Task Scheduling for Heterogeneous Architectures

Nuno Miguel Nobre

Abstract—Task scheduling on heterogeneous (and homogeneous) architectures has been long influenced by data flow principles. However, current programming models require the user to manually specify the data dependences and, by extension, to determine task granularity. Hence, this report motivates future research on trying to circunvent the burdens currently posed to users of such programming models, whilst still preserving the known advantages of data flow inspired task scheduling.

1 Introduction

Current computing systems usually exploit multiple processors working in parallel to (ideally) enhance the performance of applications. However, such systems are increasingly restricted by energy efficiency constraints [1]. As such, new heterogeneous multi-core architectures featuring distinct types of processing cores designed to target different performance and power optimization points have become common practice. The advent of ARM's big.LITTLE architecture [2] and its most recent iteration, ARM's DynamIQ [3], are two such examples. Coupling this reality with the increasing interest in hardware accelerator-based computation [4] using General Purpose Graphics Processing Units (GPGPUs), Field-Programmable Gate Arrays (FPGAs) or Digital Signal Processors (DSPs), makes the efficient scheduling of the tasks of an application on such a range of available resources simultaneously a hard problem and a crucial aspect in the quest for high performance or, if need be, for the right balance between performance and energy efficiency.

By scheduling we usually mean not only assigning the tasks, i.e., the (usually programmer-defined) application's units of work, to the appropriate resources, but also ordering task execution on each of those resources [5]. Moreover, as program correctness should not be violated, the mapping and ordering must guarantee that inter-task dependences are, ultimately[1], respected. This raises the problem of representing a program in a manner that not only makes these task-precedence requirements explicit but also lends itself to the application of a scheduling algorithm. Therefore, this report starts by exploring some program representation models that expose dependences in different forms (Section 2). Then, it describes how the principles behind one of those models, data flow graphs, are currently being used to implement task schedulers and what the current burdens are for the users willing to use them (Section 3).

2 Program representation models

Determining how a given task depends on another is critical to determining the degree of concurrency, i.e., the number of tasks that can execute in parallel, and how that concurrency can be exploited. First, however, it is important to clarify that tasks can be decomposed into basic blocks, each of which is a sequence of instructions with no branch instructions, except possibly the last one, and no branch targets, except perhaps the initial one. Each basic block may correspond to a straight-line sequence of statements (source-level statements or intermediate language statements that are subsequently compiled to one or more instructions), a single statement or even an individual instruction [6]. Fig. 1 depicts this decomposition.

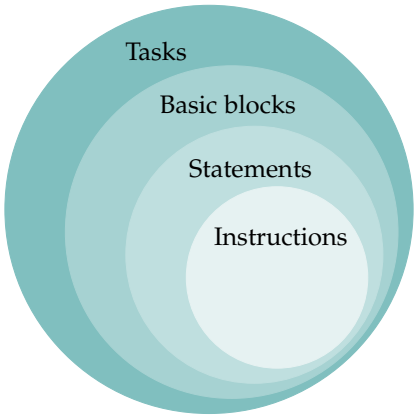

Fig. 1: Tasks are composed of one or more basic blocks. In turn, these are sequences of one or more statements which, finally, are compiled into one or more machine-level instructions.

There are two main types of dependences: control dependences and data dependences. The former determine the ordering of a basic block[2] with respect to a branch instruction in another basic block so that the first basic block is executed in correct program order and only when it should be [7]. On the other hand, data dependences can be further split into three categories:

- Flow dependences, also known as true data dependences or read-after-write (RAW) dependences, occur when a variable is assigned or defined in one statement and used in a subsequently executed statement.

N. M. Nobre (nunomiguel.nobre@postgrad.manchester.ac.uk) is with the School of Computer Science, The University of Manchester, Oxford Road, Manchester, UK, M13 9PL.

1. In the sense that if, for example, speculative execution is allowed, we may execute code that should not have been executed, thereby violating, even if momentaneously, the dependences of the program.

2. Note that this can still be a fine-grain block, e.g. a single instruction.

- Anti-dependences, also known as write-after-read (WAR) dependences, happen when a variable is used and subsequently reassigned.
- Output dependences, otherwise known as write-after-write (WAW) dependences, arise when a variable is assigned and subsequently reassigned.

The last two are also commonly termed false or name dependences given that they can be eliminated by appropriate variable renaming [6].

