Compile-time Cost Analysis for Parallel Programming

Roopa Rangaswami

The University of Edinburgh,
Department of Computer Science,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ

Abstract. This paper focuses on the compile-time cost analysis of programs expressed in the BMF-style, which results in the selection of a cost-effective parallel implementation on a given topology.

1 Introduction

Developing efficient software for parallel computers is a difficult task, even for the specialist. An ideal parallel programming model should provide architecture-independence, a high level of abstraction, and accurate performance estimates.
The HOPP (Higher-order Parallel Programming) model is based on the Bird-Meertens Formalism (BMF) [Bir89]. BMF comprises of a set of useful higher-order functions, many of which are implicitly-parallel, and which enables programs to be expressed at a high level of abstraction. HOPP is based on a functional paradigm, and therefore, automatically inherits architecture-independence. The behaviour of the BMF functions is predetermined and this feature is exploited in building a cost model that aims to accurately predict the costs of programs. HOPP models the execution of data-parallel programs on MIMD (Multiple Instructions Multiple Data) machines to produce SPMD (Single Program Multiple Data) code.
In the HOPP model, parallelism in programs is expressed implicitly using the fixed set of BMF functions. The idea of using a fixed set of higher-order functions to express skeleton-based parallelism was originally proposed in [Col88]. A cost calculus based on the BMF model of parallel programming is presented in [SC93]. However, the calculus is at too high a level of abstraction to produce accurate performance estimates, and also only handles parallelism at the level of the outermost higher-order function. This paper focuses on developing a *realistic hierarchical* cost model for parallel computation based on BMF. The set of higher-order constructs will henceforth be referred to as the set of *recognised functions*.

2 The Features of the HOPP Model

The HOPP model comprises of three parts - the *program* model, the *machine* model and the *cost* model.

2.1 The Program Model

A program in HOPP is expressed as a *composition* of functions. *Composition* refers to functional composition. Each component is referred to as a *phase* of the program. A program could comprise of one or more phases, each with nested instantiations of recognised and user-defined functions. This is the *only* allowable structure for a program in HOPP. The set of recognised functions includes those that are borrowed from BMF [Bir89], and some additional ones [Ran96]. The latter functions can indeed be expressed in terms of the former ones. However, they have been included as recognised functions in their own right, not only because they commonly occur in problems, but also because they can be efficiently implemented in parallel. All the recognised functions operate on the *list*, which is the only data structure. In the following example, *plus* and *times* correspond to user-defined functions which add and multiply two integers, respectively. The program consists of three phases, each with instances of recognised (represented in boldface) and user-defined functions.

prog xs = (**fold** plus 0 ∘ **map** (**scan** plus 0)
∘ **cross_product** times ys) xs;

The HOPP model is independent of the base language. User-defined functions are implemented in a chosen strict functional language. However, these functions are not allowed to perform I/O operations [Ran96].
The cost analysis can be expected to produce accurate performance estimates only for *regular* problems. *Regularity* in this context implies a predictable communication structure and a resilience in performance to different data sets. The former is imposed by the nature of recognised functions. The latter is a limitation arising due to compile-time analysis, but cannot be enforced. The implications of *regularity* restrict polymorphism. The cost analysis needs type information in order to compute communication costs. Consequently, definitions of sequential functions that allow for full or restricted polymorphism are not permitted.

2.2 The Machine Model

The programs are targeted at distributed-memory machines which consist of a set of interconnected processors. The following topologies were considered: hypercube, 2-D torus, linear array and tree.

2.3 The Cost Model

For each recognised function, the cost model determines its cost of parallel execution on a given machine topology [Ran96]. User-defined functions only have a sequential implementation and are also referred to as *sequential functions*. Effectively, the cost model computes the costs of possible implementations for a given program on a given machine topology. The cost model has been implemented in the form of an *analyser*.

