Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

'Paradigm-oriented' design of parallel iterative programs using functional languages

F.A. Rabhi, J. Schwarz

Department of Computer Science, University of Hull, Hull HU6 7RX, UK

ABSTRACT

Functional languages offer a high degree of abstraction to the programmer while containing a great deal of implicit parallelism. This parallelism could be efficiently exploited if *parallel algorithmic structures* were used in the design of algorithms. A structure captures the behaviour of a *parallel programming paradigm* and acts as a template in the design of an algorithm. This paper addresses the issue of defining a structure for static iterative transformation (SIT) algorithms which are coarse-grained data parallel algorithms with an iterative control structure. The parameters required by the structure are supplied by the programmer using a functional language, forming the problem specification. This specification can then be successively turned into a sequential functional program, then into a parallel program for a graph reduction machine, and finally into a program that maps on a specific parallel architecture.

INTRODUCTION

Two major problems are hampering the acceptance of functional languages as a means of programming parallel systems. First is the difficulty in writing some programs with sufficient inherent parallelism. Second is the unpredictable performance of programs due to problems of load-balancing, grain size, and locality of references.

One potential solution to both these problems involves the use of constructs known as *parallel algorithmic structures* (or skeletons). Such structures capture the behaviour of an entire class of parallel algorithms or a *paradigm*. Any algorithm that obeys a known paradigm can then be specified by using its corresponding parallel algorithmic structure as a template, leaving the lower level details of exploiting parallelism to the implementation. Therefore, the process of designing a program is entirely "paradigm-oriented": a user

enters the problem specification by defining the parameters required by the corresponding structure. This approach benefits the programmer in providing convenient high-level parallel concepts, while expressing the parallelism in a non-architectural way provides the opportunity for efficient implementation across parallel architectures. The paradigm-oriented approach is explained in detail in [17].

This paper addresses the issue of defining a parallel algorithmic structure for a specific class of parallel algorithms, called *static iterative transformation* (or SIT) algorithms. We show what the user needs to define in the specification, and how this specification can be successively converted into a sequential functional program, then into a parallel program for a graph reduction machine, and finally into a program that maps onto a specific parallel architecture.

ITERATIVE TRANSFORMATION ALGORITHMS

These algorithms operate on a set of homogeneous data objects. These objects are transformed through several *iteration* steps. During an iteration step, each object performs a computation using local data or data received from other objects. There is also global shared information which is updated at each step by *combining* local data from the objects (ie "reduce" operations). The sharing of global information is made available usually via some form of "broadcast" operation.

We only intend to consider the *static* case where the number of objects does not change at run-time. An enhancement of this are genetic algorithms which have iterations which can produce new objects. This class of algorithms is of great importance in scientific and engineering applications such as image processing, numerical analysis and finite element methods. They can be considered as data-parallel algorithms [7] with an iterative control structure and a coarser grain than simple vector processing. Most of these algorithms correspond to *domain partition* algorithms [12] or are simply referred to as *geometric parallelism*.

THE USER'S SPECIFICATION

To design a SIT algorithm, the user has to specify a description of the local state, and how a state changes from one iteration to another. The user also identifies the global variables and how their value changes after each iteration step. The rest of this section shows how to enter these various parameters using a functional language. In the examples, functions are defined using the Haskell[9] syntax. Some knowledge of the language is assumed.

Transformations

As the number of objects does not change, we assume that the size of the set is defined as a special constant *wsetsize* whose value can be accessed anywhere

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

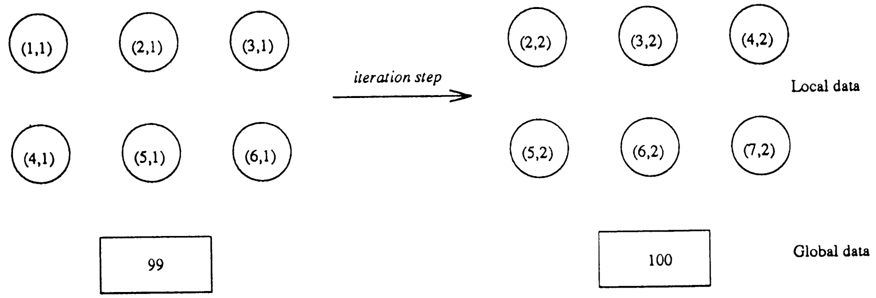

Figure 1: A totally distributed SIT algorithm

in the specification. Each object carries p items of information in a tuple, and is uniquely identified in the set through a global coordinates system. Global information that is accessible by all the objects is also represented as a tuple. A transformation occurs during an iteration cycle. To make the process of defining a transformation easy, local state and global variables are paired into a tuple $((s_1, s_2, \ldots, s_p), (g_1, g_2, \ldots, g_q))$ and transformations can be defined as:

