Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 146 – 155

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Parallelism Reduction Based on Pattern Substitution in Dataflow Oriented Programming Languages

Loïc Cudennec, Renaud Sirdey

name.surname@cea.fr
CEA LIST, Nano-Innov / Saclay
91191 Gif-sur-Yvette cedex
FRANCE

Abstract

In this paper, we present a compiler extension for applications targeting high performance embedded systems. It analyzes the graph of a dataflow application in order to adapt its parallelism degree. Our approach consists in the detection and the substitution of built-in patterns in the dataflow. Modifications applied on the graph do not alter the semantic of the application. A parallelism reduction engine is also described to perform an exhaustive search of the best reduction. Our proposition has been implemented within an industry-grade compiler for the Sigma-C dataflow language. It shows that for dataflow applications, the parallelism reduction extension helps the user focus on the algorithm by hiding all parallelism tuning considerations. Experimentations demonstrate the accuracy and the performance of the reduction engine for both synthetic and real applications.

Keywords: Parallelism, Dataflow Programming, Pattern Detection and Substitution, Reduction Engine, Sigma-C

1. Introduction

Nowadays, it turns out that the quest for computational power leads - again - to an important increase of the number of processing units [1]. Computing grids and many-cores are examples of systems that rely on massive scaling, up to thousands of units connected over continents or within a single chip. There is a real challenge to make parallel programming efficient on large-scale systems while staying appealing to developers. Most of the modern approaches still rely on explicit parallel programming in which an emphasis has to be made on managing communications and synchronizations between tasks. This is particularly true with programming languages like MPI [2], OpenMP [3] and OpenCL [4], all languages being widely used on distributed systems. Some paradigms have been introduced to limit these drawbacks: for example, agent-based and dataflow programming languages offer implicit mechanisms intended to hide low-level communications, as well as inter-tasks synchronizations.

However, in the field of high performance computing, applications have to be finely tuned in order to take benefit from the underlying execution infrastructure. This step largely modifies the application design by adding constraints that are not related to solve the original problem, as well as portability issues to target different architectures. One relevant example is the sizing of the application regarding the number of concurrent tasks. This sizing has to ensure the best execution speedup: too few concurrent tasks results in an *idle system* while too many concurrency implies an overhead due to task communications and context switches. It also implies the allocation of enough space to store all task contexts, where memory is essentially a precious resource in embedded high performance computing. In

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2012.04.016

both cases, execution speedup will not be optimal. Furthermore, the sizing problem closely depends on parallelism granularity. For example, a video processing application can be designed creating concurrent tasks per frame, per row, per macro-block, or even per pixel. In most cases, the smaller the granularity is, the better the parallelism degree can be tuned, what helps in reaching the best speedup. As a counterpart, this will also require a tight understanding of the application design and may involve a very time-consuming iterative process to find the best solution. This process is oftenly based on experimentations and know-how.

In order to deal with the application sizing, several approaches have been proposed: from the widely-used hand-made configuration files specifically designed for a given execution platform, to the use of pragma instructions indicating that the compiler or the runtime can modify the code to fit into the host target. Pragma instructions are part of the CUDA [5], OpenMP and the upcoming MP Designer [6] languages. In these situations, applications still have to be written with explicit parallelism instructions. One approach to ease parallelism development is to rely on automatic parallelism extraction. In this scenario, a static analysis is performed on the source code in order to detect loops that meet all the requirements allowing to execute inner instructions in a concurrent way. This approach has several drawbacks: static analysis is still a very complex process that makes loop detection difficult to perform, some loops and other code statements may not be detected even if they are eligible for the treatment and last but not least, this approach encourages developers to keep writing applications in the single threaded model instead of moving to the distributed model.

With the democratization of parallel programming, application developers cannot be expected to master architecture related intricacies and should be allowed to focus on designing algorithms with only two goals in mind: 1) solve a problem and 2) get the smallest parallelism granularity in order to keep enough tuning possibilities for speedup. The overall optimizations and parallelism tuning should be transparently handled by the compiler, without any pragma hints or user text decorations. In this paper, we focus on the Sigma-C dataflow programming language [7], a language which has been specifically designed for programming high performance computing applications over massively parallel architectures. One of the key aspect of this language, over all aspects offered by dataflow programming, is the ability to specify the productions and consumptions of each task. This crucial information is used at compile-time for checkings such as buffer sizing, placement, routing, deadlock detection and, as presented in this paper, parallelism tuning.

In this paper, we propose a compiler extension that analyzes the application dataflow graph and the tasks communication behavior in order to tune the parallelism degree of a dataflow program. This extension is called *parallelism reduction*. Our approach is based on the detection and substitution of patterns within the application task instantiation graph. These patterns are part of a built-in library. On top of that, we designed a *parallelism reduction engine* that is able to select relevant patterns and to calculate a substitution order by making an exhaustive or heuristic search. This engine ensures to get, if possible, the desired parallelism degree. This paper is organized as follow. Section 2 introduces the Sigma-C dataflow programming language. Sections 3 and 4 respectively present the parallelism reduction patterns and engine. Section 5 gives some insights on the implementation and the preliminary evaluation. Section 6 discusses the related works. Finally, section 7 concludes and gives some perspectives.

