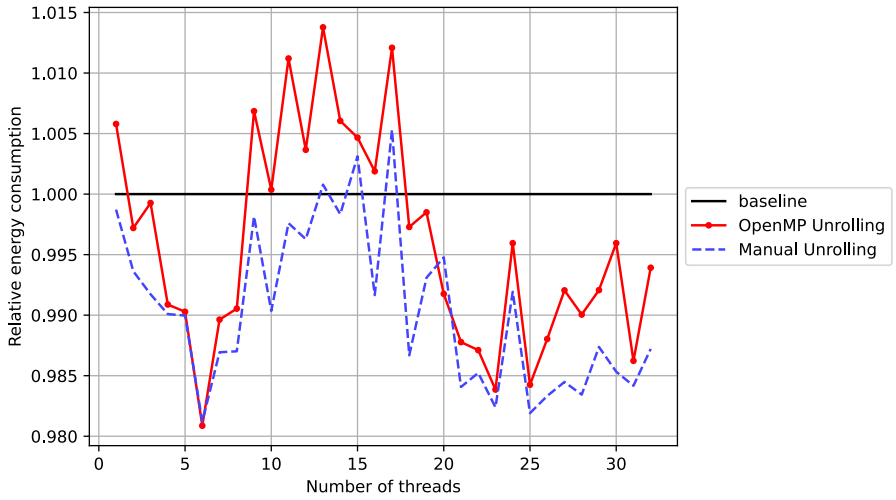


Figure 5.20: Relative energy consumption of unrolling the EP program.

loops. After this, we apply full loop unrolling both with OpenMP and manually. While the baseline is best for low numbers of threads, unrolling becomes better with more threads. The CG program, which computes a conjugate gradient, spends most of its execution time in the *conj_grad* function. Here we apply manual and OpenMP unrolling. Unrolling using OpenMP yields the best results, especially for low numbers of threads, even in comparison to the manual unrolling. In the EP program we apply both manual and OpenMP unrolling. We see that manual unrolling is better than OpenMP unrolling in general, but that the difference is very small. In the MG program we apply unrolling to the *resid* function. Results show that unrolling yields worse results than the baseline.

These results are summarized in table 5.14. We observe that the differences between any of the configurations are very small, usually about just one percent. We also observe that on average, unrolling in this way yields worse energy results of about a percent for OpenMP unrolling and two percent for manual unrolling.

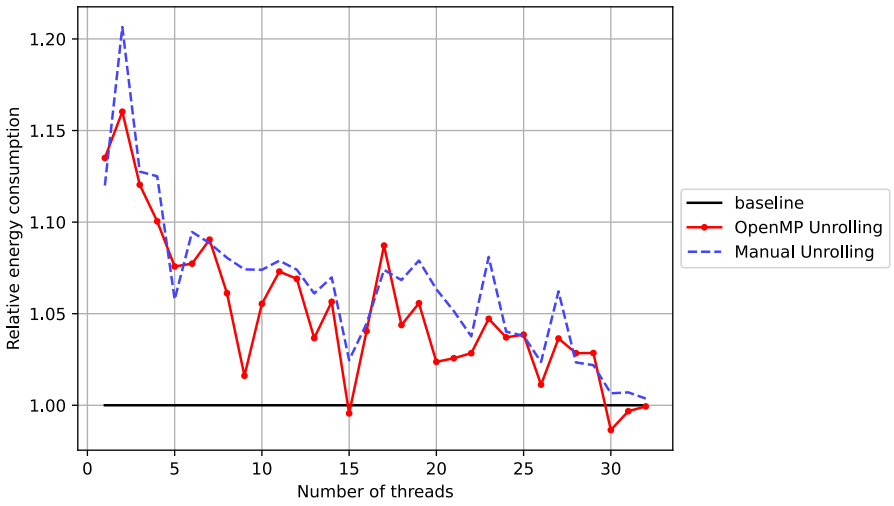


Figure 5.21: Relative energy consumption of unrolling the MG program.

Table 5.14: The improvement of unrolling each program in NPB compared to the baseline, averaged across all numbers of threads.

Program	Unrolling Method	Absolute			Relative		
		T [s]	P [W]	E [J]	T	P	**E**
BT	Baseline	8.812	191.2	1685	1	1	**1**
BT	Rolled	9.058	189.0	1712	1.028	0.988	**1.016**
BT	OpenMP	9.060	188.9	1712	1.028	0.988	**1.016**
BT	Manual	9.002	188.3	1695	1.021	0.985	**1.006**
CG	Baseline	0.112	178.3	19.94	1	1	**1**
CG	Rolled	0.111	178.7	19.80	0.991	1.002	**0.993**
CG	OpenMP	0.109	178.5	19.49	0.976	1.001	**0.977**
CG	Manual	0.110	188.5	20.74	0.984	1.057	**1.040**
EP	Baseline	1.946	148.3	288.5	1	1	**1**
EP	OpenMP	1.945	147.7	287.3	1.000	0.996	**0.996**
EP	Manual	1.936	147.6	285.8	0.995	0.995	**0.990**
MG	Baseline	1.946	148.3	288.5	1	1	**1**
MG	OpenMP	1.945	147.7	287.3	1.000	0.996	**0.996**
MG	Manual	1.936	147.6	285.8	0.995	0.995	**0.990**

5.8 PARSEC benchmark

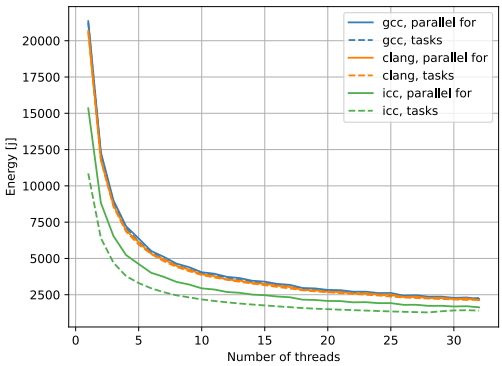
In this section we present the results from the two programs tested for the PARSEC benchmark suit, *blackscholes* and *fluidanimate*. The programs were tested using the *parallel for* and *tasking* implementations as well as with all available compilers and waiting policies. The parallel for versions were tested using the default *static* scheduling policy, unless otherwise stated.
In figure 5.22 the energy usage for the two programs can be seen while using the two parallelisation implementation. The results shown are obtained by taking the

geometric mean using the three different waiting policies. For blackscholes, figure 5.22a, the tasking version performs better on average for all three compilers. For both implementations, using the ICC compiler performed significantly better than their GCC and Clang counterparts. When taking the geometric mean of the energy saved by using tasks compared to the parallel for implementation over all threads, the tasking version for GCC uses 2.5 % less energy, Clang uses 2.0 % less and for ICC it is 27 %. For this program some additional scheduling policies were tested, specifically the *guided* setting, which resulted in *parallel for* being at best roughly 5 % better than their tasking version while using GCC and Clang. For ICC the *tasking* version was, however, consistently the best. For fluidanimate, figure 5.22b, the opposite relation is true where instead *parallel for* is the best option. The energy saved for the three compilers is between 10 - 15 % compared to their tasking versions. Overall this program has some unusual performance characteristics. From one to four threads the performance improves linearly, then little improvement is seen up until eight threads, where the performance improves substantially. After eight threads the programs execution time stays the same, leading the energy used to increase when more threads are added.

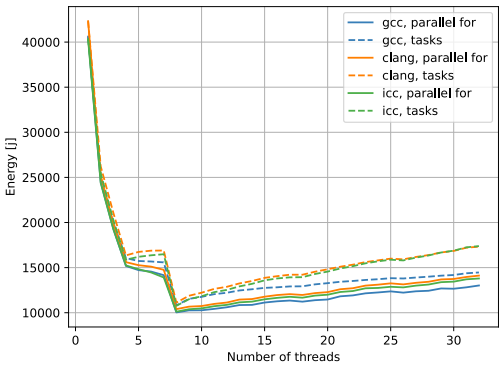
To figure out why ICC performed so much better than the two other compilers for the blackscholes program, further analysis was conducted using the perf tool. There we noticed that GCC and Clang spent the majority of their execution time in the `exp` function, which computes the Euler number raised to some given input, included in the `math.h` library, something the ICC didn't spend nearly the same amount of time in. The `exp` function was therefore tested in isolation for the different compilers. The results from these test was that the ICC implementation of the function was 4x faster than for the two other compilers. This finding explains why ICC had the best performance for the blackscholes program.

In figure 5.23 the energy usage for the two programs can be seen while using the three waiting policies. The results shown are obtained by taking the geometric mean using the two parallelisation implementation. For blackscholes, figure 5.23a, the default option performs best for all three compilers, but the difference compared to the other waiting policies are very slight, only 1 - 2 %. An outlier is the active option for ICC, where from 28 to 32 threads it gets noticeable slower. For fluidanimate, figure 5.23b, what waiting policy performed best heavily depends on how many threads was used. For eight threads, which was the thread count that overall used the least amount of energy, the active and default setting performed the best, with passive being 5 - 10 % worse depending on the compiler. As more threads are used this trend reverses and the active and default setting gets significantly worse while the passive just increasing slightly in energy. For 32 threads the passive setting uses between 10 - 20 % less energy compared to the active setting for the different compilers.