$$transf((s_1, s_2, \ldots, s_p), (g_1, g_2, \ldots, g_q)) = ((s_1', s_2', \ldots, s_p'), (g_1', g_2', \ldots, g_q'))$$

The left component of the result $(s_1', s_2', \ldots, s_p')$ represents the new state (in each object) and the right hand side $(g_1', g_2', \ldots, g_q')$ represents the new global data. *All communication is implicit through variable name references.* For example, a global variable g_i referred to in the right hand side corresponds to a local access, while if it appears on the left hand side it corresponds to a broadcast operation. An example of a transformation is shown in figure 1.

The left component may also contain references to the self index *coord* in case objects need to know their position in the set. Local neighbourhood communication is achieved through references to *external expression lists.* An external expression list is of the form *exp@dest* where *exp* is an arbitrary expression and *dest* is a list of neighbour coordinates. It means that the expression *exp* is computed by each of the objects whose coordinates are in *dest* and the cumulated results are returned as a list. An expression computed remotely may refer to state variables before or after they are modified in the iteration cycle. An example is illustrated in figure 2.

Initial conditions and termination

The initial state $(s_{1_i}, s_{2_i}, \ldots, s_{p_i})$ for each object and the initial global values

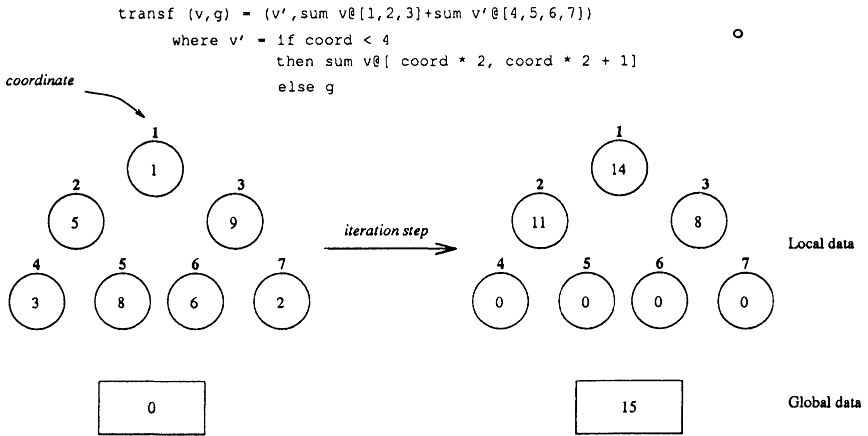

Figure 2: A SIT algorithm with neighbourhood communication

$(g_{1_i}, g_{2_i}, \ldots, g_{q_i})$ are defined using a parameterless transformation *init*.

$$init = ((s_{1_i}, s_{2_i}, \ldots, s_{p_i}), (g_{1_i}, g_{2_i}, \ldots, g_{q_i}))$$

The initial state values may contain references to the self-index *coord* but should not contain external expression lists. The final component in the specification is the termination condition *terminate*. Termination is decided based upon the value of the global data only.

Problem specification

Given a transformation *transf* to be executed at each iteration cycle, the initial values and a termination condition, the entire SIT problem can be expressed using a special function *sit$*:

$$problem = sit\$\ terminate\ transf\ init$$

There is a program transformation called $\mathcal{S}$ that can be applied to the specification producing a sequential functional program which is the *executable specification*. For details about this transformation and the definition of the function `sit$`, see [17].

Examples

The following examples show how to define specifications for various SIT algorithms.

