ITAT 2013 Proceedings, CEUR Workshop Proceedings Vol. 1003, pp. 82–87
http://ceur-ws.org/Vol-1003, Series ISSN 1613-0073, © 2013 M. Brabec, D. Bednárek

Programming parallel pipelines using non-parallel C# code

Michal Brabec, David Bednárek

Department of Software Engineering*
Faculty of Mathematics and Physics, Charles University Prague
{brabec,bednarek}@ksi.mff.cuni.cz

Abstract. *Parallel and high-performance code is usually created as imperative code in FORTRAN, C, C++ with the help of parallel environments like OpenMP or Intel TBB. However, learning these languages is quite difficult compared to C# or Java. Although these modern languages have numerous parallel features, they lack the automatic parallelization or load distribution features known from specialized parallel environments. Due to the referential nature of C# and Java, the principles of parallel environments like OpenMP cannot be directly transferred to these languages. We investigated the idea of using C# as a programming language for a parallel system based on nonlinear pipelines. In this paper, we propose the architecture of such system and describe some key steps that we have already taken towards the future goal of extracting both the pipeline structure and the code of the nodes from the C# source code.*

1 Introduction

Parallel programs are usually designed within the framework of a specific paradigm like thread-based, task-based, or pipeline paralelism. Such a framework is either explicitly used by the programmer in the form of a library like Intel TBB, or it is hidden inside a compiler capable of automatic parallelization like C++/OpenMP.

Pipeline parallelism is a paradigm which receives increasing attention due to its relation to stream processing; in its generalized, branched pipeline form it is also sufficient for data-processing applications including relational or RDF databases [6]. The explicit specification of data flow in a pipeline also helps in NUMA or distributed applications where the cost of data movement is important [3].

Unfortunately, pipeline parallelism was not studied as thoroughly as other forms of parallelism – while automatic parallelization within a thread-based or task-based framework has been implemented in many systems including FORTRAN, C, and C++ compilers, extracting pipeline structure from program code is still in the stage of experiments [11].

Bobox [3] is a parallel execution enviroment based on generalized branched pipelines which connect a set of execution units called *boxes*. A Bobox application is composed of two components: the *model* which describes how boxes are interconnected and the *box code*, i.e. the implementation of all boxes used in the model.

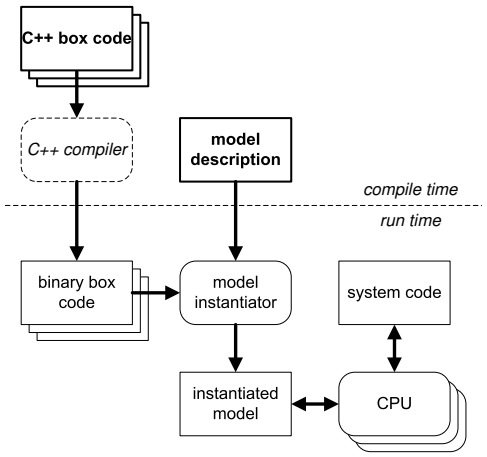

Fig. 1. The basic architecture of Bobox

As shown in Fig. 1, the code of individual boxes is compiled from their C++ source code and linked together with Bobox system code at run time. The run-time representation of the model, called *instantiated model*, is created by the *model instantiator* from the text-based model description and the binary box code. After instantiation, the model is assigned to a set of CPUs and executed.

When created by humans, Bobox models are usually written in *Bobolang* [3], a declarative language whose purpose and principles are similar to netlist languages like SPICE [9]. Bobox models may also be generated from a query language using a *language front-end*, e.g. the SPARQL front-end [6].

Nowadays, Bobox boxes are programmed in C++ within tight restrictions imposed by the framework interface. Although simpler than within explicit thread-based or message-based parallelism, programming in Bobox is still a tedious and error-prone task.

In this paper, we propose a Bobox front-end which transforms the box code from C# to C++. During the

* This work was supported in part by the grant P103/13/08195S of the Grant Agency of the Czech Republic.

transformation, code is added to control pipelines and synchronization.

In the advanced version of the architecture, the front-end also extracts the model from a C# program, approaching the goal of automatic parallelization in the Bobox environment.

While the advanced version is a future goal, we have already taken the critical steps towards the basic version. We have studied and implemented the key analytical part of the proposed system, the *CIL analyzer*; in particular, we thoroughly studied the aspects of C# which made the problem different from known compiler algorithms for C++ or FORTRAN code.

