Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming

Dalvan Griebler and Luiz Gustavo Fernandes

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS),
GMAP Research Group (FACIN/PPGCC), Brazil
Av. Ipiranga, 6681 - Prédio 32, 90619-900 - Porto Alegre, RS, Brazil
`dalvan.griebler@acad.pucrs.br, luiz.fernandes@pucrs.br`

Abstract. Pattern-oriented programming has been used in parallel code development for many years now. During this time, several tools (mainly frameworks and libraries) proposed the use of patterns based on programming primitives or templates. The implementation of patterns using those tools usually requires human expertise to correctly set up communication/synchronization among processes. In this work, we propose the use of a Domain Specific Language to create pattern-oriented parallel programs (DSL-POPP). This approach has the advantage of offering a higher programming abstraction level in which communication/synchronization among processes is hidden from programmers. We compensate the reduction in programming flexibility offering the possibility to use combined and/or nested parallel patterns (*i.e.*, parallelism in levels), allowing the design of more complex parallel applications. We conclude this work presenting an experiment in which we develop a parallel application exploiting combined and nested parallel patterns in order to demonstrate the main properties of DSL-POPP.

1 Introduction

In recent years, High Performance Computing (HPC) has become a wide spread research field which is no more restricted to highly specialized research centers. The use of HPC is crucial to achieve significant research goals in many segments of the modern Computer Science. In this scenario, multi-core processors are now a mainstream approach to deliver higher performance to parallel applications and they are commonly available in workstations and servers.

Although these architectures present a high computing power, developers still have to acquire technical skills to take advantage of the available parallelism. This can lead developers to deal with complex mechanisms, which in addition may result in very specialized solutions [1]. In this sense, programmers may prefer to stay away from parallel programming due to the required efforts to learn how to correctly use it. For that reason, it becomes necessary to investigate alternatives to face this complexity offering to developers different ways to create efficient scalable parallel applications for current architectures.

In the HPC literature, many libraries and frameworks based on the pattern-oriented approach or similar were proposed to make parallel programming easier.

As recent successful examples, it is possible to cite FastFlow [2], Muesli [3], SkeTo [4] and Skandium [5] (among many others that will be discussed in Section 2). This scenario is an evidence that parallel patterns, initially known as skeletons [6], provide a high-level abstraction to develop algorithms while taking advantage of the benefits of parallel architectures. Thus, besides improving the productivity of expert parallel code developers, the use of specific patterns or combinations of them can help less experienced parallel code developers to create efficient and scalable applications [7].

Most part of the pattern-oriented environments proposed so far were designed for clusters and computational grids [8]. With the advance of multi-core platforms, this tendency is changing. Pattern-oriented libraries and frameworks for shared memory systems are becoming more and more necessary. In this context, we believe that parallel patterns along with an expert code generation can guide developers to efficiently create parallel applications for those kind of platforms.

In this paper, we intend to explore the use of patterns an their features through a Domain-Specific Language for Patterns-Oriented Parallel Programming (DSL-POPP) designed for multi-core platforms. By doing that, we intend to hide from developers low level mechanisms such as load balance, flow control schemes and synchronization operations needed to implement parallel applications using patterns. Additionally, we want to provide a way for developers to easily combine patterns in different levels of parallelism (nested and fused patterns). One of the main reasons of using the DSL approach is because it allows minimal changes in a general purpose language (such as the C language for instance, which is familiar to many programmers). Besides, it is crucial for our goal because it makes it easier to change code to experiment different nesting of parallel patterns. Summarizing, the main contributions of our paper are the following:

- we introduce a Domain-Specific Language for Patterns-Oriented Parallel Programming;
- we propose a programming model to achieve nested parallelism through different combinations of patterns based on routines and code blocks structures;
- we show an experimental image processing scenario in which we carry out implementations and tests with the combination of pipeline and master/slave patterns.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 discusses the Pattern-Oriented Parallel Programming paradigm (POPP). The DSL-POPP environment is introduced in Section 4. Section 5 presents performance evaluation experiments of parallel code developed using DSL-POPP. Finally, Section 6 concludes this work.

2 Related Work

Since the emergence of the structured programming concept with parallel skeletons introduced by Murray Cole [6], several libraries, frameworks and languages employed this approach on parallel and distributed systems. Murray proposed eSkel, an environment that allows skeletons constructions of parallel programming using similar MPI primitives in C code [9]. More recently, the SkeTo parallel skeleton library (a C++ library coupled with MPI) implements two-stage dynamic task scheduling to support multi-core clusters [4]. The nestable parallelism pattern is not an objective, but SkeTo provides data structures (lists, trees and matrices) implemented using templates, and parallel skeletons operations (map, reduce, scan and several others) can be invoked on them.

