Coarsening functional parallelism using intelligent search algorithms

Utrecht University

Oscar Leijendekker[1]

July 1, 2016

[1]Supervised by Ana-Lucia Varbanescu

Abstract

The increase in parallelism in modern-day computer architectures requires programs capable of exploiting that parallelism. With the goal of automating the process of creating parallel implementations, we want to extract implicit parallelism from a program. This can be done easily when dealing with functionally pure languages but doing so may generate a parallel implementation that spends more time on communication than it saves on computation. To this end, we present a method capable of iteratively reducing the amount of parallelism until a proper balance between computation and communication is reached. Our method may be able to find more useful implicit parallelism than simply selecting tasks based on their size, as has been done in previous work.

Acknowledgements

I want to thank my supervisor, Ana-Lucia Varbanescu for our inspiring and fruitful discussions and managing to keep my enthusiasm intact despite my initial underestimation of the problem. I'd also like to thank my examiners Gerard Vreeswijk and Atze Dijkstra, for putting up with my desire of writing a thesis steeped in a field outside both my and their own comfort zones.

Contents

Chapter 1

Introduction

Future microprocessors are expected to be increasingly parallel [1]. Moreover, while parallel processing power is still expected to increase, sequential execution times are not. This creates a need to write programs that can exploit this parallelism.

Unfortunately, writing such programs is a hard task for programmers and new methods must be developed to ensure the broad adaptation of parallelism, such as ensuring the correctness of parallel implementations and improving current programmer productivity [2].

A key part of writing parallel implementations is deciding which parts, or *tasks*, will run concurrently with each other. Such decisions are made based on two criteria. First, we must know which tasks can run concurrently based on how they depend on each other's outputs. Secondly, we must consider how parallelizing the tasks will affect the program's overall execution time.

Since communication between processing units takes some amount of time, the design of parallel programs requires balancing the time spent on computation with the time spent on communication. Using tasks with a *fine granularity*, i.e. tasks which take little time to compute, will increase the time spent on communication, whereas using tasks with a *coarse granularity* will reduce the amount of parallelism available and may therefore increase the time spent on computation.

Traditionally, methods have attempted to address this issue by restricting which tasks are allowed to incur a communication overhead. Only those tasks who's expected execution time is greater than some threshold value were allowed to be offloaded to other processing units. This has a major drawback in that opportunities for parallelism are only considered if they can be found in single large tasks.

We attempt to improve on such approached by defining a method capable of creating coarse-grained tasks from fine-grained one. This could be used to decompose a program into its smallest elements and assembling suitable tasks from them, thereby considering more situations than just singular big tasks.

Our research will therefore attempt to answer the following question: how can fine-grained tasks be merged intro coarser ones while respecting execution-order restrictions so that a balance is found between computation and communication.

The rest of this paper is structured as follows. In chapter 2 we will provide the background knowledge required in the other chapters. Chapter 3 contains the details of our method, explaining which tasks may be merged, when to stop merging and how to quantify how 'good' a merge is. We discuss how this method is implemented in chapter 4, where we also present a sample language used to decompose program into its smallest parts. A case study for a sample program is introduced in chapter 5. We discuss the strengths and weaknesses of our method in chapter 6 and provide ideas on how it could be extended. In chapter 7 we place our work in the context of previous research efforts. We conclude in chapter 8.

Chapter 2

Background

In this chapter, we introduce the background information required to fully understand the rest of this paper.

Section 2.1 elaborates on why parallelism is required.

Section 2.2 explains the distinction between data parallelism and task parallelism.

Section 2.3 describes some existing parallel architectures and what their strengths and limitations are.

Section 2.4 explains what data races are and some ways to avoid them.

Section 2.5 elaborates on the importance of having tasks with the right granularity

Section 2.6 briefly mentions what schedulers are and what types of schedulers exist.

Section 2.7 describes some methods that can be used to estimate the execution time of a task.

2.1 The need for parallelism

In the past, speed improvements in program execution times could be achieved by simply running it on a faster machine. An increase in instruction throughput was achieved through higher clock speeds and architecture design improvements that detected and exploited parallelism at instruction level. This trend ensured programs were expected to run faster on future machines without having to rewrite them. In recent years, this increase in execution speed has been stalled.

Increasing the clock frequency requires more electrical power. Furthermore, the energy requirement increases faster than the clock frequency. This is due, among other things, to 'leak' currents. These are small leaks of electricity that become larger as more power is put on a microprocessor. Along with the obvious cost of using additional power comes the increased cooling requirement as microprocessors essentially convert electric energy into heat. This situation leads to a practical maximum for clock frequency which is often called the *power wall*.

Another problem with increasing processor frequency lies in relatively slow memory. Although small memory, e.g. registers, exist that can match a processor's speed, the larger main memories have lacked behind. Not only is its frequency generally much lower than the processor's, communicating with main memory also takes a relatively long time. Although this issue is partially addressed through the use of caches, these caches are subjected to the same physical limitations as main memory; fetching data from them is relatively slow and it will only become slower as caches increase in size. As a task must wait for data to be loaded, a processor which can perform no other operations in the meantime will stall, limiting it's instruction throughput. This limitation is called the *memory wall* and its impact is further elaborated in [3].

Although some tricks have been used to improve sequential execution times, there is a limit to how far we can take them. Scalar processors, which pipeline instructions, are now commonplace. They allow for multiple subsequent instructions to be processed at every clock tick, each in a different stage in the pipeline. This approach requires us to fill the pipeline and as such, to predict what instructions will be executed in the future, before the current one has been fully computed. The larger the pipeline, i.e. the more parallelism, the more instructions we need to predict. Superscalar processors will try to execute multiple instructions concurrently in separate pipelines if they detect that this is possible, i.e., that instructions don't rely on each other's results. Such systems can only analyse a program at instruction level, after it has been compiled. This means that high-level parallelism between instructions that are farther apart is unknown to them, which limits their usability. Superscalar processors suffer the same prediction problems as scalar processors as they too require us to know what instructions will be executed in the future, so that we may compute them now instead.

Besides the limitations in their use, neither of these approaches helps us deal with the memory wall.

So as past strategies reach their limits, further speed improvements must be found elsewhere. Specifically, in explicit parallel programs. If we have many tasks that can execute concurrently we could process multiple of them at the same time instead of processing each one faster, reducing the need for a higher clock frequency. Furthermore, while one task waits for data from memory, we can process another one. For this reason, future microprocessors are expected to be explicitly parallel [1], requiring program implementations capable of exploiting this parallelism. Writing such programs, however, is a difficult task requiring new approaches aimed at increasing programmer productivity while ensuring the results' correctness [2].

2.2 Data parallelism and task parallelism

There are, broadly speaking, two types of parallelism: *data parallelism* and *task parallelism*.

Data parallelism occurs when the same operation is applied to multiple data-points. For example, when rotating a geometry made of triangles, every point, i.e. vertex, needs to be multiplied with a matrix, the multiplications are not interdependent so they may be executed out of order and consequently, in parallel.

Figure 2.1 is a graphic representation of a data parallel operation; we apply the same function, $\lambda x \to f(x)$, to every data-point to get our result.

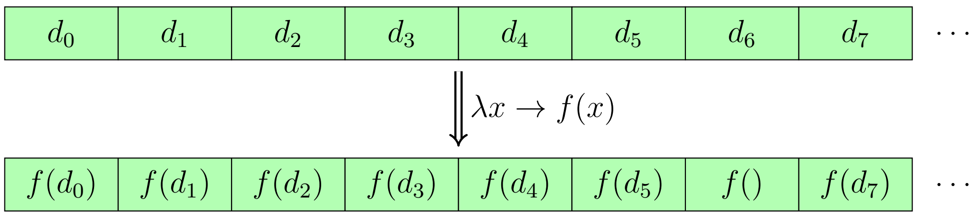

Figure 2.1: A data parallel operation

An interesting member of data-parallel operations are vector operations. Vector operations are not only data parallel, they perform exactly the same instructions for every data-point. The example about geometry transformation we gave is a vector operation. If instructions executed differ due to e.g. the presence of conditionals, a data-parallel operation is not a vector operation.

Task parallelism occurs when two, potentially different, tasks do not directly or indirectly depend on each other's outputs. Figure 2.2 is a graphical representation of task parallelism.

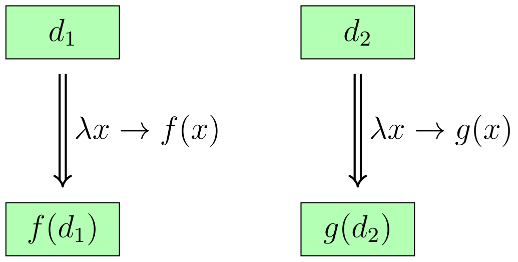

Figure 2.2: An example of task parallelism

In that situation, $\lambda x \to f(x)$ can be executed on one processing unit while $\lambda x \to g(x)$ is offloaded to another.

In our method we will look exclusively at task parallelism. In chapter 6 we will theorize how the method could be expanded by considering data-parallelism as a special case.

2.3 Parallel architectures

Different parallel architecture exists, each with their own advantages and drawbacks. We will briefly mention a few common ones.

Multi-core processors are the industry's answer to hitting the limitations of sequential instruction execution [2]. They essentially have multiple semi-independent processing units. It varies per architecture what resources are shared as there is no exact definition of a 'core'. Multiple cores are capable of exploiting both data and task parallelism.

SMT cores may have multiple threads in their pipeline at the same time. This is an efficient form of hardware multithreading, essentially simulating multiple cores. Sharing a core between threads means that while one thread is waiting for a memory read or unsure about future instructions, the core can run another one. Since multiple hardware threads share a pipeline, they are generally expected to be slower in throughput than multiple cores, although this may not always be the case as cache-sharing may reduce communication overhead. Some of the benefits of SMT cores are further explained in [4].

SIMD processing units, such as GPUs, execute an instruction on multiple data-points with a single operation. This makes them exceptionally fast for vectorization but incapable of task parallelism. Other forms of data-parallelism can sometimes be simulated, but this is slower than real vector operations.

*FPGA*s are programmable hardware components which can be used to create program-specific pipelines.

Most present-day systems are *heterogeneous*, e.g. they contain both multiple cores and a SIMD unit.

2.4 Data-races, data-flow dependencies and functionally pure languages

When tasks are run concurrently we no longer know when they will access data in memory. If tasks read and write to the same memory location, scheduling them concurrently can change the order in which this happens. A tasks may read data that does not exist yet, as the task that should write that data has not done so yet, resulting in undefined behaviour. Such a situation is called a *data race*. We can avoid data races through synchronisation, e.g. not running one task before all the data it requires has been written. To do this we must be able to identify what a task's *data-flow dependencies* are. A task has a data-flow dependency from another task if it requires data which that other task provides.

The difficulty of extracting data-flow dependencies from a program varies based on what programming language is used.

Functionally pure languages such as Haskell and Mercury ensure that a function[1] will not read or modify any global state; it depends entirely on its inputs and outputs. Thanks to this property, we know that functions written in those languages only have data-flow dependencies with tasks that provide their inputs or read their outputs. Data-flow dependencies have been extracted for functionally pure languages abundantly in previous research [5, 6, 7]. Furthermore, new languages developed with the intention of easing the creation of parallel program implementations are often functionally pure [6, 8, 9].

On the other hand, traditional imperative languages such as C do allow

[1]or predicate in the case of Mercury

for global state access. This adds a degree to uncertainty concerning what parts of memory a function will use. Methods used to find data-flow dependencies for such languages, e.g. [10], will therefore require confirmation from a programmer who knows which tasks are indeed data-flow independent.

By finding all data-flow dependencies we can extract all the parallelism that is implicitly present in a program's code.

