Appeared in S. Storøy, S. Hadjerrouit & al., eds.: *Norsk Informatikkonferanse NIK'98*, Tapir, Trondheim Norway 1998, pp. 310–321.

Solving recursive problems in linear time using Constructive Recursion*

Magne Haveraaen and Steinar Søreide
University of Bergen, Norway

Abstract

Recurrence relations are common in the mathematical formulation of many important problems. Being able to rapidly and efficiently prototype a recurrence will aid research and engineering in diverse areas such as partial differential equations and dynamic programming. Unfortunately, the conventional techniques available are either to program the recurrence as a recursive program with exponential growth in time complexity, or to unfold the recursion and hand code an efficient solution, with associated high development costs. Constructive recursion promises to deliver a space and time optimal compilation of recurrences in general. The scheme has also shown to provide a simple way to generate parallel code on high performance computing architectures.

1 Introduction

Computational modelling using partial differential equations (PDEs) is becoming an important tool for industry. Currently this is hampered by the time and effort needed to develop good computational models, and the time and cost needed to port such a model onto a high performance computer (HPC).

Recurrence relations form the core of a computational model of a PDE. For many problems, such as those in computational fluid dynamics, formulating an appropriate recurrence relation is a non-trivial task. Often it will require several iterations of formulating a recurrence, creating a prototype implementation and trying it on selected data sets, before a usable strategy is found. The test problems may require the use of an HPC to increase the turn-around time during the computational model development cycle. Being able to rapidly prototype reasonably efficient implementations of recurrence relations may significantly reduce the time and effort needed to find the appropriate recurrence. But programming a recurrence top-down as a recursively defined function in a normal stack-based programming language gives exponential running time, which is impractical for all purposes. Techniques for elimination of tail recursion etc. (see, e.g., [4]) are not powerful enough to improve on execution time complexity of a program, even though a marked improvement in actual execution times may occur.

More promising routes seem to come from the study of data dependencies. In Crystal, [1], data dependencies were extracted from recursive programs and inverted by the compiler during compilation. For the simpler dependency patterns the opposite graph was easily computed, and optimal execution time complexity was achieved, but in general finding the opposite of the dependency graph is NP-complete.

The constructive recursion (CR) approach was developed along the line of utilising the data dependency patterns, but by making them an explicit structure in the program text, [7], [6] and [3]. Then both the dependency pattern and its opposite are controlled by the programmer, rather than being extracted by the compiler. Traditionally a recursive

*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.

program implicitly defines an exponentially growing tree-structured graph. In the constructive recursive case the graph is explicitly defined by the user, allowing linear solution time for recurrences, even for higher order recurrences. The information in the graph allows the translation of the recursive program to recursion free loop programs, giving a bottom-up execution scheme for the compile program. By distributing the nodes of the graph explicitly on the nodes of a parallel machine, parallel programs may also be generated, as proposed in [9]. Together this allows for efficient use of many high performance computing (HPC) architectures.

Sapphire is a functional language designed to code CR programs. A Sapphire program consists of three parts: a definition of the data dependency in the form of a Data Dependency Algebra (DDA), a definition of the recurrence, and a configuration binding the two together.

We have developed a prototype Sapphire compiler generating ANSI C code. The translator has also been enhanced to generate parallel code on top of MPI. MPI hides important aspects of the communication structure of the target architecture, and we would like to explore other translation schemes that allow explicit use of this structure. Several PDE problems have been implemented using Sapphire, including heatflow and the flow of electric currents in cardiac fibres. We have also made a trial implementation of the stagecoach dynamic programming problem.

This presentation is organised as follows. Section two briefly presents the concept of recurrences. Then we show our programming approach through the development of a solver for the linear heatflow problem and provides sequential execution times for the generated code. In section four we present some butterfly algorithms. The fifth section discusses the parallel code generation and gives parallel performance figures for the heatflow example. Section six discusses the implications for dynamic programming by showing a solution to the stagecoach problem. We end with a brief summary and comparison with other work.