Muesli is a C++ template library that uses MPI and OpenMP to support multi-processor and multi-core architectures [3]. Contrary to SkeTo, Muesli supports nesting data and task parallel skeletons. Also, Lithium [10] and its successor Muskel [11] are skeletons libraries for clusters and both support the nestable skeletons using the macro data-flow model. Inspired on Lithium and Muskel frameworks, Skandium is a Java library for shared memory systems that provides task and data nested skeletons, instantiated via parametric objects [5].

FastFlow is a programming framework for shared memory systems which implements a stack of C++ template libraries using lock-free synchronization mechanisms [2]. However, in FastFlow, developers must implement the pattern through framework routines as in others frameworks and libraries previously mentioned. Differently, we propose the use of a Domain Specific Language to abstract the low level parallel mechanisms necessary to implement parallel patterns, such as load balance, control flow, tasks splitting and synchronization operations. In DSL-POPP user interface environment, programmers develop sequential code in code blocks inside the pattern predefined structure. Our language offers the possibility of nesting (parallelism in levels) and combining parallel patterns.

Finally, there are research works that focus on pattern-oriented parallel languages. These languages and domain-specific languages share similar features such as the existence of a compiler and automatic code generation. P3L is an explicit parallel language that provides skeletons constructions to explore parallelism [12]. In other words, P3L defines skeletons constructions with input/output and sequential modules. More recently, Skil [13] offers a subset of C language (high order functions, curring and polymorphic types) which should be used to implement patterns. Skill does not offer pre-implemented patterns and does not allow nested patterns. In contrast to those languages, we have designed DSL-POPP to explore different combinations of patterns and levels of parallelism in shared memory systems. Additionally, we have introduced an alternative way to implement patterns in user level interface through routines and code blocks directly integrated in the C language code.

3 Patterns-Oriented Parallel Programming

Parallel skeletons have been an alternative to create parallel programming interfaces since the early 90s. As the name implies, they are algorithms skeletons to develop structured parallel programs. Skeletons are similar to Software Engineering concept of Design Patterns [14]. Patterns became popular in parallel programming with object-oriented programming. One of the main reasons to use parallel patterns, is because they allow the programming environment designers to generate parallel codes freely from the parametrization of the abstractions and from the addition of sequential code [1,7]. Also, a pattern-oriented approach can help programmers to develop complex parallel applications since a pattern provide the structure of the program implementation. This reduces considerably the efforts to learn how to use parallelism techniques to take advantage from high performance architectures [1,14].

We propose the use of POPP (Patterns-Oriented Parallel Programming) generic model to create an interface programming environment based on the patterns approach. This model is potentially designed to explore different levels and combinations of patterns implementations. We chose to offer the POPP model through a domain-specific language programming interface what allow us to automatically generate parallel code. Our objective is not to create a new and independent parallel language, but extend a general purpose language offering a higher abstraction layer over it in which we intend to make low level parallelism mechanisms as abstract as possible for developers.

The POPP model relies on a combination of patterns routines conceptually defined and code blocks corresponding to the parallel pattern. Since programs can be composed by different types of computations (routines), the parallelism may not be expressed by a single routine. For that reason, our model allows the inclusion of subroutines which can be used to compose patterns in a hierarchical way offering different parallelism levels. A representation of the POPP generic model is illustrated in Figure 1.

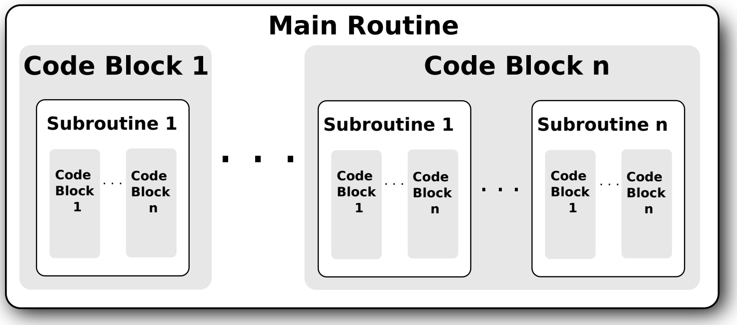


Fig. 1. The POPP model

In order to illustrate how to implement a parallel pattern in the POPP model, we describe two examples of classical parallel patterns: master/slave and pipeline (Figure 2). In master/slave pattern, the master is responsible for sending the computational tasks for all slaves. Then, once all tasks have been computed, results are sent back to the master to finalize the whole computation. For this pattern, both POPP routine and subroutines can implement their own master and slaves code blocks. For instance, it is possible to have inside a slave code block a subroutine composed of other master and slaves code blocks.