2.5 Task granularity and load-balancing

Even if we can extract all the implicit parallelism present in a program, this will create tasks with a very fine granularity. As mentioned in the chapter 1, having many parallel tasks will require more communication operations. If the tasks are particularly small, we may be spending more time communicating than was saved on computation time.

On the other hand, if we have very large tasks, *load balancing*, i.e. distributing the tasks over available hardware so that the program will run fastest, becomes more difficult. For example, if a system has 4 cores and we wish to execute 5 equally-sized tasks on them, there is no way to distribute them evenly.

This dichotomy of reducing communication overhead and maintaining parallelism, using coarse or fine granularity, is often addressed by selecting tasks. The effect of offloading a task to another processing unit is estimated based on e.g. input sizes, heuristics or profiling. If the effect is expected to be positive for program execution time, we allow the task to be offloaded. Any other task is run sequentially, avoiding communications overhead. This approach has been implemented in various ways [11, 7, 5]. One possibility is to compare a task's expected execution time with a threshold. If the threshold is at least the time spent on communication for a particular task, the time saved by offloading that task will be greater than the communication time, provided there is enough unused hardware to execute the task. Our method will try to improve on this approach.

2.6 Scheduling

After having divided a program into tasks we must decide how to distribute them over the available hardware resources. Although it is possible to make a static schedule, doing so in an efficient manner would require us to know at compile time what the execution time of each task is. This is difficult for multiple reasons:

- Execution times of tasks may vary depending on their inputs.
- Other processes may contend the hardware we plan to use, slowing down a task.
- The effect of caches can be very hard to predict, especially if multiple cores share one or more caches.
- Its often the case that not all targeted systems have the same hardware.

These problems can be mitigated by using a dynamic scheduler. Schedulers will distribute the workload over available hardware at runtime. This can be based on the expected execution time of tasks or simply by letting inactive threads check if they can take over any work from active threads. Examples of schedulers are Cilk [12, 13], Wool [14], StarPU [15] and StarSS [16].

We will not use schedulers in our work but the task we generate should be fed to a scheduler to create a working parallel program implementation.

2.7 Execution time estimation and profiling

Execution time estimation is a broad problem in informatics.

For critical systems it is sometimes required to know an upper bound of the execution time of certain programs. This is a much researched topic [17, 18, 19], but proposed methods do not work for more advanced instructions which introduce a degree of uncertainty. Although execution time estimates can be made for modern linear languages [20], estimating the run-time of complex parallel systems is a difficult problem. Such estimates have been made, but are often on a very coarse scale [21] or require running the program first [22].

An obvious way to estimate the execution time of a program or task is to run it, measure how long it takes and expect this value to be indicative of the execution times of future runs. This is called profiling. There are two main ways of profiling code: instrumentation profiling, where the time between specific instructions is measured, and statistical profiling, where the program is interrupted multiple times and its stack is analysed. This gives us an idea of which functions are executed the most and therefore take the most time. Instrumentation profiling is simpler but increases the program's execution time, making the profiling results less accurate. Statistical profiling on the other hand requires a sufficiently large number of samples to be accurate. Furthermore it only gives us the total execution time of a task, i.e. its execution time multiplied by the number of times it has run, not the time required to run the task once.

Chapter 3

Design

To ease the creation of parallel implementations of programs, we require a way to ensure tasks selected for concurrent execution have the right granularity, so that we may balance computation time and communication time.

In this chapter we will elaborate the design of our method. We will first give a brief overview of how our method works.

3.1 Overview

We design a method that can take a set of fine-grained task with data-flow dependencies and from them generate sufficiently coarse-grained tasks, fit for parallel execution. The method will attempt to preserve parallelism as much as possible and it will respect the restrictions set by data-flow dependencies. Thus we will improve on previous methods that relied on simple selection of tasks, thereby requiring these tasks to be sufficiently coarse to start with in order to find any useful parallelism. That is, these methods do not consider the possibility of merging multiple tasks together, which would incur the communication overhead only once for all the merged tasks.

Our method can be used to decompose a program into very small parts and merge these parts into tasks of an acceptably coarse granularity. It finds which tasks may be merged based on data-flow dependency restrictions, as we detail in section 3.4. The merging of tasks needs to stop at some point, or we would end up with a sequential implementation. To this end we will define a *stop criterion* in section 3.6. Since there are multiple ways in which tasks can be merged we need to find the best one. This is done using an intelligent search algorithm. Any such algorithm requires a *fitness function* to determine how good a certain result is. We will present our search function in section 3.7.

A naive solution could be constructed by using profiled execution time as a fitness function. The stop criterion could be reached once no merges exist that further improve the fitness value. However, this would require a lot of profiling which may be too computationally expensive. This problem is further increased as values yielded by profiling may be noisy and require many runs to be representative. The stop criterion and fitness function we present can be performed statically provided we can estimate the sequential execution time of single tasks. How we make this estimation can be found in section 3.5, it will require only one profiling run.

Our method requires a *data-flow dependency graph*, defined in section 3.3, and produces an *execution order graph*, defined in section 3.2.

Our method does not commit to one search algorithm or another but instead provides a context which may be used by different algorithms. In chapter 4 we will further elaborate what algorithm we used for the implementation.

3.2 The execution-order graph

The output and intermediate results of our method are expressed as an *execution-order graph.* This is a graph that denotes what tasks exist and how they depend on each other. We formally define the graph in definition 1.

Definition 1 *A* program execution-order graph $G = \{T, D, r\}$ *is an acyclic, directed, antitransitive graph representing a program's task execution order. In the above formula, T is a set of tasks and D the set of data-flow dependencies between them. r is a root node s.t. $r \notin T$ but $\exists t \in T(r \in D_{in}(t))$. For $t \in T$ we have $D_{out}(t) \in D$, defining all outgoing data-dependencies for t and $D_{in}(t)$ defining the input dependencies. We note that iff. $t^2 \in D_{out}(t^1)$ then $t_1 \in D_{in}(t_2)$. An execution-order graph is always* antitransitive.

An example execution order graph is provided in figure 3.1. This is a possible graph for a program that computes values for some variables x_1 and x_2, uses x_2 to compute y and finally computes z using all previously computed variables.

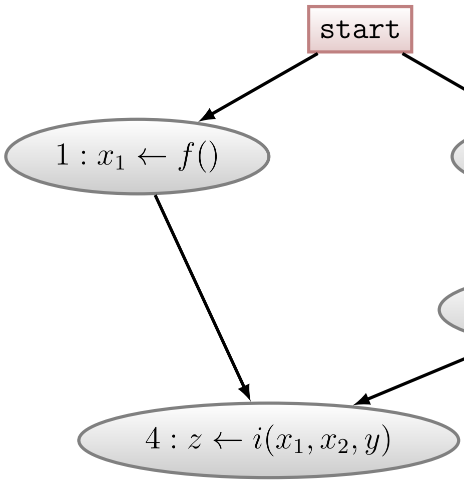

Figure 3.1: An example of an execution-order graph

A task may require data provided from multiple tasks, but may not be connected to all these tasks. A connection between two tasks can only exist if there is no chain of tasks already connecting them. This means every task is only connected to those tasks that it must explicitly wait for, which makes visual representations of the graph prettier as it is closer to a possible implementation. Although this property is stronger than antitransitivity[1], we denote it as antitransitivity in the rest of this document.

The graph is also acyclic. Recursion, such as in function calls or loops, is represented by a task that starts another graph. In the case of loops this graph is 'run' multiple times.

Every task in the graph has a set of sub-tasks. Initially this set contains one sub-task for every task. When two tasks are merged together the resulting task will contain the sub-tasks of both merged tasks. Sub-tasks within the same task are executed sequentially and they are ordered.

Visually we will represent both tasks $t \in T$ and r by nodes and D by edges between those nodes. The root nodes r will be coloured differently.

We point out the absence of a connection between nodes 2 and 4, even though the latter depends on data (x_2) from the former. This is because there already exists a chain of nodes connecting them, as explained above.

[1]for antitransitivity this only holds for chains with a single intermediate node

3.3 The data-flow dependency graph

Input to our method is a data-flow dependency graph, which we will use to construct a first execution-order graph. A data-flow dependency graph is a graph representing how tasks depend on each other's inputs and outputs. It is similar to an execution-order graph but it is not necessarily antitransitive. Furthermore, tasks in a data-flow dependency graph have exactly one sub-task. This is not a necessity for our method, but, ideally, a data-flow dependency graph would contain the maximal implicit parallelism of a program, so tasks should not be pre-merged. Figure 3.2 depicts the data-flow dependency graph that generated the execution-order graph from figure 3.1.

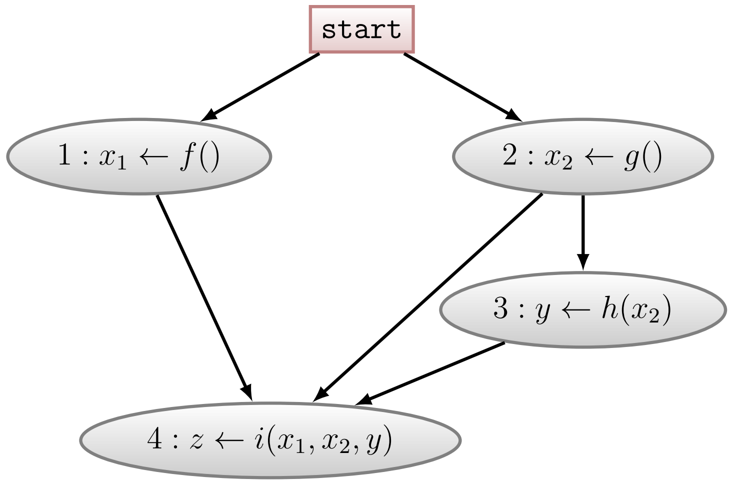

Figure 3.2: An example of a data-flow dependency graph

The dependency graph is defined in definition 2

Definition 2 *A* data-flow dependency graph $G_{data} = \{T, D_{data}, r\}$ *consists of all tasks T a program must execute and how they depend on each other. Iff. task $t^2 \in T$ requires data provided by $t^1 \in T$, then $(t^1, t^2) \in D_{data}$. Additionally, iff. task $t^1 \in T$ requires no data from other task, then $(r, t^1) \in D_{data}$. No other elements exist in D_{data}*

The dependency graph is directed and acyclic. It is directed because an edge expresses that one node (a task) requires data from another, it is acyclic because cyclic data-flow dependencies would be unresolvable: we can't execute any tasks in the cycle as all such tasks would depend on data that is not computed yet.

To get from a data-flow dependency graph to an execution-order graph, we must also make the former antitransitive by pruning any superfluous dependencies.

For this we define the transitive operator $\triangleright$, given D_{data}. We note that this operator has higher priority than any others.

$$t^1 \triangleright t^2 \longleftrightarrow (t^1, t^2) \in D_{data} \vee \exists t^3 (t^1 \triangleright t^3 \wedge t^3 \triangleright t^2)$$

This operator essentially denotes that t^2 must execute after t^1, although it may not directly depend on it. Given a data-flow dependency graph $G_{data} = \{T, D_{data}, r\}$, we construct an execution-order graph $G = \{T, D, r\}$. Only the edges

$$\left\{(t^1, t^2) \in D_{data} \mid \neg\exists t^3((t^3, t^2) \in D_{data} \wedge t^1 \triangleright t^3)\right\}$$

are in D, all others are pruned.

The execution-order graph so constructed is the starting point for our search algorithms. It will gradually be improved.

A data-flow dependency graph can be constructed from a program through either static analysis or supervised profiling of memory accesses such as in [10].

3.4 Allowed merge operations

When deciding which tasks may be merged together, it is important to ensure that merging those tasks will not result in any data-flow dependency violations. That is, if an execution-order graph denotes that task t^1 should be executed after task t^2, any acceptable merge operation should result in a graph were this is still the case. Furthermore, merging should not create any cycles, as this would be unresolvable, as explained in section 3.2. In figure 3.3 we see an obvious example of a bad merge: not only have we inversed the order in which t^1 and t^4 are executed, we have created a cycle, meaning none of the task will ever start executing as they all wait for data from other tasks.