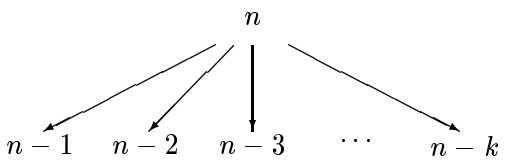
2 Recurrences and recursive programming

2.1 Recurrence relations

An order k, $k \geq 0$, linearly dependent recurrence f with natural numbers as *index domain* is given by the form

$$\begin{aligned} f_n &= \phi(f_{n-1}, f_{n-2}, \ldots, f_{n-k}), \qquad \text{for } n \geq k, \\ f_{k-1} &= \epsilon_{k-1}, \\ &\vdots \\ f_0 &= \epsilon_0, \end{aligned}$$

where ϕ is an expression not referring to f and the ϵ_i are constants. The dependency can be visualised by



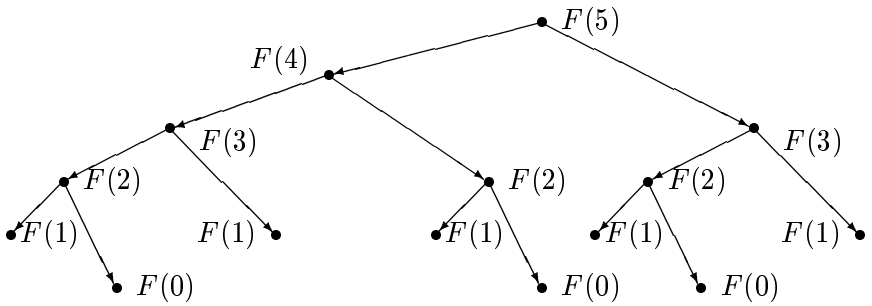
which shows how step n of the recurrence depends on steps $n-1, \ldots, n-k$. A standard example of an order 2, linearly dependent recurrence is the Fibonacci function, given by

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2}, \qquad \text{for } n \geq 2, \\ F_1 &= 1, \\ F_0 &= 1. \end{aligned}$$

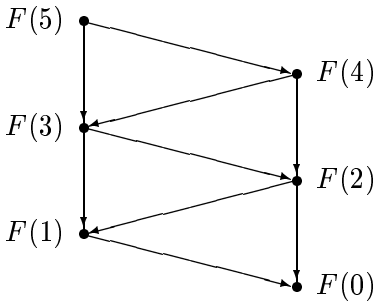
It is obvious that this can be expressed as a recursive program

```
F(n) = if n > 1 then F(n-1) + F(n-2) else 1
```

When compiled in a conventional language this program will give the following execution tree for F(5).



Note how the subtrees for F(3) are duplicated and F(2) are triplicated. This multiplicity of identical subtrees gives an exponential execution time, which is prohibitively expensive even for moderately sized arguments to this simple function. By merging subtrees, we get the much more compact execution graph



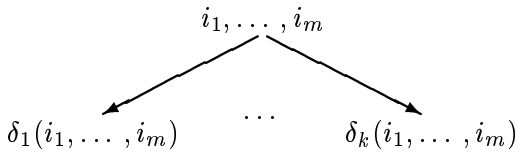
which only grows linearly in size with the argument to the function. This graph is the one we achieve by the normal hand coded iterative version of the Fibonacci function.

2.2 Generalised recurrence relations

The concept of a recurrence relation can be generalised to arbitrary discrete index domains. Given a collection of m, $0 \leq m$, index variables $i_1, \ldots, i_m$, each ranging over the natural numbers, the generalised recurrence has the form

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

which exhibits the following pattern



The dependency pattern, often called a stencil, is given by the k expressions $\delta_1(i_1, \ldots, i_m)$, $\ldots$, $\delta_k(i_1, \ldots, i_m)$ each returning a full index set of m index domain values. The recurrence f is fully defined from the stencil δ, the expression ϕ, and a suitable collection of initial values ϵ. Note that we have to put certain conditions of wellfoundedness on the dependency pattern and choice of initial values in order to make f well defined. Thus one might view the recurrence as composed from three parts: a structural part (the stencil δ), an expression part (ϕ with the recursive part of the recurrence definition), and an initialisation part ϵ.

The general form of the recurrence relation encompasses a large class of recursive equations. Some problems include

- Discretisations of partial differential equations, such as the FDM discretisation of the heatflow equation in one dimension.
- Butterfly algorithms, such as the Fast Fourier Transform.
- Dynamic programming problems, such as the Stagecoach problem.

Many of these programs are simple to parallelise using the information given by the constructive recursion programming scheme.