2.1 Polyhedral model

The polyhedral model is an algebraic representation of programs now commonly used for automatic optimization and parallelization [8]. Its applicability has been historically limited to the Static Control Parts (SCoPs) of programs. These are a subset of general loop nests where both loop bounds and conditional branches are affine functions of the surrounding loop iterators and of constant parameters whose values are unknown at compile time. Given these constraints, the loop's iteration space can then be described by a set of affine inequalities:

$$P = \{\mathbf{x} \,|\, \mathbf{x} \in \mathbb{Z}, \boldsymbol{A}\mathbf{x} + \mathbf{a} \geq 0\}, \tag{1}$$

where $\mathbf{x}$ is the iteration vector and $\boldsymbol{A}$ and $\mathbf{a}$ are a constant matrix and vector, respectively. The polyhedron P is then coupled with a set of data dependence equations, and the resulting dependence system of equalities and inequalities can then be solved, even if just approximately (and, thus, conservatively), with a dependence system solver [6]. Later, some algorithm is used to restructure the program based on the data dependence analysis for arrays performed in the previous step. The outcome is a change of variables to be applied to the iteration space [8]. Finally, a high level representation is again obtained in the code generation step [8], [9].

It is important to note that although these methods were initially aimed for regular access patterns, there has been some work in extending them to more general pieces of code [9]. However, despite these efforts directed graphs remain the preferred way of representing programs with general, potentially irregular, array accesses. For this reason, they are our focus from this point onwards.

2.2 Directed graphs

There are several directed graphs that can abstract various types of information about a program. A graph G consists of a set of vertices (also nodes or points) V and a set of edges E, each of which is an ordered pair of vertices. In more formal words: $G = \langle V, E\rangle$ where $E \subseteq V \times V$. In a directed graph, the edges have a direction and, as such, $\langle X, Y\rangle \in E \implies \langle Y, X\rangle \in E$ where $X, Y \in V$.

2.2.1 Control flow graph

In a Control Flow Graph (CFG), edges represent potential flow of control between basic blocks which are represented by vertices. A conditional branch is denoted by a vertex with two or more successors. Additionally, the CFG has a single source vertex from which all other vertices are reachable and a single sink vertex reachable from all other vertices. An example is given in Fig. 2 for the following C program which computes an approximation for the integral of the function $y = x^2$ over the interval $[0, 1]$ using the trapezoidal rule.

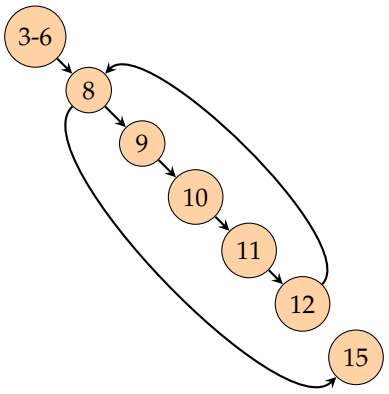

Fig. 2: Control flow graph for the numerical integration code given in the text. Each vertex corresponds to the statement(s) with the same numerical label(s). Notice how vertex 8 corresponds to a conditional branch.

```
 1 double integrate()
 2 {
 3   double old_y = 0.0;
 4   double x = 0.0;
 5   double y = 0.0;
 6   double s = 0.0;
 7
 8   while (x < 1.0){
 9     old_y = x * x;
10     x = x + 0.02;
11     y = x * x;
12     s = s + 0.01 * (old_y + y);
13   }
14
15   return s;
16 }
```

2.2.2 Control dependence graph

The Control Dependence Graph (CDG) is, as its name would suggest, used to represent the control dependence relationships between any two basic blocks. It can be derived from the CFG but, unlike the latter which enforces a fixed sequencing of operations, it only determines necessary sequencing, exposing potential parallelism [10]. The CDG has the same vertices as the CFG but only adds an edge $X \to Y$ whenever vertex Y is control dependent on vertex X. Formally, this happens when Y postdominates[3] some successor of X but does not strictly postdominate X [6]. An example is given in Fig. 3 for the code given earlier.

2.2.3 Data dependence graph

A Data Dependence Graph (DDG) is the equivalent of the CDG for the data dependences defined at the beginning of Section 2. Fig. 4 depicts such a graph for the numerical integration code. Clearly, this is a multigraph, i.e., one where two vertices might be connected by several edges.