To formally define these restrictions, we first reintroduce the $\triangleright$ operator, used earlier in the context of a data-flow dependency graph, within the

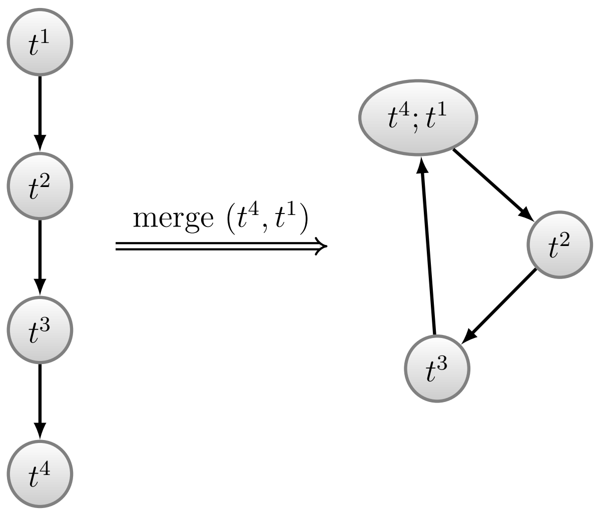

Figure 3.3: An obvious example of a bad merge

context of an execution-order graph:

$$t^1 \triangleright t^2 \longleftrightarrow (t^1 \in D_{in}(t^2) \vee \exists t^3(t^1 \in D_{in}(t^3) \wedge t^3 \triangleright t^2)) \tag{3.1}$$

To denote how tasks and their connections change after a merge operation, we use the subscript m. For some $t \in T$, t_m is the task containing all subtasks of t after the merge operation is executed. That is, if m merges the pair (t^1, t^2), then $t^1_m = t^2_m = (t^1; t^2)$, for any task t that is not being merged $t_m = t$. We clarify that $(t^1; t^2)$ is a task which executes the subtasks from t^1 first, followed by the subtasks from t^2. Note that $t_m = t \not\models D_{in}(t_m) = D_{in}(t)$, in other words, the connections of a task may change even if the task itself does not.

Now we can formalize our restrictions. Each of the following must hold:

$$t^1 \triangleright t^2 \models t^1_m \triangleright t^2_m \vee t^1_m = t^2_m \tag{3.2}$$
$$t^1 \triangleright t^2,\ t^1_m = t^2_m \models t^1_m = (t^1; t^2) \tag{3.3}$$
$$t^1_m \triangleright t^2_m \models \neg t^2_m \triangleright t^1_m \tag{3.4}$$

To ensure a merge meets these requirements, we only merge a pair of tasks if it matches any patterns described in definition 3. We will prove that these pairs meet our restrictions later, after we have further detailed how a merge is performed.

Definition 3 *For a graph $G = \{T, D, r\}$, we allow the following tasks to be merged together:*

- Siblings.

  $$\big\{\{t^1, t^2\} \mid \exists t^3 \left(t^3 \in D_{in}(t^1) \wedge (t^3 \in D_{in}(t^2)\right)\big\}$$

  I.e. tasks which both depend on a third task to provide some data.

- Co-parents.

  $$\big\{\{t^1, t^2\} \mid \exists t^3 \left(t^3 \in D_{out}(t^1) \wedge (t^3 \in D_{out}(t^2)\right)\big\}$$

  I.e. tasks that provide data to a common third task.

- Parent-child pairs.

  $$\big\{(t^1, t^2) \mid t^2 \in D_{out}(t^1)\big\}$$

  I.e. two task of which one provides a data to the other.

We can merge a pair (t^1, t^2) creating a new task $t^1_m = t^2_m = (t^1; t^2)$ with the following connections:

$$D_{in}(t^1_m) = makeAntitransitive(D_{in}(t^1) \cup D_{in}(t^2)) \tag{3.5}$$
$$D_{out}(t^1_m) = makeAntitransitive(D_{out}(t^1) \cup D_{out}(t^2)) \tag{3.6}$$

Where *makeAntitransitive*() is a function which prunes superfluous connections in the same manner as is used for the creation of execution-order graphs from a data-flow dependency graphs, explained in section 3.3. It also removes reflexive connections.

We will now prove our restriction is met for all pairs considered in definition 3. By equation (3.5) and (3.6), a merge will only affect tasks directly connected to those that are being merged. We will limit our proof to only these affected tasks for the sake of brevity.

Proof 1 *We note that t^1 and t^2 are not necessarily the nodes being merged, unless otherwise stated.*

First, we will prove that the way we merge tasks respects the order of operations between the nodes after a merge, the restriction in equation (3.2)*. From equation* (3.5)*:*

$$t^1 \in D_{in}(t^2) \models \left(t^1_m \in D_{in}(t^2_m)\right) \vee \left(t^1_m \triangleright t^2_m\right) \vee \left(t^2_m = t^1_m\right) \tag{3.7}$$

That is, if t^1 *was an input dependency to some other task, the new node will be as well, unless it was pruned by makeAntitransitive, which would also remove reflexive dependencies.*

By the definition of $\triangleright$*, i.e. equation* (3.1)*, we can abbreviate this to:*

$$t^1 \in D_{in}(t^2) \models t^1_m \triangleright t^2_m \vee t^2_m = t^1_m$$

We can use this equation as the base case for a proof by induction. For the sequential case we will prove $t^1 \in D_{in}(t^3),\ t^3_m \triangleright t^2_m \models t^1_m \triangleright t^2_m \vee t^1_m = t^2_m$*:*

$$\begin{aligned} t^1 \in D_{in}(t^3),\ t^3_m \triangleright t^2_m &\models \left(t^1_m \in D_{in}(t^3_m) \vee t^1_m = t^2_m\right) \wedge t^3_m \triangleright t^2_m \\ &\models \left(t^1_m \in D_{in}(t^3_m) \wedge t^3_m \triangleright t^2_m\right) \vee t^1_m = t^2_m \\ &\models t^1_m \triangleright t^2_m \vee t^1_m = t^2_m \end{aligned}$$

With this we have proven:

$$t^1 \triangleright t^2 \models t^1_m \triangleright t^2_m \vee t^1_m = t^2_m \tag{3.8}$$

Which is our first restriction.

Now we must prove that each of the proposed merge-pairs will not cause any cycles when merged (3.4) *and that merging such pairs will respect any order restrictions that exists between them* (3.3)*.*

For this we will require the following observation. Since the graph is antitransitive:

$$t^1 \in D_{in}(t^2) \models \neg\exists t^3 (t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \tag{3.9}$$

The proof for siblings and co-parents.

We will first prove that there exist no order requirement between siblings.

$$\begin{aligned}
\exists t^3(t^3 \in D_{in}(t^1) \wedge (t^3 \in D_{in}(t^2)),\ t^1 \rhd t^2 &\models \exists t^3((t^3 \in D_{in}(t^1) \wedge \neg\exists t^4(t^3 \rhd t^4 \wedge t^4 \rhd t^2))\\
&\models \exists t^3((t^3 \in D_{in}(t^1) \wedge \neg(t^3 \rhd t^1 \wedge t^1 \rhd t^2))\\
&\models \exists t^3((t^3 \rhd t^1 \wedge t^1 \rhd t^2 \wedge \neg(t^3 \rhd t^1 \wedge t^1 \rhd t^2))\\
&\models \perp\\
\exists t^3(t^3 \in D_{in}(t^1) \wedge (t^3 \in D_{in}(t^2)) &\models \neg t^1 \rhd t^2
\end{aligned} \tag{3.10}$$

This obviously satisfies (3.3).

We can also use it for our proof that restriction (3.4) *is respected. Since only connections to and from the merged nodes will be altered, those nodes would have to be part of any cycles created:*

$$t^3_m \rhd t^4_m,\ t^4_m \rhd t^3_m \models t^3_m = t^1_m \vee t^4_m = t^1_m \tag{3.11}$$

Where t^1 and t^2 are the nodes we chose for our merge while t^3 and t^4 are unbound, i.e. they may be any task $\in T$. We also note that

$$t^1_m \rhd t^3_m \models t^1 \rhd t^3 \vee t^2 \rhd t^3 \tag{3.12}$$

follows from (3.5). *This means the execution order restrictions of the new task must originate from either or both of the merged tasks.*

We will now prove that merging tasks with no order-restriction between them will not result in any cycles in G.

$$\begin{aligned}
\neg t^1 \rhd t^2,\ \neg t^2 \rhd t^1,\ t^3_m \rhd t^4_m,\ t^4_m \rhd t^3_m &\models (t^3_m = t^1_m) \vee (t^4_m = t^1_m)\\
&\models (t^1_m \rhd t^4_m \wedge t^4_m \rhd t^1_m) \vee (t^1_m \rhd t^3_m \wedge t^3_m \rhd t^1_m)\\
&\models (t^1_m \rhd t^3_m \wedge t^3_m \rhd t^1_m)\\
&\models \big((t^1 \rhd t^3) \vee (t^2 \rhd t^3)\big) \wedge \big((t^3 \rhd t^1) \vee (t^3 \rhd t^2)\big)\\
&\models (t^1 \rhd t^3 \wedge t^3 \rhd t^1) \vee (t^1 \rhd t^3 \wedge t^3 \rhd t^2)\\
&\quad \vee (t^2 \rhd t^3 \wedge t^3 \rhd t^1) \vee (t^2 \rhd t^3 \wedge t^3 \rhd t^2)\\
&\models \perp \vee (t^1 \rhd t^2) \vee (t^2 \rhd t^1) \vee \perp\\
&\models \perp\\
\neg t^1_m \rhd t^2_m,\ \neg t^2_m \rhd t^1_m &\models \neg(t^3_m \rhd t^4_m \wedge t^4_m \rhd t^3_m)
\end{aligned} \tag{3.13}$$

Similarly, we can prove that co-parents have no order restrictions between them:

$$\begin{aligned}
\exists t^3\left(t^3 \in D_{out}(t^1) \wedge (t^3 \in D_{out}(t^2)\right), t^1 \triangleright t^2 &\models \exists t^3(t^1 \in D_{in}(t^3) \wedge (t^2 \triangleright t^3)) \\
&\models \exists t^3\left(\neg\exists t^4(t^1 \triangleright t^4 \wedge t^4 \triangleright t^3) \wedge (t^2 \triangleright t^3)\right) \\
&\models \exists t^3\left(\neg(t^1 \triangleright t^2 \wedge t^2 \triangleright t^3) \wedge (t^2 \triangleright t^3)\right) \\
&\models \perp \\
\exists t^3(t^3 \in D_{out}(t^1) \wedge (t^3 \in D_{out}(t^2)) &\models \neg t^1 \triangleright t^2
\end{aligned} \tag{3.14}$$

Thereby proving our restrictions are met in the same manner as for siblings.