The rest of the paper is organized as follows: In Section 2, we describe the motivation for our project and the goals that resulted from the motivation. Section 3 describes the architecture of the proposed solution as well as the justification for the use of C#. We will also compare our approach to related work throughout this section. In Section 4, we will discuss technical details associated with the choice of C# and the key components of our system. In the Conclusion, we will describe the current status and the future development of the project.

2 Motivation

The principles of Bobox, developed in accordance with the general pipeline parallelism paradigm, determine the means that a developer in Bobox possess. As we will show in the following paragraphs, the stress on maximum performance causes that programming in Bobox is not as straightforward as the pipeline approach promises.

2.1 Parallelism in Bobox

Bobox design principles impose some crucial restrictions upon the behavior of individual boxes. In particular, a box shall always execute purely serially, thus, any parallel execution occurs only among boxes at the plan level. This approach corresponds to inter-operator parallelism in databases.

In order to improve the degree of parallelism, most Bobox models require replication of boxes and introduction of data splitters and mergers as described in [6]. The replication is done by *model parallelizer* at compile time, using the knowledge of crucial parameters of the run-time environment like the number of cores.

The architecture of the compile-time part of the improved Bobox system is shown at Fig. 2; the run-time part remains the same as in Fig. 1.

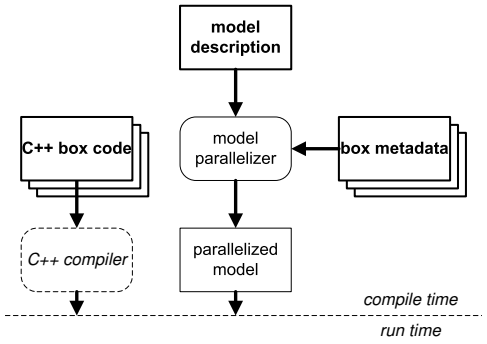

Fig. 2. Improving parallelism of Bobox models

For correct and meaningful transformation, the model parallelizer must know essential properties of the boxes, like statelessness or order sensitivity, as well as estimates of their quantitative behavior (e.g. input-to-output data size ratio). These properties are described in *box metadata*.

Currently, there is no mechanism to check whether the implementation of a box really satisfies the properties declared in its metadata. For database-like applications, this fact is negligible because the effect of individual boxes corresponds to physical algebra operators whose properties are well understood.

On the other hand, when Bobox is used as a parallel engine for general computing, the individual boxes correspond to routines, tasks, or similar elements of a parallel algorithm whose behavior is not always clearly defined. An error in box metadata may cause troubles similar to errors known from parallel programming like race conditions. Detecting and correcting these errors may be as demanding as checking race conditions. This fact undermines the Bobox aspiration to be a simpler programming environment than general parallel programming systems.

2.2 Programming in Bobox

Furthermore, coding the individual boxes is not as simple as it may seem with respect to the simple principles of Bobox.

Most algorithms are described naturally using loops taking input data elements one by one. However, for performance reasons, the data in Bobox are transmitted and received in blocks called envelopes. Consequently, the code of a box must explicitly handle envelope receiving and sending and, thus, deviate from the simple one-by-one arrangement. Explicit envelope handling may be quite painful, especially in cases where the inputs and/or outputs are not synchronous (e.g. in the ordered merge algorithm).

In addition, the original Bobox principles required that the code of a box should never enter a block-

ing call. This required restructuring the code so that envelope handling is done outside of the main box routine. Although this principle corresponds to event-driven programming which has been successfully used for years, it is unnatural in the context of most numerical and many data-processing algorithms.

The problem of blocking calls was solved in later versions of Bobox by the use of *fibers*, i.e. lightweight threads allowing to suspend execution of a box code anywhere. However, this solution comes at the cost of stack switching and, thus, slightly worse performance mainly due to larger number of cache misses.

2.3 Goals

As demonstrated in the previous paragraphs, using the pipeline paradigm under ultimate performance requirements lead to several problematic arrangements in Bobox. It became obvious that implementing boxes directly is quite difficult task and that returning back to natural implementation of algorithms would require a substantial change to the programming environment.

A natural programming environment for Bobox shall unload the burden of communication and envelope handling from the programmer. For performance reasons, the envelope handling shall not be hidden in runtime libraries – it is necessary to transform the code from natural one-by-one loops into event-driven code.

In addition, the programming environment shall also maintain the coherence between box code and box metadata, either by checking whether the box code satisfies the box properties given in advance or by generating the box metadata from the box implementation.