Differently, the pipeline pattern is based on a line of stages, in which each stage performs part of the computational workload. The output of each stage is the input of the next one [1]. The POPP model allows the creation of a pipeline in which stages can be implemented as subroutines with their own stages. The final composition can be represented by several pipeline stages.

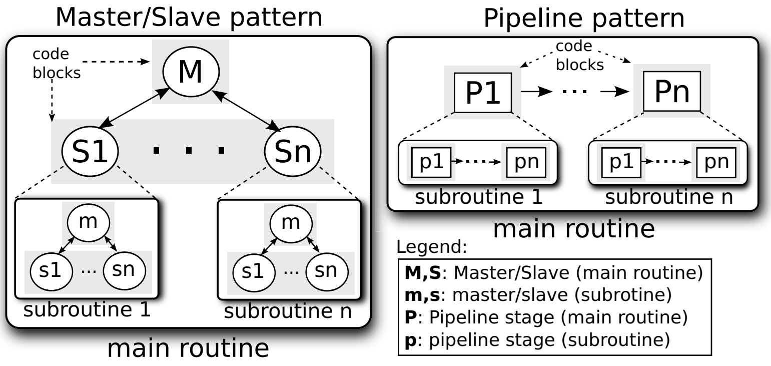


Fig. 2. Master/slave and pipeline patterns

As previously mentioned, patterns can be combined in the POPP model using main routine and subroutines combinations. We present a version of hybrid patterns in Figure 3. In this example, the main routine uses the pipeline pattern and two of its stage blocks parallelize their operations using the master/slave pattern. This configuration is only an example. Others combinations of patterns can be also implemented such as a master/slave pattern in which slaves apply the pipeline pattern as subroutines.

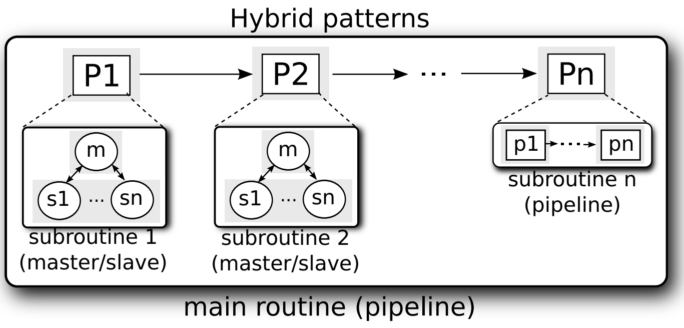


Fig. 3. An example of combined patterns

In this section we presented the abstract idea of the POPP model, which includes different levels of patterns implementation and their combination. These features became clearer in the next section, in which a domain-specific language is proposed based on this programming model.

4 DSL-POPP in a Nutshell

The structure of the POPP model is generic enough to support different parallel patterns. However, in this paper we intend to demonstrate its usability through the implementation of master/slave and pipeline patterns on DSL-POPP. It is important to highlight that our DSL can be extended to provide other parallel patterns. For each new pattern, a new set of routines should be defined.

4.1 Compilation

In order to use our DSL, developers have to include our library (`poppLinux.h`) in the source code and use our compiler (named `popp`). This library includes all the routines, code blocks and primitives definitions. The compilation process of the source code is depicted in Figure 4. The source-to-source code transformation between our language and C code is automatically performed by our pre-compiler system, which is responsible for checking syntax and semantic errors. Then, the pre-compiler systems generates the C parallel code using the Pthreads library based on the parallel patterns used. Besides, for the source-to-source code transformation, we created a Shared Memory Message Passing Interface (SMMPI) to carry out threads communication. Finally, we use the GNU C compiler to generate binary code for the target platform.