Iterative methods: Iterative methods work by continuously refining a solution to a problem until an acceptable one is reached. A well known example of iterative methods is when solving a set of equations $Ax = b$ where A is an $n \times n$ matrix, b a vector of size n and x is the vector of size n to be determined. These methods only converge when some conditions apply but this is not discussed in this paper for the sake of simplicity.

Two methods are considered here: the Jacobi relaxation and the Gauss-Seidel relaxation. For each of these methods, the corresponding specification will be given. In the Jacobi relaxation, each point x_i in the vector x is refined according to the following equation:

$$x_i(t+1) = \frac{-1}{a_{ii}} \left(\sum_{j \neq i} a_{ij} x_j(t) - b_i \right)$$

We assume that the algorithm stops when the difference between the old value and the new one at every point is less than some threshold. The objects are arranged into a chain of size *wsetsize*. Each object is uniquely identified by its position i in the chain where $1 \leq i \leq wsetsize$. In the calculation, an object i only requires the row i of the matrix A. Therefore, the state of an object consists of the row A_i, the constant b_i and the variable x_i. We choose to keep the maximum difference between two successive values in all the grid as a global variable. To write the specification, we need a function `sumprod` that computes the sum of the product between a row in the matrix (represented as a Haskell array `r`) and a list of values `xs`:

```
sumprod j i r []                          = 0
sumprod j i r (x:xs) | (j==i)       = sumprod (j+1) i r (x:xs)
                     | otherwise    = y*r!j + sumprod (j+1) i r ys
```

Each object needs to communicate with all the other objects so a function `others` is defined. The function `all` allows the access to `abs(x-x')` from all the objects. The transformation can now be written as:

```
others x = [ y | y <- [1..wsetsize] , y /= x]

all = [1..wsetsize]

jacobi ((row,bi,x), g) = ((row,bi,x'), g')
   where x' = - ((sumprod 1 coord row x@(others coord))-bi)
                /row!coord
         g' = maximum (abs (x-x'))@all
```

```
terminate g = (g < threshold)

init = ( ...initial values at point coord..., threshold+1)

problem = it$ terminate jacobi init
```

where the function maximum returns the maximum element in a list.

The second method is the Gauss-Seidel relaxation which computes the new value $x_i(t+1)$ using the new values $x_j(t+1)$ for $j < i$ and the old values $x_j(t)$ for $j > i$ as follows:

$$x_i(t+1) = \frac{-1}{a_{ii}} \left(\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) - b_i \right)$$

Now, the communication pattern is different as the values imported from the right are different from the values imported from the left. Therefore, two new functions left and right are introduced. The Gauss-Seidel transformation can now be written as:

```
left x = [ y | y <- [1..x-1] ]

right x = [ y | y <- [x+1..wsetsize] ]

gaussseidel ((column,bi,x), g) = ((column,bi,x'), g')
   where x' = - (1/column!coord)
              *( (sumprod 1          coord column x'@(left coord ))
                +(sumprod (coord+1) coord column  x@(right coord))
                 -bi)

         g' = maximum (abs (x-x'))@all

problem = it$ terminate gaussseidel init
```

Solving Laplace's equation on a square: The iterative methods described can be extended to equations with more than one dimension. These equations generally arise from from a physical problem in two or three dimensions which can be described using a set of partial differential equations. In this example, the Jacobi iterative method is used to solve Laplace's equation on a square, whose solution is given by the following equation:

$$x_{i,j}(t+1) = \frac{x_{i-1,j}(t) + x_{i,j-1}(t) + x_{i,j+1}(t) + x_{i+1,j}(t)}{4}$$

Given a matrix of discrete values, a new value is computed after each iteration. This value is the average of the 4-neighbours' values. As before, the algorithm stops when the difference between the old value and the new one at every point is less than some tolerance.

The objects are arranged into an $n \times n$ grid, n being considered as the size of the set. Each object is uniquely identified by its row-column coordinates (i, j) where $1 \leq i, j \leq n$. The state of each object consists of one variable only, which is the value of the corresponding point. The maximum difference between two successive values in all the grid is kept as a global variable.