Furthermore, the programming environment may assist with fine-grained parallelism: If the box as a whole satisfies the conditions necessary for coarse-grained parallelism achieved by pipelining or partitioning, then it likely satisfies similar conditions for applying vector instructions.

3 Approach

The goals defined in the previous section naturally lead to the concept of code transformation and/or translation from a user-friendly programming environment to the C++ box code.

3.1 Language

The use of C++ at the output stage is dictated by the implementation language of the Bobox core and the unmatched performance of the code generated by C++ compilers.

On the other hand, the language at the input side is a subject of discussion. Given the output language, the use of C++ would be natural; however, analyzing and transforming C++ code is extremely difficult because of its complex syntax and permissive pointer semantics. Furthermore, the formerly widespread knowledge of C++ has nowadays retracted to devoted programming professionals – in e-science environment, they are not always available.

Since Bobox is targeted at scientific and data-intensive computation beyond the borders of numerical computation, languages like FORTRAN or Mathematica were disqualified due to their poor ability to handle sophisticated data structures.

There were many attempts to introduce a non-imperative programming language for parallel programming like Lustre, F#, or PigLatin. None of the new languages attracted sufficient attention of programmers, rendering them useless for a general-programming environment.

Given the observations mentioned above, our choice narrowed to modern, widely-accepted strongly-typed general-programming languages – Java and C#. Although they are only the least bad choice among our options, there are at least two important advantages of these languages:

First, there are many programmers fluent in these languages.

Second, both languages compile via standardized bytecodes – thus, our implementation may, hopefully, use the bytecode produced by standard compilers, bypassing the tedious implementation of specialized language front-end.

For our system, we finally decided to use C#, although the preference over Java was somewhat arbitrary.

3.2 Architecture

The architecture of the proposed system is shown in Fig. 3. The boxes are implemented in C# and compiled by a third-party C# compiler (Microsoft Visual Studio or Mono). The compiler produces an intermediate representation called CIL and standardized by ECMA/ISO/IEC [1]. The CIL code is then analyzed and box metadata are created. The analyzed intermediate code is then passed to the *box generator* which generates C++ source code of the boxes. The rest of the process is the same as in Fig. 2 – the code is compiled by a third-party C++ compiler (Microsoft Visual Studio or GNU C++) while the box metadata is used by the model parallelizer.

In the proposed system, an application consists of model description and box code just like in the plain system from Fig. 1 with the visible difference that the

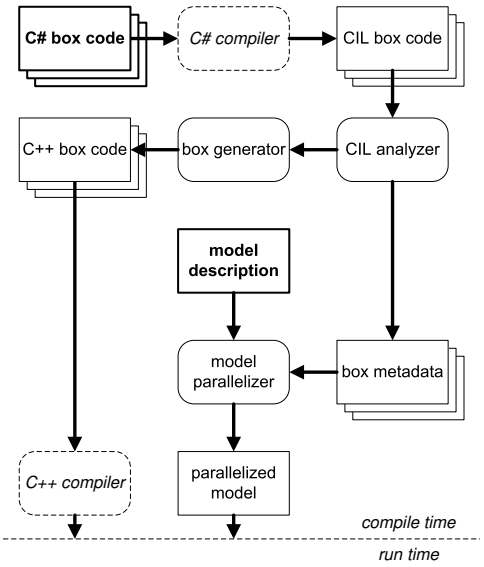

Fig. 3. C#-to-C++ box code generator

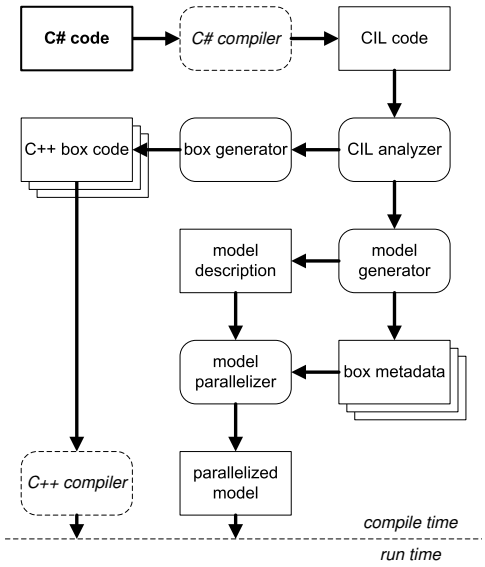

Fig. 4. C#-to-Bobox compiler architecture

code of the boxes is implemented in C# instead of C++. Nevertheless, the new system offers the following advantages:

The envelope handling is added to the code automatically, allowing the programmer to focus on the nature of the algorithm. The box metadata, required for the application of model parallelizer as in Fig. 2, are extracted automatically from the source code, ensuring their coherence.