2. Sigma-C: a Dataflow Programming language for Large-Scale Infrastructures

Sigma-C [7] is an agent-based dataflow programming model and language designed for efficient parallel programming on large-scale infrastructures such as many-core processors and computing grids. The model is based on process networks with process behavior specifications. The language is an extension of the ANSI C language and provides keywords to define and connect agents. The Sigma-C application is described as a static instantiation graph with no change during the execution. Agents communicate through point-to-point, unidirectional and typed links. They are defined using three main sections, as shown in Listing 1. The *interface* section is used to declare input and output communication ports, as well as a specification of the consumptions and productions. This specification allows formal analysis to enforce properties such as absence of deadlock and memory bounded execution. In Listing 1, the *specification* section of agent *Filter* specifies the consumption of *width* integers on the *input* port, 1 float on the *random* port and the production of *width* floats on the *output* port. The *map* section is used to instantiate agents and connect ports. The last section is dedicated to user functions: the *start* function is the entry point of the agent and is repeatedly executed. The Sigma-C language also provides system agents that ease data reorganization. The three main agents are *Split*, *Join* (both for round-robin distribution of data) and *Dup* (duplicate data).

Listing 1: A simple filter agent definition in Sigma-C

```
agent Filter (int width) {
 interface {
  in<int>   input;
  in<float> random;
  out<float> output;
  spec {input[width]; random; output[width]};
 }
 map {
   agent myRandom = new Random();
   connect(myRandom.output, random);
 }
 void start() exchange(input i[width], random r, output o[width]) {
   int k = 0;
   for (k = 0; k < width; k++)
     o[k] = i[k] + r;
 }
}
```

One *leitmotiv* of the Sigma-C language is to let the programmer determine the finest granularity level according to his *à priori* knowledge of the application, not regarding the final execution performances. This generally leads to rather highly parallel applications. For example, a matrix multiplication application can intuitively be written using one instance per resulting cell, leading to millions of agents. The parallelism speedup is nonetheless a trade-off between the number of parallel tasks and their management overhead. Too few concurrent tasks leads to idle processing units, too many concurrent tasks lead to significant memory use in order to store contexts. The optimal number of concurrent tasks depends on several parameters: from the tasks execution time, physical memory use and data communication intensity, to the final hardware number of processing units and network topology. These parameters are either hard to calculate, or target dependent, making close to impossible to reuse the code as this. The developer should focus on algorithm parts of the program and the compiler should take care of optimizations to ensure best (at least acceptable) execution performances. Our approach consists in giving a target number of instances per processing units (*ratio*), the number of processing units and let the compiler adapt the application with these parameters. We propose to add a compiler extension to the Sigma-C toolchain, dedicated to parallelism tuning. This extension takes the dataflow instantiation graph and applies modifications using pattern detection and substitution.

3. Parametrized Dataflow Patterns

The application instantiation graph describes all the instances and connections declared in the Sigma-C source file. In order to fit a given execution infrastructure with a given degree of parallelism, we choose to apply a set of modifications on this instantiation graph. These modifications are not subject to limitations: instance and port creation, deletion or modification can be achieved, as long as these operations hardly respect the following rule: no alteration of the semantic of the application, neither of the user code. Our approach relies on the use of pattern detection and substitution, in which some parts of the graph - called *subgraphs* in the remaining of the paper - are recognized and replaced by another subgraph. A *parametrized subgraph* is a subgraph description in which some of the structural aspects or instance properties are set by parameters. Therefore, *patterns* are defined as two parametrized subgraphs and a set of *substitution rules*. These rules are applied to modify the first parametrized subgraph in order to build the second one.

We motivate the need for parametrized subgraphs with the possibility to identify several class of subgraph. For example, the *Split-*-Join* pattern is defined by a *Split* system agent, connected to a set of equivalent subgraphs - the star in the pattern name - that are in turn connected to a *Join* system agent. It is parametrized by the number of subgraphs sitting in the middle of the Split-Join scheme. This pattern, widely used in dataflow applications, is illustrated twice in the simple version of the Laplacian image processing application, as shown in Figure 1. In this application, images are loaded as a sequence of lines in a contiguous memory block. A first set of filters processes the image line-by-line, while a second set processes column-by-column. Split and Join system agents are used to access, reorganize and distribute data to feed filters. Consumption and production of system agents and filters are given by the *k* parameters. Using this level of granularity, the application size directly depends on the image size: a $W * H$ image