What the results from two PARSEC benchmarks have demonstrated is that there is not one size fit all when it comes to optimising for energy usage for parallel programs. The two programs tested shared no parameter when it came to their overall best configurations. For blackscholes the overall best configuration was with active waiting policiy, tasking implementation and ICC as the compiler, while for fluidanimate the best configuration was instead passive waiting policy, parallel for implementation and GCC as the compiler.

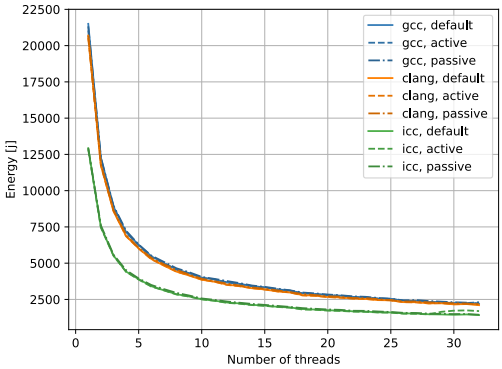


(a) Blackscholes.

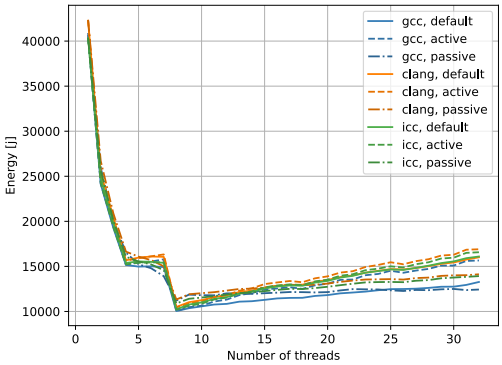


(b) Fluidanimate.

Figure 5.22: Energy used for the two PARSEC programs, for the *parallel for* and *tasking* implementations, using the three different compilers.



(a) Blackscholes.



(b) Fluidanimate

Figure 5.23: Energy used for the two PARSEC programs, for the three different waiting policies, using the three different compilers.

5.9 Summary in context of research questions

The results presented thus far have been in the context of each program. This section aims to summarise the results and draw possible conclusions in terms of each research question. The final question, which concerns novel directives for energy consumption, is covered on its own in chapter 6.

RQ1: How much impact do code transformations such the recently introduced unrolling and tiling have on energy consumption?

This question has been evaluated using our own matrix multiplication (both naive and reordered) and 2D stencil programs, as well as our efforts of applying the transformations to the common benchmark suites NPB and BOTS. Unrolling specifically has also been evaluated using the unrolling microbenchmark.
We have seen that tiling can provide a significant energy reduction (50-60 %) in the naive matrix multiplication, with similar results for the 2D stencil. However, we have also seen that tiling with OpenMP has consistently provided worse results than manual unrolling. In the benchmark suites, we have not managed to apply tiling with much success due to the strict requirements for doing so. We have also seen how tiling can drastically deteriorate performance in the reordered matrix multiplication. In general, we have also seen that the effectiveness of tiling slightly decreases with higher numbers of threads.
To conclude, loop tiling can potentially provide huge improvements to both execution time and power, and in turn energy consumption. However, it is often not applicable to real programmes, because it requires nested loops where memory is accessed in a certain way. The programmer can not blindly apply tiling and expect their programs to be faster, but must instead empirically profile their program with and without tiling to not risk a major performance degradation, likely due to preventing other optimisations.
The effects of explicit unrolling, either through the OpenMP directive or manually modifying the code, differs heavily between programs, but overall it tends to have either an insignificant or negative impact when done naively. This can be seen for the unrolling results for the Alignment program in BOTS, the four benchmarks tested in NAS, or the `SIMPLE_COMP_DEPEND` program in the unrolling microbenchmark. In some rare cases, performance can also significantly degrade, like for the `SIMPLE_MEM` program, where the unrolling disturbed automatic optimisations by the compiler, increasing total energy usage by 70 %. Lastly there were some programs that had an noticeable positive effect on overall performance, like the 2D stencil benchmark, with an improvement of about 5 %.
An area which showed more promising results was manual unrolling where some additional optimisation had been applied, especially if that optimisation exposes some opportunity for ILP by the compiler. Examples of this can be seen for the Strassen benchmark from BOTS, or the `COMPLEX_NESTED` in the unrolling microbenchmark. In those cases *loop jamming* was used, which groups the logic from several loops into one, making it so SIMD instructions could be utilised, significantly reducing execution time and energy usage by 45 % and 80 % respectively.

In conclusion, it is best to let the compiler handle unrolling optimisations except in cases where the programmer can expose opportunities for ILP to the compiler by manually unrolling, like for the *loop jamming* optimisation. Naively adding unrolling has in many cases at best a very small positive impact, and at worst a big performance downgrade, making it so the safest option is to not unroll at all.

RQ2: How do the runtime parallelism constructs of OpenMP (tasks and parallel loops) compare in terms of energy?

This has been examined using our parallel construct microbenchmarks at a fine-grained level. Single- and multithreaded task generation has been analysed using the BOTS programs where both methods are available. Parallel loops versus tasking has also been tested using PARSEC.
We first consider the different variants of tasking. From the microbenchmark we have seen that single-threaded task generation is ineffective for very small tasks (tens of microseconds), but that the difference becomes negligible for larger task granularities. This is confirmed by the BOTS programs, where we see that single-threaded task generation is about 1.5 % faster and consumes 1.6 % less energy. We can conclude that in most cases, there is no major difference between the task generation methods. Single-threaded is, however, easier to implement most of the time and should be preferred for tasks of granularities beyond tens of microseconds.
In the parallel constructs microbenchmark we have compared the best version of tasking versus parallel loops. Here, we conclude that parallel for loops provide the best energy reduction given that the correct scheduling is used. In the case where parallel loops use static scheduling, task generation techniques can provide better results due to uneven work sizes. However, we have shown that parallel loops will perform better than tasking in that case given that dynamic scheduling is used.
For the PARSEC benchmark suite, while for one program the most energy efficient implementation ultimately used tasking, in general when looking at the performance across all three compilers, the parallel for was on average better when using an appropriate scheduling policy.
We conclude that in cases where both parallel loops and tasking can be used, parallel loops should be preferred.

RQ3: How does the waiting policy of OpenMP affect power and energy?

This is perhaps the aspect we have seen the biggest improvements from using the different configurations, and has been examined at a fine level using our inactivity microbenchmark and on a coarse level using the BOTS, NPB and PARSEC benchmark suites, as well as the parallel constructs microbenchmark to a degree.
From the inactivity benchmark we have seen that active waiting is better than passive waiting for very small task granularities (tens of microseconds) while passive waiting becomes better for larger tasks. Threads must be inactive for fairly long periods of time (30-40 microseconds for Clang and ICC, and around 60-70 microseconds for GCC) for passive waiting to yield lower energy consumption. Furthermore, in the parallel constructs microbenchmark, we have seen active waiting outperform passive waiting as long as threads have a reasonable amount of work to perform. In

situations where this is not the case, such as with single-threaded task generation with more threads than can be utilized, energy is saved by using passive waiting. Passive waiting also tends to be favored with parallel for loops, especially when using GCC.
In a more realistic context, represented by BOTS and NPB, we have seen that active waiting is favorable in general, as this consumes the least energy and has the best execution times. This is however only true for Clang and ICC, as GCC performs the best with passive waiting in this case due to active and default waiting scaling poorly with many threads. From PARSEC we have seen that active waiting is preferred when using tasking while passive waiting is preferred with parallel for loops, which confirms the results from the parallel constructs microbenchmark.
To conclude, the OpenMP waiting policy often has a major impact on both power and energy in programs where threads are blocked by each other. Passive waiting generally provides lower power at the cost of higher execution time and active waiting does the opposite. The used compiler has also a major impact. The choice of waiting policy should be a major configuration parameter for the programmer or data center system administrator.

RQ4: How do different implementations of OpenMP impact the research questions above?

In our analysis we have used three different compilers which includes its own unique implementation of OpenMP. The differences we have observed between them is a results of both in the standalone compiler and their respective OpenMP implementation. In the following discussion we won't make any attempts to isolate the contribution from the specific OpenMP implementation but instead compare them as a whole.
For GCC and Clang, the results for loop transformations have been very similar. Where a transformation has been beneficial for one, it has been for the other as well and vice versa. ICC has however shown more varied results, which has been the most prominent for matrix multiplication. Here, where loop tiling has been beneficial for GCC and Clang, it has been a major detriment to ICC. This is due to ICC performing loop reordering by itself, which it fails to do if tiling is applied. We have also seen ICC benefit the most from assuming that the loop iterations is divisible by tile size or unrolling factor.
For the 2D stencil program, we have seen that both tiling and unrolling consistently provide some decreased energy consumption. ICC has however seen the most benefit, followed by GCC, and then Clang. Also note that for both matrix multiplication and the 2D stencil, the best results have been from some version with ICC.
Next is the question of waiting policies and parallel constructs. In BOTS, we have seen that ICC and Clang have fairly similar results regardless of waiting policy, but that clang performs somewhat better for high numbers of threads. GCC however, show very poor results, only scaling well up to about 10 threads before becoming worse with more threads regardless of waiting policy, and also performing the best with passive waiting. Strangely, it simultaneously performs the best with low numbers of threads. We draw the conclusion that GCC (with passive waiting) should be used with low numbers of threads to optimise for power, while ICC or Clang with

active waiting should be used in all other cases.
For NPB, while there is significantly less variance in the results, about the same conclusions can be drawn. The main difference here is that GCC performs much better, about equal to Clang. ICC outperforms them both by far at any numbers of threads.