3.3 Advanced architecture

Figure 4 shows an advanced version of the proposed architecture. Here, the source code consists of C# code of the complete application. This is compiled into CIL as in the previous case. The advanced analyzer breaks the application code into boxes and extracts the model automatically from the global structure of the code. The following phases are the same as before.

The advanced version is far more ambitious than the basic architecture, it essentially consists of automatic coarse-grained parallelization of C# code. Such level of program transformation is long known for FORTRAN [7], it was succesfully implemented for C [12] and similar goal was achieved with the help of profiling information in [11]. Among languages with referential semantics, coarse-grained parallelization was attempted in Java [10]. However, no such attempt was described for C# yet.

3.4 The effect of referential semantics

Of source, C# and Java differ from the target C++ language by their reliance on referential semantics – to compile from C# or Java to C++, one must either simulate the referential semantics in C++, or restrain the input code from using the referential semantics.

When used on local variables or stand-alone classes, the reference nature may be stripped off by object inlining as shown in [5]. However, this technique does not work on link-based data structures including many standard containers. It means that the standard container library must be replaced by a different set of containers that will discourage the use of references. This fact may certainly confuse programmers used to standard containers; nevertheless, learning a new set of containers is certainly easier than switching to another language (C++) completely.

4 The analyzer

The structure of the CIL analyzer closely follows the series of transformations and analyses used to prepare the code for parallelization. The optimization steps and their order are as follows:

- Preliminary transformations
- Preliminary code analysis
- Dependence testing

The following paragraphs briefly discuss the most important steps; details may be found in [4].

4.1 Preliminary transformations

This step includes procedure integration and code verification. Procedure integration (also called inlining)

replaces calls to procedures with their bodies – of course, such a transformation leads to code expansion and is impossible in the case of recursion. However, given our motivation and architecture, it is applicable and it is easier than inter-procedural analysis that is usually necessary before automatic parallelization.

The main purpose of procedure integration in our system is to remove unnecessary dependencies caused by parameter passing in the referential semantics of C#. Even though the procedures bound by a call could be analyzed independently, the integration allows the flow of data be accurately analyzed.

Code verification is a process designed to check if the code follows the restrictions required for translation to box code. It is performed after procedure integration and it must make sure that the final code does not contain any unsafe code, forbidden instructions or constructs, including prohibited library elements.

4.2 Preliminary code analysis

This step gathers information about the control-flow constructs and then creates a list of all variables used in the method. Both types of information are later used during dependence detection, since a dependence may be based on data or control-flow. This step does not contain any transformations or optimizations and the code is not modified here.

This analysis recognizes five different types of constructs: loops, if/else branches, switch statements, protected blocks and return statements.

Variable recognition is not a simple task since the fields of an object shall be considered separate variables whenever possible; however, a fall-back to considering the object as a whole must be available when necessary.

In addition, there are special temporary variables created on the stack as a result of some operation and later consumed by some other instruction. These variables are recognized by a stack simulator and they represent the relationship between instructions that constitute separate commands.

4.3 Aliasing

Aliasing is the name for the fact that multiple symbols (may) represent the same memory location. If the analyzer is not able to determine what pointers or references reference the same memory then it must conservatively assume that they can reference the same memory.

Aliasing in .NET is simplified by two important facts. There are no pointers and the references are controlled by the type system which forbids certain references to address the same object. Another important fact is that the reference must always address a valid object; it cannot be assigned some random address.

In addition, procedure integration used in this work can remove parameter aliasing because the formal parameters are removed in the process.

Regardless of these factors, exact analysis of aliasing is an algorithmically unsolvable problem so the analyzer always uses a heuristic-based conservative approximation.

4.4 Dependence testing

Dependence testing is the most difficult part of this project. The CIL code is transformed to a structure that can be analyzed by well-known algorithms of dependence testing [2]. Note that the procedure integration done previously allows to bypass inter-procedural version of dependence testing.

There are two important facts that help dependence testing in our case. First, there are no pointers allowed and there are no arbitrary addresses, because everything must represent valid, allocated objects. Second, local variables are completely private and they cannot be modified anywhere outside the method, with the only exception of reference parameters and it is possible to check if a local variable have been passed by reference or not.