Each object that is not on the border needs values from its North, East, West, and South neighbours, so a function `news` as well as a predicate function `border` are defined. Communication is also needed to collect all differences between two successive values in order to compute the global value. Therefore a function `interior` which returns the coordinates of all the objects (except those on the border) is defined. Next, the transformation `laplacetransf` that should be applied between two iteration steps is defined. Finally, the last components to be defined are the initial state and global values, and the termination condition.

```
news (i,j) = [ (i,j-1),(i+1,j),(i-1,j),(i,j+1) ]

border (i,j) = or [ (i==1),(j==1),(i==wsetsize),(j==wsetsize)]

interior = [ (i,j) | i <- [2..wsetsize-1] , j <- [2..wsetsize-1] ]

laplacetransf (v , g) = ( v'  , g' )
                 where
                    v' = if not (border coord)
                         then (sum (+) v@(news coord)) / 4
                         else v
                    g' = maximum abs(v-v')@interior

tolerance = ... constant ...

init = ( ... to be defined for object coord ... , (tolerance + 1))

terminate g = (g < tolerance)

laplace = sit$ terminate laplacetransf init
```

CONSTRUCTING THE PARALLEL PROGRAM

Static iterative transformation algorithms show a great deal of locality so they are particularly suitable for distributed memory architectures. We first assume that the architecture has as many processor/memory pairs as there are

objects in the problem, and that the physical interconnection network matches exactly the logical communication pattern. We also assume that there is a processor called the *host* that synchronises the iteration steps in the algorithm and handles the global data.

Parallel evaluation model

The abstract parallel evaluation model used in the parallel program is based on *graph reduction*[1, 15], which is a simple demand driven model of computation suitable for functional languages. In graph reduction, a program is represented by a graph of expressions and the execution of this program consists of reducing the corresponding graph until the normal form, i.e. the result, is reached. This process may be carried out in parallel since any subgraph can be reduced independently from the others by a parallel task. A task is to executed by a *reduction agent*, which corresponds to a processor/memory pair. Examples of experimental parallel graph reduction machines for distributed memory architectures include the HDG-machine[11] and PAM[13].

Synchronisation is achieved entirely through the graph. Any node being evaluated by a task is *blocked* until it is overwritten by the result. Any task attempting to read the value of a blocked node is *suspended* and reawakened only when the evaluation of the node is completed. In general, a task creates parallel subtasks so the run-time system is in charge of dynamically allocating tasks to reduction agents. A reduction agent has an *active tasks queue* from which it selects the next task to be executed. There is a *suspended tasks pool* which contains the tasks that cannot proceed until some value becomes available. In our case, no tasks are dynamically generated so each reduction agent has a fixed number of tasks between its active tasks queue and the suspended tasks pool.

Parallel and communication routines

For our purpose, we assume that the following special functions are available:

- *parlet v=e in e'*: a local task that forces the evaluation of the expression e is sparked. Its value is later bound to the variable v. The parent task continues evaluating the expression e'.
- *request(d, e)* : requests from destination d the value of the expression e, whose the value is returned as the result of the function call.
- *update(d, e, i)*: sends an update message to destination d containing the value e for the appropriate slot in the input array i (see below). This function is always executed for its effect rather than its result which is generally assigned to a dummy variable.
- *broadcast(e, i)*: broadcasts the value of e to all objects, overwriting the variable i. If a list *all* yields all possible coordinates in the system, $broadcast(e, i)$ is equivalent to $map(\lambda d \rightarrow update(d, e, i))all$.

- *wait(dest)* : takes a list of destinations, allocates an *input array* where every slot corresponds to a destination. An input array is marked as *blocked* with a counter equal to its size. On receiving an update message for that particular array, the slot corresponding to the source of the update packet is updated with the expression contained in the update packet and the counter is decremented. When the counter reaches 0, the array is converted into a list and its status becomes *evaluated* so that any task waiting for its value is reactivated.

The $\mathcal{P}$ transformation

The aim is to produce a parallel program which assumes that each reduction agent is in charge of an object. Therefore, each reduction agent can be identified with the same coordinates as the object it holds and communication between objects occurs between the corresponding reduction agents. The $\mathcal{P}$ transformation transforms each transformation *transf* into a version $transf_{\mathcal{P}}$ with the local transformation running on every reduction agent and the global transformation running on the host. In this paper, we will describe two versions of the $\mathcal{P}$ transformation: the *request-update* transformation $\mathcal{P}_1$ and the *update-only* transformation $\mathcal{P}_2$. These are now described in turn.

