A Methodology for the Derivation of Parallel Programs

Joy Goodman

Department of Computer Science, University of Glasgow

Abstract. I am currently developing a methodology for deriving parallel programs from functional specifications. By specifying the basic program in a functional language and using guided equational reasoning, a more efficient parallel program in a variety of languages and styles can be derived.

1 Introduction

Parallel programming is an important and growing area of computer science. It aims to provide the greater performance which is required in many areas of computing today. However it is not without problems, and these hinder its widespread use. These include difficulties in the following areas:

Writing parallel programs. It is far harder to write parallel programs than it is to write sequential ones, due to the larger number of details which must be kept track of.

Portability. With sequential programming, a single model of computation suffices for most machines, but this is not the case in parallel programming. Thus it is extremely hard, if not impossible, to write programs which will run on all parallel machines efficiently. In order to obtain efficiency, the programs must be rewritten to run on the new machine.

Choices in the course of writing programs. There are many more choices to be made when writing a parallel program than when writing a sequential one, and each one must be made correctly in order to obtain maximum efficiency.

Demonstration of correctness. According to Pierre America, "Because of the non-determinism inherent in parallel systems, testing alone cannot provide much confidence in their correctness. Rather, it should be proved rigorously (and possibly even formally) that a program meets its specifications." [1].

In conjunction with Gudula Rünger and my supervisor John O'Donnell, I am developing a methodology to address and help overcome these problems.

2 The Methodology

The methodology which I present provides a framework for the derivation of parallel programs. It divides this derivation into a sequence of stages at many of which decisions must be made. This leads to a tree-like structure of possible derivations.

This method of parallel program derivation is based on that used by Rünger and Rauber in TwoL [4], and the methodology I am working on is based on the one presented in [3].

2.1 Program Transformation

The basic mechanism used in the methodology is program transformation. Program transformation is the process of transforming one program into another which produces the same result. We do this via equational reasoning, *i.e.*, by replacing parts of a program by other code which is equivalent. In a functional language with no side effects such as Haskell, equational reasoning preserves the meaning of the overall program. Thus by applying equational reasoning steps to a program we can transform the program into another one which produces the same result but not necessarily in the same way.

However if each step must be done from first principles there will be very many steps and the transformations will take a long time. Therefore, in order to assist the programmer, we aim to provide a set of theorems and lemmas which capture common transformation steps. We will prove by equational reasoning that these hold, so that the programmer can use them and be assured that their use will lead to correct programs without having to go through all the steps him/herself.

Example An example of such a lemma is:

$$\begin{gathered} \mathit{foldl1}\ f\ xs = \mathit{tfold}\ f\ xs \\ \text{provided } f \text{ is associative} \end{gathered}$$

In this lemma *foldl1* is the standard fold left function over non-empty lists. It is inherently sequential and takes $O(n)$ time. *tfold* is a parallel tree fold in which the processors are arranged in a tree structure. The data enters the tree at the leaves and each processor applies f to its two inputs and then passes the result to its parent in the tree. This function takes $O(\log\ n)$ time.

Therefore by applying this lemma the programmer can introduce parallelism into a program and reduce its time complexity while ensuring that the result of the program does not change.

2.2 Stages

There are many stages in the derivation, each of which will be clearly defined. However they fall into three main categories:

Specification These stages occur at the start of the derivation and the programs in them are written in a subset of Haskell. [1]

Intermediate During these stages the programmer can introduce features such as load-balancing into the program, and details of parallelism are also introduced. Again the programs are written in Haskell.

Implementation During these stages the program is transformed firstly into a version of Haskell which is similar to the target language and finally into the target language itself. This language can be any parallel programming language, although so far we have only used C+MPI.

The stages (and therefore the choices) which are more machine-specific are arranged later in the derivation tree. Therefore, in rewriting a program for a different architecture, you can reuse the programs from earlier stages in the tree. Only the later ones which are dependent on the particular architecture or type of architecture need to be redone.

2.3 Transformation Steps

As mentioned in Section 2.1 we transform the programs using equational reasoning aided by the use of theorems and lemmas. Since the

[1] Although we have chosen to use Haskell in this methodology, there is nothing inherent in the methodology to prevent the use of any pure functional language.

stages will be clearly defined, the transformations between them can also be of set forms. We aim to encapsulate these set forms in the provided set of lemmas so that a program can be transformed from one stage to the next by applying one or more of these lemmas.

Although the method allows the transformations to be done by equational reasoning if required, it does not insist that this is so. If proof of correctness is not required and the programmer can do some of the steps without using formal transformations, then he or she is free to do so.

2.4 APMs

The methodology also provides specific help with the parallelism by providing a mechanism for separating algorithmic and machine-dependent concerns. This mechanism is the ability to define abstract parallel machines (APMs) [3]. These sit alongside the main programs and provide the parallel operations which are used within them. There are a variety of APMs, modelling different architectures at various levels of abstraction. It is possible to change the APM used by a program without affecting the program itself in any major way (at least up to a certain stage in the derivation). Thus it is possible to modify a program for different architectures and programming models relatively easily.

One of our aims is to develop a library of useful APMs, but this does not prevent users from adding APMs and parallel operations of their own.

The APMs are defined in a set format and style, known as a ParOp definition(first given in [3] and further discussed in Section 2.5 below). This style is particularly useful in making explicit the placement of data and its communication, but is not in general executable: it is used primarily for specification purposes. Therefore we are working on the development of methods for deriving Haskell implementations from ParOp definitions.

