Parallelization of C# Programs Through Annotations

Cristian Dittamo, Antonio Cisternino, and Marco Danelutto

Dept. of Computer Science, University of Pisa,
Largo B. Pontecorvo 3, 56127 Pisa, Italy
{dittamo,cisterni,marcod}@di.unipi.it

Abstract. In this paper we discuss how extensible meta-data featured by virtual machines, such as JVM and CLR, can be used to specify the parallelization aspect of annotated programs. Our study focuses on annotated CLR programs written using a variant of C#; we developed a meta-program that processes these sequential programs in their binary form and generates optimized parallel code. We illustrate the techniques used in the implementation of our tool and provide some experimental results that validate the approach.

Keywords: Code annotations, generative and parallel programming.

1 Introduction

The correct and efficient design of parallel programs require to consider several different concerns, that are both difficult to separate during program development and depend on the target architecture features. Parallel aspects include how the computation is performed using processing elements (such as processes and threads), and how these communicate. These are non-functional aspects of a program since they do not contribute to define the computation but only how it is performed. Subscribing the separation of concerns concept, typical of Aspect Oriented Programming (AOP) [3], we recognize the importance of using proper tools (meta-data) to program the *non-functional* aspects related to parallelism exploitation. Meta-data (annotation) can be used by the programmer to "suggest" consciously how a parallel application can be automatically derived from the code. We will describe a proper meta-program (instead of compiler) that can efficiently handle such meta-data. This is different from the standard AOP approach where join points are defined using patterns, making the programmer unaware of program transformation details that will be applied afterwards. Moreover, in our case the meta-program does not restrict itself to inject code at the join points (identified by the annotations) as an aspect weaver would do, but it also perform a local analysis of the program to make its decisions about how to render parallel the sequential code. Since annotations are explicitly defined in the program code, it is possible to operate on the binary form of the program, allowing to adapt the transformation to a specific execution environment. Several approaches have been proposed in the past to relieve programmers from the

Y. Shi et al. (Eds.): ICCS 2007, Part II, LNCS 4488, pp. 585–592, 2007.

burden associated with the design of parallel applications: *parallel compilers* [5], *parallel languages* [10], *parallel libraries* [11] and more advanced programming models [12,13]. All these proposals rely on explicit support from the compiler (or tools involved in the compilation process) to generate a parallel version of the program and they don't provide any mechanism to avoid substantial code-tangling where the application core functionality cross-cut with the concern of parallel execution. In structured parallel programming systems based on the algorithmic skeleton concept the programmers qualitatively express parallelism directly at the source level, by properly instantiating and composing a set of pre-defined parallelism exploitation patterns/skeletons. The compiler and run time system is then in charge of adapting the provided skeleton structure to the target architecture features. Our approach is similar, though in our case the code transformation is performed at load time on the binary form of the program, relying on the compilation infrastructure typical of the virtual machine. Programming languages targeting runtimes, such as [1,2], provide the programmers with general mechanisms (extensible meta-data) to insert annotations into programs that can be read at runtime through the reflection `APIs`. Annotations are ignored by the execution engine, but they can be accessed by meta-programs in order to perform program analysis and manipulation at runtime. Annotation let programmers to provide all kinds of *non*-functional knowledge to the runtime system. In this respect annotations differ from directives traditionally used by compilation systems as in `OpenMP` and `HPF`: the compiler is not responsible for their processing. The prior knowledge about these system, however, may guide us in defining annotations to express parallel computations, and the semantics that a meta-program will attribute to them. In this paper we present `Particular`[1], a meta-program which reads annotated programs written in $[a]C\#$ [9] (an extension of $C\#$, read *annotated* $C\#$). $[a]C\#$'s main contribution is to extend the custom annotation mechanism provided by $C\#$ by allowing to annotate code blocks inside method bodies, other than declaration (classes, fields, and methods are examples). As shown in the following $[a]C\#$ code

```
class MyAnnotAttribute : ACS.CodeAttribute {}
 class AnotherAnnotAttribute : ACS.CodeAttribute {}
 class Example {
  public static void Main(string[] args) {
   [MyAnnot] { /* Code under the aegis of the MyAnnot attribute */
     [AnotherAnnot] { /* Code inside a nested annotation */ }
   }
   [AnotherAnnot, MyAnnot] /* Single statement */
  }
 }
}
```