The proof for parent-child pairs. *For parent-child pairs, the correct order is enforced, so the proof for restriction* (3.3) *is obvious:*

$$t^1 \in D_{in}(t^2),\ t^1_m = (t^1;t^2),\ t^1_m \triangleright t^2_m \models t^1_m = (t^1;t^2)$$

As for cycles:

$$\begin{aligned}
t^1 \in D_{in}(t^2), t^1_m = t^2_m,\ C &\models \exists t^3(t^1_m \triangleright t^3_m \wedge (t^3_m \triangleright t^1_m) \\
&\models \exists t^3((t^1 \triangleright t^3 \vee t^2 \triangleright t^3) \wedge (t^3 \triangleright t^1 \vee t^3 \triangleright t^2)) \\
&\models \exists t^3\left(\perp \vee (t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \vee (t^2 \triangleright t^3 \wedge t^3 \triangleright t^1) \vee \perp\right) \\
&\models \exists t^3\left((t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \vee (t^2 \triangleright t^1)\right) \\
&\models \exists t^3\left((t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \vee (t^2 \triangleright t^1)\right) \wedge t^1 \triangleright t^2 \\
&\models \exists t^3(t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \\
&\models \exists t^3(t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \wedge \neg\exists t^3(t^1 \triangleright t^3 \wedge t^3 \triangleright t^2) \\
&\models \perp \\
t^1 \in D_{in}(t^2), t^1_m = t^2_m &\models \neg C
\end{aligned} \tag{3.15}$$

Where $C = \exists t^3(t^1_m \triangleright t^3_m \wedge (t^3_m \triangleright t^1_m)$, *for brevity. In this last proof we've used equation* (3.12), *the acyclic property of our input graph, transitivity of* $\triangleright$, *equation* (3.1) *and the antitransitivity of our graph, in that order.*

3.5 Task execution time estimation

Although we can now merge tasks without causing errors in the order they are executed, we still need a stop criterion and a fitness function. As we will explain later, we need to estimate the execution time of a task for either of these.

We expect profiling to generally be too expensive to be used whenever we require a new task's execution time, so we will need a way to estimate their execution times, or costs, through some kind of static analysis.

Although methods exist to estimate the execution time of a sequential program based on static analysis of the code, as mentioned in 2.7, such methods are much too complex and language-specific for the scope of our research.

Considering the fact that our tasks consist of multiple subtasks that are executed sequentially, it is fair to assume the cost of a task is the sum of the cost of its subtasks:

$$c(t) = \sum_{t_i \in t} c(t_i) \tag{3.16}$$

For instance, a task created by some merges $c\left((t^1; t^2; t^3)\right) = c(t^1) + c(t^2) + c(t^3)$. This requires knowing the costs for tasks in the original data-flow dependency graph. How these costs are computed exactly is irrelevant to our method, although they should be somewhat accurate for the result to be useful. We propose profiling tasks with a non-minimal execution time and assuming short tasks with $O(1)$ complexity, consisting of few instructions to have a cost of 0.

We note that using equation (3.16) as execution time estimation is not entirely correct. The effect of cache misses, for one, may have unknown consequences for the actual execution time of a particular task when it follows some other one. We don't expect this to be a real problem, but if greater accuracy is necessary, the formula could be replaced by a new profiling run once it depends on too many terms.

3.6 Stop criterion

As we mentioned earlier in this chapter we need to know when to stop merging nodes. Otherwise we will end up with a linear program again. We wish to stop when there is a good balance between time spent for communications and time spent for computations. This happens when there are no tasks for which we spend more time communicating than we save by offloading their computation. So, assuming we have a minimal cost for tasks that are allowed to be offloaded, called s, we have to ensure that no task which causes a communication overhead is smaller than s. We note that s is at least as large as the overhead of offloading a task, we cannot increase the total execution time by doing so with tasks that take longer than s.

With this in mind, we define our stop criterion:

Definition 4 Stop criterion*:*
A graph $G = \{T, D, r\}$ *is accepted iff.* $\exists! t \in T$ *or* $\forall t \in T$ *any of the following holds:*

$$c(t) \geq s \tag{3.17}$$

$$\exists t^2 \in D_{out}(t) \left(D_{in}(t^2) = \{t\} \wedge c(t^2) \geq s \right) \tag{3.18}$$

$$\exists t^2 \in D_{in}(t) \left(D_{out}(t^2) = \{t\} \wedge c(t^2) \geq s \right) \tag{3.19}$$

The notion that we should stop if $\exists! t \in T$ is trivial: if the graph is completely sequential we stop merging. For the second part, we check if every node matches one of three conditions: it is larger than s, it has a task depending on it and it alone that is larger than s or a task larger than s provides data to it and it alone.

The first condition is obvious, if a task is large enough we do not mind an overhead. But this condition alone would be too strict, as demonstrated in figure 3.4.

In this figure, if $c(t^1) < s$, $c(t^4) < s$, $c(t^2) \geq s$ and $c(t^3) \geq s$, using only (3.17) we would have to merge t^1 and t^4 with something to trigger our criterion, either operation reducing the parallelism more than desired.

More generally, if a task follows or is followed by another task with no waiting operations between them, those two tasks can be scheduled on the same processing unit, causing an overhead only once. We get, as it were, the smaller task 'for free'.

The inclusion of criteria 3.18 and 3.19 does have the effect of accepting some strictly sequential connections that are not fully merged, as in figure 3.5, where if $c(t^1) > s$ or $c(t^2) > s$, the stop criterion would trigger. Although this is not a case of excessive parallelism, it may be undesirable. Luckily it is trivial to merge such cases further.

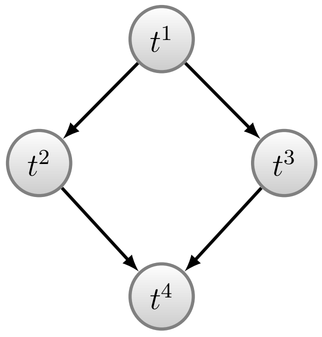

Figure 3.4: An execution-order graph that should sometimes trigger our criterion

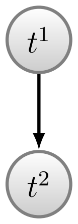

Figure 3.5: Another execution-order graph that could trigger our criterion

3.7 Execution-order graph evaluation

Although we know which nodes can be merged together to reduce parallelism and when to stop merging them, not all results are going to be equally efficient. For instance, if we continually merge small tasks into a specific big one, the stop criterion will trigger once only the big task remains, even though we may have been able to merge the small tasks with each other and end up with 2 big tasks that could run in parallel.

We need a way to compare graphs and decide which one is 'best'; a fitness function. Ideally, this function should yield a one-dimensional value as this makes implementing intelligent algorithms much simpler and leaves us with a single clear 'winner' instead of a set of them.

Considering that our end-goal is to improve the speed of concurrent programs, we use the expected execution time of the *critical path* as a fitness measure. This time is the minimum we will require to compute every task in the graph, given the current connections. It is what we would get on a machine with no communication overhead and infinite processing units. It is therefore a good measure to keep useful parallelism high and avoid creating too many sequential dependencies.

We note that there might be better evaluation strategies if the target machine is known. We will elaborate on this in our discussion in chapter 6.

Chapter 4

Implementation

In this chapter we will specify which parts of our method and its requirements we implemented and how we did so.

First of all, we had to be able to extract a data-flow dependency graph from a program. This meant either implementing a complex method that profiles memory accesses to *speculate* which functions are independent or just using a functionally pure, eagerly evaluated language and know for sure. A choice was easily made for the second option.

Since being able to parse a full, modern language would constitute a lot of work that is outside the scope of our research, we have created a very simple language for demonstration purposes. We present it in section 4.1. Our method is, however, applicable to other functionally pure, eagerly evaluated languages as well. Section 4.2 explains how we extracted tasks from programs in our language.

We have created a simple profiler to get the costs of our initial tasks, it is described in section 4.3. We have also implemented a greedy maximizing algorithm and all the functions our method requires (e.g. creating an execution-order graph from a data-flow dependency graph, finding merge pairs, comparing graphs by their critical path and deciding whether a graph meets our stop criterion).

Our code is written in C++, a repository with the source code and the sample program used for the validation in chapter 5 is available at https://bitbucket.org/DrBearhands/master-thesis.

4.1 Language design and syntax

The grammar for our simple language is as follows:

$$\begin{aligned}
\langle program\rangle &::= \langle function\rangle +\\
\langle function\rangle &::= \text{“function”}\ \langle name\rangle\ \text{“(”}((\langle arg\rangle\ (\text{“,”}\ \langle arg\rangle)) * | \text{“”})\text{“) \{”}\ \langle statement\rangle + \text{“\}”}\\
\langle arg\rangle &::= (\text{“in”} | \text{“out”})\ \langle type\rangle\ \langle name\rangle\\
\langle statement\rangle &::= \langle declaration\rangle\\
&\quad | \langle while\rangle\\
&\quad | \langle conditional\rangle\\
&\quad | \langle function\ call\rangle\\
&\quad | \langle assignment\rangle\\
\langle declaration\rangle &::= \langle type\rangle\ \langle name\rangle\ (\text{“,”}\ \langle name\rangle)*\\
&\quad | \langle type\rangle\ \text{“[”}\ \langle expression\rangle\ \text{“]”}\ \langle name\rangle\ (\text{“,”}\ \langle name\rangle)*\\
&\quad | \langle name\rangle\ (\text{“,”}\ \langle name\rangle) * \text{“isSplit”}\ \langle name\rangle\ \text{“at”}\ \langle expression\rangle\\
\langle while\rangle &::= \text{“while (”}\ \langle expression\rangle\ \text{“) \{”}\ \langle statement\rangle + \text{“\}”}\\
\langle conditional\rangle &::= \text{“if (”}\ \langle expression\rangle\ \text{“) \{”}\ \langle statement\rangle + \text{“\}”}\\
\langle function\ call\rangle &::= \langle name\rangle\ \text{“(”}((\langle expression\rangle\ (\text{“,”}\ \langle expression\rangle)* \mid \text{“”})\text{“)”}\\
\langle assignment\rangle &::= \langle variable\rangle\ \text{“<-”}\ \langle expression\rangle\\
\langle variable\rangle &::= \langle name\rangle\\
&\quad | \langle name\rangle\ \text{“[”}\ \langle expression\rangle\ \text{“]”}
\end{aligned}$$

Most of the syntax should be familiar to imperative programmers. We also point the reader to the sample program in appendix A. We have omitted the rules for expressions, types and names as they are not particularly interesting but are somewhat complex.

One unusual feature of our language is splitting declaration for arrays. This declaration creates two arrays by dividing the accessible indices of a third one so that the two new arrays do not share any memory. A graphical depiction is available in figure 4.1.

We use this feature to pass parts of arrays to a function without requiring memory copy operations. This potentially allows to discover more parallelism since the parts are disjoint: if two function calls require access to disjoint parts of a data-structure, passing the whole structure might create a dependency between them even though this is not necessary, obfuscating opportunities for parallelism.

Although thinking about these split operations and how memory is accessed may seem like an additional burden on the programmer, which is something we want to avoid by automating parallelization, making good use of them can be summarized with one simple rule of thumb: to only give functions access to data they actually need. We believe this is generally a good design strategy in any case.

4.2 Task and data-flow dependency extraction

To extract a data-flow dependency graph for a program in our language, we have implemented a parser. A graph is made for every scoped statement block, i.e. the bodies of functions, conditionals and while loops. The statements inside a block are the tasks of the graph. Conditionals and while loops are both nodes and graphs. To extract the actual dependencies, i.e., the edges, we track the order in which variables are being read and written. If a statement reads a variable, we add a dependency to the last statement that wrote that variable. If a statement writes a variable, we add a dependency to all the statements that have read that variable since it was previously written.

This process is slightly more complex when split declarations are concerned. In figure 4.1 we see an array A which has been split in two different ways. Division B splits the array into B_1 and B_2, while division C splits it in C_1 and C_2. The locations in memory $A_0 - A_9$ are the same for all arrays.

We know that modifying B_2 will not affect B_1, but it might affect A, C_1 and C_2. For this reason, we group arrays we created from one split operations into a *division*. We know that if we modify one array of a division the other arrays of that division will remain unchanged, but that might not be the case for arrays in other divisions. Therefore, if we read a variable from the division of an array, we also consider to have read that original array and all arrays in any other divisions we have created from that original array.

Although in our example we know that modifying B_1 will not affect C_2, these kind of situations are not necessarily known at compile-time since split operations depend on non-constant expressions.