The *request-update* transformation $\mathcal{P}_1$: This first transformation removes references to external variables lists and replaces them by a variable name. An internal task is created (using a *parlet* statement) which allocates an input array and sends request messages to the list of destinations specified. When the other end receives a request message, it sends an update message containing the value of the expression requested and the address of the input array. If the main task tries to access the value of the variable before all its values have arrived, it is suspended. When all the slots of the input array have been updated, the array is converted into a list and the suspended task is reactivated.

$$
\begin{aligned}
transf(s,g) &= (s',g') \\
&\quad where \\
&\qquad s' = \ \ldots e_1 @ dest \ldots \\
&\qquad g' = \ e_2
\end{aligned}
$$

$\Rightarrow_{\mathcal{P}_1}$

$$
\begin{aligned}
ltransf_p\ s &= s' \\
&\quad where \\
&\qquad s' = \ \ldots \\
&\qquad\qquad parlet\ v_i = map(\lambda x \rightarrow request(x, e_1))\ dest \\
&\qquad\qquad in \ldots v_i \ldots \\
&\qquad\qquad \ldots \\
gtransf_p\ g &= g' \\
&\quad where \\
&\qquad g' = \ e_2
\end{aligned}
$$

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

where v_i is a unique variable name. The same transformation is applied for external expression lists contained in the host's global cycle function:

$$\begin{array}{lll} transf(s,g) & = & (s',g') \\ & & where \\ & & \quad s' = \; e_1 \\ & & \quad g' = \; \ldots e_2@dest \ldots \end{array}$$

$$\Rightarrow_{\mathcal{P}_1}$$

$$\begin{array}{lll} ltransf_p\ s & = & s' \\ & & where \\ & & \quad s' = \; e_1 \\ gtransf_p\ g & = & g' \\ & & where \\ & & \quad g' = \; \ldots \\ & & \qquad parlet\ v_i = map\ (\lambda x \rightarrow request(x,e_2))\ dest \\ & & \qquad in \ldots v_i \ldots \\ & & \qquad \ldots \end{array}$$

To avoid objects requesting global variables, these variables are broadcasted by the host in advance. Therefore, a *parlet* declaration containing a *broadcast* operation is added into the global cycle.

$$\begin{array}{lll} transf(s,(\ldots g_i \ldots)) & = & (s',g') \\ & & where \\ & & \quad s' = \; \ldots g_i \ldots \\ & & \quad g' = \; e_2 \end{array}$$

$$\Rightarrow_{\mathcal{P}_1}$$

$$\begin{array}{lll} ltransf_p\ s & = & s' \\ & & where \\ & & \quad s' = \; parlet\ v_i \; = wait([host]) \\ & & \qquad in \ldots (head\ v_i) \ldots \\ gtransf_p\ (\ldots g_i \ldots) & = & g' \\ & & where \\ & & \quad g' = \; parlet\ d = broadcast(g_i, v_i) \\ & & \qquad in\ e_2 \end{array}$$

where v_i is also a unique variable name and d a dummy variable. The variable g_i should not have been broadcasted before in another *parlet* statement. The variable v_i is an input array with one value only which is marked as *blocked* by the *wait* procedure. The broadcast generated by the host will update the input array's slot that corresponds to the host.

There are three problems with this transformation: Firstly, the host may broadcast some values which are not needed by all the objects. Secondly, during a local communication, a request message is sent just before the value

is needed causing a task suspension which involves expensive context switching and queue maintenance overheads. Often, it is possible to request the values well in advance (e.g. at the start of the cycle). Thirdly, as all reduction agents are executing the same task, it is often possible to predict the local data requirements of an object. This means that request messages between neighbours can be eliminated by generating update messages in advance. The next program transformation addresses the last two problems.