6

Directives and Programmer Recommendations

In this chapter we further interpret the results and suggest how to apply the findings. Some findings are more coarse-grained and general and, therefore, fit better as a recommendation for the programmer, while others are more suitable to be applied in the form of OpenMP directives. We begin with the general programmer recommendations, found in section 6.1. The purpose of this section is not to present new data or conclusions, but to concretise the findings for the common programmer. After that, section 6.2, we discuss how a descriptive extension to the unrolling and tiling directives could be added to alleviate compiler optimisations and estimate the effects of such an extension. Finally, section 6.3 covers how unrolling can be used to remove dependencies between variables leading to faster reduction operations.

6.1 Programmer recommendations

In the following list, we cover broad programmer recommendations aimed at increasing the energy efficiency for parallel programs. These recommendations have been created by condensing the key finding of our results into easily applicable guidelines that do not require deep understanding of the inner workings of OpenMP by the programmer. We list the recommendations roughly in the order of greatest to lowest significance in the context of reducing energy consumption.

- **Compiler choice** has shown to be one of the main factors in reducing energy consumption. For single-threaded programs, GCC tends to perform the best, while ICC and Clang tend to perform better with many threads. GCC should be avoided in tasking-heavy programs as it tends to scale poorly. Using the active waiting policy tends to be the best choice when using Clang or ICC due to shortening the execution time, while GCC often performs the best with passive waiting due to a lower power consumption.
- The **OpenMP waiting policy** has also a major impact on energy consumption. Using *active* waiting tends to be the best as it shortens execution time. The waiting policy is especially important in tasking-heavy programs.
- In cases where **tasking and parallel for loops** can be used, they tend to perform about the same when the granularity of loop iterations are larger than about 100 microseconds. This is the case in most programs. Smaller than that, we have seen that parallel for loops tend to perform best in terms of energy consumption as well as execution time as long as the correct scheduling policy

is used.

- When applying parallelism to a loop using tasking, it can be done in a **single-threaded or multi-threaded task generation** manner. For realistic workloads, there is almost no difference. Only when the tasks are very small, in the order of tens of microseconds, single-threaded task generation should be avoided as it throttles parallelism.
- **Loop tiling** is a complex subject. On the one hand, it can improve performance considerably in programs such as matrix multiplication and stencil applications, which act on multidimensional data. In these programs, we have seen energy reductions of the scale 40-50 %, mainly due to shortened execution time. On the other hand, it can be greatly detrimental to performance if it causes other optimisations not to be applied by the compiler, if these would otherwise be applied if tiling was not used. In these cases, we have seen loop tiling increase energy consumption by a factor 2-3×. It should also be mentioned that loop tiling is very situational due to being limited to nested loops, and often nontrivial to apply to existing programs.
- **Loop unrolling** is equally complex, but for other reasons. Explicit loop unrolling by the programmer should be considered only in very specific cases. Altering source code and applying unrolling naively will in the best case have a very minor benefit (1-2 % energy reduction), and in the worst case worsen performance drastically (2-3X energy consumption) due to preventing other optimisations. It does have a potential use if it can be combined with additional clever optimisations and allow for techniques such as auto-vectorisation, in which case it can be highly beneficial. However, as this benefit is not due to the unrolling itself as well as being a highly advanced optimisation technique, we conclude that the common programmer should almost always just leave unrolling to the compiler.
- In general, when minimising energy consumption on a homogeneous platform such as the one used in this project, *minimising energy consumption is most easily done by minimising execution time rather than power consumption.*

As a final note, we emphasise that there are always exceptions to these guidelines. When applying an optimisation, it should be empirically evaluated to see if the program performs better or worse with the optimisation compared to before. This is especially important for tiling and unrolling, which we have seen both drastically improve and impair performance. OpenMP-applied tiling and unrolling can be helpful in this regard, as it requires less effort from the programmer to apply than doing so manually.

6.2 Loop transformation length check clauses

Throughout our testing we have used two variants of manual tiling and unrolling: one where the tile size or unrolling factor is assumed to evenly divide the number of loop iterations, and one where this is not assumed. In the case where passing this additional information would make a significant difference, it would make for a good optional modifier to the *tile* and *unroll* directives, exemplified by the listings 6.1 and 6.2. The potential gain from such an extension would be that overhead logic

from uneven tile sizes and unroll factors could be avoided.

Listing 6.1: Possible extension to the unrolling directive.

```
int i;
int n[N];
#pragma omp unroll partial(5) nocheck
for(i = 0; i < N; i++)
{
  n[i] = 10 * i
}
```

Listing 6.2: Possible extension to the tiling directive.

```
#pragma omp tile sizes(8, 8, 8) nocheck
for (int row = 0; row < N; row++) {
  for (int col = 0; col < N; col++) {
    for (int k = 0; k < N; k++) {
      C[row*N + col] += A[row*N + k] * B[k*N + col];
    }
  }
}
```

To see how much effect this could have in practise, we first consider the data gathered from applying tiling to matrix multiplication, which is best summarised in table 5.2. By comparing the tiling versions (Manual v1 and v2), we can estimate the improvement of this optimisation. For Clang, we see that the possible energy reduction is $1 - \frac{22.47}{29.66} \approx 0.24 = 24\%$, which is a fairly significant amount. For GCC, the improvement is even higher at 65 %! Such high numbers are, however, unlikely to be just due to this optimisation, but rather because the logic is simplified for the compiler, which then can optimise better. The corresponding numbers for the 2D stencil are much more conservative, with only a 2.4 % improvement for Clang and no difference for GCC. This is more in line with what is expected from such a small change. For unrolling, we see that the change would have next to no effect in practice, which is confirmed by the unrolling microbenchmark.
A compiler could choose to generate two versions of an unrolled or tiled loop, one where the property is assumed, and one where it is not.

6.3 Loop unrolling reduction clause

In OpenMP there is a clause called `reduction` which can be used together with the `parallel`, `for` or `sections` directives. It specifies that at the end of that parallel region, a specified variable should be subject to a reduction using some specified operation. The clause works by first making a private copy of the specified variable for each thread, then the threads do their calculations saving the results to their own local copy. Lastly, at the end of the parallel region, all the private copies are reduced using the specified operation, such as addition.
What we propose is to extend the OpenMP unrolling directive so that it can also

use the reduction clause, but instead of a tool to allow for TLP, it enables ILP. An example of how the directive would work can be seen for the program `SIMPLE_COMP`, with the modified implementation in listing 3.11, and the unmodified version found in listing 3.8. In this example the specified directive would look like this: `#pragma omp unroll partial(2) reduction(sum:+)`. The directive splits up the unrolling so several independent variables are used for the calculations, then does a reduction on these variables after the loop.
The goal of adding this clause is to allow the compiler to optimise for the use of vector instructions, which can significantly improve performance. When separating the calculation in this way into several independent parts, dependencies that would otherwise prevent the use of vectorisation are removed. For the previously mentioned `SIMPLE_COMP` program, execution time and energy were reduced by about 80 % using this method.
To provide some additional examples of situations where the directive is, and is not useful, another program was tested using two slightly different versions. The program calculates the sum of all elements in an array. The first version took the sum of all the elements in ascending order, while the second version accessed the array based on another array that contained all the indices shuffled in random order. The two versions were specifically chosen with the aim that the first version would be favourable for vectorisation and the second not. The code for both versions, with both the original code and with the proposed directive applied, can be seen in the listings A.1, A.2, A.3 and A.4. As expected, the first version had a significant performance improvement, reducing energy usage by 85 %, while the second version saw no improvement. Both of these results were achieved using an unrolling factor of eight, instead of the factor two shown in the listings. These results support our claim that the proposed reduction unroll clause is only effective when vectorisation is possible and give no performance improvements otherwise.

7
Conclusion

In this report, we analysed aspects of OpenMP from an energy consumption perspective. This is accomplished by executing novel microbenchmarks and common benchmark suites on HPC cluster nodes and measuring the energy consumption. Three main aspects are analysed: loop tiling and unrolling, methods for generating parallelism, and the policy of handling blocked threads. For the first aspect, we find that tiling can yield significant energy savings for some, mostly unoptimised programs, while directive-generated unrolling yields no improvements in practice. For the second aspect, we find that parallel for loops in general yield better results than explicit tasking loops in cases both can be used, but that the differences are very minor for everything but the smallest task sizes. For the third, we find that significant energy savings can be made by not descheduling waiting threads but instead have them spin, at the cost of a higher power consumption, and that this holds even for realistic workloads. We also analyse how the choice of compiler affects the above questions by compiling programs with each of *ICC*, *Clang* and *GCC*, and find that neither is strictly better than the others.