3 Example: Heatflow in one dimension

The one-dimensional heatflow equation is a simple example, well suited for a detailed explanation. It describes the heat distribution $u(x,t)$ in a one-dimensional rod along the x-axis at time t,

$$\alpha^2 \frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}.$$

Here α is a physical constant describing the heat transfusion properties of the rod. One may think of the rod as being perfectly insulated along its length, only allowing heat to dissipate from the ends. To turn the equation into a recurrence relation we may use the finite difference method both in time and space, achieving

$$\alpha^2 \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} = \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t}$$

Rearranging terms, multiplying by Δt and replacing t by $t - \Delta t$ we get

$$u(x,t) = \alpha^2\sigma u(x+\Delta x, t-\Delta t) + (1-2\alpha^2\sigma)u(x,t-\Delta t) + \alpha^2\sigma u(x-\Delta x, t-\Delta t)$$

where $\sigma = \Delta t/\Delta x^2$. Here $\sigma \leq c$, for some numerical constant c, relating the size of the time steps to the size of the space steps. If the quotient is larger than c we loose numerical stability of the method.

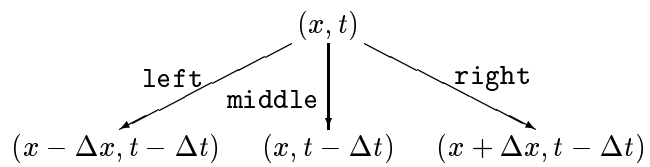
The Data Dependency Algebra (DDA) in Sapphire is defined by a sort, the nodes of the data dependency graph, and a set of operations that define the stencil. The sort defines information needed to describe the dependency. A suitable sort definition for the DDA of the heatflow equation is `DDA = time int * length int`. This gives us (implicitly) two projection functions

```
length :  DDA -> int
time :  DDA -> int
```

which pick out the first and second components of the pair, and one injection function

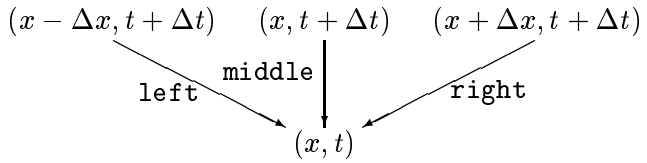
```
DDA : int,int -> DDA
```

which defines the data pair from two components. The stencil is given by the **r**(equest) function. It describes which nodes in the DDA-graph that a given node depends on. From equation (3) we deduce the request data dependency.



The directions are enumerated `left`, `middle` and `right`.

The s(upply) function describes the opposite of the **r** function. It tells us which nodes in the DDA-graph that depends on a given node and also follows from equation 3.



By convention we always draw the arrows in the requesting direction, even when we focus on supplying data. Both of these functions are required when defining a DDA.

We relate the discretised version to the continuous version by relating the number of grid points to the physical distances. The length X of the rod is represented by N_x grid points, thus $\Delta x = \frac{X}{N_x - 1}$. The time T of the simulation is divided into N_t timesteps, starting with the initial values at timestep 0 and ending at timestep N_t. Then $\Delta t = \frac{T}{N_t}$. The stability requirement now turns into a condition on the number of timesteps, $N_t \geq (N_x - 1)^2 \frac{T}{cX^2}$.

As the computation graph is to evolve over a discrete grid defined from pairs of integers, the **r** and **s** functions become

```
r(p,d) = if   d = left -> DDA(time(p)-1,length(p)-1)
         |  d = middle -> DDA(time(p)-1,length(p))
         |   d = right -> DDA(time(p)-1,length(p)+1)
         fi;
s(p,d) = if   d = left -> DDA(time(p)+1,length(p)-1)
         |  d = middle -> DDA(time(p)+1,length(p))
         |   d = right -> DDA(time(p)+1,length(p)+1)
         fi;
```

We may express the recurrence as a recursive top-down program with the explicit stencil in the form of the **r** function

$$u(p) = \alpha^2\sigma u(\mathtt{r}(p, \mathtt{left})) + (1 - 2\alpha^2\sigma)u(r(\mathtt{p}, \mathtt{middle})) + \alpha^2\sigma u(\mathtt{r}(p, \mathtt{right}))$$

To make the recurrence fully defined we introduce boundaries: a set of initial values at timestep 0, and one at each end of the rod. We will apply the same temperature at the boundaries throughout the computation.

