Chapter 3
Transforming Rapid Prototypes to Efficient Parallel Programs

Christoph A. Herrmann and Christian Lengauer

3.1 Introduction

Sequential programming has overcome its preoccupation with details of the machine organisation several decades ago. In contrast, the average parallel programmer spends a large amount of development time on organisational details like synchronisation, communication, marshalling, etc. Message passing libraries like MPI are useful aids at this level – but they give the programmer a large choice of complex options rather than removing organisational issues with bold abstractions.

We believe that, in many applications, the structure of the algorithm has a higher impact on performance than organisational details. In this case, it appears more useful to explore the design space by developing and evaluating several alternative prototypes rather than to choose one design more or less arbitrarily and optimise it. A rapid development of alternative prototypes is only feasible if the parallel code can be generated automatically from the prototype at an early design stage. Of course, one cannot expect the automatically generated code of the prototype to exhibit the performance of a manually tuned parallel program. On the other hand, it makes sense to base successive optimisations on a prototype with good asymptotic performance.

Our approach is to have application programmers construct prototypes in a functional language, without consideration of the target language which could later be used to implement time-critical parts of the final implementation directly. The functional language is supposed to provide a high level of abstraction, suitable for convenient prototyping. To liberate the application programmer from considerations of machine organisation and program optimisation, we provide a library of hand-optimised, efficient, parallel implementations for a set of powerful skeletons. A skeleton is a programming template which is parameterised by the application programmer with problem-specific customising
functions and commonly specified as a higher-order function. Thus, in contrast to a macro, a skeleton can perform recursive computations. Simple skeletons are, e.g. `map` or `reduce` which describe a data-parallel or tree computation. Powerful skeletons correspond to entire programming paradigms, e.g. `divide-and-conquer` or `branch-and-bound`.

In our past work with skeletons, we have concentrated on the divide-and-conquer paradigm (DC) as an example of a powerful, non-trivial source of massive parallelism. We have named our programming language for rapid prototyping \texttt{HDC} (for \texttt{H}igher-order \texttt{D}ivide-and-\texttt{C}onquer). The focus of our recent work has been on the development of an experimental compiler for \texttt{HDC} \cite{15}. \texttt{HDC} is syntactically a subset of the language Haskell, but with strict semantics. Strictness is needed to specify the time step and processor location of each operation. We gave Haskell preference among several advanced functional languages, due to its uncompromising referential transparency, its concise syntax for index-based computations and its rich support by compilers and tools for program development.

In \texttt{HDC}, a skeleton is simply a function for which the compiler expands and links in a hand-optimised implementation, rather than compiling the function from a defining body in the source language. To the caller, a skeleton is indistinguishable from an ordinary \texttt{HDC} function. In contrast to some other skeleton approaches \cite{2, 7}, \texttt{HDC} programs are not divided into a coordination level at which the parallel program structure is composed of a set of sequential processes and a level at which these processes are implemented. These two levels require the programmer to think in terms of parallelism from the beginning. In our approach, the functions in the program are initially not given a particular parallel structure, i.e. the program is implicitly parallel. Unfortunately, an automatic parallelisation often fails to achieve high efficiency, because static program analysis cannot always reveal which parts increase efficiency if parallelised and which parts mainly incur overhead.

During program refinement, in which the efficiency of the parallel program is going to be improved, the programmer can enforce the parallelisation or the production of sequential code for a particular function application, just by choosing a variant of that function.

We found that the integration of a small set of skeletons into a compiler is not sufficient for the efficient parallelisation of a large variety of algorithms. Thus, the \texttt{HDC} compiler provides a flexible mechanism for adding new skeletons. The advanced user can add a Haskell module to the \texttt{HDC} compiler which generates the skeleton implementation for all different contexts in which it is applied in the source program.

Even advanced compilation techniques have a very limited ability to exploit domain-specific knowledge, although this often makes all the difference in high-performance applications. Recently, the point has been made for a way of customising a compiler for special-purpose optimisations \cite{20}. Our approach is a step in this direction: it isolates domain-specific implementation issues, takes them away from the (general-purpose) compilation and gives them into the hands of an implementation expert of the domain – here, divide-and-conquer.