7.1 Future Work

Our finding indicate that the waiting policy of OpenMP has a major impact on execution time, power and energy. In general, an *active* waiting policy increases the power consumption but lowers execution time for realistic workloads, with a *passive* waiting policy doing the opposite. This could be used as a major parameter in the previous work OpenMPE [1], which was covered in section 2.3. In short, it introduces directives for optimising a combination of energy, power, performance and quality of service.
Enhancements can also be made to the experimental methodology. For example, the chosen method of measuring energy limits us to measuring average power only, and not peak power. Future work could consider performing energy measurements in regular intervals during execution to accommodate for this.
We also perform all empirical experiments on a single, homogeneous platform, which to an extent limit the validity of results to that platform only. Including other platforms, especially with heterogeneous architectures, would be highly interesting.

Bibliography

[1] Ferdinando Alessi et al. “Application-level Energy Awareness for OpenMP”. In: *International Workshop on OpenMP* 9342 (Oct. 2015). DOI: 10.1007/978-3-319-24595-9_16.

[2] S Anil Kumar. “Enhancing the Scope for Automated Code Generation and Parallelism by Optimizing Loops through Loop Unrolling”. In: *2020 Fourth International Conference on Inventive Systems and Control (ICISC)*. 2020 Fourth International Conference on Inventive Systems and Control (ICISC). Jan. 2020, pp. 790–795. DOI: 10.1109/ICISC47916.2020.9171081.

[3] R. Bianchini and R. Rajamony. “Power and energy management for server systems”. In: *Computer* 37.11 (Nov. 2004). Conference Name: Computer, pp. 68–76. ISSN: 1558-0814. DOI: 10.1109/MC.2004.217.

[4] Christian Bienia et al. “The PARSEC benchmark suite: characterization and architectural implications”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. PACT ’08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: https://doi.org/10.1145/1454115.1454128 (visited on 03/24/2022).

[5] Wesley Brewer et al. “Inference Benchmarking on HPC Systems”. In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. ISSN: 2643-1971. Sept. 2020, pp. 1–9. DOI: 10.1109/HPEC43674.2020.9286138.

[6] Howard David et al. “RAPL: memory power estimation and capping”. In: *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. ISLPED ’10. New York, NY, USA: Association for Computing Machinery, Aug. 2010, pp. 189–194. ISBN: 978-1-4503-0146-6. DOI: 10.1145/1840845.1840883. URL: https://doi.org/10.1145/1840845.1840883 (visited on 02/28/2022).

[7] Bhavishya Goel and Sally A. McKee. “A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors”. en. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Chicago, IL, USA: IEEE, May 2016, pp. 273–282. ISBN: 978-1-5090-2140-6. DOI: 10.1109/IPDPS.2016.118. URL: http://ieeexplore.ieee.org/document/7516023/ (visited on 01/27/2022).

[8] *Inseime compiler and runtime infrastructure.* http://insieme-compiler.org/. Accessed: 2022-05-12.

[9] Herbert Jordan et al. “A multi-objective auto-tuning framework for parallel codes”. In: *SC ’12: Proceedings of the International Conference on High*

Performance Computing, Networking, Storage and Analysis. ISSN: 2167-4337. Nov. 2012, pp. 1–12. DOI: 10.1109/SC.2012.7.

[10] Stefanos Kaxiras and Margaret Martonosi. "Computer Architecture Techniques for Power-Efficiency". In: *Synthesis Lectures on Computer Architecture* 3.1 (Jan. 2008). Publisher: Morgan & Claypool Publishers, pp. 1–207. ISSN: 1935-3235. DOI: 10.2200/S00119ED1V01Y200805CAC004. URL: https://www.morganclaypool.com/doi/abs/10.2200/S00119ED1V01Y200805CAC004 (visited on 01/18/2022).

[11] Marius Knaust, Florian Mayer, and Thomas Steinke. "OpenMP to FPGA Offloading Prototype Using OpenCL SDK". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 387–390. DOI: 10.1109/IPDPSW.2019.00072.

[12] "Chapter 12 - Toolchain Primer". In: *Power and Performance*. Ed. by Jim Kukunas. Boston: Morgan Kaufmann, 2015, pp. 207–239. ISBN: 978-0-12-800726-6. DOI: https://doi.org/10.1016/B978-0-12-800726-6.00012-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128007266000124.

[13] E.A. Lee. "The problem with threads". In: *Computer* 39.5 (May 2006). Conference Name: Computer, pp. 33–42. ISSN: 1558-0814. DOI: 10.1109/MC.2006.180.

[14] *libgomp Github page, file implementing scheduling policy.* https://github.com/gcc-mirror/gcc/blob/master/libgomp/loop.c. line 405 - 410.

[15] Arthur Francisco Lorenzon et al. "Aurora: Seamless Optimization of OpenMP Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 30.5 (May 2019). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1007–1021. ISSN: 1558-2183. DOI: 10.1109/TPDS.2018.2872992.

[16] Alexander Matthes et al. "Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library". In: *High Performance Computing*. Ed. by Julian M. Kunkel et al. Cham: Springer International Publishing, 2017, pp. 496–514. ISBN: 978-3-319-67630-2.

[17] Thiarles S. Medeiros et al. "Mitigating the processor aging through dynamic concurrency throttling". en. In: *Journal of Parallel and Distributed Computing* 156 (Oct. 2021), pp. 86–100. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2021.05.006. URL: https://www.sciencedirect.com/science/article/pii/S0743731521001118 (visited on 02/25/2022).

[18] Anilkumar Nandamuri et al. "Power and Energy Footprint of OpenMP Programs Using OpenMP Runtime API". In: *2014 Energy Efficient Supercomputing Workshop*. Nov. 2014, pp. 79–88. DOI: 10.1109/E2SC.2014.11.

[19] *NAS Parallel Benchmarks*. URL: https://www.nas.nasa.gov/software/npb.html (visited on 03/15/2022).

[20] Nwe Zin Oo and Panyayot Chaikan. "The Effect of Loop Unrolling in Energy Efficient Strassen's Algorithm on Shared Memory Architecture". In: *2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*. June 2021, pp. 1–4. DOI: 10.1109/ITC-CSCC52171.2021.9501472.

[21] *OpenMP 5.1 Specifications*. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf (visited on 03/11/2022).

[22] *OpenMP 5.2 Specifications*. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (visited on 03/16/2022).

[23] Zakaria Ournani et al. "Taming Energy Consumption Variations In Systems Benchmarking". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 36–47. ISBN: 978-1-4503-6991-6. DOI: 10.1145/3358960.3379142. URL: https://doi.org/10.1145/3358960.3379142 (visited on 02/09/2022).

[24] Thomas Rauber and Gudula Rünger. "Thread Programming". In: *Parallel Programming: for Multicore and Cluster Systems*. Ed. by Thomas Rauber and Gudula Rünger. Berlin, Heidelberg: Springer, 2013, pp. 287–386. ISBN: 978-3-642-37801-0. DOI: 10.1007/978-3-642-37801-0_6. URL: https://doi.org/10.1007/978-3-642-37801-0_6 (visited on 03/14/2022).

[25] Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras. *Power-Efficient Computer Architectures: Recent Advances*. en-US. 2008. URL: https://ieeexplore.ieee.org/document/7036193?arnumber=7036193 (visited on 01/18/2022).

[26] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs". In: *ACM SIGARCH Computer Architecture News* 36.1 (Mar. 2008), pp. 277–286. ISSN: 0163-5964. DOI: 10.1145/1353534.1346317. URL: https://doi.org/10.1145/1353534.1346317 (visited on 02/28/2022).

[27] Peter Zangerl. "Compiler and Runtime System Optimizations for the Insieme Compiler Infrastructure". en. In: (), p. 138.

A

Appendix

A.1 Matrix Multiplication

In figures A.1 and A.2 we see the energy consumption of the naive and reordered matrix multiplication, respectively. The results are similar to when using Clang, with the naive algorithm heavily benefiting from tiling and the reordered algorithm becomes significantly worse. Corresponding results for ICC are seen in figures A.3 and A.4.

