Efficient Parallelisation of Recursive Problems Using Constructive Recursion*

Magne Haveraaen

Institutt for Informatikk, Universitetet i Bergen, HiB, N-5020 Bergen, Norway
http://www.ii.uib.no/~magne

Abstract. Many problems from science and engineering may be nicely formulated recursively. As a problem solving technique this is very efficient. Due to the high cost of executing recursively formulated programs, typically an exponential growth in time complexity, this is rarely done in practice. Constructive recursion promises to deliver a space and time optimal compilation of recursive programs. The constructive recursive scheme also straight forwardly generates parallel code on high performance computing architectures.

1 Introduction

Recursive formulation of algorithms has for decades been considered an efficient problem solving method in computer science. The high run-time costs for most recursive programs has been hindering its general applicability. The standard operational interpretation of a recursive program forms a tree of recursive calls spreading out from the node of the initial invocation, i.e., *the root node*, and this is where the result of the computation is gathered. The shape of this tree is defined by the data dependency pattern of the recursive calls. The tree often has many nodes with identical subtrees. The efficiency of the computation, and its parallelisation, crucially depends on identical subtrees being merged into one, thus forming a compact *data dependency graph*. Also, the traversal order of this graph is important: if we trace the data dependency graph from the root, we need to instantiate all the nodes of the data dependency graph before any computation takes place. If we instead proceed in a *data-driven* fashion, starting at the leaves, we will only instantiate nodes when the computation reaches them. A node may be deleted as soon as the execution has passed on the value computed at it to the next node.

Of the three computational models, operational (Turing machines, imperative programs), syntactic (general grammars, term rewriting, λ-calculus and related functional programming), and semantic (μ-recursion), the semantic model readily generalises to recursion on data dependency graphs (from its traditional

* This investigation has been carried out with the support of the European Union, ESPRIT project 21871 SAGA (Scientific Computing and Algebraic Abstractions) and a grant of computing resources from the Norwegian Supercomputer Committee.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 758–761, 2000.

recursion on natural numbers). This generalisation is called *constructive recursion* [3, 2]. It includes the definition of the data dependency graph as an explicit component of a recursive program. A compiler may then use this information to create data-driven code distributed on the processors of a parallel machine. We have developed a programming language, Sapphire, embodying these ideas. A Sapphire program consists of three parts: the definition of the data dependency, the definition of the recursive function, and the initiation of the computation.

The next section sketches the principles of constructive recursion and associated compilation technique. Section 3 presents an example with timings. The conclusion gives some references to related work.

2 Constructive Recursion

Somewhat simplistically, a recursively defined function f can be abstracted to the form

$$f_{i_1,\ldots,i_m} = \phi(f_{\delta_1(i_1,\ldots,i_m)}, \ldots, f_{\delta_k(i_1,\ldots,i_m)}),$$

where the tuple $(i_1, \ldots, i_m)$ of variables range over some subset $D \subseteq \mathbb{N}^m$, the *index domain*, of the m-tuple of natural numbers. The functions $\delta_\ell : D \to \mathbb{N}^m$ define the *dependency pattern* or *stencil* of the recursion[1]. The recursion is well defined if the δ_ℓ define a well-funded partial order on the m-tuple of natural numbers $\mathbb{N}^m$, i.e., all paths generated by the δ_ℓ are finite, and there is a set of initial values $\epsilon_{i_1,\ldots,i_m}$ for every index-tuple $(i_1, \ldots, i_m) \in (\mathbb{N}^m \setminus D)$.

To get a constructive recursive definition of f, we also need to know the opposite of the stencil δ_ℓ. The opposite does not mean the inverse of each δ_ℓ. Instead we need a set of functions $\delta'_{\ell'}$ for $\ell' \in \{1, 2, \ldots, k'\}$, such that for every $(i'_1, \ldots, i'_m) \in \mathbb{N}^m$ for which there exists a δ_ℓ and $(i_1, \ldots, i_m)$ such that $\delta_\ell(i_1, \ldots, i_m) = (i'_1, \ldots, i'_m)$, then there exists a $\delta'_{\ell'}$ such that $\delta'_{\ell'}(i'_1, \ldots, i'_m) = (i_1, \ldots, i_m)$. (The number k' of opposite functions may be larger or smaller than k.) Knowing the stencil and its opposite makes us able to trace out the compact data dependency graph from the initial values.

Following [5], parallelising a program is to embed the program's data dependency pattern into the space-time of the target computer's communication structure. This information can be provided by a mapping of the nodes of the dependency graph to space-time coordinates of the target computer. Allowing the programmer to define this mapping, gives explicit control over the parallelisation at a very high abstraction level.

The Sapphire compiler uses this information to generate an imperative program which traces the compact data dependency graph from the initial values. The imperative program generates the nodes of the graph when they are first activated in the computation, and reclaims the nodes as soon as they have been computed and their values passed on in the graph. Parallel code is generated using MPI.