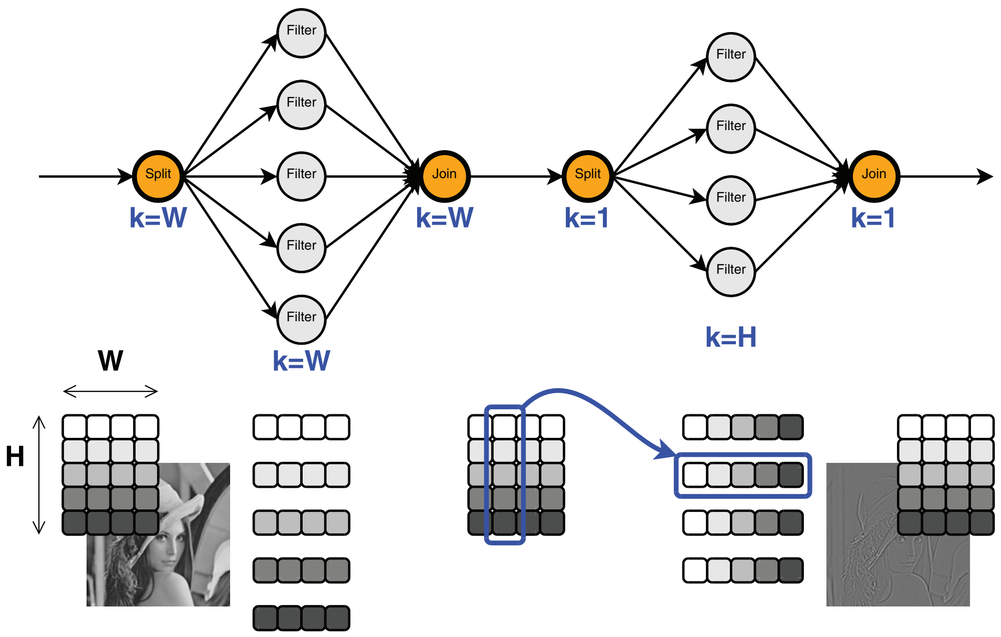


Figure 1: Split-*-Join-Split-*-Join pattern sequence as used in a simple Laplacian image filter (Huertas-Medioni operator [8]).

instantiates H line filters and W column filters. This intuitive implementation of the Laplacian filter can generate large applications if applied to large images: as an example, a 1080p HDTV frame would require 3000 filters, far more processes than today's regular embedded encoder-decoder platforms are able to run.

In some particular conditions regarding productions and consumptions, the *Split-*-Join* pattern can safely be modified by removing or adding some of the subgraphs connected between Split and Join system agents. Such modifications are used to adapt the number of instances started at runtime. Figure 2 shows how this can be applied on the simple Laplacian image filter: some filters are removed in both line and column filter sets. According to the round-robin data distribution, the remaining filters have to process more data per frame. The reduction works fine for line filters that may receive more than one line to process. However, data reorganization does not work for column filters: the resulting input columns are not well-formed [1] and the filters do not read the expected data. As a matter of fact, the reduction of parallelism ensures to preserve data reorganizations if and only if the *equivalence of pointer* is preserved within the whole pattern. It is also a desired behavior that simplifies the programmer's work.

A connexion between two agents preserves the equivalence of pointer if all reads and writes are made into *contiguous blocks of memory*. In Figure 2, line filters read exactly the same number of data (W) than written by the split agent. In this first configuration, the round-robin distribution ensures to preserve the equivalence of pointer. In the second configuration, column filters read H data while the split agent writes one by one. The H data written in the input block of the column filter come from non-contiguous addresses in the original input of the split. Here, the equivalence of pointer property is not preserved and the reduction cannot be applied. Therefore, the detection of the Split-*-Join pattern must check the following condition: $remaining(S_k/F_k) = 0$, where S_k refers to the Split productions and F_k to the filters consumptions. This example illustrates that some pattern detections have to match data reorganizations: this is possible to analyze if the language provides a specification of the consumptions and productions of the instances. Finally, it is only possible to add or remove subgraphs between the Split and Join system agents if these subgraphs are *stateless*, that is, subgraphs do not keep local information between two invocations. This property is determined thanks to a static analysis of the source code during the parse phase.

[1] Some unfriendly data reorganization can be directly fixed in the application design. As an example, a trivial work-around for the Laplacian application consists in applying two matrix transposition operators, before and after the column filters, hence falling back into the advantageous line filters case. A matrix transposition operator can be expressed using a Split and a Join, directly connected through their W pins, with respectively 1 production and H consumptions.

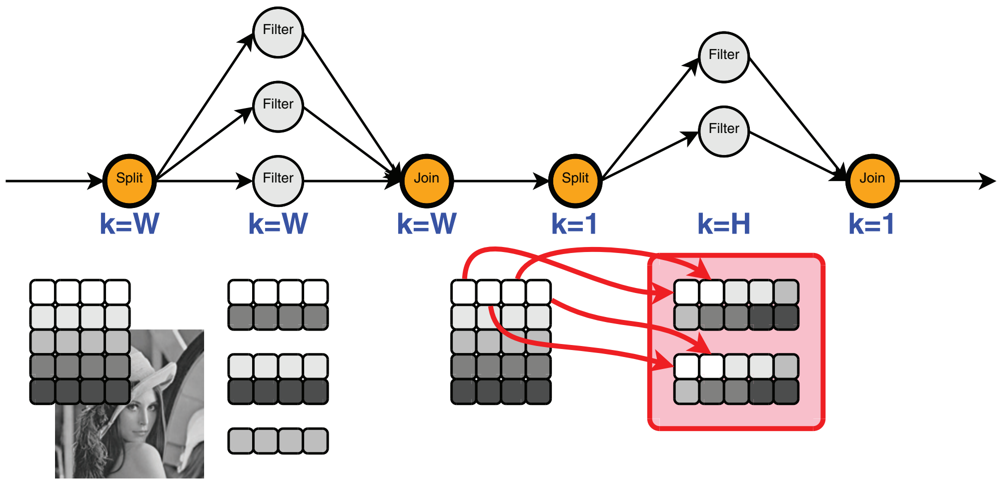


Figure 2: Split-*-Join pattern reduction on the simple Laplacian image filter. Left pattern preserves pointer equivalence while the right pattern does not organize data as expected.