The constructive recursive program follows. It takes into account the boundary conditions, a constant temperature given at the start of the computation, and assumes the initial temperature distribution of the rod has been set at the start of the computation.

```
u(p) = if left_boundary(p) = true or right_boundary(p) = true
       then u(r(p,middle))
       else as*u(r(p,left)) + (1-2*as)*u(r(p,middle)) + as*u(r(p,right))
```

The constant $\mathtt{as} = \alpha^2\sigma$.

Normally, a recursively defined function like `u` would cause execution time to grow exponentially with the number of timesteps. The Sapphire compiler, using the definition of the DDA, translates this code to non-recursive ANSI C, which executes the code bottom-up. A series of tests were run sequentially on one processor of a SGI Cray Origin 2000 machine. As Figure 1 shows, the running time of the generated C program is clearly linear.

Figure 2 shows test runs of the heatflow problem for a fixed number of time-steps, but with different numbers of grid-nodes. We see that the scaling is close to linear, as hoped for. The slight deviation as size grows is probably due to the use of non-local memory as the data size increases.

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

Figure 1: Test runs showing scalability in number of timesteps.

No. of Grid Points	No. of Timesteps	Execution Time(sec)	Measured growth	Ideal growth
1000	1000	2.0	1	1
2000	1000	3.9	2.0	2
4000	1000	7.8	3.9	4
8000	1000	15.6	7.8	8
16000	1000	31.3	15.7	16

Figure 2: Test runs showing scalability in problem size.

4 Example: Butterfly stencil for generalised recurrences

The heatflow example had a very simple, albeit common, data dependency pattern. A more complex pattern is the butterfly dependency pattern. In the butterfly the connections depend on which node and which size butterfly we are looking at. A height 4 butterfly is depicted in figure 3. A butterfly of height h will have $h+1$ rows (indexed

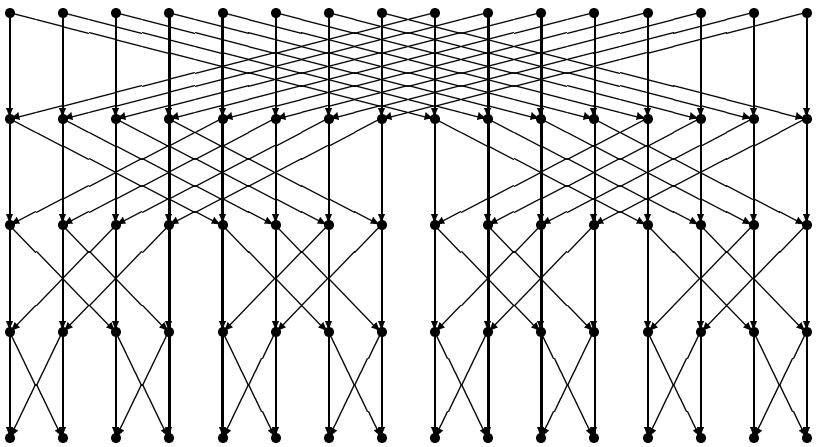


Figure 3: Butterfly graph (height 4).

0 through h) and 2^h columns (indexed 0 through $2^h - 1$). A node is identified by index pairs in the row-column order, thus for a height h butterfly we will use indices in the range $(0,0)$ through $(h, 2^h - 1)$. An inductive definition of the butterfly is given by the following:

- A height 0 butterfly has 1 node indexed by $(0,0)$.

- A height h, for $h > 0$, butterfly is composed horizontally from two height $h-1$ butterflies placed side by side, such that $2^{(h-1)}$ is added to the column numbers of the rightmost subbutterfly. Then a new top row with indices $(h,0)$ through $(h,2^h-1)$ is added. Each node (h,j) in the top row is connected by a vertical arc to the node $(h-1,j)$ (a top node in one of the butterflies of height $h-1$) and by a diagonal arc to the node $(h-1,\texttt{complementbit}(j,h-1))$. That is, each new node is connected to the node immediately beneath it (a topmost node in one of the subbutterflies) and diagonally to the corresponding top node of the other subbutterfly.