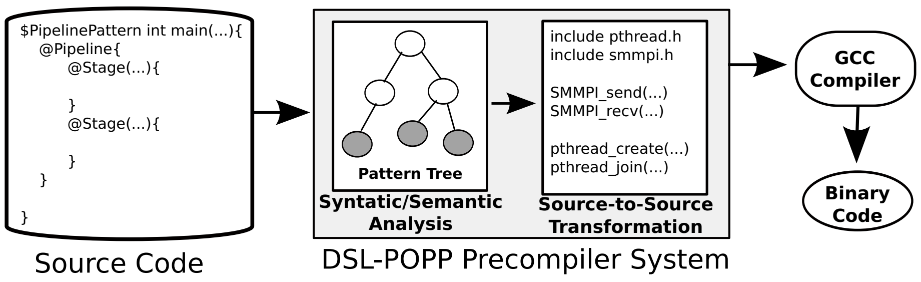


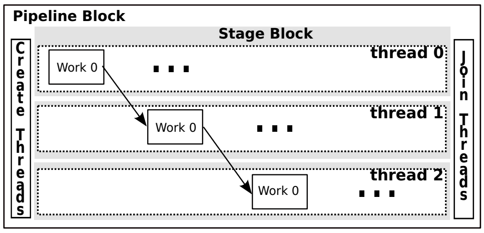
Fig. 4. Overview of the DSL-POPP compilation process

4.2 Programming Interface

In DSL-POPP, language interface specification routines begin with "$" and code blocks with "@". The pattern routine should be declared in a function followed by the return data type and its name. Code blocks should be used inside of the pattern routine and they contain full C code. Figure 5 describes the syntax and logical structure of DSL-POPP constructions.

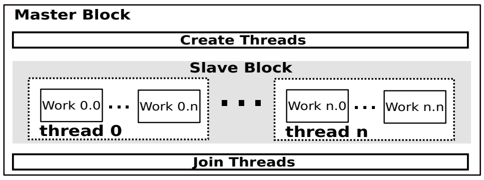
As we can see in Figure 5(a) at the left side, a `$PipelinePattern` routine supports two code blocks: pipeline block (@`Pipeline`{}) and stage block (@`Stage(...)`{}). The pipeline block should be declared at least once and it is responsible for coordinating the pipeline flow. Each stage block corresponds to a stage in the pipeline and can be declared as many times as necessary inside the pipeline code block. In stage blocks, parameters are the number of threads, the buffer to be sent to the next stage and the buffer size.

```
$PipelinePattern void func_name(...){
   @Pipeline{
      @Stage(int num_th, void* buffer, int buf_size){

      }
      @Stage(int num_th, void* buffer, int buf_size){

      }
      @Stage(int num_th, void* buffer, int buf_size){

      }
   }
}
```



(a) Pipeline

```
$MasterSlavePattern void func_name(...){
   @Master{
     @Slave(int num_th, void* buffer, int buf_size,
     const POPP_LB_Policy){

     }
   }
}
```



(b) Master/Slave

Fig. 5. Syntax and logical structure of the DSL-POPP

In Figure 5(b) also at the left side, we show how a `$MasterSlavePattern` routine implements algorithms using master and slave blocks (@`Master`{} and @`Slave(...)`{}). The master coordinates all computation flow and starts the slave blocks. The slave blocks must be used to obtain parallelism inside of the master. Additionally, it is necessary to inform the number of slave threads, the buffer to be sent to the master, the size of the buffer and the load balance policy.

4.3 Patterns Implementation

Still in Figure 5, we show how DSL-POPP organizes routines and code blocks and implements the pattern-oriented parallelism. Basic parallel patterns such as Master/Slave or Pipeline are structured based on data exchange through messages. Aiming at simplifying the Pthreads code generation phase for multi-core platforms, we created a Shared Memory Message Passing Interface (SMMPI) which implements threads communication through semaphore routines. The send and receive operations are carried out based on threads identification. For instance, when a thread send a buffer to another thread, in reality it is using `sem_post` to unblock the destination thread which is waiting on a `sem_wait` to receive the buffer and start its work. In the pipeline pattern code generation, we use SMMPI routines to perform send and receive through pipeline stages transferring data through the buffer defined as a parameter of the stage block.

A pipeline routine can be implemented in several ways. The example presented at the right side of Figure 5(a) shows the classic scenario in which all stages blocks have only one thread per stage. In this scenario, the pipeline block creates all necessary threads and wait them all to finish their works. This procedure is repeated transparently from the first up to the last stage block declaration. As it is possible to notice, communication between stages is also implicit and it happens through the buffer declared as a parameter. When stages have different workloads, a non-linear pipeline can be implemented using more threads on the unbalanced stage. This feature avoids significant performance losses in pipeline implementations.