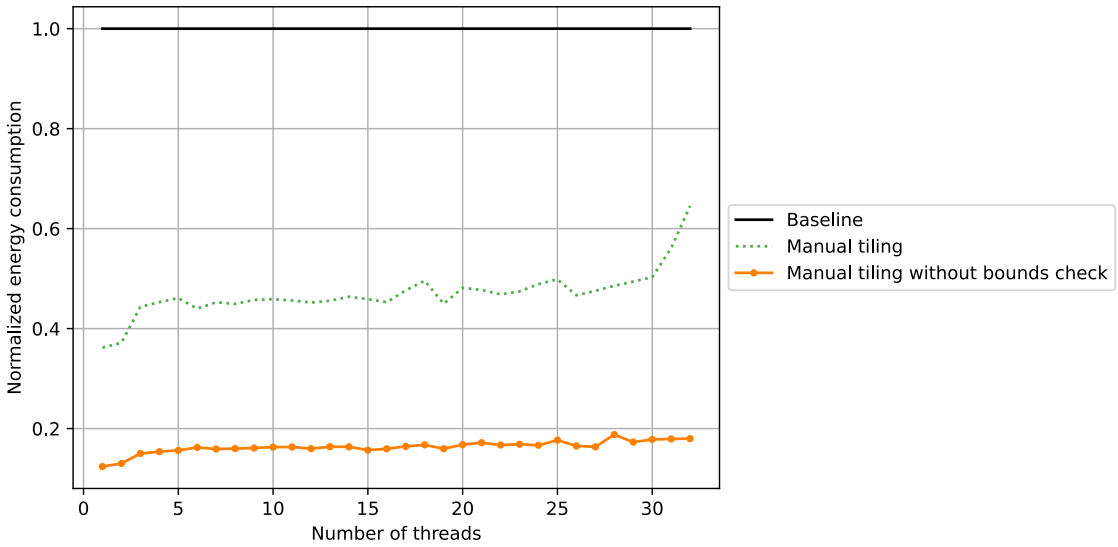


Figure A.1: Relative energy consumption for the naive matrix multiplication with tiling using GCC.

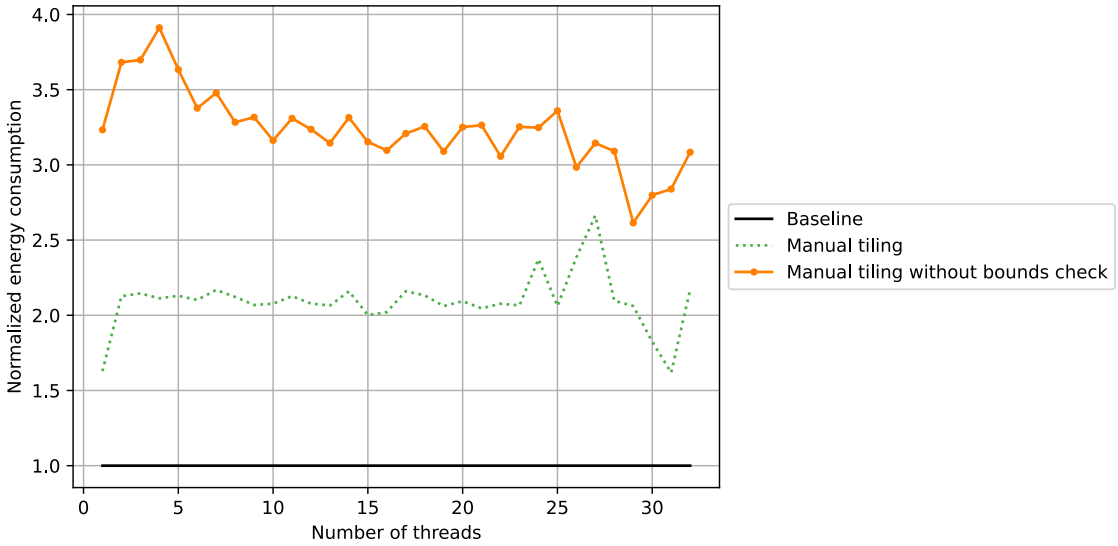


Figure A.2: Relative energy consumption for the reordered matrix multiplication with tiling using GCC.

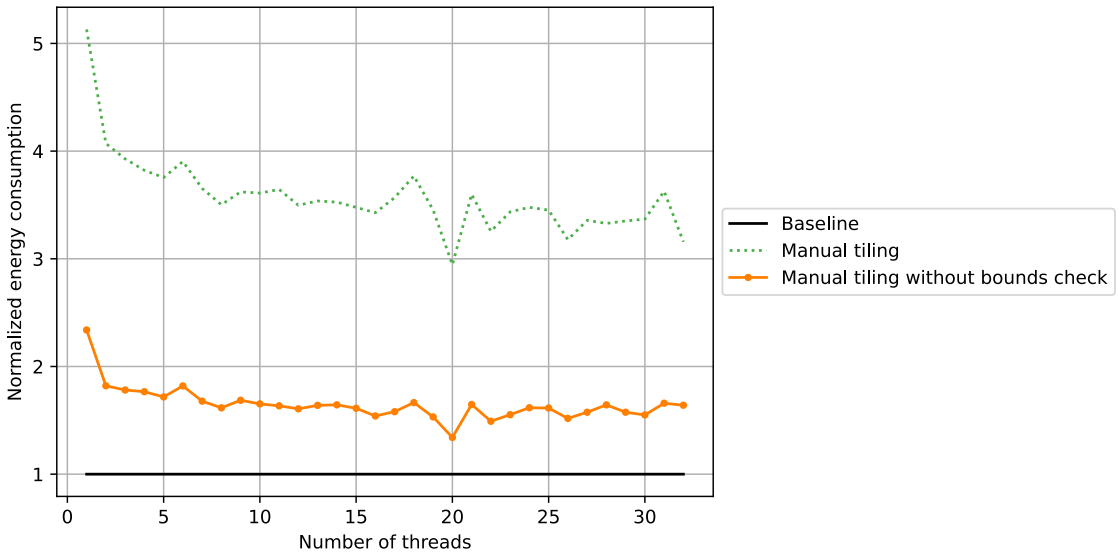


Figure A.3: Relative energy consumption for the naive matrix multiplication with tiling using ICC.

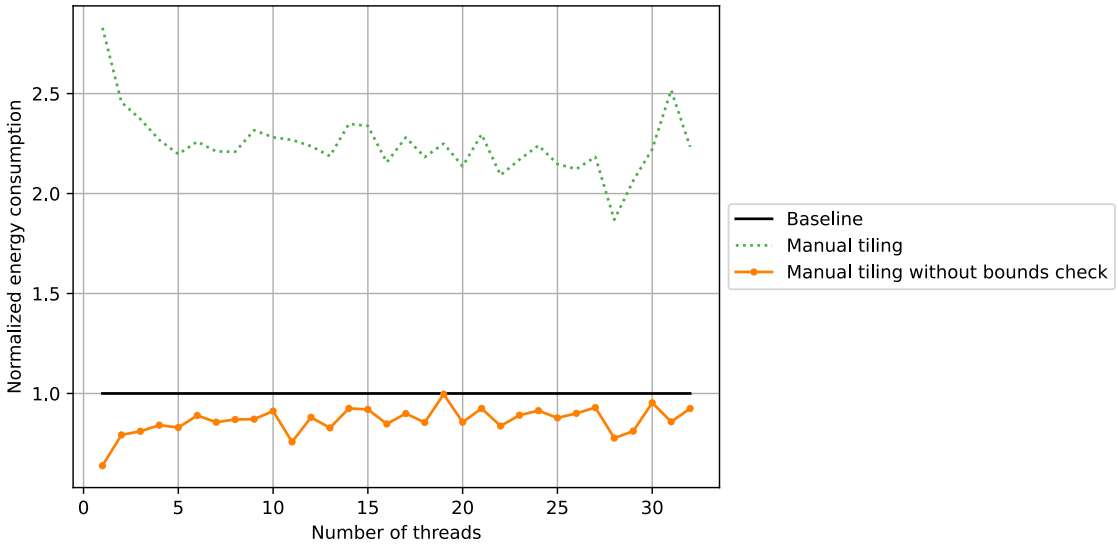


Figure A.4: Relative energy consumption for the reordered matrix multiplication with tiling using ICC.

Unrolling results for GCC are seen in figures A.5 and A.6. Corresponding results for ICC are seen in figures A.7 and A.8.

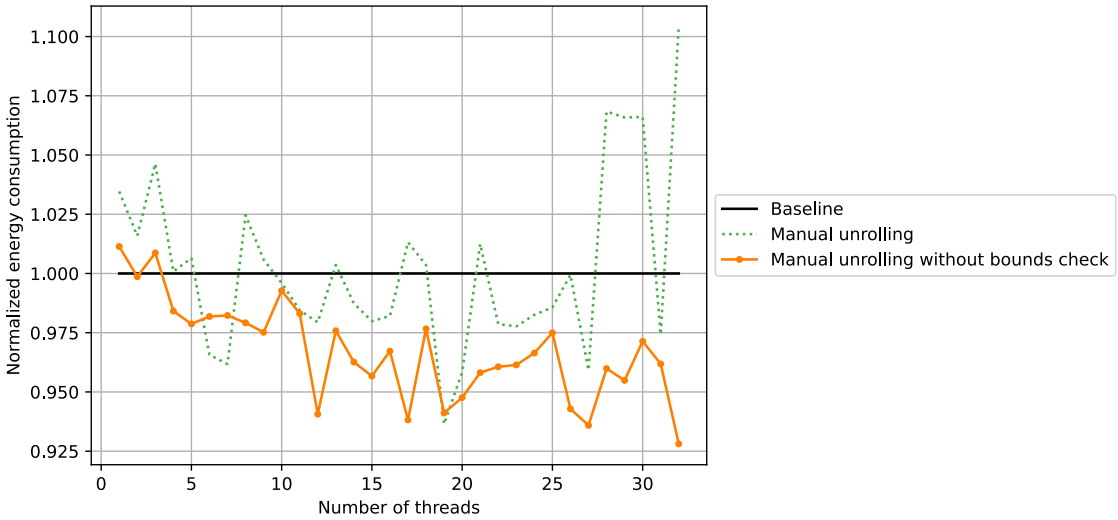


Figure A.5: Relative energy consumption for the naive matrix multiplication with unrolling using GCC.