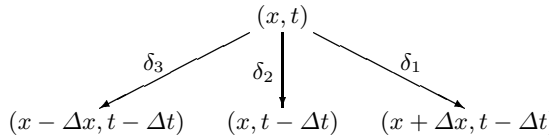
[1] If we limit the form of the δ_ℓ, we get a *recurrence relation*.

3 Example: Heatflow in One Dimension

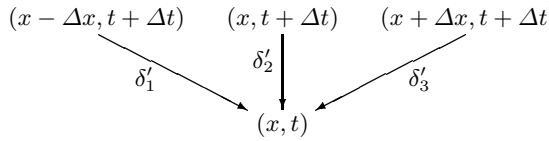
The one-dimensional heat flow equation, $\alpha^2 \frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$, describes the heat distribution $u(x,t)$ in a one-dimensional rod along the x-axis at time t. Here α is a physical constant describing the heat transfusion properties of the rod. To turn the equation into a recursion we may use the finite difference method both in time and space, giving

$$u(x,t) = \kappa u(x+\Delta x, t-\Delta t) + (1-2\kappa)u(x,t-\Delta t) + \kappa u(x-\Delta x, t-\Delta t),$$

where $\kappa = \alpha^2 \Delta t/\Delta x^2$. This simple program has 3 recursive calls per timestep. Its stencil is given by the δ_ℓ functions.



As opposites we get the $\delta'_{\ell'}$ functions.



(By convention we always draw the arrows in the δ_ℓ direction.)

The Sapphire compiler translates this code to non-recursive ANSI C, which traverses the compact graph of the stencils bottom-up. A series of tests were run sequentially on one processor of a SGI Cray Origin 2000 machine. At each timestep the heat distribution for all discrete x-values is computed. The running time of the generated C program clearly grows linearly with the number of timesteps.

No. of Grid Points	No. of Timesteps	Execution Time(sec)	Measured growth	Ideal growth
4000	1000	7.83	1	1
4000	2000	15.64	2.0	2
4000	4000	31.24	4.0	4
4000	8000	62.46	8.0	8

Tests of the MPI version on the SGI Cray Origin 2000 shows good parallel efficiency (1000 timesteps and 80 000 grid points).

No. of Processors	Execution Time(sec)	Measured speed-up	Ideal speed-up	Efficiency Measured/Ideal
2	298	1	1	100%
4	150	2.0	2	100%
8	75	4.0	4	100%
16	38	7.8	8	98%
32	20.5	14.5	16	91%

All tests were run with the machine in ordinary production mode.

4 Conclusion

Recursive programming is a very efficient problem solving technique, but with traditional compiler technology this is often prohibitively expensive to compute. We have suggested supplying the stencil and its opposite as an explicit part of the program. This makes it possible for a compiler to generate code that traces the compact data dependency graph in an efficient data-driven fashion. We can also augment the data dependency graph with information on the distribution of the computation on the processors of a parallel computer. This yields full control over the parallel execution of the code, without the need for the programmer to get involved in the parallel code itself.

Similar approaches have been investigated by other researchers, but in more restricted contexts. In [6] many approaches based on functional programming are reported. Some of these utilise the dependency in order to generate data driven code and parallelise programs, but they do not give the user explicit control over the opposite dependency. This limits the efficiency of the approaches to cases where the dependency may be automatically inverted. In an imperative context Čyras & al. [1] have used dependencies to provide a modular decomposition of the recursive function (with stencils) from code tracing out the graph (the opposite stencils). A detailed comparison with our technique is presented in [2]. See [4] for more detailed examples of the constructive recursion technique.

Acknowledgements Thanks to Steinar Søreide who implemented the Sapphire compiler and did the timings.

References

[1] Vytautas Čyras. Loop synthesis over data structures in program packages. *Computer Programming*, 7:27–50, 1983. (in Russian).

[2] Vytautas Čyras and Magne Haveraaen. Modular programming of recurrencies: a comparison of two approaches. *Informatica*, 6(4):397–444, 1995.

[3] Magne Haveraaen. How to create parallel programs without knowing it. In *Proceedings of the 4th Nordic Workshop on Program Correctness – NWPC4*, number 78 in Reports in Informatics, pages 165–176, P.O.Box 7800, N-5020 Bergen, Norway, April 1993.

[4] Magne Haveraaen and Steinar Søreide. Solving recursive problems in linear time using constructive recursion. In Sverre Storøy, Said Hadjerrouit, et al., editors, *Norsk Informatikk Konferanse – NIK'98*, pages 310–321. Tapir, Trondheim, Norway, 1998.

[5] W.L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32:93–114, 1983.

[6] B.K. Szymanski, editor. *Parallel Functional Languages and Compilers.* ACM Press; New York / Addison Wesley; Reading, Mass., 1991.