In the current scheme, parallelism is only exploited within each phase. The phases themselves are sequential and phase i does not commence until phase $i-1$ is completed. The cost of a program comprising of k phases is given by:

$$Cost = \sum_{i=1}^{k} C_{p_i} + \sum_{i=0}^{k-1} C_{i,i+1}$$

C_{p_i} is the cost of phase i, and depends on the nature and costs of its composing functions. $C_{i,i+1}$ is the cost of phase transition. It accounts for any communication cost that may be incurred in rearranging the output data of phase i to suit the implementation of phase $i+1$.

The cost of implementing a recognised function, F, operating on k input lists of sizes $n_1, n_2, \ldots, n_k$, in parallel on p processors, is represented by:

$$C = F((n_1, n_2, \ldots, n_k), p, C_f).$$

where, C_f is the cost of F's argument function. In the following representation, each recognised function is assumed to have only one list argument. The analyser performs a cost analysis for each phase in the program, for a given topology. The cost of the sequential implementation is also computed in each case.

1. For a phase consisting entirely of user-defined functions, no parallel implementation is possible. The cost of the sequential implementation is determined by that of the user-defined functions.
2. The phase could contain only one occurrence of a recognised function, F.
$$C_1 = F(m, p, C_f). \tag{1}$$
3. The phase could contain a recognised function, F, that has another recognised function, G, as its argument.
   (a) The cost of implementing F in parallel and G sequentially is:
   $$C_2^1 = F(m, p, G(n, 1, C_f)). \tag{2}$$
   (b) The cost of implementing G in parallel and F sequentially is:
   $$C_2^2 = F(m, 1, G(n, p, C_f)). \tag{3}$$
   (c) F and G, could be implemented in parallel on p_1 and p_2 processors, respectively, $(p_1 \times p_2 \leq p)$. However, on certain topologies, logical neighbours may no longer be physical ones. The cost function is, therefore, parametrised on the maximum number of hops to be traversed between logical neighbours, and is represented by l (≥ 1, depending on the nature of the topology).
   $$C_2^3 = F(m, (p_1, l_1), G(n, (p_2, l_2), C_f)). \tag{4}$$
4. The phase could contain three nested recognised functions. Seven possibilities arise, corresponding to the implementations of each of the three functions in parallel, any two in parallel and all three in parallel. Arguments similar to those in 3 can be applied. For a phase with more than three nested recognised functions, potential parallelism is exploited only up to the first three levels and the previous discussion applies.

The Search Tree The analysis of the program results in the construction of a search tree, in which the weights on the nodes at a level represent the costs of the possible implementations for the corresponding phase. The weights on the edges represent the costs of phase transition. The least-cost path in the search tree corresponds to the most cost-effective implementation for the program. There are four different costs involved.

- The predicted cost of sequential execution, T^s_{pred}.
- The predicted cost of the selected parallel implementation on p processors, T^p_{pred}.
- The actual sequential execution cost, T^s_{prac}.
- The actual cost of the selected parallel implementation, T^p_{prac}.

If the model reflects the practical performance, then

$$\frac{T^p_{pred}}{T^s_{pred}} \approx \frac{T^p_{prac}}{T^s_{prac}}. \tag{5}$$

2.4 The Performance of the HOPP Model

The accuracy of the predictions of the analyser has been studied on three example programs, targeted at the hypercube topology - Matrix Multiplication, Merge Sort [Ran95] and Jacobi Method for solving partial differential equations. The predictions of the analyser were confirmed by practical implementations on the Meiko Computing Surface [Ran96].

3 Conclusion

A cost analysis for the HOPP model has been discussed. Early results indicate that the cost predictions match practical behaviour. The main limitation is the non-applicability of HOPP to irregular problems. The expressiveness of the model constitutes an area of further research.

References

[Bir89] Richard Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[Col88] Murray Cole. Higher-order functions for parallel evaluation. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 8–20, August 1988.

[Ran95] Roopa Rangaswami. HOPP - a higher-order parallel programming model. In Marc Moonen, editor, *Algorithms and Parallel VLSI Architectures*. Elsevier, 1995.

[Ran96] Roopa Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD thesis, University of Edinburgh, 1996. To be published.

[SC93] David B Skillicorn and Wentong Cai. A cost calculus for parallel functional programming. Technical report, Department of Computing and Information Sciences, Queens University, Kingston, Canada, March 1993.