At the right side of Figure 5(b), we illustrate how DSL-POPP organizes the Master/Slave pattern implementation. The master block creates as many threads as defined in the slave block and it waits until all slave threads finish their works. This is transparent for programmers, since threads creation occurs where the slave block starts and the synchronization occurs automatically at the end of the slave block. Besides, at the end of slave block, all slave threads send their works back to master thread (using the slave block parameter buffer) in order to allow it to merge all results. This communication procedure is also transparent to developers. Finally, the implementation of the load balance policy is also hidden from developers. In fact, slave threads receive their workloads according to the policy informed by parameter in the slave block. The implementation of these policies is entirely automatic generated by our pre-compiler system. Three load balance policies are available:

– `POPP_LB_STATIC`: the workload is divided by the number of threads. The resulting chunks are then statically assigned to slave threads as they start their computation;
– `POPP_LB_DYNAMIC`: the workload is divided in chunks, each one containing a number of tasks defined by the number of slave threads (finer task grain). When a thread finishes a task, it dynamically asks for another one to the master thread until there are no more tasks to be computed;
– `POPP_LB_COST`: the workload is divided in chunks, each one containing a number of tasks defined by the number of slave threads. Tasks are reorganized in such a way chunks have similar computational costs. Chunks are dynamically assigned to slave threads during execution time.

4.4 Levels of Parallelism

In the DSL-POPP, we use nested and combined patterns to achieve sub-level parallelism and hybrid patterns combination. It is important to mention that our implementation allows the use of nested patterns only inside slave (Master/Slave) and stage (Pipeline) blocks, not in the master block. Figure 6 shows the threads flow control graph for possible uses of nested and combined patterns implementations for a two level parallelism.

In Figure 6(a), we illustrate the use of nested pipeline patterns. Figure 6(b) and (d) present combination of pipeline and master/slave patterns (hybrid versions). Finally, Figure 6(c) shows how is the control flow when nested master/slave patterns are used inside of the slave block.

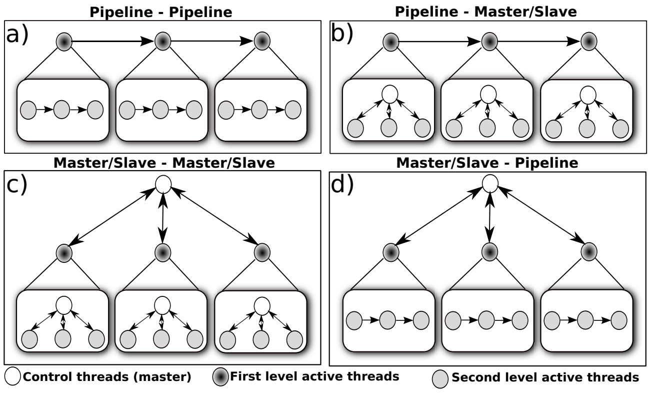


Fig. 6. Overview of thread graph in DSL-POPP

Also, it is possible to notice through an analysis of the control flows in Figure 6 what are the active threads during the execution time of a given application (taken in account for performance evaluation measurements). Control threads (representing the master thread) do not perform significant computation and appear only to facilitate the comprehension of the abstract representation of the master/slave pattern control flow. An example demonstrating the use of nested and combined patterns over a real application is presented in the next section.

5 Experimental Evaluation

In order to carry out an experimental evaluation of DSL-POPP and its features, we use it to parallelize an image processing application which applies a sequence of filters in a set of input images. In this section, we start briefly describing how we parallelize the application using DSL-POPP (Section 5.1). After, we introduce our experimental scenario in terms of platform and set of tests (Section 5.2). Finally, we present and discuss the performance results (Section 5.3).