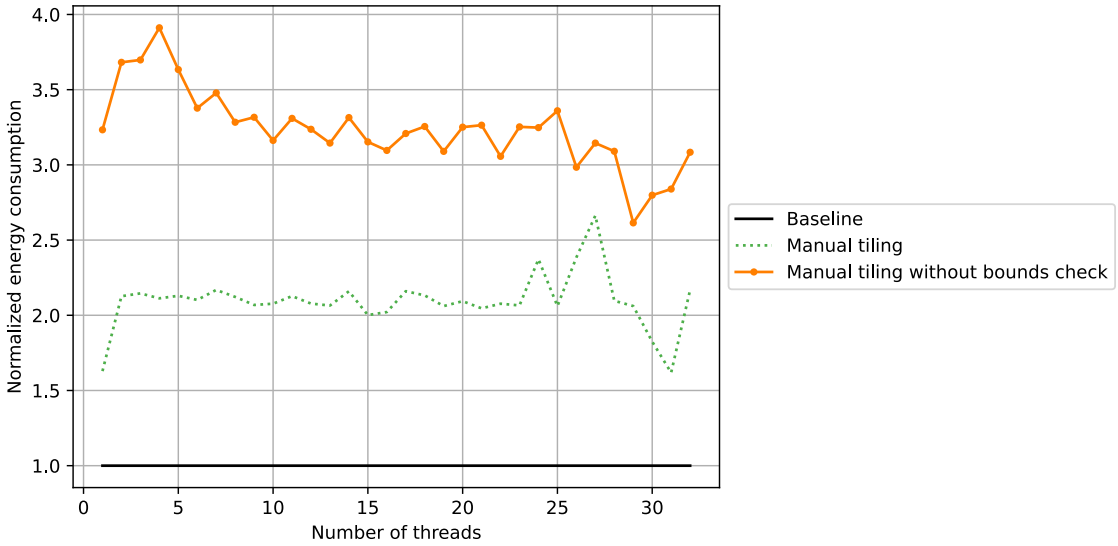


Figure A.6: Relative energy consumption for the reordered matrix multiplication with unrolling using GCC.

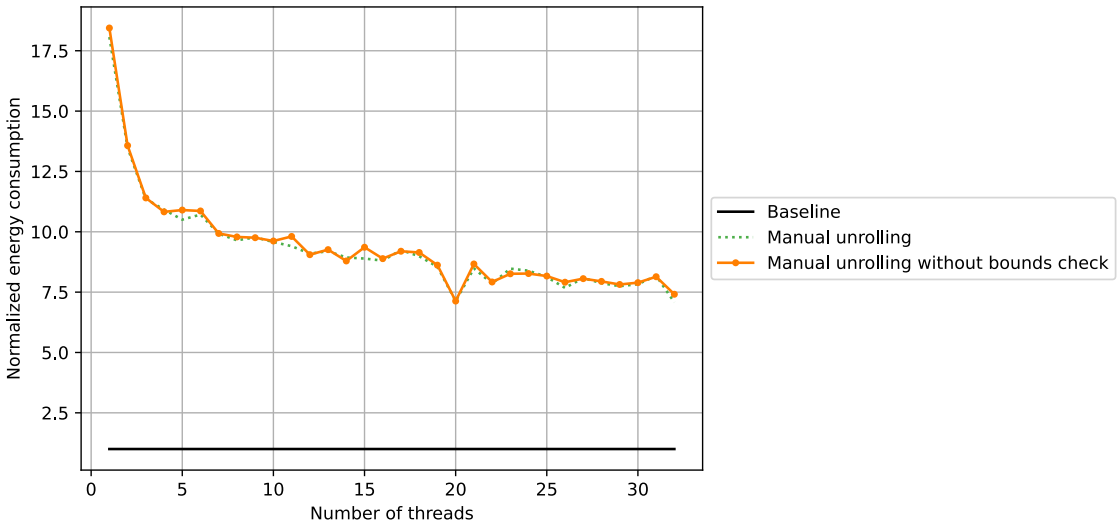


Figure A.7: Relative energy consumption for the naive matrix multiplication with unrolling using ICC.

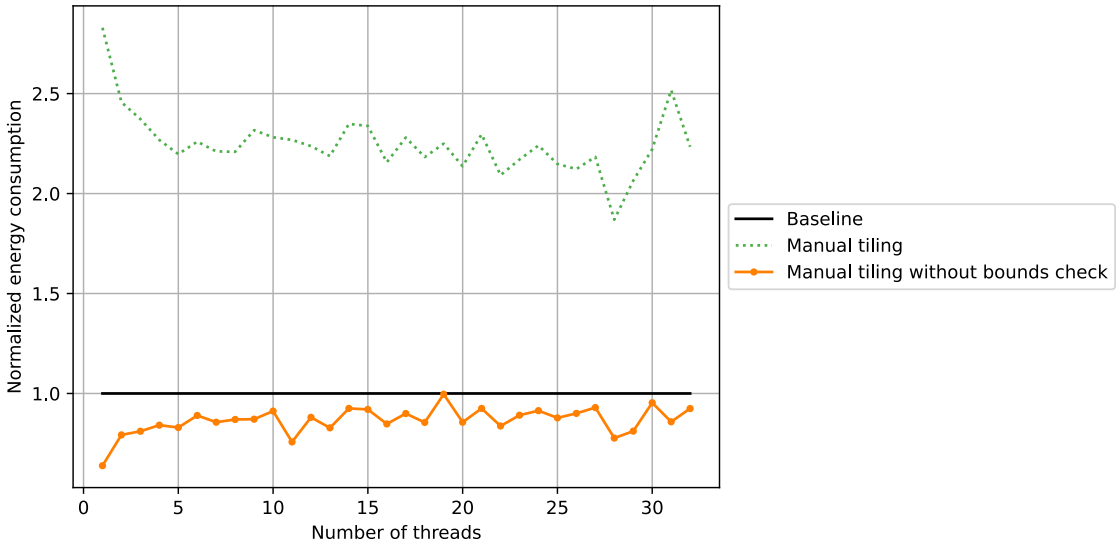


Figure A.8: Relative energy consumption for the reordered matrix multiplication with unrolling using ICC.

A.2 2D stencil

In figures A.9 and A.10 we observe the energy consumption results for applying loop tiling to the 2D stencil program, compiled with GCC and ICC respectively. Figure A.11 shows the results of unrolling for Clang.

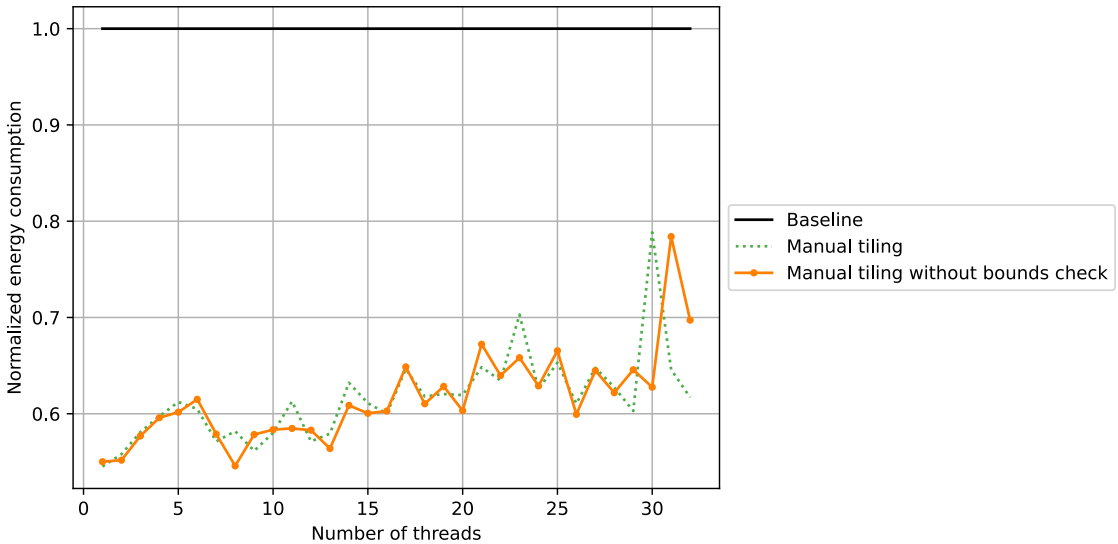


Figure A.9: Relative energy consumption from applying tiling to the 2D stencil program, compiled with GCC.