The function `complementbit` for integers $j,k \geq 0$ is defined by

$$\texttt{complementbit}(j,k) = \begin{cases} (j \operatorname{div} 2^{k+1})2^{k+1} + 2^k + (j \bmod 2^k) & \text{if } \texttt{bit}(j,k) = 0 \\ (j \operatorname{div} 2^{k+1})2^{k+1} + (j \bmod 2^k) & \text{if } \texttt{bit}(j,k) = 1 \end{cases}$$

where $\texttt{bit}(j,k) = (j \operatorname{div} 2^k) \bmod 2$.Thus the function call $\texttt{complementbit}(j,k)$ will yield the integer corresponding to j, but with bit k complemented. The function call $\texttt{bit}(j,k)$ extracts bit k from j.

In Sapphire we may define a height h butterfly with a DDA sort `BF = row int * col int`, giving us the correspondingly named projection and injection functions. Choosing direction `1` as keeping to the left and `2` as keeping to the right when moving in the `r` and `s` directions, these functions are defined by

```
r(p,d) = if d = 1 and bit(col(p),row(p)-1) = 0
              -> BF(row(p)-1,col(p))
         |  d = 1 and bit(col(p),row(p)-1) = 1
              -> BF(row(p)-1,complementbit(col(p),row(p)-1))
         |  d = 2 and bit(col(p),row(p)-1) = 0
              -> BF(row(p)-1,complementbit(col(p),row(p)-1))
         |  d = 2 and bit(col(p),row(p)-1) = 1
              -> BF(row(p)-1,col(p))
         fi;
s(p,d) = if d = 1 and bit(col(p),row(p)) = 0
              -> BF(row(p)+1,col(p))
         |  d = 1 and bit(col(p),row(p)) = 1
              -> BF(row(p)+1,complementbit(col(p),row(p)-1))
         |  d = 2 and bit(col(p),row(p)) = 0
              -> BF(row(p)+1,complementbit(col(p),row(p)-1))
         |  d = 2 and bit(col(p),row(p)) = 1
              -> BF(row(p)+1,col(p))
         fi;
```

One way to utilise the butterfly stencil is to efficiently compute and replicate the sum of 2^h numbers on 2^h nodes. Assume that each node in the 0'th row of the butterfly has been assigned some number. The replicated sum will add these numbers together, so that each node at the top will request a copy of the sum. The intermediate nodes will contain various subsums.

```
sum(p) = sum(r(p,1)) + sum(r(p,2))
```

The fast Fourier transform, the FFT, is a standard algorithm which utilises the butterfly structure fully. The inputs are placed on the bottom row, and the outputs appear on the top row.

```
fft(p) = fft(r(p,1)) + exp(a*row(p)/(2**col(p)))*fft(r(p,2))
```

where `a` $= 2\pi i$, i being the unit along the imaginary axis. In both these examples the sequential execution time is $O(n \log n)$, where $n = 2^h$ is the number of input nodes.

Figure 4: 16 processor linear array.

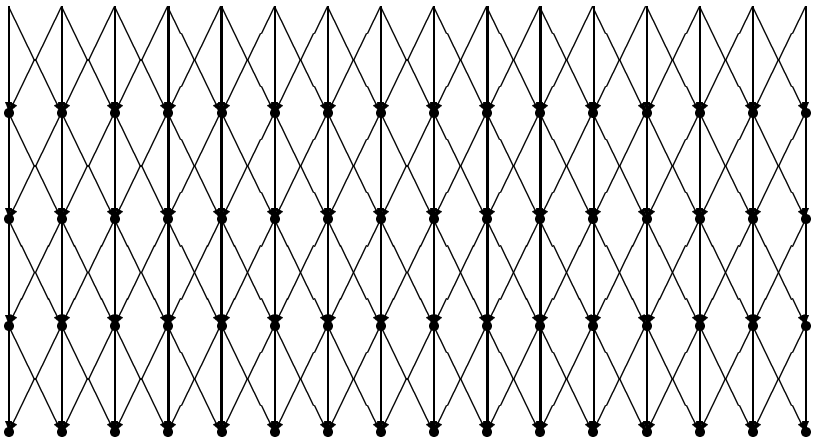


Figure 5: Space-time view of 16 processor linear array.

5 Parallelising constructive recursive programs

For Constructive Recursion programs, the data dependency (and its opposite) is known from the DDA. Following [9], the program's dependency pattern has to be embedded in the computer's communication structure to give a parallel program. The communication structure is representable as a Space-Time Algebra (STA). An STA is a special form of a DDA with space and time coordinates for the nodes. The embedding is expressible as program code, allowing the programmer full control. Since an embedding only depends on the DDA and the STA, it can be reused for many different programs with the same dependency pattern. See [6] and [8] for a more detailed discussion of parallelisation issues.