5.1 Application Description and Implementation

We chose an image processing application because it allows us to explore parallelism combining both patterns available in DSL-POPP. The input is a list of bitmap images over which three different edge detection filters are applied (Prewitt, Sobel and Roberts) sequentially.

```
#include<poppLinux.h>
int list_size , num_th;
char **list_buffer;
$MasterSlavePattern unsigned char *do_sobel(unsigned char *image,
int width,int height){
 @Master{
  unsigned char *filter_sobel=(unsigned char*) malloc(height*width);
  @Slave(num_th,filter_sobel ,height*width,POPP_LB_STATIC){
   unsigned char *filter_sobel=(unsigned char*)malloc(height*width);
   int x,y,u,v;
   unsigned char image_buffer[3][3];
   for(y=1;y<height-1;y++)
    for(x=1;x<width-1;x++)
     for(v=0;v<3;v++)
      for(u=0;u<3;u++)
       image_buffer[v][u]=image[(((y+v-1)*width)+(x+u-1))];
       filter_sobel[((y*width)+x)]=sobel(image_buffer);
  } //slave
 return filter_sobel;
 } //master
}
$PipelinePattern int main(int argc, char **argv){
 @Pipeline{
  //....pre processing...
  @Stage(1, filter, height * width){
   int width, height, row;
   unsigned char *image, *filter;
   for(row=0; row<list_size; row++){
    getImageSize(list_buffer[row],&width,&height);
    image=(unsigned char *) malloc(height*width);
    filter=(unsigned char *) malloc(height*width);
    memcpy(image,save_bmp2binary(list_buffer[row],image,width,height),
    width*height);
    memcpy(filter,do_prewitt(image,width,height),width*height);
   }
  }//stage
  @Stage(1, filter, height * width){
   int width, height, row;
   unsigned char *filter;
   for(row=0; row<list_size; row++){
    getImageSize(list_buffer[row], &width, &height);
    filter = (unsigned char *) malloc(height * width);
    memcpy(filter, do_sobel(filter, width, height), width*height);
   }
  }//stage
  @Stage(1, filter, height * width){
   int width, height, row;
   unsigned char *filter;
   for(row=0; row<list_size; row++){
    getImageSize(list_buffer[row], &width, &height);
    filter = (unsigned char *) malloc(height * width);
    memcpy(filter,do_roberts(filter,width,height),width*height);
    save_bmp(convertName_bmp2filter(list_buffer[row]),filter,width,
    height);
   }
  }//stage
  //pos processing
 }
}
```

Listing 1.1. Overview of DSL-POPP Image Processing Algorithm Implementation

In Listing 1.1, we present one possible way to parallelize the application using patterns available in DSL-POPP. Only parts of the application code are shown in order to evaluate the use DSL-POPP key features. In this example, we implemented the sequence of filter as a pipeline and each filter was individually parallelized using master/slave pattern.

In line 4, the Sobel filter function is implemented using the master/slave pattern. We do not show here, but both Prewitt and Roberts filters apply the same master/slave pattern since the base edge detection algorithm is quite similar. Readers can notice that the use of the master/slave routines does not require significant changes in the sequential code. We only declare the Sobel function using master/slave syntax routines, and the `filter_sobel` variable (line 7) defined in the master block receives the slave results. The double declaration of `filter_sobel` is necessary because variables are private in each code block. Moreover, we used the static load balance policy (`POPP_LB_STATIC`) in the slave block (line 8). In this implementation, the outermost for-loop (line 12) will be automatically split among the slave threads during the automatic code generation.

For the main function (line 22), a pipeline routine (lines 23 to 54) performs all pre-processing instructions (*e.g.*, input images list allocation and organization) in the pipeline block and introduces the declaration of all pipelines stages (lines 25, 36 and 45). Again, no significant changes in the sequential code were necessary. We kept the same for-loop construction for all stage blocks. The course of a single input image through the pipeline stages is transparent for the developer since DSL-POPP analyzes the code blocks declarations and automatically generates a code in which the image is moved to the next stage through the buffer parameter. Once again, all internally declared code blocks variables are private.

5.2 Tests Scenario

We created a set of different implementations of the image processing application intending to highlight how patterns can be easily combined in different ways using DSL-POPP. Evidently, some of the pattern combinations presented better results than the others and many others combinations were possible. What is important in our point of view is that DSL-POPP make it easier to create those parallel versions offering a way to compare parallel solutions for the same problem with less development effort. The following implementations were evaluated:

– **Test-1**: implements just one level of parallelism using master/slave for the image filter functions. In this scenario, we employed 3 to N slave threads;
– **Test-2**: achieves parallelism using pipeline in the main function and master/slave for filter functions. Tests in this implementation were carried out using one thread per stage combined with 1 to N slaves threads;
– **Test-3.1**: both main and filter functions were implemented using master/slave. For this test, we used 3 slaves threads in the main function with 1 to N slaves in the filter functions;
– **Test-3.2**: implements the same patterns routines that Test-3.1, but the main function combines from 1 to N slaves threads and the filter functions only execute with 3 slave threads;

- **Test-4**: the main function is implemented using pipeline in which each stage employs 1 to N threads;
- **Test-5**: just implements master/slave in the main function in which 3 to N slaves threads are used.