the programmer simply encloses the name of the annotation inside square braces, as it is custom in $C\#$, to annotate a block or a statement. A pre-compilation step must be done in order to transform `ACS` code to standard $C\#$ code. Annotations can be retrieved at runtime through an extension of the reflection `API` that returns attributes in the form of a forest (to represent annotations with nested

[1] PARallelizaTIon of CUstom annotated intermediate LAnguage programs at Runtime.

scopes), where each node allows obtaining a cursor to the intermediate languages' instructions enclosed in the annotation.

In the rest of this paper we discuss the set of annotations introduced by our system and made available to the programmer for providing hints to our tool. It also explains the strategies adopted for rendering the parallel version of an annotated program. We finally provide experimental results on the performance of the generated code.

2 Prototype Framework

`Particular` defines a **set of the annotations** to provide hints on how to transform a sequential `C#` program into a parallel one. Since we were interested in exploring the viability of the approach, we decided to use a small and simple set of annotations. After the results we obtained, we believe that more annotations could be used, supporting a wider set of parallel programming paradigms, and perhaps leading to better performance. We introduced two annotations named `Parallel` and `Process`. The former denotes the parts of the code subject to parallel execution, and the latter denotes the specific parts that have to be included in an independent control flow (a thread when targeting `SMP` architectures, or a remote process when targeting clusters/networks of workstations or grids). In general, it is `Particular` that defines the semantics of annotations by manipulating the annotated code. The `Particular` tool should be used before an annotated example is loaded. This way, it is possible to customize the schema used for generating the parallel version depending on the actual architecture where the program should run. `Particular` reads an annotated executable program, in its intermediate (bytecode) form, and produces a new parallel program exploiting these annotations. It is important to notice that the entire transformation does *not* involve the original source program. Our approach is based on the manipulation of binary programs, therefore `Particular` can adopt strategies depending on the target architecture used for program execution. It may decide how annotations should be "better" rendered in the final program and what mechanisms should be used (threads, processes, etc.). Since our annotations may be of some help to the execution environment, a `Just-in-time` (`JIT`) compiler could be used to read these annotations. This extension, however, would require a tighter integration between annotated binaries and the execution environment. The basic parallelism exploitation pattern used to handle `Parallel` annotated sections of code is the master/worker paradigm: a master process delivers "tasks" to be computed to a worker, picked up from a worker pool according to a proper scheduling strategy. The worker completely processes the task, computing a result which is eventually given back to the master process. This process is repeated until there are new tasks to be computed. In our approach each `Process` annotation leads to the generation of a master process/thread and of a set of worker processes/threads. Whether processes or threads are generated depends on the kind of target architecture at hand, and "generation" in this context is a synonym of proper `IL` code generation. With our framework the developer can use any standard compiler (Microsoft, Rotor, Mono) targeting `CLR`. Whenever a

new architecture or pattern for parallelism will be introduced just `Particular` will be modified.

Design pattern to achieve parallelism. Our aim is to let the programmer focus on the functional aspects of the application, without having to concentrate at the same time on execution aspects. Annotations serve this purpose, and the user may even be unaware of the implementation details of parallel execution (though it is expected that he understand the meaning of annotations). For each annotation found, the meta-program creates a method containing the `IL` code of the annotated section (taking care of ensuring the appropriate handling of variables access); the annotated block is replaced by an asynchronous method call to the correspondent (new) method. Moreover `Particular` emits the `IL` instructions needed to call the (new) method (i.e. for reading actual parameters, to return values). Every new method code and meta-data are loaded into new library referenced by the original one.