To illustrate the parallelisation technique we will look at a very simple parallel computer architecture and show a couple of embeddings.

A linear array is a computer where the processing elements are connected in a linear fashion, i.e., each node is connected to two other nodes, except the two end nodes which are connected to just one neighbour. Figure 4 shows a 16 node linear array connected computer.

The communications of a linear array in time appears schematically in Figure 5. Here all the processors of the linear array are depicted at every time-step, as if all computations were taking place instantaneously and synchronously. The communications now go between the timesteps, to the neighbours as defined by the static configuration of the linear array. Requesting arcs are diagonal lines backward in time. In addition, a processor may retain information between computation steps. This is depicted as a communication from a processor to itself (vertical arrows).

The one-dimensional heatflow computation from Section 3 may easily be embedded in the linear array's communication structure. The dependencies of this computation exhibit the same graph as the communications of the linear array, i.e., the communication structure in Figure 5.

The butterfly computations are more difficult to embedd in the linear array. We need to use more than one time-step for communications, and progressively more timesteps as the communications of the butterfly grow longer. This can be seen in Figure 6, which shows a height 4 butterfly embedded in a 16 node linear array using 16 timesteps. The scale is logarithmic in the time-direction, and the processors are drawn as open circles

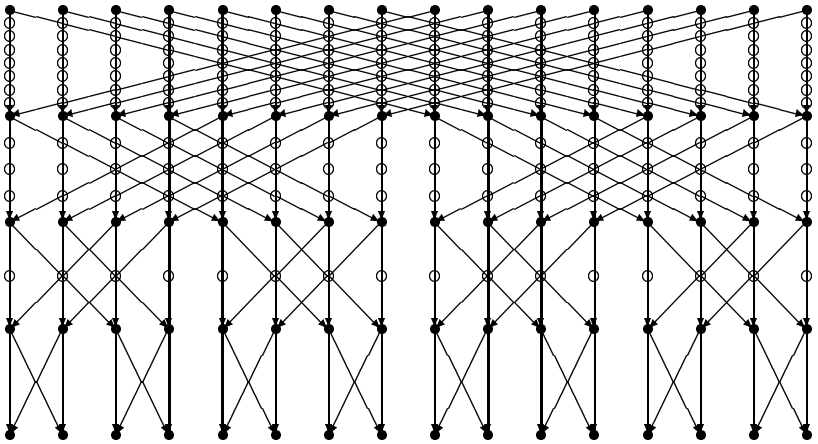


Figure 6: Butterfly embedded in space-time of linear array. Unused connections are not shown. Unused nodes are indicated with an open circle. The time-scale is logarithmic.

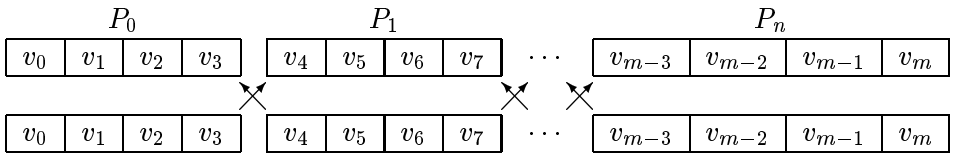


Figure 7: Inter-processor communication pattern of one-dimensional heatflow from time-step t to time-step $t+1$ with grouping of virtual nodes on processors.

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

Figure 8: Test runs of parallel code for 1000 timesteps and 80 000 grid nodes.

when they do not compute but just forward communications. This embedding utilises only a fraction of the communication capasity of the machine.

Parallelisation is to embed a computation in a parallel machine's space-time such that

- computations map to processors at specific time-steps,
- dependencies map to paths in the parallel machine's space-time communication structure.

The mapping is user definable and expressible as program code. This gives the programmer full control over the computation, but still avoids lots of detail in the writing of parallel programs.