3. In a CFG, a vertex Y is said to (strictly) postdominate X if ($Y \neq X$ and) every path from X to the sink node includes Y.

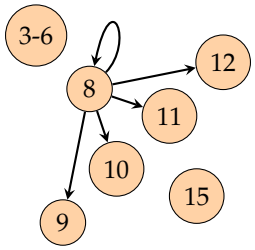

Fig. 3: Control dependence graph for the numerical integration code given in the text. Each vertex corresponds to the statement(s) with the same numerical label(s). Notice how vertices 3-6 and 15 could be executed in parallel, barring data dependences. In fact, they execute unconditionally whenever the program executes.

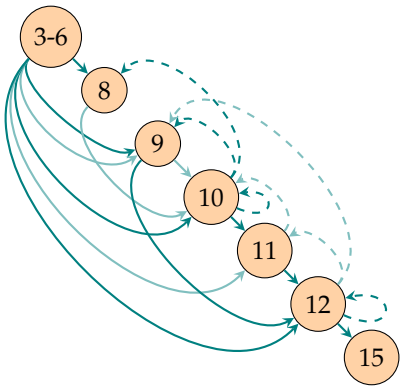

Fig. 4: Data dependence graph for the numerical integration code given in the text. Each vertex corresponds to the statement(s) with the same numerical label(s). Darker edges denote flow dependences, lighter edges designate both types of name dependences and dashed edges represent loop carried data dependences from one iteration to the next.

For the purposes of this report, where human readability is a priority, we assumed the loop in the code is executed at least once and used a value-based approach to data dependences in order to decrease the number of edges in the graph. This contrasts with an address-based approach which would, for example, make vertex 15 in Fig. 4 flow dependent on vertex 3-6. Using the value-based definition, statement 12 effectively "kills" such dependence. The attentive reader will also notice that whenever there is a chain of dependences such that statement j is flow dependent on statement i and statement k is anti-dependent on statement j, the resulting output dependence between i and k is omitted.

2.2.4 Program dependence graph

The Program Dependence Graph (PDG), pioneered by Ferrante et al. [11] to aid in compiler optimization techniques, simply adds the control dependence relations to the same graph as the data dependence relations. Thus, this model captures both the data and control precedence constraints thereby ensuring program correctness. The example depicted in Fig. 5 is simply the result of merging the CDG and the DDG of Figs. 3 and 4, respectively.

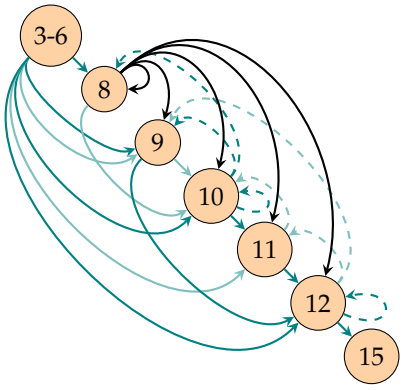

Fig. 5: Program dependence graph for the numerical integration code given in the text. Each vertex corresponds to the statement(s) with the same numerical label(s). This PDG results from adding the black edges denoting control dependences in Fig. 3 to the DDG in Fig. 4.

2.2.5 Data flow graph

In a Data Flow Graph (DFG), edges are limited to, as the name implies, data flow dependences. Name dependences are, by construction, non-existent, and control dependences are cast into flow dependences as necessary. Dennis' efforts [12] laid the groundwork in the area of data flow computation and, among others, led to the work of Gurd et al. [13] developed at the University of Manchester in the early 1980s. The Manchester data flow computer implements a data flow architecture, an alternative to the classic, control-driven von Neumann architecture, in the sense that instructions do not reference memory cells: data is allowed to be transmitted directly from generating instruction to the subsequent (consuming) instruction(s) [14]. Execution then commences as soon as all required inputs for an instruction have arrived rather than depending on separate control signals.