The *update-only* transformation $\mathcal{P}_2$: Often, if an object is expected to request the value of an expression e from a set of input neighbours $dest$, then this processor is likely to send the value of the expression e (computed locally) to a set of output neighbours $dest'$. Therefore, update messages instead of request messages can be generated prior to starting the iteration cycle. This transformation is not always applicable because the set of output neighbours $dest'$ cannot always be statically determined, mostly due to boundary conditions. In addition, all the objects do not have the same the same data requirements so a predicate $update_p$ is needed to distinguish between the objects that participate in the update and those which do not need to send update messages at all,

Therefore, the *update-only* transformation $\mathcal{P}_2$ can only be applied to an external variable list if the corresponding inverse list $dest'$ and the predicate $update_p$ have been determined. Automatically determining these parameters is still being investigated. We will only show how the transformation uses them to generate update messages. If the external expression list is present in the local cycle, the $\mathcal{P}_2$ program transformation works as follows:

$$
\begin{array}{l}
transf(s,g) = (s',g') \\
\quad where \\
\qquad s' = \;\; \ldots e_1@dest \ldots \\
\qquad g' = \;\; e_2 \\
\Rightarrow_{\mathcal{P}_2} \\
ltransf_p \; s = s' \\
\quad where \\
\qquad s' = \;\; parlet \; d \;\; = if \; (update_p \; coord) \\
\qquad\qquad\qquad\qquad\quad then \; map \;\; (\lambda x \rightarrow update(x, e_1, v_i)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad dest' \\
\qquad\qquad\qquad\qquad\quad else \; 0 \\
\qquad\qquad\quad parlet \; v_i \;\; = wait(dest) \\
\qquad\qquad\quad in \\
\qquad\qquad\quad \ldots v_i \ldots \\
gtransf_p \; g = g' \\
\quad where \\
\qquad g' = e_2
\end{array}
$$

where v_i is a unique variable name and d a dummy variable. If the exter-

nal expression list occurs in the global cycle, only its corresponding predicate $update_p$ is needed by the $\mathcal{P}_2$ transformation.

$$
\begin{array}{lll}
transf(s,g) & = & (s',g') \\
 & & where \\
 & & \quad s' = e_1 \\
 & & \quad g' = \ldots e_2@dest \ldots
\end{array}
$$

$\Rightarrow_{\mathcal{P}_2}$

$$
\begin{array}{lll}
ltransf_p\ s & = & s' \\
 & & where \\
 & & \quad s' = \ parlet\ d\ = \ if\ (update_p\ coord) \\
 & & \qquad\qquad\qquad\qquad then\ update(host, e_2, v_i) \\
 & & \qquad\qquad\qquad\qquad else\ 0 \\
 & & \qquad\quad in\ e_1 \\
gtransf_p\ g & = & g' \\
 & & where \\
 & & \quad g' = \ parlet\ v_i\ = wait(dest) \\
 & & \qquad\quad in\ \ldots v_i \ldots
\end{array}
$$

Example: Deriving the parallel version of Laplace's problem

Considering Laplace's problem, there are two external lists in the transformation `laplacetransf`. To the first list `v@(news coord)` corresponds a predicate $update_p$ which is always *true* as all the objects send update messages. The difficulty is to determine $dest'$ which is the inverse of the list of destinations `(news coord)` in the presence of boundary conditions. By hand, we can determine `dest'=(news' coord)` where:

```
news' (i,j) = (if (i/=1)        then [((i-1),j)] else [])++
              (if (i/=wsetsize) then [((i+1),j)] else [])++
              (if (j/=1)        then [(i,(j-1))] else [])++
              (if (j/=wsetsize) then [(i,(j+1))] else [])
```

As the second external expression list `abs(v-v')@interior` occurs in the host's cycle, all is needed is the predicate $update_p$. Following the argument that only objects not on the border need to send `abs(v-v')` to the host, this predicate is equal to `coord 'in' interior` where the function `in` returns `true` only if `coord` is in the list `interior`. The transformation `laplacetransfp` can now be expressed with *update-only* messages (transformation $\mathcal{P}_2$):

```
llaplacetransfp v = v'
   where
       v' = parlet d1 = if true
                        then map (\x -> update(x,v,y))
```

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517


```
                                    (news' coord)
                            else 0
                       d2 = if (coord 'in' interior)
                            then update(host,abs(v-v'),z)
                            else 0
                       y = wait(news coord)
            in
               if not(border coord)
               then (foldr1 (+) y) / 4
               else v

glaplacetransfp g = g'
   where
       parlet z = wait(interior)
       in
        g' = maximum z
```

THE RUN-TIME SYSTEM

Assuming that the $\mathcal{P}$ transformation has derived the local and global transformations, the next stage consists of organising the run-time system based around an underlying parallel graph reduction machine. The run-time system comprises a set of reduction agents that implement the local transformations and a host that implements the global transformation.