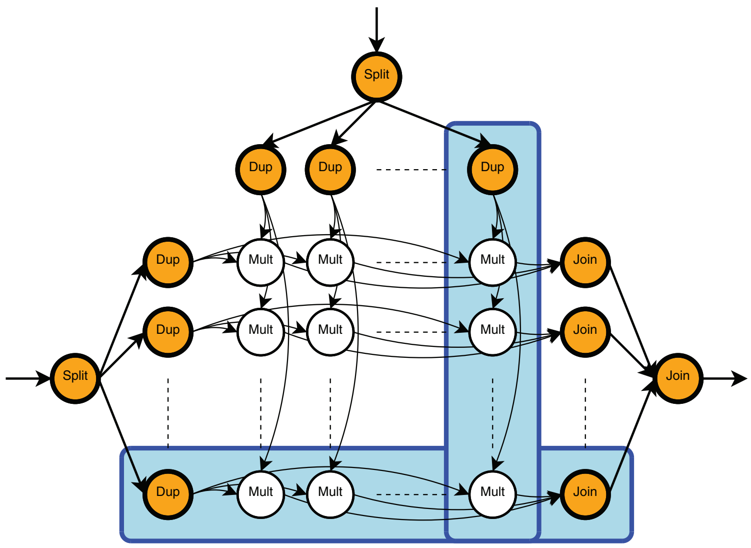


Figure 3: Matrix Multiplication pattern.

Other often met patterns based on the Split-Join scheme include the Cascade pattern, the Butterfly pattern and the Matrix Multiplication pattern, as shown in Figures 4 and 3. The Cascade pattern (Figure 4(a)) is a recursive case of the Split-*-Join pattern, each stage of which being another Split-*-Join pattern. This pattern is used in image processing, for macro-block decomposition and regular strided memory access patterns. By definition, all patterns of the same stage share the same properties. Therefore, the reduction of parallelism can be processed on each level of the tree. Applying reduction on the root node removes a large number of instances but also offers a few control on the tuning of parallelism compared to if applied on the leaves. The Butterfly pattern (Figure 4(b)) is an other example of recursive Split-*-Join pattern. It is used in the Deriche image processing application [9] and differs from the Cascade pattern by interlacing the leaves outputs to spread data to all filters. Finally, the Matrix Multiplication pattern (Figure 3) consists in a double Split-Dup cascade and a Join cascade. It splits the rows of matrix A and the columns of matrix B in order to feed each multiplication cell and calculate the resulting matrix. This pattern can be reduced by removing rows and columns.

All patterns include both detection and substitution functions. These functions are specific to each pattern. The *detection function* takes a *root instance* and returns true if there is a match between the given pattern and the instantiation graph starting at the root instance. Therefore, the detection of all pattern instances is done by applying this function to all eligible root instances. Pattern may also include the type of the root instance. For example, patterns presented in this paper are detected starting with a *Split* root instance. This speeds the process up by allowing to only apply the function to a small list of instances instead of processing the entire graph. The detection is programmatically done by navigating through the application instance graph and using built-in matching functions offered by the compiler

API. Some of these functions rely on Floyd-Warshall routing tables [10] to determine the shortest paths between two agents, as well as graph isomorphism algorithms to determine if two subgraphs are equivalent. In case of a positive pattern match, the removable parts of the subgraphs are identified and a *reduction capacity indicator* is calculated by counting all the removable user instances. This indicator will be later used by the reduction engine.

The *substitution function* takes a root instance and replaces the detected subgraph by a semantic-equivalent subgraph. It is parametrized by a *factor of reduction* that is used to control the level of parallelism reduction. This factor is a floating point number ranging between 0 and 1, where 0 means no reduction and 1 applies the full reduction, based on the reduction capacity indicator. The substitution is programmatically done by using a graph altering API that allows to remove agent and subgraph instances with port auto-reconnect. The function then returns the number of user instances that have been removed. This result will be later used by the reduction engine to evaluate a solution.

Patterns presented in this section are built-in patterns, but one can imagine that some specific patterns, tightly related to an application field, can be added to the engine.