Parameters and local variables represent independent memory locations that can be accessed only by the method itself because passing parameters by reference was ruled out by procedure integration. Therefore, all reads and writes to different local variables or arguments are independent operations that do not collide with each other. However, there may be collisions when a field is accessed using two local variables referring to a single object.

Stack variables represent values added and removed from the stack and every variable is written and read just once. Every stack variable simply represents a single true dependence with a source in the instruction that created the variable and the sink is in the instruction that consumes it.

Two field variables can access the same memory, only when they access the same field in the same object, otherwise they are independent. To prove independence between fields, it is necessary to keep track of the object they belong to and all possible dependences must be considered when this object cannot be properly monitored.

Arrays represent the best opportunity for parallelization, but their analysis is the most difficult. The subscript analysis is a complex problem which can be

handled in several degrees of conservative approximation, presented for instance in [2].

Induction variables are defined by loop iterations and they are essential to understand the behavior of a loop. Given the syntax of loops in C#, it is more reliable to analyze the behavior of individual variables regardless of their presence in the loop heading.

Before the core dependence testing, the loops and their induction variables have been identified and array subscripts have been reconstructed, along with multidimensional arrays. The analysis of aliasing should provide some help for the testing and all the variables which have not been separated may be treated as a single variable for the purposes of this analysis.

With all this information at hand, dependence testing is a matter of applying appropriate algorithms presented in [8].

5 Conclusion and future work

We have successfully implemented key parts of the CIL analyzer as described in the previous section. This implementation answered the main open problems associated to the proposed architecture, namely it allowed us to state that:

The reference nature of C# does not create significant additional obstacles in the code analysis required for parallelization. In particular, most aliases and false dependences generated by references may be removed by procedure integration. The intermediate language (CIL) used by C# compilers does contain enough information to perform the required analysis. In particular, we developed the stack simulator to accurately analyse the data flow in a CIL procedure.

Note however, that these observations are valid when assuming C# code that serves the motivation described in Sec. 2.

It is doubtful whether our observations may apply for arbitrary C# code – at least, the use of procedure integration disqualifies recursive code. Nevertheless, some phases of analysis may be usable also outside our constraints.

To complete our goals, the box generator has to be implemented. We believe that all the evil was hidden in the details of the analyzer, so there is hopefully no algorithmically difficult part in the generator. On the other hand, the quality of the code produced by the generator strongly affects the performance of the system; thus, it requires extreme care when designing the generator.

Last but not least, although the system may be essentially usable as is, any real-life use of our system will require a set of containers to replace the prohibited reference-based standard containers.

In the advanced version of the architecture, the model generator must transform the dependence graph of the analyzed code into a Bobox model. Although it is essentially possible to do it in one-to-one manner, such a model will contain boxes so small that the execution will suffer from communication overhead and cache misses. To create effective models, careful cache aware decomposition strategy will be required – this is the most intricate item in our future work.

References

1. ISO/IEC 23271:2012. information technology. common language infrastructure (CLI). Technical report, ISO/IEC JTC1/SC22, 2006.
2. Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann San Francisco, 2002.
3. David Bednárek, Jiří Dokulil, Jakub Yaghob, and Filip Zavoral. Data-flow awareness in parallel data processing. In Giancarlo Fortino, Costin Badica, Michele Malgeri, and Rainer Unland, editors, *Intelligent Distributed Computing VI*, volume 446 of *Studies in Computational Intelligence*, pages 149–154. Springer Berlin Heidelberg, 2013.
4. Michal Brabec. Analysis of automatic program parallelization based on bytecode. Diploma thesis, 2013.
5. Zoran Budimlić, Mackale Joyner, and Ken Kennedy. Improving compilation of java scientific applications. *Int. J. High Perform. Comput. Appl.*, 21(3):251–265, August 2007.
6. Zbyněk Falt, Miroslav Čermák, Jiří Dokulil, and Filip Zavoral. Parallel SPARQL query processing using bobox. *International Journal On Advances in Intelligent Systems*, 5(3 and 4):302–314, 2012.
7. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35(8):66–80, August 1992.
8. Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann Publishers, 1997.
9. Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.
10. Frank Otto, Victor Pankratius, and WalterF. Tichy. XJava: Exploiting parallelism with object-oriented stream programming. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 875–886. Springer Berlin Heidelberg, 2009.
11. Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531 – 551, 2010.
12. W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 356–369, 2007.