Our results were obtained by computing the average execution time of 40 executions for each thread count. We fixed the number of necessary samples using a 95% confidence interval. For our experimental evaluation, we used 40 input images of size 3000×2550. The target architecture is composed of two Intel Xeon E7-2850 (ten cores each) at 2.0GHz and 80GB of main memory running Ubuntu-Linux-10.04-server-64bits. It is important to mention that experiments with more than 20 threads were possible due to virtual nodes (hyper-threading).

5.3 Performance Results

For the performance evaluation, we calculate the speedup as well as the efficiency through an average of 40 executions. We plotted the performance results in Figure 7.

The experiments results show that Test-1 does not achieve acceptable performance when the number of threads increases. This occurs in this scenario because we are doing the parallelism only in the image filters and the grain becomes too small as the number of threads grows. A better performance is achieved in the Test-2, in which we also explore parallelism in the main function with a pipeline routine at the same time as the filter functions with master/slave routine. Nevertheless, Test-2 does not scale well after 15 threads.

Test-3.1, which uses nested master/slave patterns, has similar performance to Test-2 even though the main routine in that case was implemented using the pipeline pattern. Test 3.2 presented the best results due to a better match between the number of active slave threads used and the static load balance policy applied. In fact, the limitation of the slave threads to 3 in the second level helped to avoid the computation of very fine grain tasks.

In Test-4, only the main routine was parallelized using the pipeline pattern. In this case, as the number of threads grows, pipeline stages become multi-threaded. Results indicate a better performance with less threads than Test-2 and Test-3.1, but they are not better than Test-3.2 possibly due to slightly different computational costs in each pipeline stage. Finally, Test-5 which implements just a one level master/slave pattern, presented loss of performance in some threads configuration (12, 18 and 24) due to a larger grain that does not match well with the static load balance policy we used.

At this point, it is important to stress out that all six implementations were carried out with very few modifications in the original source code. The essence of the algorithms itself was not changed, only the structure of the parallel solutions were inserted in the code. Even for more complex hybrid implementations, the effort to modify the code was minimum. Thus, we could test different solutions very quickly and find one with satisfactory performance and scalability.