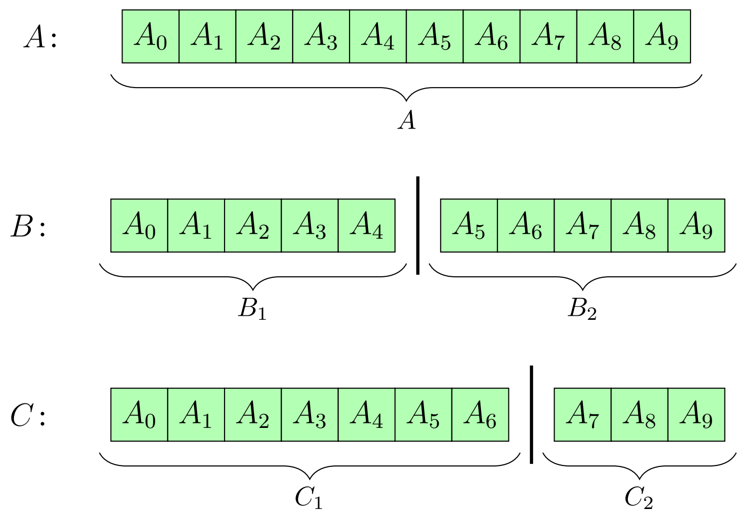

Figure 4.1: Multiple splits for array A

4.3 Initial task cost estimation

To estimate the costs of tasks in a graph, we compile programs from our language to `C++`, adding the necessary instructions for instrumentation profiling.

We profile tasks containing loops, conditional statement blocks and functions. The costs of all other tasks we initialized to 0, as they have complexity $O(1)$ and should consist of few instructions.

We have chosen instrumentation profiling over statistical profiling because, even though it is less accurate, we are interested in the cost of running a task once, not how much time it takes in our program in total.

We have chosen for `C++` over `C` as a target language because it allowed us to use some more advanced features, specifically templates and operator overloading, which made implementing the source-to-source compiler much easier. Since those are the only features of `C++` we use that are not in the common subset with `C`, we do not expect our decision to have negatively affected execution times.

We can profile either the average or maximal execution times of tasks. This will result in either realistic or pessimistic estimates, respectively. Using maxima, the graph's nodes' costs will be higher, causing us to hit the stop criterion sooner and therefore retaining more parallelism. In our tests, we will use maximal values.

The generated `C++` code is compiled into binaries using the `g++` compiler with flags `-std=c++11 -O2`.

4.4 Search algorithm

We have implemented a simple greedy search algorithm to determine what nodes to pick. We have chosen a greedy approach because it is simple and requires few fitness function evaluations. This makes it a logical choice for a first algorithm as it would be wasteful to use a slower and more complex algorithm without knowing if a simpler one might suffice. We simply do not know yet how problematic local optima might be in the field of implicit parallelism extraction and reduction, if at all.

At every iteration, we find the set of all allowed merges. For each of them, we evaluate the fitness function for the graph created by applying that merge. This gives us a score for every merge, we then chose the merge with the best score. The algorithm stops iterating when the graph meets our stop criterion.

After this, we merge whatever sequential tasks are left into a single task/node.

4.5 Visualization

We have written functions to export a graph to the `dot` format, used by e.g. Graphviz. Graphs presented in chapter 5 were created in such a way.

Chapter 5

Case study

5.1 Setup

In this chapter we perform a case-study of our method using a merge-sort algorithm. We have chosen this algorithm for three reasons:

- It is easy to implement
- Thanks to its similarity to a naive Fibonacci program, it is obvious for a human programmer what the correct parallelization is, making it easy to validate our results.
- It has a very consistent execution time, making profiling results a good estimation of future runs for similar data-sizes.

The program's implementation in our testing language can be found in appendix A. The generated `C++` program that was used for profiling is attached in appendix B.1, appendix B also contains other source files used for profiling, e.g. to initialize the values to be sorted.

As input to the merge-sort program we use an array of 10 million `float` values.

We extract the program's data-flow dependency graphs and turn them into execution-order graphs. We then feed the execution-order graphs to our search algorithm. We will examine the resulting graphs to determine whether they coincide with our expectation of what a human programmer would create.

Although comparing execution times may seem obvious, we do not do this. Our method is aimed at finding more opportunities for parallelism than simple selection of tasks does, if the opportunities it finds are equally fast, a faster alternative may simply not exist. Execution time comparisons would therefore only be meaningful on a large, representative dataset of programs, which we do not have for our language. Furthermore, our method does not require the use of any specific parallel programming paradigm, so an exporting the execution-order graph into a program for a particular paradigm may well yield biased results. Considering these issues, creating an exporter is more work than we believe it's worth.

Figures 5.1, 5.2, 5.3 and 5.4 show the execution-order graphs our method generated for the various functions. These are the graphs we start with, before any tasks have been merged.

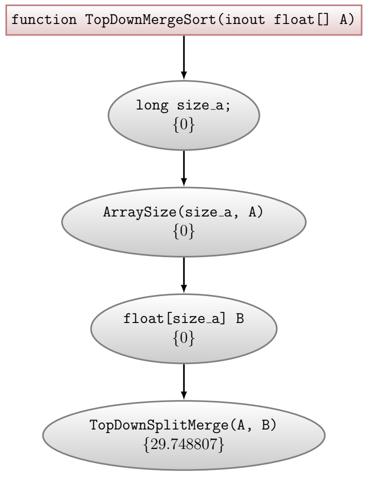

Figure 5.1: The execution-order graph for the `TopDownMergeSort` function at iteration 0

In these graphs, we have represented tasks containing statements which have both a scoped statement block and a tree of their own by creating a dashed edge between that node and the root of its tree. Root nodes are represented by coloured, rectangular nodes. The execution time estimations are written in the nodes within curly braces.

In figure 5.1 we see the graph for `TopDownMergeSort`. This graph is essentially sequential and therefore not interesting for our method.

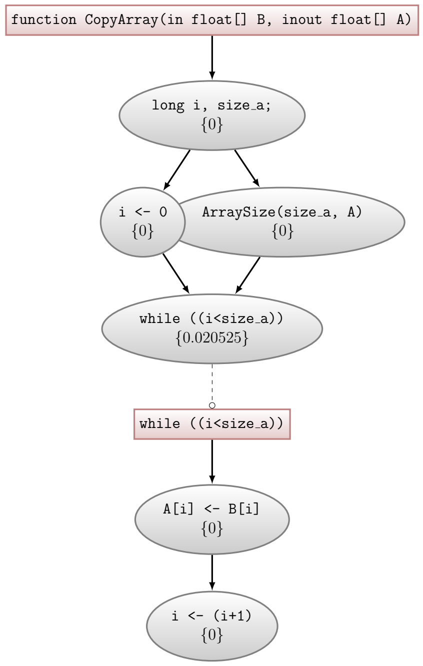

Figure 5.2: The execution-order graph for the `CopyArray` function at iteration 0

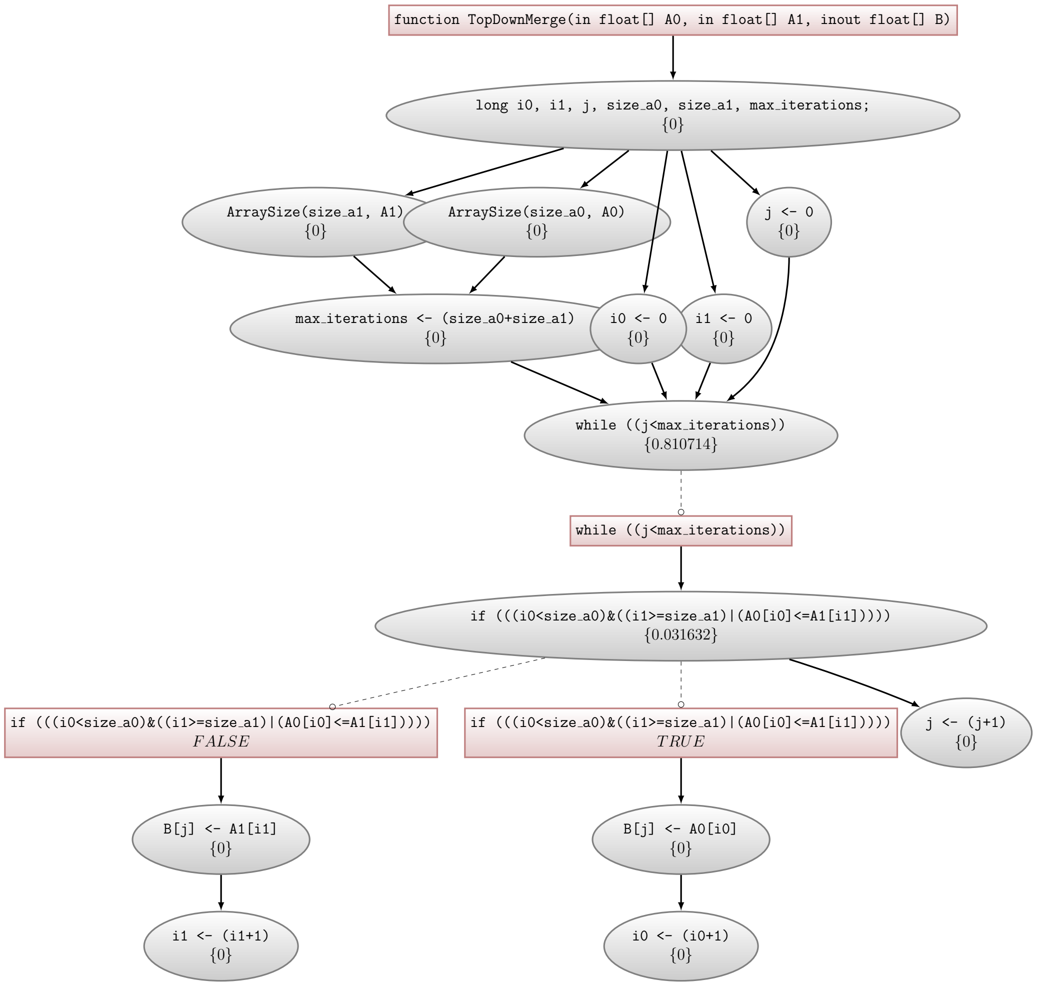
function TopDownMerge(in float[] A0, in float[] A1, inout float[] B)
long i0, i1, j, size_a0, size_a1, max_iterations; {0}
ArraySize(size_a1, A1) {0}
ArraySize(size_a0, A0) {0}
j <- 0 {0}
max_iterations <- (size_a0+size_a1) {0}
i0 <- 0 {0}
i1 <- 0 {0}
while ((j<max_iterations)) {0.810714}
while ((j<max_iterations))
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1])))) {0.031632}
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1])))) FALSE
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1])))) TRUE
j <- (j+1) {0}
B[j] <- A1[i1] {0}
B[j] <- A0[i0] {0}
i1 <- (i1+1) {0}
i0 <- (i0+1) {0}

Figure 5.3: The execution-order graph for the `TopDownMerge` function at iteration 0

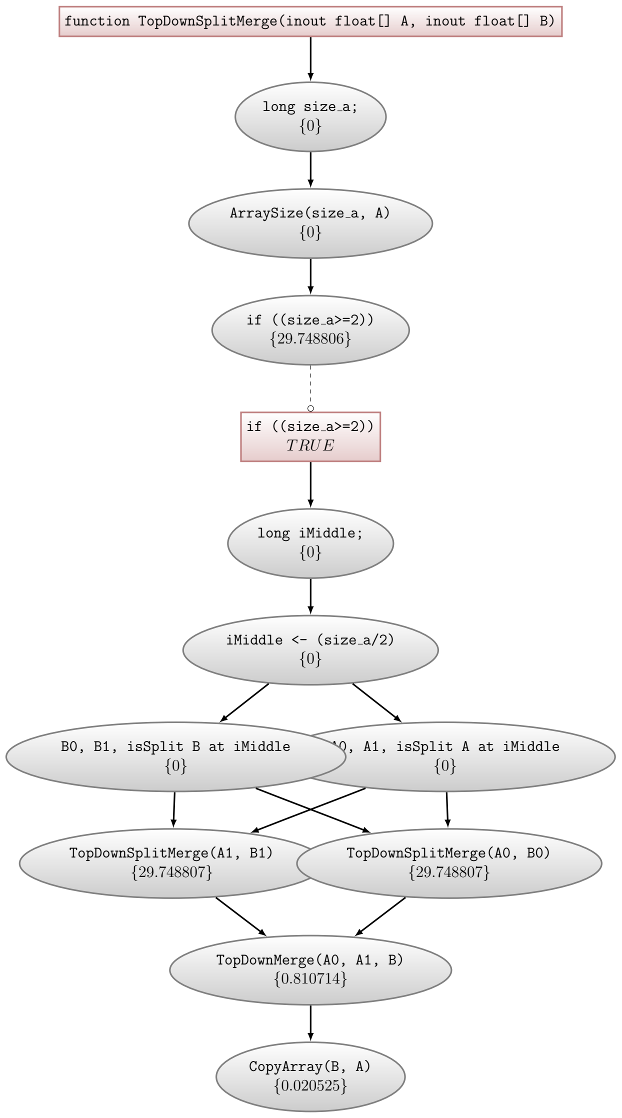

Figure 5.4: The execution-order graph for the `TopDownSplitMerge` function at iteration 0

In figure 5.2 we see the graph for a `CopyArray` function. This graph illustrates a weakness of our method: a vector operation, copying one array to another, something that can often profit easily and massively from parallel architectures, is expressed as a linear loop. Although it is not relevant in this specific example as its estimated cost is relatively low, our method should eventually be improved to deal with these cases. We will discuss how we might improve on this later, in section 6.1.