State Analysis. Using annotations developers define new scopes inside a method body. Local variables can either be inside or outside a particular scope: when a scope is extruded from its original location, the meta-program should ensure that the state is properly handled, and the portion of the local state of the original method body should be made accessible to all the fragments generated from it. Unfortunately, the intermediate language form of the code does not provide a notion of scope for local variables, which are flattened into an array, thus we have to reconstruct a plausible scope for each of them. When an annotated scope is extruded into a new method m, a new type *State* is created; *State* contains a field for each variable that is not local with respect to an annotated scope. The signature of method m is extended with a parameter of type *State* in order to receive the non local state of the annotated scope. Therefore, values, not local to m, are passed as input arguments to it, thus making them local. These arguments are exposed directly in the signature of extruded methods generated by the tool. The only technical difference is that these variables are accessed with different `IL` instructions. Another critical issue in dealing with variables not local with respect to a scope, is that they are potentially accessible by many concurrent scopes `Particular` has to inject `IL` instructions in order to synchronize the access to the variables. It is important to note that the programmer must be aware of this behaviour, since the order of access into these shared variables can differ from the sequential execution of the original annotated program. After the analysis phase, the tool should rewrite the `IL` instructions of the annotated scope, so that references to arguments and local variables are consistent with the original method definition. If the instruction refers to a variable that should be synchronized, additional instructions are injected in order to avoid race conditions; the locking pattern used depends on the specific mechanisms employed to render the parallel execution (threads or processes): locks in the case of threads are achieved using monitors provided by the `CLI` infrastructure, in the case of multiple processes we delegate the master to deal with the synchronization and semaphores are used. At first sight, one might think that the annotations we

have introduced allow to express only a fixed number of processes to be spawned by the translated code. This is not true: in our parallelization pattern, if we annotate the body of a `for` loop, the main thread will spawn a worker for each execution of the body. Again, it is the responsibility of the programmer to ensure that the unbounded spawn is preferable, although the master can use schemas that impose an upper bound to the number of workers.

Meta-program implementation. As discussed in the previous sections, the transformation program processes the program in its binary form. The main advantage of this approach is that the strategy used to render the annotations is deferred at deploy time, or even at runtime[2]. Therefore, the transformation can take into account the particular geometry of a distributed/parallel system and make specific decisions on how to perform the rendering of the annotations. Another advantage of the approach is that the programmer can rely on the standard development toolchain to develop and debug the program in its sequential form, focusing on the functional aspects of the application. Code generation exploits the facilities provided by the `CLI` infrastructure in the reflection `APIs` The library is responsible for generating the bytes of an executable, allowing the programmer to emit opcodes and arguments in a symbolic form; it is also responsible for performing the back-patching required to resolve forward branches. The algorithm adopts a two-step strategy:

- *step 1.* **Code Analysis**, analysis of the code as discussed previously, where the algorithm takes as input a forest of annotations retrieved through the `[a]C#` runtime. It returns an array of `Master`s that has as many elements as the `Parallel` blocks inside the annotated method;
- *step 2.* **Code Generation:** Annotated methods are processed and a new assembly is generated with the newer version of the method; the algorithm scans the method body and generates a new method for each worker by invoking the appropriate methods of the actual `Master`. The algorithm creates all the required types and their members (i.e. fields, methods, constructors, properties, and events) using the reflection `APIs`. The code generation strategy is based on copying instructions of the original program, without any attempt of code optimization. This is not an issue, however, since the `JIT` compiler already performs optimizations on the intermediate language.