Data flow programs are most naturally written in a single-assignment language like SISAL [15], [16]. This dictates that each variable be assigned only once, thereby eliminating naming dependences. Additionally, unlike usual imperative languages like C, it supports nonsequential semantics by offering concurrent control constructs. The following is equivalent to the numerical integration code given earlier, now written in SISAL.

```
 1 function integrate(returns double_real)
 2
 3   for initial
 4     s := 0.0;
 5     y := 0.0;
 6     x := 0.0;
 7   while x < 1.0 repeat
 8     s := 0.01 * (old y + y);
 9     y := x * x;
10     x := old x + 0.02;
11   returns value of sum s
12   end for
13
14 end function
```

Notice how statement 8 uses a value defined in statement 9 and how the `for initial` loop allows each iteration to depend on variables defined in the previous iteration, e.g. `old x` gives the value of `x` in the previous iteration. The machine code generated after compilation can be represented graphically as (partially) shown in Fig. 6 for the Manchester data flow computer.

It is interesting to observe how the control dependence associated with the branch (BRR) in statement 7 was cast into data flow dependences into BRR itself, which only executes when both the comparison (CGR) result and the value of `x` from the same iteration are available; and into the adding instruction (ADR), which only begins executing when the current value of `x` is passed onto it by BRR. This happens whenever CGR outputs `True`, i.e., `1.0 > x`. The single-assignment property of SISAL is translated here in terms of tags. These effectively allow the coexistence of different versions of the same variable for different iterations of the `for initial` loop, thereby eliminating name dependences.

2.2.6 *Task dependence graph*

One striking characteristic of the graphs covered so far is that although they are all directed, none is necessarily acyclic. There is a tendency [5], [17]–[19] however, to model an application's Task Dependence Graph (TDG) as a Directed Acyclic Graph (DAG) where vertices represent the application's tasks and edges represent intertask dependences. The DAG formulation has the advantage of avoiding complications of reentrant code like dealing with operands from different loop iterations or having to keep track of recursions. In fact, it has been shown that if each vertex's task is only allowed to be executed exactly once, a program is feasible[4] if and only if its graph representation is acyclic [20].

In order to obtain this DAG, we can either generalise a PDG or a DFG, given that these are the only graph models presented here that can capture both control and data dependences simultaneously. Then, the graph is made acyclic either by loop (or other form of reentrant code) unrolling/unfolding [21] or by choosing the task granularity (see Section 3.2) coarse enough to hide these structures inside the vertices themselves [20]. As this process potentially involves encapsulating control structures inside the nodes, it is customary to simply resort to a data flow approach which casts the remaining control dependences as flow dependences (e.g. as described by Banerjee et al. [22]), with the added benefit of eradicating name dependences. Often, however, the difference between the three data dependence types (i.e. flow dependences, anti-dependences and output dependences) is not relevant. In such cases, a TDG can be defined, whose edges can represent all types of data dependences. In short, in the literature, a TDG can be shaped very similarly to an unrolled DDG or DFG [20].

3 Task Scheduling

The problem of scheduling a TDG of a program onto a (possibly heterogeneous) computing system is a well-defined NP-complete and, thus, NP-hard problem, considered one of the most challenging problems in parallel computing [23]. It involves mapping a DAG for a set of computational tasks while ensuring that precedence requirements for these tasks are satisfied and, at the same time, the overall execution length is minimized [5], [19].

4. A program is said to be feasible if and only if a task order can be found that complies with the precedence constraints of the dependence relations [20].