In figure 5.3 we see a clear example of too fine granularity. The depicted `TopDownMerge` function should be linear for maximal efficiency.[1]

Finally, in figure 5.4 we see the graph of the function `TopDownSplitMerge`. This is the function in which we'd like to keep some parallelism. Specifically, the recursive function calls `TopDownSplitMerge(A0, B0)` and `TopDownSplitMerge(A1, B1)` should stay parallel to each other, as these are two large tasks that will take longer than the communication overhead incurred by offloading them.

Besides those two function calls, we want to get rid of all parallelism from these graphs.

We have used a threshold value $s = 0.5$. In our test case however, we expect that any non-zero value lower than the cost of `TopDownSplitMerge` should yield the same results since all other non-zero cost tasks are sequential.

5.2 Results

After feeding the execution-order graphs from last section to our search algorithm, we ended up with the graph depicted in figures 5.5, 5.6, 5.7 and 5.8.

It is immediately obvious that we have reached our goal for figures 5.5 and 5.6, they are both completely sequential as evident from the graph, although this was already the case for the former.

Figure 5.5 also shows a completely linear graph. Although there appear to be two parallel nodes, those are actually two separate graphs of which either the one or the other is called, depending on the result of evaluating of the conditional.

Finally, in figure 5.7 we see that the parallelism between statements `TopDownSplitMerge(A0, B0)` and `TopDownSplitMerge(A1, B1)` has been

[1]barring, perhaps, VLIW machines or similar architectures where single threads execute multiple instructions as one, creating no communication.

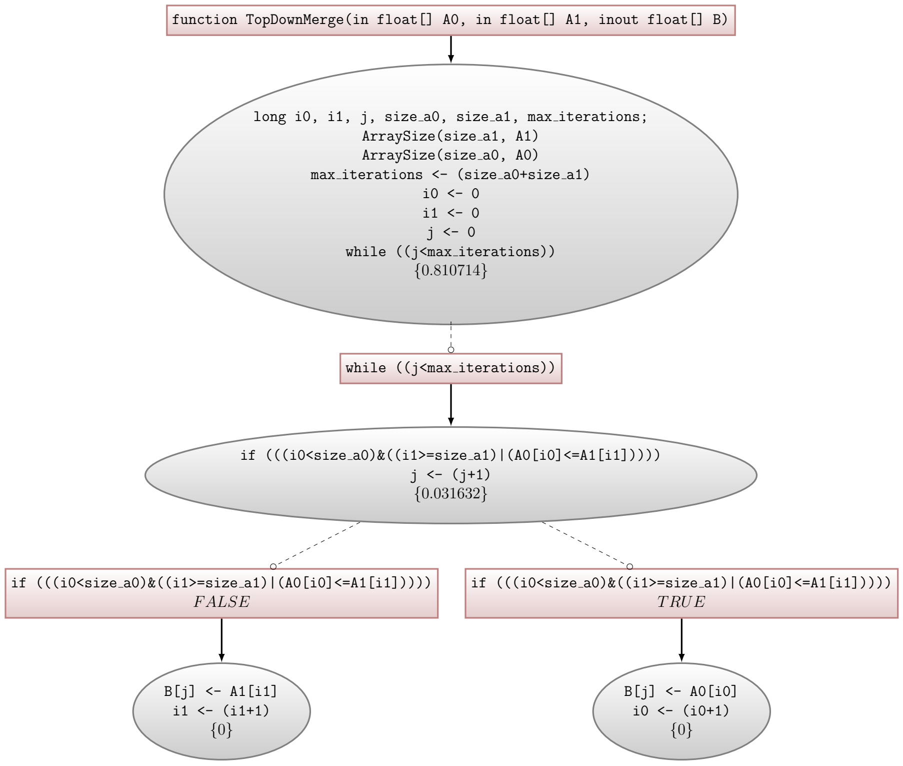
function TopDownMerge(in float[] A0, in float[] A1, inout float[] B)
long i0, i1, j, size_a0, size_a1, max_iterations;
ArraySize(size_a1, A1)
ArraySize(size_a0, A0)
max_iterations <- (size_a0+size_a1)
i0 <- 0
i1 <- 0
j <- 0
while ((j<max_iterations))
{0.810714}
while ((j<max_iterations))
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1]))))
j <- (j+1)
{0.031632}
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1]))))
FALSE
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1]))))
TRUE
B[j] <- A1[i1]
i1 <- (i1+1)
{0}
B[j] <- A0[i0]
i0 <- (i0+1)
{0}

Figure 5.5: The resulting graph for the `TopDownMerge` function

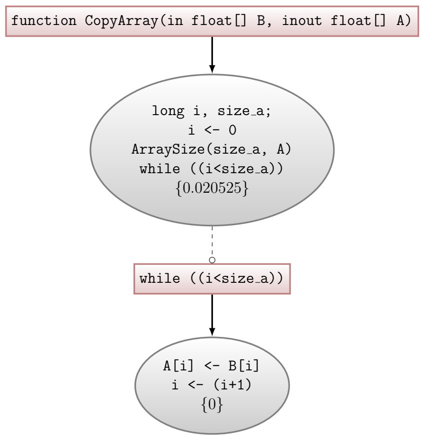

Figure 5.6: The resulting graph for the `CopyArray` function

preserved while everything else has become sequential, which is exactly what we wanted.

In short, our method managed to meet our expectations in finding useful parallelism for the program presented in this case study.

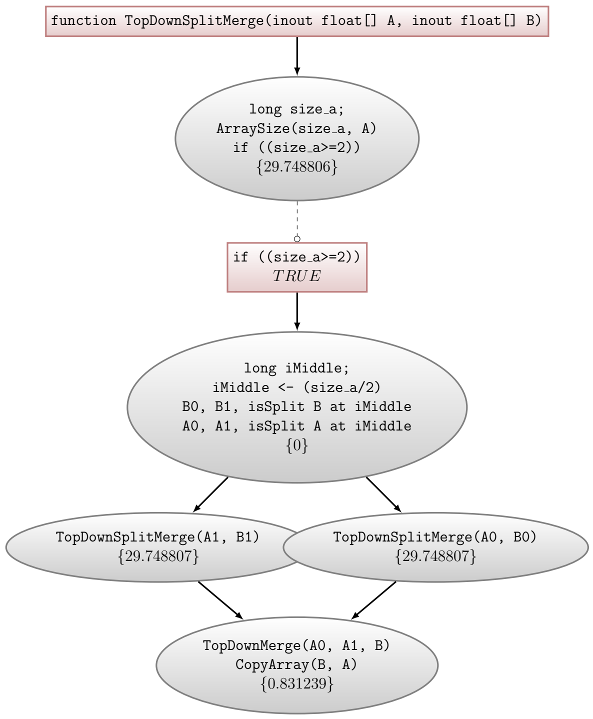

Figure 5.7: The resulting graph for the `TopDownSplitMerge` function

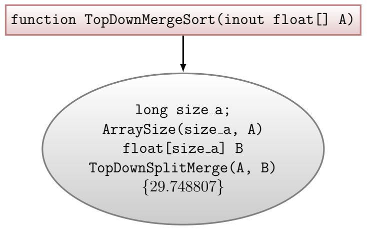

Figure 5.8: The resulting graph for the `TopDownMergeSort` function

Chapter 6

Discussion

We created a method to extract implicit parallelism from a program with a good balance between time spent on communication and time spent on computation. Our method improves on existing approaches, such as the one used in [5], as it is capable of considering more cases and may therefore find better parallel program implementations.

Our case study has shown that, for the tested program, the method is capable of generating an execution-order graph similar to what we expected a human programmer would design. Unfortunately, the lack of a sufficient sample programs prevents us from drawing any final conclusion about the effectiveness of this method. Similarly, while a greedy search algorithm appeared sufficient to produce correct results, this may not be the case for more complex programs.

We believe that this method, with some additions, may in the future be used to extract efficient implicit parallelism from programs automatically. This in turn could make the development of parallel implementations for these programs much easier.

One particular field that could benefit from this is game development. In video games performance is very important and the existence of multiple subsystems provides plenty of useful functional parallelism. Additionally, data-parallelism is often already very well exploited as GPUs often tend to be a system's bottleneck when running video games.

There are several improvements that could be made to our method, of which we consider two to be most relevant. First, it is currently unable to deal with data-parallelism and vector operations. Second, data-flow dependency graph extraction is not implemented for any useful languages. We will address these two issues in the next sections, followed by other, lesser issues.

6.1 Data-parallelism

Since data-parallel operations are a great source for parallelism, we want our method to be compatible with it. That means we must be able to somehow represent data-parallel operations in our graph to determine the critical path, even if we do not wish to merge them in any way.

Data-parallel operations are characterized by the fact that one function is applied to many data-points. As such, we could represent them as many nodes/tasks that are parallel to each other, e.g. as depicted in 6.1.

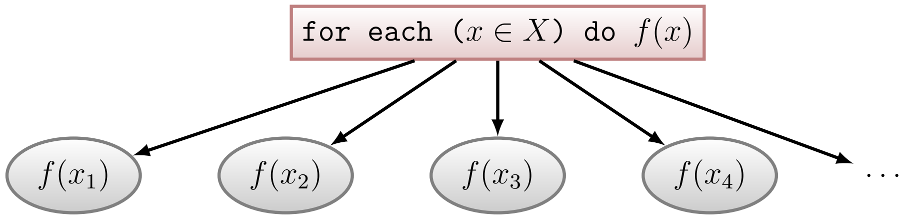

Figure 6.1: A possible data-parallel execution-order graph

In this case, we could merge the data-parallel nodes together until they reach the desired granularity, but this is something which can be done more effectively by merging tasks so that each new task consists of a fixed amount of initial tasks. How many tasks should be grouped can be easily determined as $n = \lceil \frac{s}{c(f)} \rceil$. This means we do not gain anything by this representation. Furthermore, it is not so trivially implemented if the number of tasks is not known at compile-time.

We propose instead, to represent parallel operations as a special type of task with additional attributes, similar to how some tasks now execute other graphs. We would then require new merge definitions for merging these data-parallel tasks together and with non-data-parallel tasks. For example, a merge could combine two for loops executing different operations on the same array into a single for loop that executes both operations.