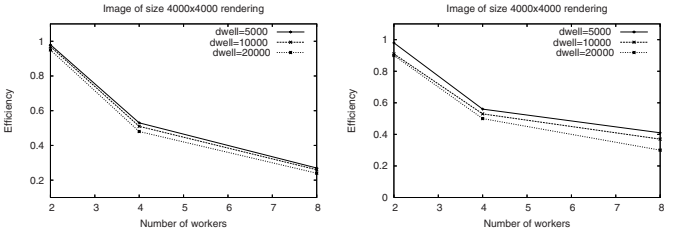
3 Experimental Results

To evaluate the effectiveness of our framework, we used `Particular` to manipulate two computations from two different classes of parallel applications: *task farm* and *static stencil* computations. We evaluated the effects of our transformation in terms of reduction in execution time for the whole application. We used heterogeneous platforms, both in operating systems (Windows and Linux)

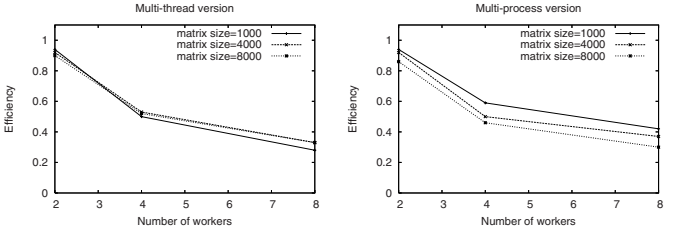
[2] Virtual machines can dynamically load types, thus a running program can invoke `Particular` and then load its output.

and architectures (uniprocessor, multiprocessor and dual core). This work should be considered preliminary, since most of the effort in developing `Particular` was spent in making the meta-program infrastructure; we expect that smarter plug-ins for the master task would lead to better performance. The `CLI execution environment` implementation used is `Microsoft .NET CLR v2.0` under `Microsoft Windows` systems, whereas `Mono v1.1.15` is used under the `Linux` system. We considered two applications, representative of a large number of parallel applications: a *task farm* (i.e. embarrassingly parallel) application (namely the parallel rendering of the Mandelbrot set) and a stencil, data parallel application (namely Jacobi's method used to approximate the solution of the Laplace Finite Difference over a matrix of values). For the task farm, the maximum number of workers used is not fixed, and the parallel pattern `MAP` manages to use the available resources according to a "best effort" policy. We rendered images of different sizes, with a different color rendering precisions. The graph `(a)` below shows the efficiency of the multi-threaded version of the renderer, with an image of 4000x4000 points and different precision values as inputs. In order to reduce the communication overhead, each worker computes the color of the points in its region and stores the results in a local queue. Once the computation ends, the worker copies its elements into the global queue. Therefore, synchronization is required only while copying results. Using a dual core processor we reached the best efficiency employing 2 workers only. As expected, increasing the number of workers causes an increase in processors contentions and race conditions while accessing the global circular queue. Therefore, the wait time spent on synchronization increases as well as the execution time for the whole application. Using larger precision values does not allow to have better performance because of the trivial type of computations made for each iteration. Similar results are obtained with the multi-processes version of the application. In this case a larger color precision causes the communication costs to rise, thus reducing the overlapping of communications with calculus. The graph `(b)` below shows the efficiency of the multi-processes version of the renderers. For our experiments concerning the data parallel stencil application, we considered different sizes of the input array. The graph `(c)` below shows the efficiency values obtained testing the multi-threaded version of the application. As expected, the best performance is obtained on a dual core processor with 2 workers. Splitting the input array into more blocks causes a decrease in the time spent for computation and increases the number of race conditions between workers when accessing the shared array. Having a larger input array increases the computations on local elements but at the same time causes an increment in the number of synchronizations required to access the shared array; therefore, the performance obtained decreases when larger arrays are utilized. Slightly better performance results are obtained with the multi-processes version as shown in the graph `(d)` below. In this case, the `Master` is responsible for updating the elements with the new values computed by the `Workers`, hence the elimination of synchronization hints.

However, the blocks of rows are exchanged between `Master` and `Worker` through the network, leading to heavy communication costs. Therefore, the best



a. Mandelbrot's multi-threaded version b. Mandelbrot's multi-processes version



c. Jacobi's multi-threaded version d. Jacobi's multi-processes version

performance results with the use of 2 workers and the smallest input array size. When increasing the latter, the processing of Laplace's Equation cannot overlap communications, forcing the `Master` to wait for updates without performing any calculus.