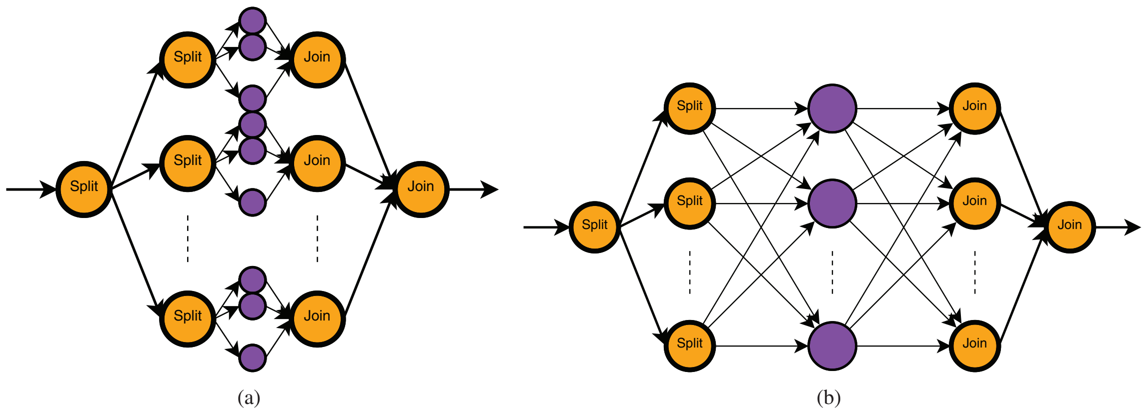


Figure 4: (4(a)) Split-*-Join Cascade pattern and (4(b)) Butterfly pattern.

4. Parallelism Reduction Engine

The parallelism reduction engine is in charge of performing the detection of patterns and applying patterns on the application graph. Figure 5 illustrates the two main steps of the engine. The first step takes an application graph, a pattern list and returns the detected patterns as a set of instantiated patterns. *Instantiated patterns* are defined by a pattern, a root instance and a capacity. The second step takes the application graph, a set of instantiated patterns and returns a sequence of instantiated patterns. This sequence gives three information: 1) the list of instantiated patterns (some of them can be discarded), 2) the order of appliance and 3) the factors of reduction attached to each pattern. Many sequences can be built by ordering instantiated patterns and modifying reduction factors. Therefore, the reduction engine has to select the best solution, according to the application and execution platform configurations.

A factor of reduction is attached to each instantiated pattern. This allows the engine to apply patterns with different strengths, going further with parallelism tuning. This is particularly relevant in the context of iterative compilation in which information gathered on target can be fed back to save or modify patterns. Other considerations such as energy savings, thermal dissipation and load balancing can be part of the decision. However, for the sake of simplicity, we choose to calculate a global reduction factor, that will be set to all instantiated pattern. This global factor is given using the following formula:

$$factor_{global} = \frac{nbinstances - (nbunits * ratio)}{capacity}$$

Where *nbinstances* is the number of user instances in the application graph, *nbunits* is the number of processing units in the targeted execution platform, *ratio* is the reduction ratio expressed in number of user instances per processing units and *capacity* is the number of user instances that can be removed if a full reduction is applied to all instantiated patterns. The *nbunits* and *ratio* constants are given as parameters to the compiler. For example, let's consider a 386-user-instance application built for a 64-processing unit target. The current ratio is 6 instances per processing unit and the user targets a 2.4 ratio. If pattern detection returns that the overall reduction capacity indicator is up to 381 removable instances, the resulting global reduction factor will be set to 0.61.

Using this global reduction factor makes possible to perform an *exhaustive search* to find the best pattern sequence. This exhaustive search consists in testing all pattern permutations and in keeping the one that is closer to the desired parallelism ratio. At this point, calculating the number of user instances that will be removed for each permutation cannot be done by just summing all pattern capacities of the sequence: some patterns, once applied, may disable or modify patterns that follow. For example, the Cascade pattern (Figure 4(a)) is made of recursively-defined patterns. If the first instantiated pattern of the sequence starts at the Split root instance of the tree, the following included patterns may be removed once this first reduction applied. Therefore, permutations have to be applied *in real conditions* on the application instance graph in order to evaluate the solution. This process ensures to find the best solution but obviously makes the algorithm more complex and slower than just guessing the final result.

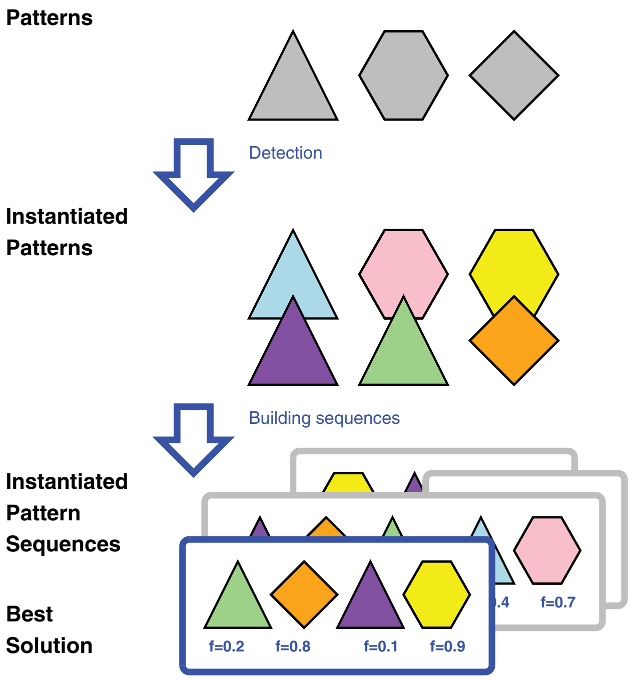


Figure 5: Parallelism Reduction Engine: from pattern detection to best solution.