Another problem with data-parallel operations is that the critical path will no longer be satisfactory: the cost of the critical path of a data-parallel operation is the cost for applying it to one of its elements, which is often tiny compared to the actual run time of the operation. Even if we score the task based on the total time of the operation, data-parallel operations are much more likely to be spread out across all processing units, leaving none for our other task and making the critical path less meaningful.

Considering the above, it would be easier if we only needed to implement special nodes for parallel tasks that will always run on specialized hardware such as GPGPUs or FPGAs, thereby not filling our 'main' processing units to capacity.

It might be the case that the critical path remains a useful fitness function, provided we use a good scheduler, but this is unknown and would require further experimentation.

6.2 Language requirements

For our implementation we required a very simple functionally pure, eagerly evaluated language. The language we implemented is too simplistic to be used directly. Furthermore, even though we could compile program from traditional languages, e.g. c, by simulating a stack and heap using arrays, this is not very useful as all operations using stack variables would depend on each other, even if they logically do not have to. Our method should therefore be implemented for a more complex language before it can really be useful.

Unfortunately most popular languages are not functionally pure and eagerly evaluated. We expect, in fact, the functional purity is often undesired in systems programming languages. During the development of the Rust language, for instance, functional purity was the default at some point, but this was later changed. One example where functional purity might get in the way is debugging, when programmers might want to perform IO operations to output certain details about the program's execution.

It is, however, overzealous to purge all impurities from a program for the glory of parallelism. It is sufficient that any impure operations be sanctioned by the programmer. That is, the programmer should mark them to denote that while they are indeed impure, the order in which they are executed is irrelevant. That is, they are *reorderable*. Examples of such operations include debug logging, random number generation and reductor operations, such as appending elements to an unordered list. By thus weakening the restriction of functional purity, languages that exploit implicit parallelism may be designed more easily.

6.3 Search strategy comparison

In our experiment we have limited ourselves to a greedy algorithm. This proved sufficient for our case study. However, greedy search algorithms are known to get stuck in local optima. More complex programs may therefore be optimized better using different algorithms. To this end, it would be beneficial to make a comparison of many such algorithms to identify which ones are most successful. These tests were well outside the scope of our research as it would require a large code-base, which we do not have, to produce accurate results.

6.4 Graph evaluation

Our fitness function is based on execution time of a sequential program, making the assumption that this will be representative of the tasks' execution time in a parallelized version. It is possible that different processing units share hardware, as may be the case when hyperthreading or by sharing a memory bus. This would make sequential speed a bad estimate for concurrent execution speed. Improvements on the fitness function based on knowledge about target architecture might therefore improve our method.

6.5 LVars

In [23] and [24], methods are discussed to partially run tasks with data-flow dependency to each other concurrently. Essentially the dependent task runs until it actually requires data from the provider, then waits until such data is made available by the provider. Exploiting this new type of dependency would require a (slightly) different fitness function, where the critical path is not computed based on the full cost of a task but only on the part required to write the LVar.

Chapter 7

Related Work

In this chapter we will place our research within the context of previous work in the field of automated or assisted program parallelization. Generally speaking, parallelization consists of two parts: identifying tasks and their dependencies and managing how these tasks are executed on the available hardware. Many approaches that focus on the second part, e.g. [13, 14, 15, 16], leave the first part to the programmer.

Cilk [12, 13], for instance, expands the `C` and `C++` languages so that a keyword can be used to denote that a function call may be executed on a different thread. It also sports keywords to denote certain vector and data-parallel operations. The runtime libraries then take care of distributing work over threads by using a work-stealing approach.

Although such methods remove the burden of making parallelism explicit and managing thread communication from the programmer, he must still split the program into parallel parts and ensure that these parts are sufficiently coarse-grained. This is partially due to the fact that they often use languages from which dependencies can not be trivially extracted. While it is possible to speculate on what the dependencies are by profiling memory accesses [10], this essentially still leaves the relevant decision-making to the programmer.

7.1 Coarseness-based parallelization decisions

There are multiple approaches that select which tasks are run in parallel based on their execution time. We list two representative ones and explain how they relate to and differ from our work.

In [5] researchers first profile programs written in Haskell to measure the execution time of various tasks. This part is similar to what we do, although their method only selects tasks, it does not aggregate them. The researchers then parallelize tasks using a work-stealing approach. If work is indeed stolen the parallelism has been exploited and was therefore useful. Our approach differs from this in that it does aggregate, potentially finding more promising opportunities for parallelism. We do not check if tasks are indeed run concurrently as the cost of creating a task that is not stolen is relatively low [13]. Furthermore, if the system is not saturated, our fitness function will perform a similar role by preferring high amounts parallelism when this matters, i.e. to reduce the critical path.

In [11], researchers make run-time decisions about whether to parallelize a `map` function, based on the size of its input data-structure and the costs of the function that is mapped. The method can therefore only detect parallelism expressed with a map function, as the cost-function for other tasks may scale differently with input size. Regardless, the idea of making run-time parallelization decisions based on task inputs is an interesting one as it potentially avoids communication overhead for tasks that turn out to be much smaller than we expected at compile-time.

7.2 Stream languages

The program execution-order graph created by our method is similar to how stream languages handle programming. An example of such a language is StreamIt [25]. In StreamIt, a program is expressed in terms of actors, executing a specific task, and FIFO data-channels between those actors. Since all communications between actors are made explicit through these channels, opportunities for parallelism can be extracted easily. In [26], the authors address problems concerning balancing communication and computation as well as reducing the critical path, similar to how we did. Stream languages however, may be too static for many applications [27].

Chapter 8

Conclusion

We have created a method capable of coarsening the granularity of fine-grained tasks through the iterative merging of tasks. This is an improvement on previous selection approaches in that it does not rely on coarse-grained tasks already explicitly existing in a program.

Our contributions are the following:

- We have defined three types of merge operations that will preserve necessary execution order restraints, thereby avoiding data-races.
- We have provided a fitness function and a stop criterion that can be used to iteratively merge tasks using intelligent/heuristic maximizing strategies.
- We implemented a parser, profiler and functions required to test such maximizing strategies.

In our case study with a merge-sort program, we have shown that iterative task merging, guided by greedy selection using our fitness function and stop condition, may produce execution-order graphs similar to what a human programmer might design. We may tentatively conclude that the method described could increase the automatically detectable, useful parallelism of a program. However, as we have used a single, relatively simple, test case, we do not know how our method will behave for larger programs. More complex programs may have local optima in which a greedy search algorithm will get stuck. Other search algorithms on the other hand may be more successful in terms of results, but might be much slower.

To make a final conclusion about the applicability of our method more work is required so that tests can be performed on a larger variety of programs.

Bibliography

[1] H Peter Hofstee. Future microprocessors and off-chip sop interconnect. *Advanced Packaging, IEEE Transactions on*, 27(2):301–303, 2004.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[3] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[4] Stijn Eyerman and Lieven Eeckhout. The benefit of smt in the multi-core era: Flexibility towards degrees of thread-level parallelism. *ACM SIGARCH Computer Architecture News*, 42(1):591–606, 2014.

[5] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ACM SIGPLAN Notices*, volume 42, pages 251–264. ACM, 2007.

[6] Paul Bone. *Automatic Parallelisation for Mercury.* PhD thesis, Department of Computing and Information Systems, The University of Melbourne, Australia, Decemeber 2012.

[7] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ACM SIGPLAN Notices*, volume 42, pages 251–264. ACM, 2007.

[8] Clemens Grelck and Sven-Bodo Scholz. Saca functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[9] Henri E Bal, Jennifer G Steiner, and Andrew S Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys (CSUR)*, 21(3):261–322, 1989.

[10] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM Sigplan Notices*, volume 44, pages 177–187. ACM, 2009.

[11] Brian Reistad and David K Gifford. *Static dependent costs for estimating execution time*, volume 7. ACM, 1994.

[12] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[13] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[14] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[16] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[17] Eugene Kligerman and Alexander D Stoyenko. Real-time euclid: A language for reliable real-time systems. *Software Engineering, IEEE Transactions on*, (9):941–949, 1986.

[18] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Proceedings of the 34th annual Design Automation Conference*, pages 147–152. ACM, 1997.

[19] Peter Puschner and Ch Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[20] Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 98–103. ACM, 2001.

[21] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 185–199. IEEE, 2000.

[22] Michael A Iverson, Fusun Ozguner, and Lee C Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 99–111. IEEE, 1999.

[23] Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. *Theory and Practice of Logic Programming*, 11(4-5):575–591, 2011.

[24] Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[25] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[26] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 151–162. ACM, 2006.

[27] Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Comparing the streamit and σc languages for manycore processors. In *Int. l Workshop on Data-Flow Models (DFM) for Extreme Scale Computing. IEEE*, 2014.

Appendices

Appendix A

Mergesort program

```
function TopDownMergeSort(out float[] A)
{
    var long size_a;
    ArraySize(size_a, A);
    var float[size_a] B;
    TopDownSplitMerge(A, B);
}

function TopDownSplitMerge(out float[] A, out float[] B
   )
{
    var long size_a;
    ArraySize(size_a, A);
    if(size_a >= 2) {
        var long iMiddle;
        iMiddle <- size_a / 2;
                var A0, A1 isSplit A at iMiddle;
                var B0, B1 isSplit B at iMiddle;
        TopDownSplitMerge(A0, B0);
        TopDownSplitMerge(A1, B1);
        TopDownMerge(A0, A1, B);
        CopyArray(B, A);
    }
}
```

```
function TopDownMerge(float[] A0, float[] A1, out float
   [] B)
{
    var long i0, i1, j, size_a0, size_a1,
       max_iterations;
    i0  <- 0;
    i1  <- 0;
    j   <- 0;
    ArraySize(size_a0, A0);
    ArraySize(size_a1, A1);
    max_iterations <- size_a0 + size_a1;
    while (j < max_iterations) {
        if (i0 < size_a0 && (i1 >= size_a1 || A0[i0] <=
            A1[i1])) {
            B[j]     <- A0[i0];
            i0       <- i0 + 1;
        } else {
            B[j]     <- A1[i1];
            i1       <- i1 + 1;
        }
        j <- j+1;
    }
}

function CopyArray(float[] B, out float[] A)
{
    var long i, size_a;
    ArraySize(size_a, A);
    i <- 0;
    while(i < size_a) {
        A[i]     <- B[i];
        i        <- i+1;
    }
}
```