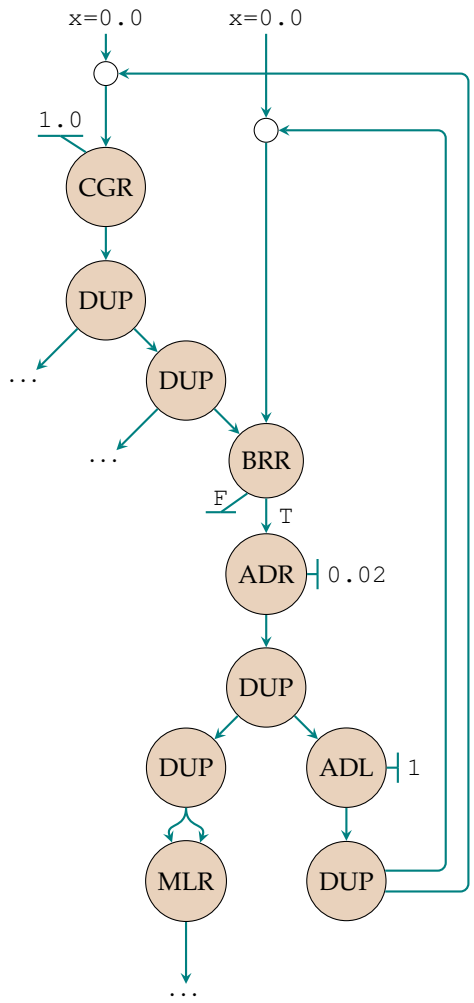

Fig. 6: Partial data flow graph for the numerical integration code given in the text. Only the fraction of the graph involving the flow of variable `x` is shown. Each vertex corresponds to machine-level instructions, as follows: ADL - add to iteration level (tag), ADR - add floating point values, BRR - branch, CGR - compare floating point values, DUP - duplicate value (to work around the specific fan-out limitation for each instruction in the Manchester system), MLR - multiply floating point values. Adapted from Gurd et al. [13].

3.1 Static vs. dynamic scheduling

When the characteristics of an application, including the execution time of individual tasks, communications costs between tasks and task dependences are known a priori, usually by relying on profiling information, the scheduling problem is solved statically, at compile time [17].

On the other hand, dynamic schedulers use information that is discoverable at runtime and take scheduling deci-

sions on-the-fly [19]. These have the advantage of being able to deal with dependence indeterminism as in cases where array indexing is determined by user input or by the result of a random generator, or where pointer aliasing can not be resolved at compile time [10]. In addition, the execution environment conditions might change at runtime, either because the resources are shared by other applications or an operating system or, even in the case of dedicated resources, by virtue of technologies like Dynamic Voltage and Frequency Scaling (DVFS) [7]. This changes the frequency and/or operating voltage of a processor based on system performance and energy efficiency requirements or power dissipation limitations as specified by the Thermal Design Power (TDP) of each component [7]. Evidently, however, the ability to take decisions at runtime is only an advantage as long as the time consumed in deliberating is balanced out by appreciable performance improvements otherwise unachievable by a static scheduler.

3.2 Scheduling algorithms and programming models

The solution to the NP-hard task scheduling problem is often approximated by resorting to heuristic approaches [17]. These are classified into a variety of categories such as list schedulers (e.g. Heterogeneous Earliest Finish Time (HEFT) [17] and Levelized-Min Time (LMT) [24]), clustering schedulers (e.g. Clustering and Scheduling System (CASS) [25]) and task duplication schedulers (e.g. Critical Path Fast Duplication (CPFD) [26] and Heterogeneous Economical Duplication (HED) [27]). Alternatively, guided random search techniques, specially in the form of Genetic Algorithms (GAs) [28], have also been used. Nevertheless, for the programming models supporting heterogeneous architectures such as OmpSs [18] and OpenStream [29], list schedulers remain the algorithms of choice mainly for their simplicity and overall superior performance [17].

Both OmpSs and OpenStream rely on programmer annotations to specify the data flow between OpenMP tasks and to build the program TDG. When compared to the standard OpenMP programming model [30], this approach has one big advantage. In fact, it unlocks the possibility of asynchronous parallelism by avoiding the explicit definition of synchronized regions (i.e. regions of code delimited by synchronization points like barriers) that could prevent the parallelization of tasks with no data conflicts [10]. In reality, the required synchronization between tasks is achieved by delaying execution until all their data prerequisites are satisfied just like we have seen before for the DFG (see Section 2.2.5 and Fig. 6). Nonetheless, by demanding the user to manually specify the dependences not only do we make the programming process harder and less intuitive, but also more error prone. This happens both as a result of errors while specifying the data dependences and because these programming models, by requiring the users to annotate tasks, implicitly require them to delimit those tasks and, thus, to make the right choices regarding task granularity. Now, although it does not affect program correctness, this decision between a large number of small, fine-grained tasks or a small number of large, coarse-grained tasks is an essential one as it influences load balancing and communication costs between processors but also the maximum degree of concurrency[5]. In fact, it implies a compromise: although load balance and concurrency improve with fine-grained tasks, the opposite happens with the communications costs.

4 Conclusions and Further Work

Clearly the latest programming models supporting asynchronous parallelism and heterogeneous architectures require the user to make the right decisions regarding task dependences and granularity. However, the user is often not interested in such intricacies as properly informed decisions might require deviating the focus from its area of expertise to elaborate programming and hardware issues. As such, the question is whether it is possible to achieve the same parallelization benefits that data flow programming models seem to give users, while still providing intuitive language constructs that do not obfuscate the underlying program's (perhaps purely scientific/mathematical) purpose.