One drawback coming with the exhaustive approach is the number of permutations. With only 10 instantiated patterns, the engine has to explore 10! = 3628800 sequences, what would be far too long to process on a regular workstation. Most of the applications of our knowledge, in the field of video encoding or motion detection use a very few patterns, up to 3 or 4 patterns, allowing to evaluate all possibilities. However, we designed the reduction engine to restrict the exhaustive field of search when the number of patterns exceed a given value P_{max}. In this case, the engine sorts patterns by decreasing capacities, keeps the P_{max} first ones and discards the others. The engine then evaluates the $P_{max}!$ possibilities, where P_{max} is set according to the hardware that hosts the compiler and the time the user is ready to pay. A more elaborated approach is to build the *pattern intersection matrix* that gives, for each pattern couple, the number of removable user instances shared by both patterns. This matrix is then used to select the P_{max} patterns from the sorted list with the least shared instances. This ensures to keep instance-independent patterns while having large capacities. The engine also evaluates the full sequence including all patterns with an arbitrary order, for example based on the order of detection.

The parallelism reduction engine uses a recursive algorithm that performs, for each sequence evaluation, a deep copy of the instantiation graph. It applies patterns sequentially and compares each step with the best solution found until then. Further developments and optimizations of the engine include code parallelization, iterative compilation in order to benefit from performance feedback, as well as greedy randomized adaptive search procedures [11].

5. Experimental Evaluation

The following results are given to demonstrate that the reduction engine is accurate and that the inner algorithms are efficient. In this paper, we do not evaluate the influence of parallelism reduction over the application execution performances. We only focus on modifications applied on the application, comparing the initial and final states and measuring how long it takes to process. All experimentations have been passed on a Intel Core 2 Duo CPU P8600 at 2.40GHz running Linux kernel 3.0.0. Only one core out of the two was used in the system. We choose to run the experimentations on this mobile device in order to demonstrate that the compiler extension can be used on a regular laptop station. Experimentations follow the same methodology: for each application, we stress the reduction engine with 64 configurations. These configurations differ on the number of processing units and the desired parallelism ratio. By multiplying these two parameters, we obtain the number of targeted user instances in the application. This number ranges from near 35 to near 2456 instances (the ratio is a floating point number). It corresponds to different reduction goals: from hard to very light, sometimes no reduction is required. Each configuration is run 21 times and we take the average results on the 20 last runs, even if we do not notice any startup effect on the first run, everything being properly loaded into memory. Five applications have been selected, three of them are made of a single pattern, the other two are real applications. Figure 6 lists all five applications with their respecting stats.

Figure 7(a) and 7(b) illustrate the accuracy of the reduction engine. In the first figure, two simple patterns are evaluated with different reduction goals. The accuracy is given as a percentage of the resulting ratio (R_r) (user instances per processing units) comparing to the expecting one (R_e), using the formula ($R_r - R_e) * 100/R_e$. Therefore, 0 means a perfect match. A negative percentage means that the engine has removed more instances than needed. The engine calculates a near-to-perfect reduction with the Split-*-Join pattern in all configurations. This is due to the simplicity of the pattern that only contains one user instance per split branch. This makes possible to remove at the instance granularity. The engine is a bit less accurate with the Matrix pattern: the reduction is at the row and column granularity. Figure 7(b) shows the number of removed instances and the factor of reduction applied to the Deriche application in all configurations. The number of removed instances also features error bars showing the number of instances that should have been removed to reach the goal. In the first part of the graph, up to 400 targeted number of instances, the reduction goal cannot be reached: the number of removable instances (only 958 instances out of 1461 instances are part of a pattern) is less than required, and the reduction factor is set to 1. Then, the factor decreases and the error bars disappear as long as the pressure on the reduction engine decreases.

Figure 8(a) shows the benefit of running the exhaustive search on as many patterns as possible. The experimentation is based on a Cascade pattern with 63 imbricated patterns. Two configurations are highlighted: one with 32 processing units and one with 64, both using a 1.2 ratio. The number of instances that could not be removed is given, according to the number of patterns selected by the exhaustive engine. This shows that the more pattern we keep in the process, the more accurate tuning we get. Figure 8(b) gives the factor of reduction and the processing time for the first configuration of this experiment. Adding patterns to the engine increases the removal capacity (going from 189 instances to 327) and decreases the reduction factor. As a counterpart, the processing time to find the best solution explodes, from 0.03 seconds for 1 pattern to 107 seconds for 7 patterns. This time corresponds to the evaluation of 7! = 5040 pattern sequences.

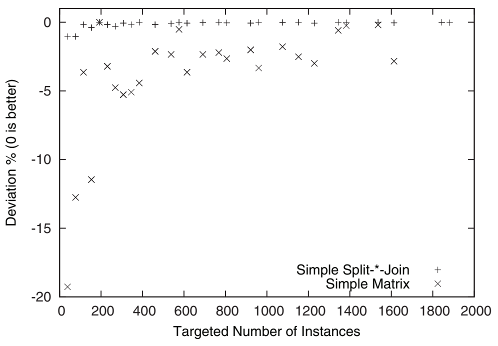
The experiments presented in this paper show that the exhaustive reduction engine returns very accurate results. However, when exceeding a given number of patterns, the factorial-growth processing time does not allow to use the exhaustive engine on the full pattern list. Discarding patterns also decreases the removal capacity and may, in turn, eliminate good solutions.