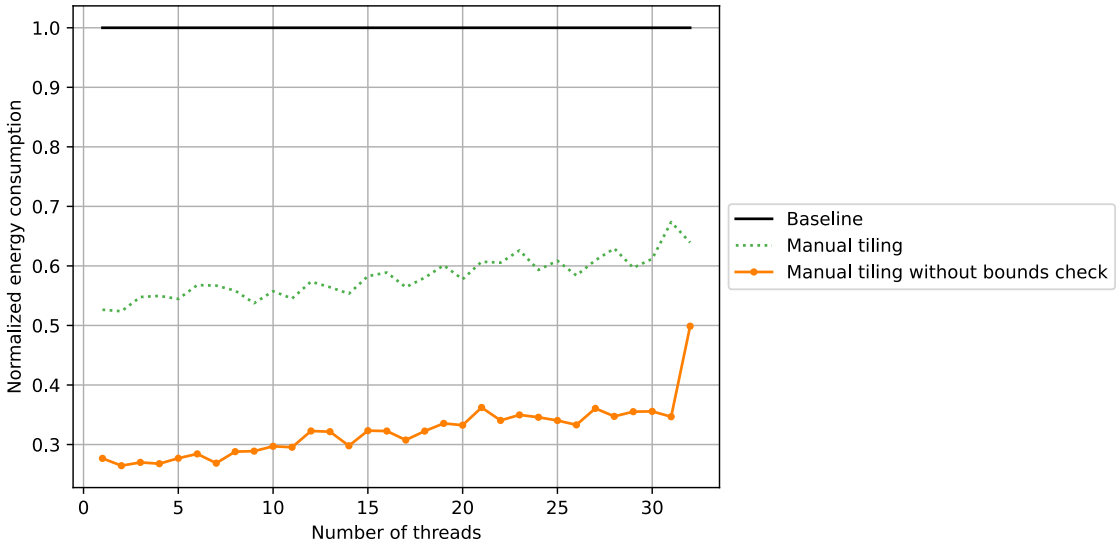


Figure A.10: Relative energy consumption from applying tiling to the 2D stencil program, compiled with ICC.

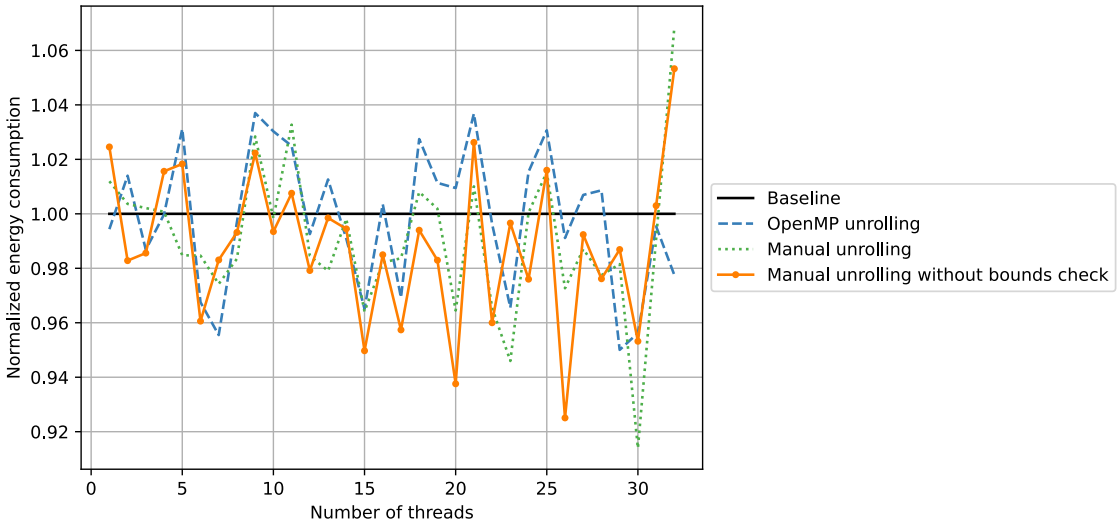


Figure A.11: Energy consumption for the 2D stencil program with different versions of unrolling, compiled with Clang.

A.3 Parallel constructs microbenchmark

In the main text only results for Clang and GCC is presented. Here, we present the corresponding data for ICC. Table A.1 shows the results for the different tasking variants: single-threaded task generation, multi-threaded task generation, and using the *taskloop* clause of OpenMP. Table A.2 shows results for parallel for loops versus the multi-threaded tasking generation.

Table A.1: Characteristics of the tasking methods for the parallelism constructs microbenchmark using ICC.

		Single-threaded			Multi-threaded			Taskloop		
Granularity	Waiting	T	P	**E**	T	P	**E**	T	P	**E**
20	Active	0.34	144	**49.39**	0.09	121	**11.34**	0.1	124	**11.99**
20	Passive	0.26	143	**37.5**	0.12	125	**14.71**	0.11	121	**13.17**
60	Active	0.12	131	**15.34**	0.09	119	**10.97**	0.09	113	**10.7**
60	Passive	0.11	121	**12.81**	0.1	121	**12.65**	0.1	115	**11.68**
100	Active	0.09	118	**11.06**	0.09	120	**11.13**	0.09	117	**11.14**
100	Passive	0.1	114	**11.43**	0.1	118	**11.9**	0.1	117	**11.82**
200	Active	0.09	113	**10.57**	0.09	120	**11.26**	0.1	121	**11.61**
200	Passive	0.1	111	**10.91**	0.1	118	**11.66**	0.1	117	**11.71**

Table A.2: Results for the parallel for versus tasking methods, with ICC. Measurements are reported as the geometric mean of said measurement for all numbers of threads.

		Tasking			ParFor, static			ParFor, dynamic		
Granularity	Waiting	T	P	**E**	T	P	**E**	T	P	**E**
20	Active	0.09	121	**11.34**	0.1	115	11.16	0.09	113	10.01
20	Passive	0.12	125	14.71	0.12	81	**9.79**	0.1	96	**9.82**
60	Active	0.09	119	**10.97**	0.1	116	11.41	0.09	109	9.8
60	Passive	0.1	121	12.65	0.11	83	**8.95**	0.09	96	**9.08**
100	Active	0.09	120	**11.13**	0.1	115	11.4	0.09	112	10.16
100	Passive	0.1	118	11.9	0.11	81	**9.05**	0.09	102	**9.53**
200	Active	0.09	120	**11.26**	0.1	114	11.43	0.09	110	10.08
200	Passive	0.1	118	11.66	0.11	84	**9.24**	0.09	99	**9.33**

A.4 Inactivity microbenchmark

In figure A.12 we see the energy consumption for ICC around the point where passive waiting overtakes active waiting in energy efficiency. It is very similar to the figure for Clang.

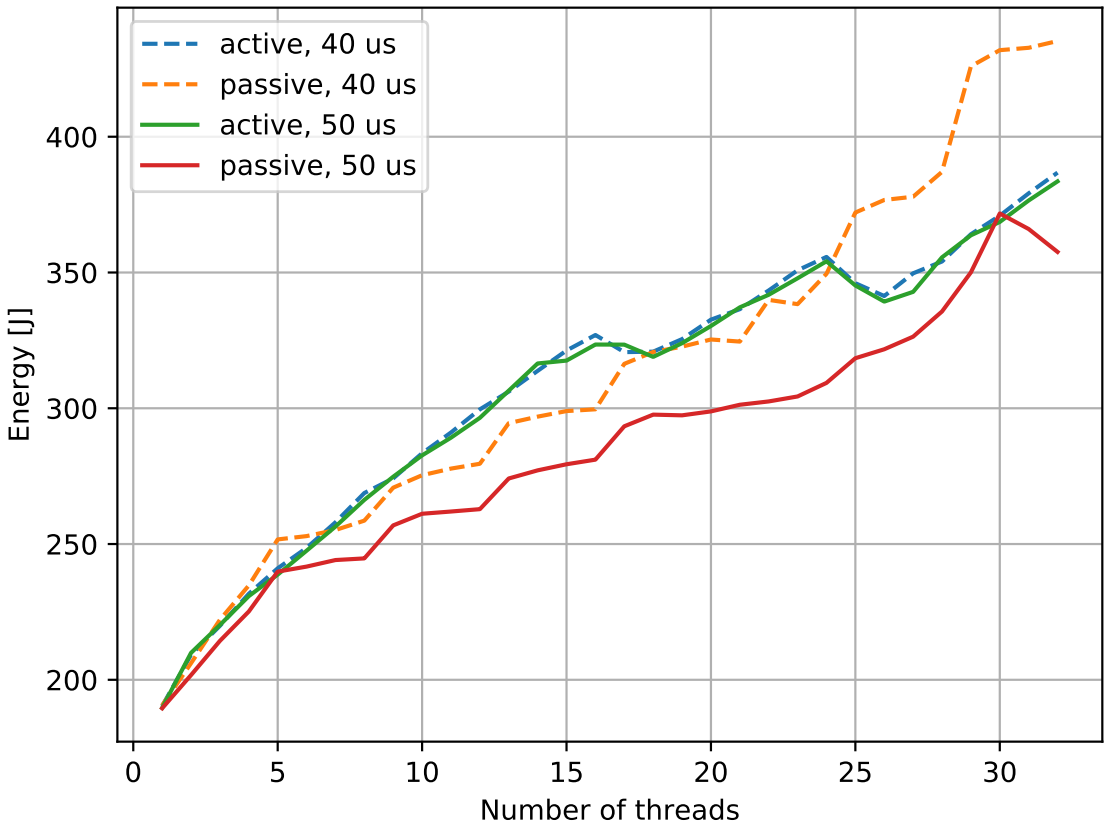


Figure A.12: Energy consumption for the inactivity microbenchmark with ICC.