2.5 ParOps

A ParOp definition is a form for writing parallel operations in which the initial and final states of the processors and the communication between them are made explicit. It is a generalised function which can represent a multitude of parallel operations and is parametrised by functions f_i and g_i which describe what the processors do to their states, what they output to the other processors and which other processors they accept inputs from.

A simplified form of the ParOp definition, in which there is no input from or output to the outside world, is as follows:[2]

$$
\begin{aligned}
\mathsf{ParOp}\ (\sigma_0, \ldots, \sigma_{n-1}) \ &= \ (\sigma'_0, \ldots, \sigma'_{n-1}) \\
\mathsf{where}\ (\sigma'_i, A_i) \ &= \ f_i(\sigma_i,\ g_i\ (V))\ \text{for}\ i \ = \ 0, \ldots, n-1 \\
V \ &= \ (A_0, \ldots, A_{n-1})
\end{aligned}
$$

where

- σ_i is the value stored in processor i at the start of the ParOp computation.
- σ'_i is the value stored in processor i at the end of the ParOp computation.
- A_i is the value sent out to the interconnection network by processor i.
- V is the vector of all the A_i's.
- g_i is a user-defined selection function which takes the values sent out by all the processors and selects which ones to take as input to processor i.
- f_i is a user-defined function which takes the value in processor i (σ_i) and the values passed to it by other processors ($g_i\ (V)$) and calculates the new value in the processor (σ'_i) and the values to pass to other processors(A_i).

Example An example of a ParOp definition is:

$$
\begin{gathered}
f_i\ (x, (in_1, in_2)) = (\tfrac{in_1 + in_2}{2}, x) \\
g_i\ (A_0, \ldots, A_{n-1}) = (A_{(i-1) \bmod n}, A_{(i+1) \bmod n}) \\
\text{for } i = 0, \ldots, n-1
\end{gathered}
$$

This definition describes a parallel operation in which the processors are arranged in a ring and each processor averages the values in its neighbouring processors.

[2] The full form can be found in [3] and [2].

For comparison purposes an equivalent Haskell function is:

$$f\ xs = [(xs!!(i-1)\text{`}mod\text{`}n + xs!!(i+1)\text{`}mod\text{`}n)/2 \mid i \leftarrow [0..n-1]]$$
$$\quad \textsf{where } n = length\ xs$$

As you can see from this example, the ParOp definition may be more complicated but makes explicit the distribution of data between processors and the communication which is going on.

2.6 Cost model

As an aid to the programmer, we aim to provide a cost model together with the methodology and a method of using it to help to derive efficient programs. The idea is that at each stage at which decisions need to be made the cost of each decision can be analysed so that the most cost-effective decision is made. As each decision is affected by the others, and the optimal solution may be obtained by taking some locally non-optimal choices, this is not a trivial task. This is work in progress, and is again based on the TwoL model [4].

2.7 Incorporation of user ideas

One question is how interesting ideas which the programmer may have are incorporated into the program. These include such ideas as load balancing, data distribution and communication patterns. This is still a bit vague at the moment, but the main idea is that the user presents the key stages of the function (*e.g.*, in a load balancing operation these might be: move data around, do calculations, move data back, combine values), leaving the actual functions as APM functions which he or she then specifies using the ParOp form and implements using provided techniques.

3 How the Method Solves the Problems

Our method helps to overcome the problems listed in the introduction as follows:

Writing parallel programs. The method guides the programmer through the transformations from the functional specification towards the target program.

Portability. The method makes it easier to rewrite programs for different architectures and machines in two main ways. It provides APMs, which separate algorithmic and machine-dependent concerns as described in Section 2.4. It also structures the derivation tree so that the least machine-specific stages are first which allows reuse as described in Section 2.2.

Choices in the course of writing programs. The method provides a cost model which helps the programmer to make good choices.

Demonstration of correctness. The method allows and eases the use of equational reasoning to prove correctness.

4 Case Studies

In order to demonstrate how the methodology works and show that it is useful in practice, we are carrying out several case studies using the model. We hope that these will highlight any problems and difficulties which exist with the model so that we can remove them.

At the moment I am working on a short simple case study in which the columns of a triangular matrix are summed. In order to achieve greater efficiency we use load-balancing to redistribute the data. A draft of this study is given in [2].

For the next case study we are looking for an application with the following characteristics:

- It should be a real application, not just made up for the purposes of the study.
- It should be non-trivial.
- It should have a potential for parallelism.
- There should be interesting choices possible in the derivation.
- It should be possible to complete it within a few months.

5 Conclusion

I have presented here work in progress on a methodology for deriving parallel programs. We hope that this will enable programmers to write parallel programs more easily and more understandably, as well as making it easier to prove correctness of such programs and to rewrite them for different architectures and machines.

Acknowledgements

I would like to thank John O'Donnell and Gudula Rünger who have worked on this methodology and Keith Wansbrough for his very helpful comments on this paper.
This work was undertaken with support from EPSRC.

References

1. Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.
2. Joy Goodman, John O'Donnell, and Gudula Rünger. Refinement transformation using abstract parallel machines. Draft paper, available at http://www.dcs.gla.ac.uk/ joy/research, Jan 1998.
3. John O'Donnell and Gudula Rünger. A methodology for deriving parallel programs with a family of parallel abstract machines. In *Third International Euro-Par Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 662–669. Springer, 1997.
4. Thomas Rauber and Gudula Rünger. Deriving structured parallel implementations for numerical methods. *The Euromicro Journal*, 41:589–608, 1995.