This approach was originally intended for massively parallel computers, assigning one computation node to each STA-node, see [6]. But as today's computers tend to have a relatively low number of powerful processors, a slightly different approach has been implemented in the current Sapphire compiler: Virtual computation nodes are grouped together

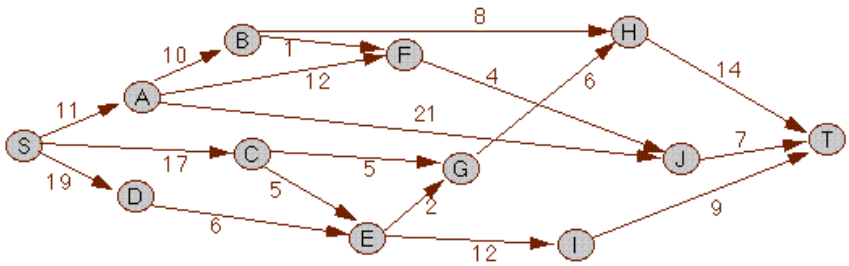


Figure 9: Schematic graph of stagecoach routes.

such that neighbours in the space-mapping given by the STA are grouped together. If the STA groups nearest neighbours of the domain together this coincides with domain splitting of partial differential equation. For the one-dimensional heatflow example, the resulting interprocessor communication pattern is shown in Figure 7.

One of Figure 8 shows test runs on an SGI Cray Origin 2000. The machine has a complex communication pattern, but the user is to think of the machine as fully connected, which trivially embeds any data dependency pattern. The tests are of the heatflow problem with constant number of timesteps, constant number of nodes and different numbers of processors. The execution times exhibit close to the optimal theoretical behaviour. The deviation when the number of processors increases is to be expected due to the relative increase in communication costs as the amount of computation on each processor decreases.

6 Dynamic programming: the stagecoach problem

The basic technique in the translation of CR programs to efficient code is quite similar to the technique of dynamic programming (see, e.g., [10]). The stagecoach problem is a classical dynamic programming problem. The background for this problem is said to be a salesman in the mid-western states of the USA. He was traveling from a small town S to another small town T, but did not want to pay more for the transport than necessary. The means of transportation was stagecoaches scheduled between the small towns in the area. The cost of travel was the sum of the cost of the individual journeys. A schematic map of the possible routes is given by the graph in Figure 9, together with the costs of the individual stretches. The problem is to find the total cost of the cheapest path from S to T in the graph.

The solution may be formulated as a recurrence. First we define the graph with the function `cost`(p, n) giving the cost of travelling from town p to town n. Now the function `cheapest`(p) is to give the total, cheapest cost of going from p to the node T, the end node for the trip. This is found by looking at all arcs from the node p to nodes n, and then adding `cost`(p, n) to the cheapest cost of going from n to T. The lowest cost from p is then the minimum of all such costs.

`cheapest(`p`) = min{(cost(`p`,`n`) + cheapest(`n`)|` there is an arc from p to n`}`,

`cheapest(`T`) = 0`.

In dynamic programming, this recurrence is manually analysed in order to create a program that computes the cheapest path in polynomial time, rather than the exponential time given by a straight forward execution of the recursive code. Taking the graph as the DDA, the CR translation technique will give a polynomial time program directly, eliminating the need to manually develop a non-recursive version. The program will input the graph in the form of arcs and associated costs, and the start and end nodes. Then it will efficiently solve any stagecoach type problem.

7 Summary and future work

The natural way of expressing a recurrence relation in a programming language such as C++ is as recursive function calls mimicking the recursive equations from the mathematical formulation. Unfortunately, this often results in a program growing exponentially both in storage and time usage. A non-recursive implementation may need only a fixed amount of storage and linear time (as in the case of the Fibonacci relation), or linear growth of storage and $O(n \log n)$ growth of time as in the FFT case. Thus naively writing a recurrence as a recursive function is not a suitable path for prototyping.

The other alternative, hand coding a non-recursive version, gives efficient code, but is a difficult, hence costly, task if the stencil is complex. Thus this is not a viable approach for prototyping recurrences.

Our approach is to use the recursive formulation of the recurrence, but augment the definition of the stencil **r**, with the definition of the opposite of the stencil, **s**. This allows us to compile the recursively structured recurrence function into non-recursive code, giving the same time and space complexity as one would expect from a reasonably efficient implementation. One can also augment **r** and **s** with information on the distribution of the computation on the processors of a parallel computer. This will also define the communication pattern of the program, thus yielding full control over the parallel execution of the code.