6. Related Work

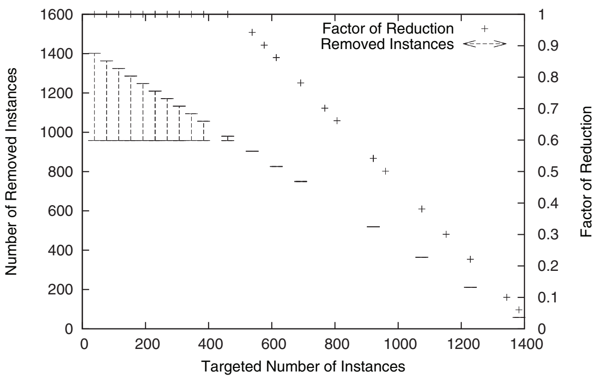
The parallelism degree has been well studied in parallel and distributed systems, from the onboard graphical processing unit to the cluster and grid infrastructures. Most of the applications that benefit from the tuning of parallelism are designed as independent processing codes. These processing codes are duplicated either by the compiler [4, 5] or in a dynamic way by the runtime [12, 13] during the execution. These systems are similar to a simple dataflow application made of a Split-*-Join pattern. Our approach is to provide a set of relevant, more complex patterns. Most of the dataflow programming language for embedded systems [14, 15, 16, 17, 18] do not address the problem of parallelism reduction. Developers have to take care of their application sizes unless the runtime will not start to execute. Some instrumentation languages [19] helps in auto-tuning the application. However, they rely on the use of pragmas and therefore add more design work to developers. This approach mainly relates to Streamit [20]. Streamit provides a high-level stream abstraction in which filters are connected within a directed graph. Its compiler performs stream-specific optimizations. In [21], several graph transformations are described: fusion, fission and re-ordering operations are applied to adjacent filters in order to modify the granularity of the program. These operations are applied on either a filter pipeline, that is, a sequence of filters with one input and one output or, on a split-join scheme in which the inner branches are pipelines. Unlike Streamit, our compiler extension is based on a pattern approach that allows to modify more complex subgraphs as shown with the cascade, the butterfly and the matrix multiplication patterns.

Application	Instances Nr	Capacity	Links Nr	WFI (s)	Patterns Nr	D_{avg} (s)	S_{min} (s)	S_{max} (s)
Simple Split-*-Join	1925	1919	3842	18.7	1	0.07	0.12	0.27
Simple Cascade	1074	963	1198	3.3	63/4	0.06	0.54	0.61
Simple Matrix	1867	1721	5319	17	1	0.006	0.10	0.13
Bitonic Sort	1132	903	1624	3.8	150/5	0.12	4.8	5.2
Deriche	1461	958	3561	8.2	2	14.8	0.24	0.26

Figure 6: Application stats: Number of instances, number of removable instances (capacity), number of links, time to build the Floyd-Warshall routing table (WFI), number of patterns (/ number of selected patterns), average time for detecting all patterns, minimum and maximum times for applying all substitutions.

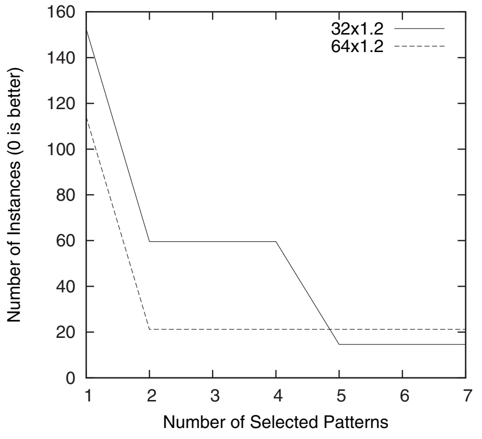


(a)

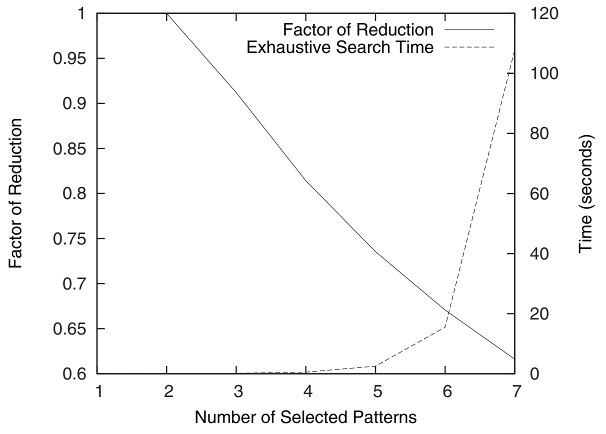


(b)

Figure 7: (7(a)) Deviation percentage of the resulting ratio comparing to the expected ratio, for both simple Split-*-Join and Matrix patterns. (7(b)) Number of removed instances, number of instances that should have been removed (error bars) and the factor of reduction applied for each configuration (the targeted number of instances) on the Deriche application [9], with two Split-*-Join patterns and 1461 instances.