During one iteration step, each reduction agent executes a main task which corresponds to the local cycle (left hand side of the transformation) and a *communication task* for every *parlet* statement. Here is a brief description of how a reduction agent operates:

step 1 : initialise the state
step 2 : execute the local cycle $ltransf_p$. During execution, spark a local task for every *parlet* statement.
step 3 : wait for a signal from the host, then stop or repeat step 2 depending on the value of the signal.

As in the reduction agents, the host runs a main task which corresponds to the global cycle and a communication task for every *parlet* statement. During execution, the main steps executed are:

step 1 : initialise the global variables
step 2 : execute the global cycle $gtransf_p$. During execution, spark a local task for every *parlet* statement.
step 3 : when all reduction agents have finished executing their cycle, evaluate the expression (*terminate g*). If the result is *false*, broadcast a signal to end the computation. Otherwise, broadcast a signal to carry on and repeat step2.

We omit to describe how the results are collected. For example, when receiving the signal to end, all the reduction agents send their final state (or part of it) to the host.

MAPPING THE PROBLEM ON A PHYSICAL ARCHITECTURE

The $\mathcal{P}$ transformation assumed an architecture with an infinite number of processors. The next step in the transformation system is to adapt the parallel program to a specific physical architecture. We assume that q objects of coordinates $coord_1 \ldots coord_q$, with states $S_1 \ldots S_q$, are mapped on a reduction agent of coordinates *pcoord*, and that there is a function *cmap* which converts logical coordinates into physical coordinates. These assumptions are made by a *mapping module* in the system.

The program produced by the $\mathcal{M}$ transformation assumes that each reduction agent will support q objects and their communication tasks. Any destination address d for an *update* or a *request* message is converted into a physical address (*cmap d*). Communication between virtual processes located on the same processor will just consist of a memory transfer between two variables.

PROJECT STATUS

We are currently implementing a Paradigm-Oriented Parallel programming Environment (POPE) for SIT algorithms. This environment consists of several transformation modules, each of which implementing one of the transformation schemes described in this paper. Direct interaction with the user will be provided at different levels. First, the user will be allowed to change his original specification after testing it using the sequential version produced by the module $\mathcal{S}$. Second the user can assist the mapping transformation by providing the mapping function. This function is used by the $\mathcal{M}$ transformation module to produce the parallel version of the program. The transformation system is based on the Glasgow Haskell Compiler which generates C code [14] but the concurrency aspects are managed by a suitable inclusion of calls to PVM (Parallel Virtual Machine [2]) C routines. The use of C and PVM ensures *instant portability* onto a variety of architectures ranging from a network of workstations running under UNIX to a dedicated parallel machine such as a multi-transputer system.

RELATED WORK

Some application-specific languages for SIT type of computations have been proposed, such as in molecular dynamics simulation [6], numerical analysis [3] and image processing. Although these packages have lot in common in terms of how to organise the data, neighbourhood communication, locality, and dealing with edge conditions, there have not been many attempts in designing a *generic* environment in which most SIT applications could be developed. The only

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

system we are aware of is the GRIDS programming environment [18] which is based on the imperative paradigm.

On the functional programming side, the proposed notation is close to Hudak's para-functional programming [8]. The "paradigm-oriented" approach is related of Cole's work on skeletons [4]. Paradigm-oriented implementations have been suggested for Divide-and-Conquer problems [15, 16] and process networks [10]. Process networks could be considered as a more general form than SIT algorithms because data objects are functions communicating with other objects using streams. However, process networks do not naturally support synchronisation, broadcasting and reduction. One of Darlington's skeletons [5] is closely related to ours but there are no suggested transformation rules from a specification to parallel code.