4 Related Work

One of the most interesting approach to separate functional aspects from parallel code is based on `AOP`. In [4] an attempt is made using `AspectJ` for encapsulating parallelization in an aspect separated from the mathematical model concerns. `Code annotations` have been mainly used to enhance the flexibility and the efficiency of the compiling step and to support new language features, see [14]. `Code generation` is another of the most diffuse application for code annotations. The `XDoclet2` tool [15] for `Java` has been successfully used for performing code generation tasks. This approach to code annotation is based on source code manipulation. Program manipulation with bytecode transformation is a technique that has been employed in several applications, see [8,16].

5 Conclusions

In this paper we have presented `Particular`, a meta-program that transforms annotated programs relying on the reflection capabilities of the `CLI` infrastructure. Our transformation schema allows deferring the decision on how to render parallel a sequential program, considering the particular distributed/parallel architecture where it will be executed. In our approach, programmers can focus on functional aspects of the problem, relying on the well established programming

toolchain for developing and debugging. Annotations will be enough to drive our meta-program in the parallelization of the program at a later stage. Although this is a form of `AOP`, in our case join-points are explicitly provided by the programmer in the form of annotations and the weaving of the parallel aspect is performed at deploy time on the binary program. The weaving process, moreover, does involve analysis of annotated blocks, not only injection of code at join-points. Another important element of our system is the ability to plug in different code generation strategies. So far we have developed the basic strategies based on thread and process execution, though we expect that smarter plugins for the master task would lead even to better performances. The results of our tests look promising and encourage us to continue in our research.

References

1. *.NET Framework Developer Center: The Common Language Runtime (CLR)*, `http://msdn2.microsoft.com/en-us/netframework/aa497266.aspx` (2007).
2. T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification* 2nd ed., Prentice Hall PTR, (1999).
3. T. Elrad, R. E. Filman and A Bader, *Aspect-oriented programming*, Communications of the ACM, Vol.44 (2001).
4. B. Harbulot and J. R. Gurd, *Using AspectJ to separate concerns in parallel scientific java code*, Proceedings of the AOSD Conference, Lancaster UK (2004).
5. T. Brandes and F. Zimmermann, *ADAPTOR - A Transformation Tool for HPF Programs*, Programming Environments for Massively Parallel Distr. Sys. (1994).
6. C. Dittamo, *Sequential program parallelization using annotations (in Italian)*, Graduation thesis, Dept. Computer Science, Univ. of Pisa, Italy, 2006.
7. J. Miller, *Common Language Infrastructure Annotated Std*, Addison-Wesley, 2003.
8. A. Cisternino, *Multi-stage and Meta-programming support in strongly typed execution engines*, PhD thesis, Dept. Computer Science, Univ. of Pisa, Italy, 2003.
9. W. Cazzola, A. Cisternino and D. Colombo, *Freely Annotating C#*, Journal of Object Technology Vol. 4, No.10, ETH Zurich (2005).
10. High Performance Fortran Forum, *High Performance Fortran language specification, version 1.0*, Technical Report CRPCTR92225 Houston, Tex (1993).
11. OpenMP Forum, *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, Technical report (1997).
12. M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi and C. Zoccolo, *ASSIST as a Research Framework for High-performance Grid Programming Environments*, (2005).
13. A. Benoit, M. Cole, J. Hillston and S. Gilmore, *Flexible Skeletal Programming with eSkel*, Proc. 11th International Euro-Par Conference (2005).
14. R. Kirner and P. Puschner, *Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis*, In Proceedings of the 5th Euromicro International Workshop on Worst-Case Execution Time Analysis (WCET05), Palma, Spain (2005).
15. C. Walls and N. Richards, *XDoclet in Action*, Manning Publications (2003).
16. H. Masuhara and A. Yonezawa, *Run-time Bytecode Specialization. A Portable Approach to Generating Optimized Specialized Code*, In Proceedings of Programs as Data Objects, Second Symposium, PADO01 (2001).