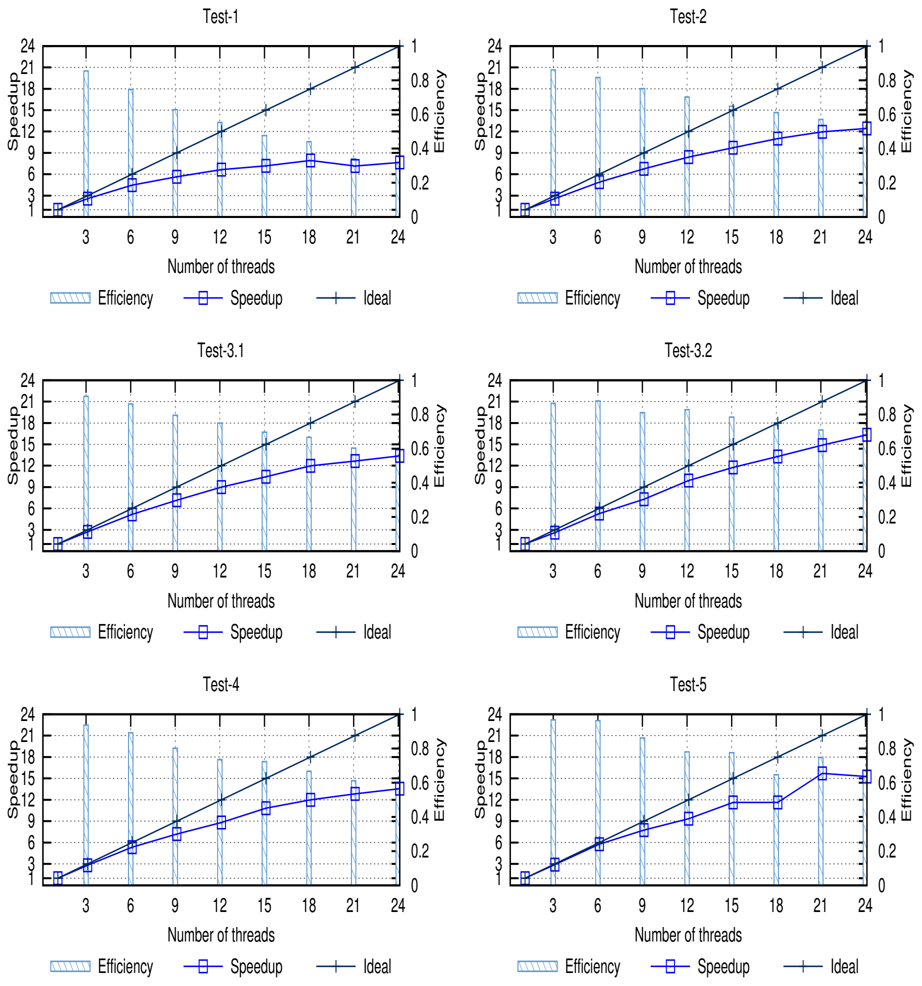


Fig. 7. DSL-POPP Results

6 Conclusions

In this paper, we proposed a Domain Specific Language for Patterns-Oriented Parallel Programming (DSL-POPP) that automatically generates parallel code for multi-core platforms. DSL-POPP offers primitives and programming environments to implement parallel code based on patterns with the C programming language. The main idea is to completely hide from developers low level mechanisms necessary to implement flow control, threads synchronization and load balance in parallel programs. Additionally, structured patterns may be easily nested or combined to create more complex parallel solutions with more than one level of parallelism.

In the experimental evaluation we have shown, our automatically generated Pthreads code proved to be capable to achieve good performances for the chosen application. It was also possible to verify that different parallel implementations were very easily produced with very few modifications in the same original code. This confirms our original idea that using a DSL would allow us to increase transparency in pattern-oriented parallel programming due to the treatment of low level parallel mechanisms at the code generation phase.

As future works we intend to include other traditional parallel patterns to our DSL (*e.g.*, divide and conquer, heartbeat, map-reduce, among others). We also consider necessary to invest more time to investigate optimized techniques in the parallel code generation exploring memory affinity for instance.

Acknowledgments. Authors wish to thank Jean-François Méhaut and Márcio B. Castro from the *Laboratoire d'Informatique de Grenoble* (LIG) for the very useful and fruitful discussions about parallel patterns which have strongly contributed to set directions for this work. Additionally, authors would like to thank the support of FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul) and CAPES (Coordenação de Aperfeiçoamento Pessoal de Nível Superior).

References

1. Mattson, G.T., Sanders, A.B., Massingill, L.B.: Patterns for Parallel Programming. Addison-Wesley, Boston (2005)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-Level and Efficient Streaming on Multi-core. In: Programming Multi-Core and Many-Core Computing Systems. Parallel and Distributed Computing, ch. 13. Wiley, Boston (2013)
3. Ciechanowicz, P., Kuchen, H.: Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In: 2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC), Melbourne, Australia, pp. 108–113 (September 2010)
4. Karasawa, Y., Iwasaki, H.: A Parallel Skeleton Library for Multi-core Clusters. In: International Conference on Parallel Processing (ICPP 2009), Vienna, Austria, pp. 84–91 (September 2009)
5. Leyton, M., Piquer, J.M.: Skandium: Multi-core Programming with Algorithmic Skeletons. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Pisa, Italy, pp. 289–296 (February 2010)
6. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)
7. Intel Mccool, D.M.: Structured Parallel Programming with Deterministic Patterns. In: HotPar-2nd USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA, pp. 1–6 (June 2010)
8. González-Vélez, H., Leyton, M.: A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. Softw. Pract. Exper. 40(12), 1135–1160 (2010)

9. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible Skeletal Programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
10. Aldinucci, M., Danelutto, M., Teti, P.: An Advanced Environment Supporting Structured Parallel Programming in Java. Future Gener. Comput. Syst. 19(5), 611–626 (2003)
11. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Skeletons for Multi/Many-core Systems. In: Proc. of the Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009), Lyon, France, pp. 265–272 (September 2009)
12. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A Structured High-Level Parallel Language, and its Structured Support. Concurrency: Practice and Experience 7(3), 225–255 (1995)
13. Botorog, G.H., Kuchen, H.: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing, Syracuse, NY, USA, pp. 243–252 (August 1996)
14. Gamma, E., Helm, R., Jonhson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (2002)