In order to continue pursuing an answer to this problem, we envisage the following roadmap:

- Improve our understanding of the scheduling algorithms used nowadays (see Section 3.2), including to what extent they influence task granularity and rely on users' annotations and the underlying program representation.
- Explore compiler and runtime system techniques for automatic discovery of task dependences (see the compiler-focused developments made by Royuela et al. [10] regarding task dependences in OmpSs).
- Explore methods to determine the optimal task granularity with little or no user intervention (see Thoman et al. [31] for a suitable starting point).
- Use OpenStream's programming model as a practical test-bed for the studies outlined in the previous points.

The Gantt chart depicted in Fig. 7 is an attempt at mapping these objectives into a time schedule for the following three years. Note that it exposes a strong overlap between theoretical and experimental (OpenStream related) methods in the hope that experimental results can be explained by the theory behind them and, conversely, that experimental results can help testify the pertinence of newly developed theoretical techniques. In addition, other required components like posters, seminars and reports whose schedule is already established are also shown.

5. Largest number of concurrent tasks at any point of the execution.

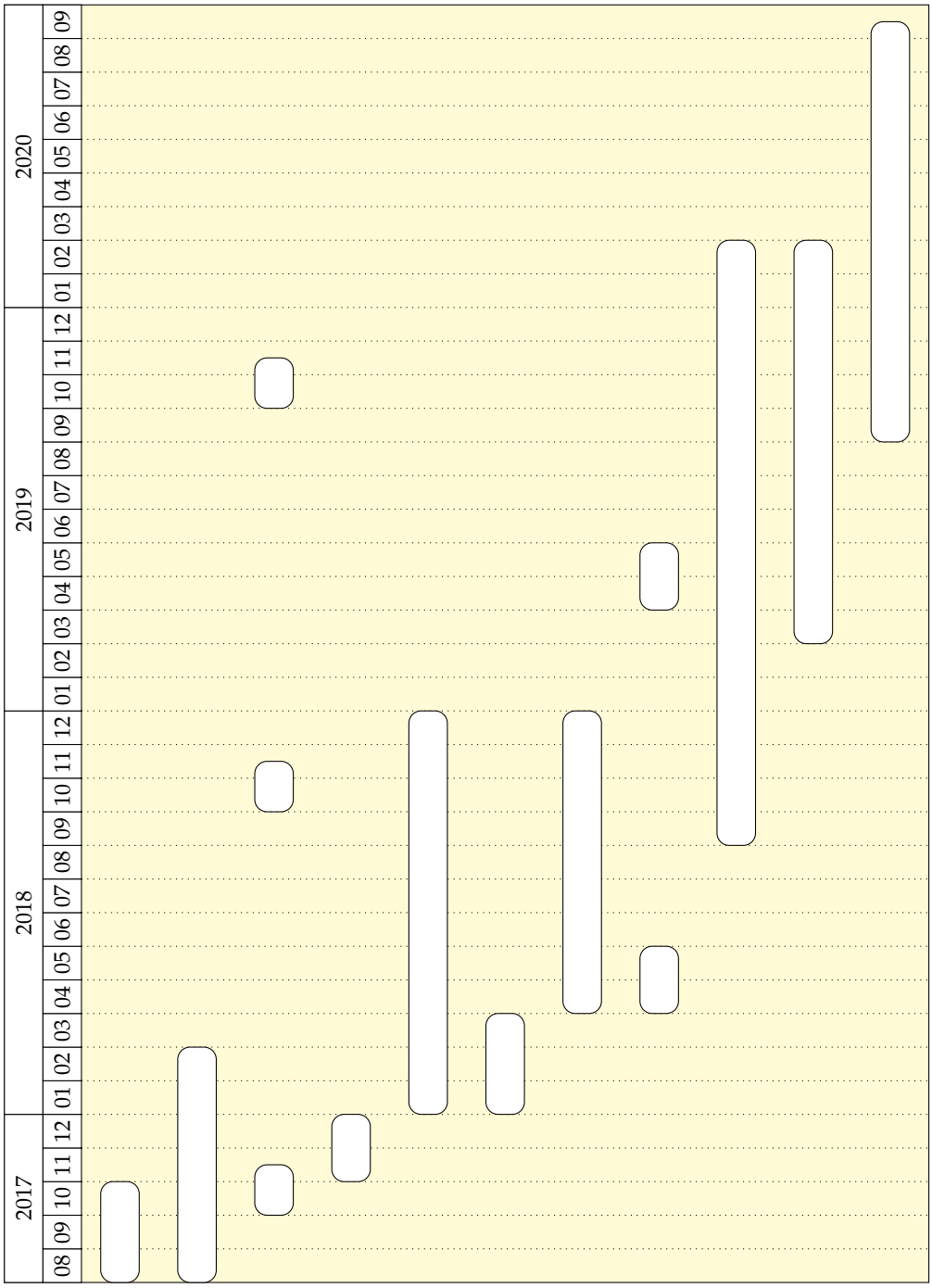

Fig. 7: Gantt chart depicting a time plan for the remaining three years of the author's project.

REFERENCES

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008.

[2] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.

[3] G. Wathan. (2017, March) ARM DynamIQ: Technology for the next era of compute. [Online]. Available: https://community.arm.com/processors/b/blog/posts/arm-dynamiq-technology-for-the-next-era-of-compute

[4] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Proceedings of the 2008 Symposium on Application Specific Processors*, ser. SASP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 101–107. [Online]. Available: http://dx.doi.org/10.1109/SASP.2008.4570793

[5] Y. Jiang, Z. Shao, and Y. Guo, "A DAG Scheduling Scheme on Heterogeneous Computing Systems Using Tuple-Based Chemical Reaction Optimization," *The Scientific World Journal*, vol. 2014, 2014. [Online]. Available: http://dx.doi.org/10.1155/2014/404375

[6] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[8] P. Feautrier, "The polytope model, past, present, future," in *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*, Newark, Delaware, 2009.

[9] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11970-5_16

[10] S. Royuela, A. Duran, and X. Martorell, *Compiler Automatic Discovery of OmpSs Task Dependencies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 234–248. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37658-0_16