A.5 Barcelona OpenMP Task Suite (BOTS)

Figures A.13 and A.14 show the execution time and power consumption of the BOTS benchmarks. The presented values are the geometric mean execution time and power of all programs.

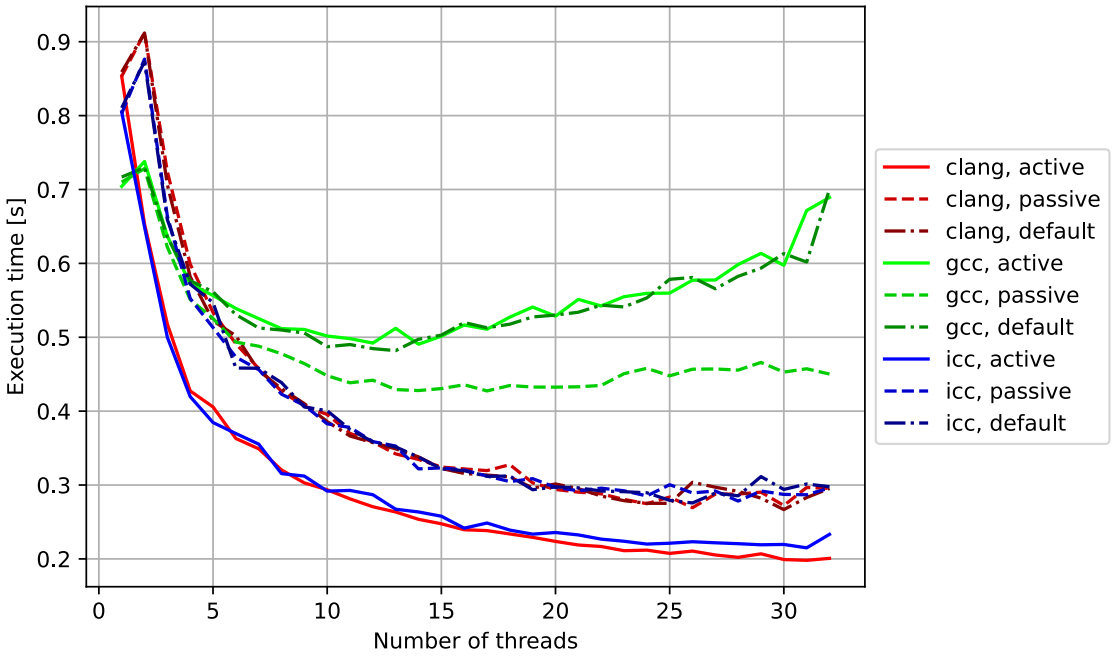


Figure A.13: Average energy consumption of all BOTS programs for the three compilers and waiting policies.

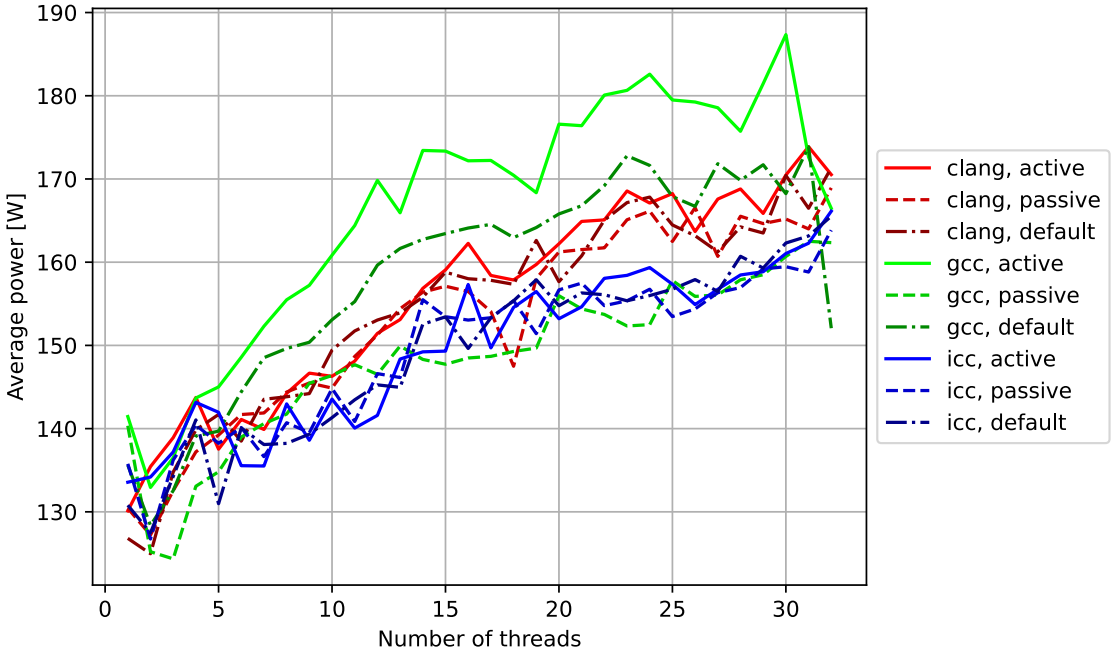


Figure A.14: Average energy consumption of all BOTS programs for the three compilers and waiting policies.

A.6 NAS Parallel Benchmarks (NPB)

Figure A.15 shows the execution times for different compilers and waiting policies for the NPB programs.

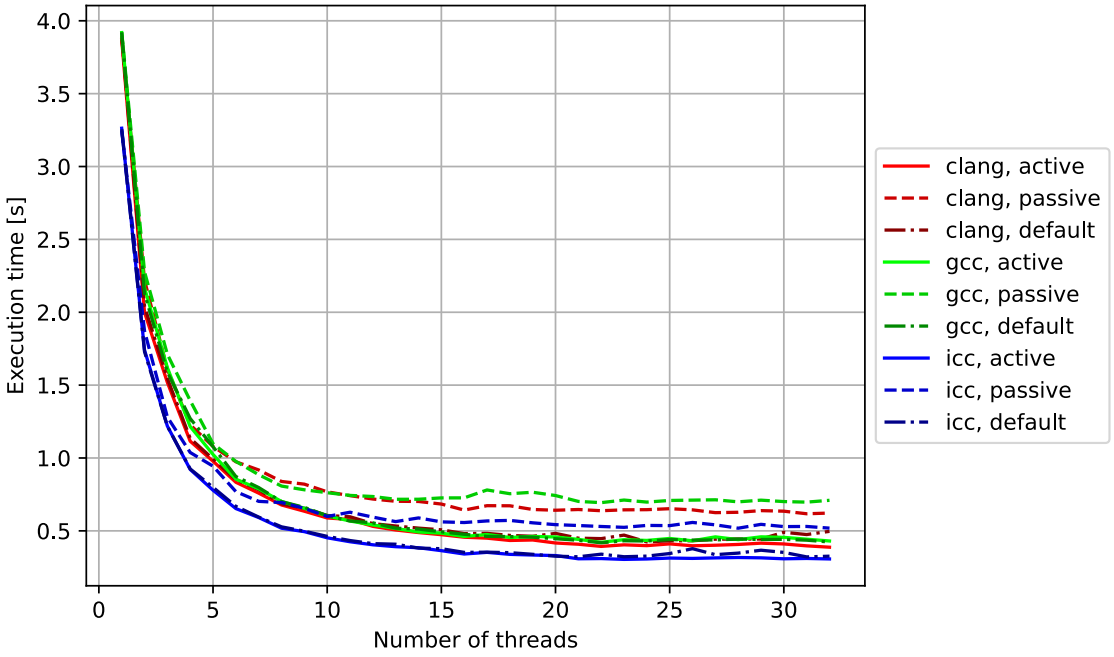


Figure A.15: Execution time of the NPB programs with different waiting policies.

A.7 Directives

In listings A.1, A.2, A.3, and A.4 the two different versions of the summation programs described in section 6.3 with their unmodified, and modified implementations are shown. The first version accesses the array in ascending order, while the second does so in a random order, with indexes precalculated and stored in the array `ind`.

Listing A.1: First version, unmodified implementation.

```
float sum = 0;
for(int i=0; i<len; i++)
  sum += A[i];
```

Listing A.2: First version, modified implementation.

```
float sum0 = 0, sum1 = 0;
for(int i=0; i<len; i+=2)
{
  sum0 += A[(i+0)];
  sum1 += A[(i+1)];
}
float sum = sum0+sum1;
```

Listing A.3: Second version, unmodified implementation.

```
float sum = 0;
for(int i=0; i<len; i++)
  sum += A[ind[i]];
```

Listing A.4: Second version, modified implementation.

```
float sum0 = 0, sum1 = 0;
for(int i=0; i<len; i+=2)
{
  sum0 += A[ind[(i+0)]];
  sum1 += A[ind[(i+1)]];
}
float sum = sum0+sum1;
```