Comparing the constructive recursive approach to the Crystal approach (Chen & al [1]), we find that a Crystal function $f : D \to t$ is expressed in terms of a specific model D for the dda. Instead of abstracting over this model as we do, they use the fact that there is an isomorphism $g : D \to E$ from the model D to the embedding E of D in the target machine's space-time algebra ST. The isomorphism is called a *reshape morphism*, since it reshapes the domain D into E. The reshape morphism is used to reformulate the function f into the function $\hat{f} : \mathsf{ST} \to t$, defined by the equation $\hat{f} = f \circ g^{-1}$, where g^{-1} is the inverse function of g. The reshape morphisms g and g^{-1} may be programmer-supplied, but in many cases it is possible to automatically derive them. This is similar to the possibility of automatic generation of space-time embeddings of dda's in the Sapphire approach.

Similar results have been achieved by Čyras & al. (see [2], [5]). They restrict their attention to dda's that are regular grid structures, and do not define the s-functions, only the r-functions, in their code. The s-function is always derivable due to the simple structure of the base domains. A detailed comparison is presented in [3]. Trishina (see [14]) generates Occam programs from reccurrence equations over discrete lattice structures by similar automatic transformations.

Thompson & Tucker ([12], [13]) have developed a general theory on the recursion over structures. Their parallel work, however, is oriented towards structures with space-time properties. This will restrict their work to the subclass of space-time algebras of our approach.

To improve the usability of the CR approach to prototyping recurrences, future work includes developing a library of DDA's representing different standard stencils. For any given stencil, the library should include both sequential and parallel versions. This allows a prototype implementation of a recurrence to be run efficiently on several HPC architectures without requiring the user to rewrite the recurrence program.

The motivation for undertaking this work has been that recurrence relations occur in many formulations of common problems: partial differential equations, Fast Fourier transform, dynamic programming, etc. A prototyping tool yielding fairly efficient code would potentially be very useful, especially when it also allows efficient generation of code for parallel high performance computers.

References

[1] M. Chen, Y. Choo and J. Li: Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991, pp. 255–308.

[2] Čyras, V. (1983). Loop synthesis over data structures in program packages. *Computer Programming*, Vol. 7, 27-50. Vilnius: Institute of Mathematics and Cybernetics (in Russian).

[3] V. Čyras & M. Haveraaen: Modular programming of recurrencies: a Comparison of Two Approaches. *Informatica*, 1995, Vol. 6 no. 4, pp. 397–444.

[4] A.J. Field and P.G. Harrison: *Functional Programming.* Addison Wesley, Wokingham UK, 1988. ISBN 0-201-19249-7.

[5] S.N.Greshnev, E.Z.Lyubimskii, V.A.Chiras. Synthesis of programs on data structures. *Programming and Computer Software*, Vol. 11, No. 5, 1985, pp. 282–291. (translated from Programmirovanie, No. 5, pp. 44–54, September–October, 1985).

[6] Magne Haveraaen: *Data Dependencies and Space Time Algebras in Parallel Programming.* University of Bergen, Reports Informatics no. 45, Department of Informatics, UiB, 1990, revised 1997, pp. 26.

[7] M. Haveraaen: How to Create Parallel Programs without Knowing it. In Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, University of Bergen, Norway, March 1993, pp. 165–176.

[8] Magne Haveraaen: An algebra of data dependencies and embeddings for parallel programming. To appear in *Formal Aspects of Computing*, 1999.

[9] W.L. Miranker & A. Winkler: Spacetime Representations of Computational Structures. *Computing* vol. 32, 1983, pp. 93–114.

[10] Smith, D.K. (1991). *Dynamic Programming: A Practical Introduction.* Ellis Horwood.

[11] B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991.

[12] B.C. Thompson and J.V. Tucker (1985): Theoretical considerations in algorithm design. In R. Earnshaw (ed.): *Fundamental algorithms for computer graphics*, Springer-Verlag, Berlin, 855–878.

[13] B.C. Thompson and J.V. Tucker (1994): *Equational specification of synchronous concurrent algorithms and architectures (second edition).* Technical Report CSR 15-94, Department of Computer Science, University of Wales Swansea.

[14] Elena Trishina: Development of Occam Programs by Stepwise Algebraic Transformations of Affine Recurrence Equations. In Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, University of Bergen, Norway, March 1993, pp. 219–228.