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[12] J. B. Dennis, *First version of a data flow procedure language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 362–376. [Online]. Available: http://dx.doi.org/10.1007/3-540-06859-7_145

[13] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985. [Online]. Available: http://doi.acm.org/10.1145/2465.2468

[14] P. C. Treleaven, R. P. Hopkins, and P. W. Rautenbach, "Combining data flow and control flow computing," *The Computer Journal*, vol. 25, no. 2, p. 207, 1982. [Online]. Available: http://dx.doi.org/10.1093/comjnl/25.2.207

[15] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1.1*, Jul 1983.

[16] L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao, *Arrays, Functional Languages, and Parallel Systems*. Boston, MA: Springer US : Imprint : Springer,, 1991. [Online]. Available: SpringerLinkhttp://dx.doi.org/10.1007/978-1-4615-4002-1MITAccessOnly

[17] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/71.993206

[18] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 329–338. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751235

[19] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 13 – 22, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731509001804

[20] O. Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[21] K. K. Parhi, "Algorithm transformation techniques for concurrent processors," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1879–1895, Dec 1989.

[22] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, Feb 1993.

[23] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[24] M. A. Iverson, F. Özgüner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," in *4th Heterogeneous Computing Workshop (HCW '95)*, 1995, pp. 93–100.

[25] J. Liou and M. A. Palis, "An efficient task clustering heuristic for scheduling dags on multiprocessors," in *Proc. Symp. Parallel and Distributed Processing*, 1996, pp. 152–156.

[26] I. Ahmad and Y. K. K. Y.-K. Kwok, "A new approach to scheduling parallel programs using task duplication," in *1994 Internatonal Conference on Parallel Processing Vol. 2*, vol. 2, Aug 1994, pp. 47–51.

[27] A. Agarwal and P. Kumar, "Economical duplication based task scheduling for heterogeneous and homogeneous computing systems," in *2009 IEEE International Advance Computing Conference*, March 2009, pp. 87–93.

[28] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, Feb 1994.

[29] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400712

[30] OpenMP ARB, "OpenMP, A Proposed Industry Standard API for Shared Memory Programming," *White Paper*, 1997.

[31] P. Thoman, H. Jordan, and T. Fahringer, "Compiler multiversioning for automatic task granularity control," *Concurr. Comput. : Pract. Exper.*, vol. 26, no. 14, pp. 2367–2385, Sep. 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3302