(a)



(b)

Figure 8: The influence of selecting different numbers of patterns: experimentations on a synthetic Split-*-Join Cascade including 63-imbricated patterns, 5 levels of deepness with 1074 instances. (8(a)) Number of instances that should have been removed, for two configurations of processing units and ratio, and (8(b)) factor of reduction and processing time given for the first configuration (32 ∗ 1.2).

7. Conclusions

This paper presents a dataflow compiler extension for parallelism tuning. It processes the graph of a dataflow application to find and substitute patterns without modifying the semantic, neither the user code. We believe this approach let the developer focus on the algorithmic part while encouraging the maximum degree of parallelism when designing applications. We have identified four patterns that are widely used in dataflow programming. These patterns are tuned by the reduction engine to reach a given goal. The extension has been implemented within an industry-grade compiler [22] for the Sigma-C language. It has shown to be very precise on tuning and efficient when used with a reasonable number of patterns. We have a number of improvements scheduled for this work. In a first step, we plan to go further with the experimentations. We have to keep in mind that the final goal of such a work is to improve the application execution performances. Therefore, the next experimentations should focus on the benefit of adapting the parallelism degree on the execution performances. We also plan to add more reduction patterns, generic and specific to industrial applications. Other improvements of the reduction engine can be obtained with the parallelization of the exhaustive search, allowing to take advantage of running over distributed computing systems. We should be able to save computing time and calculate solutions with more patterns. In a second step, we plan to integrate the reduction extension within a performance feedback loop. Iterative compilation would make possible to auto-adapt the parallelism ratio and calculate an appropriate reduction factor for each pattern. As a perspective, we plan to take benefit of the SJD intermediate representation [23] for data reorganization.

References

[1] System driver chapter 2010 updates, International Technology Working Group, http://www.itrs.net (2011).

[2] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[3] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1998) 46–55.

[4] OpenCL: Introduction and overview, Khronos OpenCL Working Group, http://www.khronos.org/opencl/ (June 2010).

[5] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2008) 40–53.

[6] G. Goossens, Enabling the design and programming of application-specific multi-processor architectures, in: Proceedings of the 20th IP-Embedded System Conference and Exhibition, Design and Reuse, Grenoble, France, 2011.

[7] T. Goubier, R. Sirdey, S. Louise, V. David, Sigma-C: A programming model and language for embedded manycores, in: Y. Xiang, A. Cuzzocrea, M. Hobbs, W. Zhou (Eds.), Algorithms and Architectures for Parallel Processing, Vol. 7016 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 385–394.

[8] A. Huertas, G. Medioni, Detection of intensity changes with subpixel accuracy using laplacian-gaussian masks, IEEE Trans. Pattern Anal. Mach. Intell. 8 (1986) 651–664.

[9] R. Deriche, Fast algorithms for low-level vision, IEEE Trans. Pattern Anal. Mach. Intell. 12 (1990) 78–87.

[10] R. W. Floyd, Algorithm 97: Shortest path, Commun. ACM 5 (1962) 345–.

[11] T. Feo, M. Resende, Greedy randomized adaptive search procedures, Journal of Global Optimization 6 (1995) 109–133.

[12] Oracle, Oracle database parallel execution fundamentals (October 2010).

[13] E. Caron, F. Desprez, DIET: A scalable toolbox to build network enabled servers on the grid, International Journal of High Performance Computing Applications 20 (3) (2006) 335–352.

[14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: Stream computing on graphics hardware, ACM TRANSACTIONS ON GRAPHICS 23 (2004) 777–786.

[15] D. Watt, Programming XC on XMOS Devices, XMOS Limited, 2009.

[16] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, C. Pu, Spidle: A DSL approach to specifying streaming application, in: Second International Conference on Generative Programming and Component Engineering, Erfurt, Germany, 2003.

[17] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, Proceedings of the IEEE 79 (9) (1991) 1305 –1320.

[18] G. Berry, G. Gonthier, A. B. G. Gonthier, P. S. Laltte, The Esterel synchronous programming language: Design, semantics, implementation (1992).

[19] C. A. Schaefer, V. Pankratius, W. F. Tichy, Atune-il: An instrumentation language for auto-tuning parallel applications, in: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 9–20.

[20] W. Thies, M. Karczmarek, S. Amarasinghe, StreamIt: A language for streaming applications, in: In International Conference on Compiler Construction, 2001, pp. 179–196.

[21] M. Gordon, W. Thies, M. Karczmarek, J. Wong, H. Hoffmann, D. Maze, S. Amarasinghe, A stream compiler for communication-exposed architectures, in: In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002, pp. 291–303.

[22] Kalray, Introduction to the MPPA technology. A technical overview, Tech. Rep. KETD-58, Kalray S. A. (2011).

[23] P. de Oliveira Castro, S. Louise, D. Barthou, Dsl stream programming on multicore architectures, in: S. Pllana, F. Xhafa (Eds.), Programming Multi-core and Many-core Computing Systems, to appear, John Wiley and Sons, 2011.