Appendix B

Mergesort C++ program

B.1 Compiled file

```
double PROFILING_VARIABLE_AVERAGE_TopDownMergeSort;
unsigned long
    PROFILING_VARIABLE_NTESTS_TopDownMergeSort;
double PROFILING_VARIABLE_AVERAGE_TopDownSplitMerge;
unsigned long
    PROFILING_VARIABLE_NTESTS_TopDownSplitMerge;
double PROFILING_VARIABLE_AVERAGE_COND_0;
unsigned long PROFILING_VARIABLE_NTESTS_COND_0;
double PROFILING_VARIABLE_AVERAGE_TopDownMerge;
unsigned long PROFILING_VARIABLE_NTESTS_TopDownMerge;
double PROFILING_VARIABLE_AVERAGE_WHILE_0;
unsigned long PROFILING_VARIABLE_NTESTS_WHILE_0;
double PROFILING_VARIABLE_AVERAGE_COND_1;
unsigned long PROFILING_VARIABLE_NTESTS_COND_1;
double PROFILING_VARIABLE_AVERAGE_CopyArray;
unsigned long PROFILING_VARIABLE_NTESTS_CopyArray;
double PROFILING_VARIABLE_AVERAGE_WHILE_1;
unsigned long PROFILING_VARIABLE_NTESTS_WHILE_1;
myTimer PROFILING_TIMER;

void TopDownMergeSort(Array<float>& A);
void TopDownSplitMerge(Array<float>& A, Array<float>& B
    );
void TopDownMerge(Array<float> const A0, Array<float>
    const A1, Array<float>& B);
void CopyArray(Array<float> const B, Array<float>& A);

void PROFILER_INIT_VALUES() {
initTimer(&PROFILING_TIMER);
PROFILING_VARIABLE_AVERAGE_TopDownMergeSort = 0.0;
PROFILING_VARIABLE_NTESTS_TopDownMergeSort = 0;
PROFILING_VARIABLE_AVERAGE_TopDownSplitMerge = 0.0;
PROFILING_VARIABLE_NTESTS_TopDownSplitMerge = 0;
PROFILING_VARIABLE_AVERAGE_COND_0 = 0.0;
PROFILING_VARIABLE_NTESTS_COND_0 = 0;
PROFILING_VARIABLE_AVERAGE_TopDownMerge = 0.0;
PROFILING_VARIABLE_NTESTS_TopDownMerge = 0;
PROFILING_VARIABLE_AVERAGE_WHILE_0 = 0.0;
PROFILING_VARIABLE_NTESTS_WHILE_0 = 0;
PROFILING_VARIABLE_AVERAGE_COND_1 = 0.0;
PROFILING_VARIABLE_NTESTS_COND_1 = 0;
PROFILING_VARIABLE_AVERAGE_CopyArray = 0.0;
PROFILING_VARIABLE_NTESTS_CopyArray = 0;
PROFILING_VARIABLE_AVERAGE_WHILE_1 = 0.0;
PROFILING_VARIABLE_NTESTS_WHILE_1 = 0;
}
void TopDownMergeSort(Array<float>& A){
myTimestamp
    PROFILING_VARIABLE_TIMER_TopDownMergeSort_start;
myTimestamp
    PROFILING_VARIABLE_TIMER_TopDownMergeSort_stop;
getTime(&
    PROFILING_VARIABLE_TIMER_TopDownMergeSort_start);
long size_a;
ArraySize(size_a, A);
Array<float> B = Array<float>(size_a);
TopDownSplitMerge(A, B);
getTime(&PROFILING_VARIABLE_TIMER_TopDownMergeSort_stop
    );
PROFILING_VARIABLE_AVERAGE_TopDownMergeSort=
    updateTimer(timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_TopDownMergeSort_start,
    PROFILING_VARIABLE_TIMER_TopDownMergeSort_stop,
    PROFILING_TIMER),
    PROFILING_VARIABLE_AVERAGE_TopDownMergeSort, ++
    PROFILING_VARIABLE_NTESTS_TopDownMergeSort);
}
void TopDownSplitMerge(Array<float>& A, Array<float>& B
    ){
myTimestamp
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_start;
myTimestamp
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_stop;
getTime(&
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_start);
long size_a;
ArraySize(size_a , A);
myTimestamp PROFILING_VARIABLE_TIMER_COND_0_start;
myTimestamp PROFILING_VARIABLE_TIMER_COND_0_stop;
getTime(&PROFILING_VARIABLE_TIMER_COND_0_start);
if ((size_a>= 2 )) {
long iMiddle;
iMiddle = (size_a/ 2 );
Array<float> A0;
Array<float> A1;
SplitArrayMemory(A0, A1, A, iMiddle);
Array<float> B0;
Array<float> B1;
SplitArrayMemory(B0, B1, B, iMiddle);
TopDownSplitMerge(A0, B0);
TopDownSplitMerge(A1, B1);
TopDownMerge(A0, A1, B);
CopyArray(B, A);
} else {
}
getTime(&PROFILING_VARIABLE_TIMER_COND_0_stop);
PROFILING_VARIABLE_AVERAGE_COND_0= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_COND_0_start,
    PROFILING_VARIABLE_TIMER_COND_0_stop,
    PROFILING_TIMER), PROFILING_VARIABLE_AVERAGE_COND_0,
     ++PROFILING_VARIABLE_NTESTS_COND_0);
getTime(&
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_stop);
PROFILING_VARIABLE_AVERAGE_TopDownSplitMerge=
    updateTimer(timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_start,
    PROFILING_VARIABLE_TIMER_TopDownSplitMerge_stop,
    PROFILING_TIMER),
    PROFILING_VARIABLE_AVERAGE_TopDownSplitMerge, ++
    PROFILING_VARIABLE_NTESTS_TopDownSplitMerge);
}
void TopDownMerge(Array<float> const A0, Array<float>
    const A1, Array<float>& B){
myTimestamp PROFILING_VARIABLE_TIMER_TopDownMerge_start
    ;
myTimestamp PROFILING_VARIABLE_TIMER_TopDownMerge_stop;
getTime(&PROFILING_VARIABLE_TIMER_TopDownMerge_start);
long i0, i1, j, size_a0, size_a1, max_iterations;
i0 =  0 ;
i1 =  0 ;
j =  0 ;
ArraySize(size_a0, A0);
ArraySize(size_a1, A1);
max_iterations = (size_a0+size_a1);
myTimestamp PROFILING_VARIABLE_TIMER_WHILE_0_start;
myTimestamp PROFILING_VARIABLE_TIMER_WHILE_0_stop;
getTime(&PROFILING_VARIABLE_TIMER_WHILE_0_start);
while ((j<max_iterations)) {
myTimestamp PROFILING_VARIABLE_TIMER_COND_1_start;
myTimestamp PROFILING_VARIABLE_TIMER_COND_1_stop;
getTime(&PROFILING_VARIABLE_TIMER_COND_1_start);
if (((i0<size_a0)&((i1>=size_a1)|(A0[i0]<=A1[i1])))) {
B[j] = A0[i0];
i0 = (i0+ 1 );
} else {
```

```
B[j] = A1[i1];
i1 = (i1+ 1 );
}
getTime(&PROFILING_VARIABLE_TIMER_COND_1_stop);
PROFILING_VARIABLE_AVERAGE_COND_1= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_COND_1_start,
    PROFILING_VARIABLE_TIMER_COND_1_stop,
    PROFILING_TIMER), PROFILING_VARIABLE_AVERAGE_COND_1,
     ++PROFILING_VARIABLE_NTESTS_COND_1);
j = (j+ 1 );
}
getTime(&PROFILING_VARIABLE_TIMER_WHILE_0_stop);
PROFILING_VARIABLE_AVERAGE_WHILE_0= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_WHILE_0_start,
    PROFILING_VARIABLE_TIMER_WHILE_0_stop,
    PROFILING_TIMER), PROFILING_VARIABLE_AVERAGE_WHILE_0
    , ++PROFILING_VARIABLE_NTESTS_WHILE_0);
getTime(&PROFILING_VARIABLE_TIMER_TopDownMerge_stop);
PROFILING_VARIABLE_AVERAGE_TopDownMerge= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_TopDownMerge_start,
    PROFILING_VARIABLE_TIMER_TopDownMerge_stop,
    PROFILING_TIMER),
    PROFILING_VARIABLE_AVERAGE_TopDownMerge, ++
    PROFILING_VARIABLE_NTESTS_TopDownMerge);
}
void CopyArray(Array<float> const B, Array<float>& A){
myTimestamp PROFILING_VARIABLE_TIMER_CopyArray_start;
myTimestamp PROFILING_VARIABLE_TIMER_CopyArray_stop;
getTime(&PROFILING_VARIABLE_TIMER_CopyArray_start);
long i, size_a;
ArraySize(size_a, A);
i =  0 ;
myTimestamp PROFILING_VARIABLE_TIMER_WHILE_1_start;
myTimestamp PROFILING_VARIABLE_TIMER_WHILE_1_stop;
getTime(&PROFILING_VARIABLE_TIMER_WHILE_1_start);
```

```
while ((i<size_a)) {
A[i] = B[i];
i = (i+ 1 );
}
getTime(&PROFILING_VARIABLE_TIMER_WHILE_1_stop);
PROFILING_VARIABLE_AVERAGE_WHILE_1= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_WHILE_1_start,
    PROFILING_VARIABLE_TIMER_WHILE_1_stop,
    PROFILING_TIMER), PROFILING_VARIABLE_AVERAGE_WHILE_1
    , ++PROFILING_VARIABLE_NTESTS_WHILE_1);
getTime(&PROFILING_VARIABLE_TIMER_CopyArray_stop);
PROFILING_VARIABLE_AVERAGE_CopyArray= updateTimer(
    timeDiffSeconds(
    PROFILING_VARIABLE_TIMER_CopyArray_start,
    PROFILING_VARIABLE_TIMER_CopyArray_stop,
    PROFILING_TIMER),
    PROFILING_VARIABLE_AVERAGE_CopyArray, ++
    PROFILING_VARIABLE_NTESTS_CopyArray);
}

void PROFILER_REPORT_VALUES() {
printf("{");
printf("\"TopDownMergeSort\" : %f,",
    PROFILING_VARIABLE_AVERAGE_TopDownMergeSort);
printf("\"TopDownSplitMerge\" : %f,",
    PROFILING_VARIABLE_AVERAGE_TopDownSplitMerge);
printf("\"COND_0\" : %f,",
    PROFILING_VARIABLE_AVERAGE_COND_0);
printf("\"TopDownMerge\" : %f,",
    PROFILING_VARIABLE_AVERAGE_TopDownMerge);
printf("\"WHILE_0\" : %f,",
    PROFILING_VARIABLE_AVERAGE_WHILE_0);
printf("\"COND_1\" : %f,",
    PROFILING_VARIABLE_AVERAGE_COND_1);
printf("\"CopyArray\" : %f,",
    PROFILING_VARIABLE_AVERAGE_CopyArray);
printf("\"WHILE_1\" : %f,",
    PROFILING_VARIABLE_AVERAGE_WHILE_1);
printf("\"pointless_variable\":0.0}");
}
```

B.2 main file

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "../src/timer.h"
#include "../src/natives.hpp"
#if PROFILE
#include "mergesort_profiler.cpp"
#else
#include "mergesort_compiled.cpp"
#endif //PROFILE

int main(int argc, char* args[]) {
        unsigned long i = 0;
        time_t t;
        srand((unsigned) time(&t));

#if PROFILE
        PROFILER_INIT_VALUES();
#endif
        Array<float> array(NUM_ELEMENTS);
        for (i = 0; i < NUM_ELEMENTS; ++i) {
                array[i] = (float)rand()/(float)(
                    RAND_MAX);
        }
        TopDownMergeSort(array);
#if PROFILE
        PROFILER_REPORT_VALUES();
#endif
}
```

B.3 library files

```
#ifndef NATIVES_HPP
#define NATIVES_HPP

template<typename Element>
class Array {
        unsigned long n_elems;
        Element* data;

        public:
        Array() {
                n_elems = 0;
        }
        Array(unsigned long _n_elems) {
                n_elems = _n_elems;
                data = new Element[_n_elems];
        }

        inline Element& operator[](unsigned long index)
             {
                return data[index];
        }

        inline Element operator[](unsigned long index)
           const{
                return data[index];
        }

        template<typename __Element, typename
           NumberType>
        friend void ArraySize(NumberType&, Array<
           __Element>);
        template<typename __Element>
        friend void SplitArrayMemory(Array<__Element>&
           p1, Array<__Element>& p2, Array<__Element>
           parent, unsigned long splitIndex);

};
```

```
template<typename Element, typename NumberType>
inline void ArraySize(NumberType& out_size, Array<
   Element> in_array) {
         out_size = in_array.n_elems;
}

template<typename Element>
inline void SplitArrayMemory(Array<Element>& p1, Array<
   Element>& p2, Array<Element> parent, unsigned long
   splitIndex) {
         p1.n_elems = splitIndex;
         p1.data = parent.data;
         p2.n_elems = parent.n_elems - splitIndex;
         p2.data = parent.data + splitIndex;
}

#endif //NATIVES_HPP
```