CONCLUSION

In this paper, we advocated a "paradigm-oriented" approach to the design and implementation of parallel functional programs. As a case study, a structure that allows a user to design iterative transformation algorithms has been defined. Compared to an implementation based on *implicit parallelism*, our approach ensures not only that a program contains sufficient parallelism but also provides crucial information for the run-time system about the nature of this parallelism. For example, the run-time system is able to statically allocate tasks to processors, partition the data in every processor's local store, and even generate update messages in advance.

Compared to an implementation based on *explicit parallelism*, the use of a parallel paradigm to express an algorithm makes it clearer, easier to modify, and allows for formal analysis and transformation into a variety of forms (sequential, parallel or adapted to a particular machine) thus making it highly portable. The drawback is that a transformation-based implementation cannot compete in efficiency terms. Each processor in the proposed system runs concurrently a main task and several update tasks and this incur an extra suspension/reactivation and context switching overheads which could be avoided if "straight-line" code with explicit communication primitives is used. It is hoped that this system would be useful for applications where portability, maintenance and a low development cost are as much important as efficiency.

REFERENCES

[1] D.I. Bevan et al., 'Design principles of a distributed memory architecture for a parallel graph reduction', *The Computer Journal*, 32(5), pp. 461-469, 1989.

[2] A. Beguelin et al., 'A User's Guide to PVM Parallel Virtual Machine', Oak Ridge National Laboratory (USA), July 1991.

[3] J.M. Boyle *et al*, 'The construction of numerical mathematical software for the AMT DAP by program transformation', In *Parallel Processing : CONPAR 92*

- *VAPP V*, L. Bougé et al. (Eds.), Lecture Notes in Computer Science 634, pp. 761-767, Sept. 1992.

[4] M. Cole, *Algorithmic skeletons : a structured approach to the management of parallel computation*, Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[5] J. Darlington *et al.*, 'Parallel Programming Using Skeleton Functions', to appear in *PARLE'93*, Munich, June 1993.

[6] P.A.J. Hilbers and K. Esselink, 'Parallel Molecular Dynamics', In *Parallel Computing : From Theory to Sound Practice*, W. Joosen and E. Milgrom (Eds), IOS Press, pp. 288-299, 1992.

[7] W.D. Hillis and G. L. Steele, 'Data Parallel Algorithms', *Communications of the ACM*, Vol 29, pp. 1170-1183, December 1986.

[8] P. Hudak, 'Para-Functional Programming in Haskell', In *Parallel Functional Languages and Compilers*, B.K. Szymanski (Ed.), ACM Press, 1991, pp. 159-196.

[9] P. Hudak *et al.*, 'Report on the Programming Language Haskell', *SIGPLAN Notices*, 27(5), May 1992.

[10] P. Kelly, *Functional Programming for Loosely-coupled Multiprocessors*, Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[11] H. Kingdon, D. Lester and G.L. Burn, 'The HDG-machine, a highly distributed graph reducer for a transputer network', Technical report 123, GEC Hirst Research Centre, March 1989.

[12] H.T. Kung, 'Computational models for parallel computers', In *Scientific Applications of Multiprocessors*, R.J. Elliott and C.A.R. Hoare (Eds), Prentice Hall International, 1989, pp. 1-15.

[13] R. Loogen et al., 'Distributed implementation of programmed graph reduction', In Proc. *PARLE '89*, June 1989, Odijk E. et al (Eds), Lecture Notes in Computer Science 365, pp. 136-157.

[14] W. Partain, 'The Glasgow Haskell Compiler', The GRASP project, Department of Computer Science, University of Glasgow, 1992.

[15] F.A. Rabhi and G.A. Manson, 'Experiments with a transputer-based parallel graph reduction machine', *Concurrency : Practice and Experience*, Vol 3 (4), Aug. 1991, pp. 413-422.

[16] F.A. Rabhi and G.A. Manson, 'Divide-and-Conquer and Parallel Graph Reduction', *Parallel Computing*, vol 17, 1991, pp. 189-205.

[17] F.A. Rabhi, 'Exploiting parallelism in functional languages: a "paradigm-oriented" approach', In *Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds, April 1993.

[18] A. Reuter, U. Geuder, M. Hardtner, B. Worner and R. Zink, 'GRIDS User's Guide', Report 4/93, University of Stuttgart, 1993.