15
Towards a theory of shared data in distributed systems

S. Dobson and C.P. Wadsworth
Rutherford Appleton Laboratory
Chilton, Didcot, Oxfordshire OX11 0QX, UK
Tel +44 1235 445867 Fax +44 1235 445945
E-mail {S.Dobson, C.P.Wadsworth}@ rl.ac.uk

Abstract

We have developed a *theory of sharing* which captures the behaviour of programs with respect to shared data into the framework of process algebra. The core theory can describe programs performing read and write access to unitary pieces of shared data. Extensions allow shared data to be decomposed and atomic copies to be made, reflecting the common operations of parallel programs. We describe the theory and give an example of its use in analysing and transforming a sample mathematical application.

Keywords

Sharing, process algebra, program analysis, program transformation

1 INTRODUCTION

Multiprocessor systems traditionally fall into two camps: shared memory, in which all processes have direct access to all the data in a computation; and distributed memory, in which access to much of the data involves explicit communication. Recently the distinction has become a little blurred through the use of virtual shared memory (Frank, 1992)(Li, 1989) and through the desire to make use of high-level data abstractions when building distributed applications.

In a typical distributed application there will be a body of data which is shared between some or all components. Examples include a mesh in a simulation, a set of tables in a distributed database, or a collection of pages in a distributed multimedia application. The efficient implementation of the application may require exploration of several different strategies before the "best" is chosen.

The danger is that decomposition blurs the programmer's conceptual model of the data being manipulated, and introduces subtle machine dependencies. These can make large applications hard to analyse and maintain, and damage their portability.

As part of the TallShiP collaborative project between RAL and the University of Leeds we have been investigating the use of high-level typed data abstractions for creating distributed applications. Our contention is that this approach raises the level of distributed programming so as to allow a

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

more structured, more portable and more easily-analysed – in short, better engineered – applications to be created. At the same time, it allows us to exploit type-specific information for optimised processing and distribution of the components of an application.

In order to explore the ways in which applications share data, we are developing a *theory of sharing*. The intention of this theory is to capture the sharing behaviour of applications, allowing different patterns of sharing to be identified, characterised and compared. We hope that this will lead to new insights into the design of efficient, portable shared data types, and to new methods for the optimised compilation and support of distributed applications.

Section two introduces the core theory of sharing. Section three describes two extensions covering wider class of systems. Section four presents some preliminary applications of the theory to program analysis. Section five relates the theory to similar work, and section six offers our conclusions and some directions for future work.

2 CORE THEORY OF SHARING

Sharing Areas and Events

Many kinds of data may be shared in an application, from simple variables to large structured types or multimedia objects. For our purposes, however, all shared data is represented by the single abstraction of a *sharing area*. A sharing area is a collection of zero or more named data items. A single variable is represented by a sharing area having one element; a large array by an area with many elements each identified by an index tuple. For the time being we consider sharing areas to be without internal structure; they are also *untyped* in that the theory does not describe the contents of elements or how they are named. All sharing areas are disjoint from all other sharing areas, in that no two areas share elements in common.

Having defined an abstraction for shared data we need ways in which to access it. Suppose that we have a set `S` of sharing areas, denoted `a,b,...`. A procedure may atomically read zero or more elements from a single sharing area: this action is denoted by `rd(a)` where `a` is the sharing area accessed. Similarly the action `wr(a)` denotes the atomic update of elements in sharing area `a`. We term these two basic actions the*events* of sharing theory.

Sharing Expressions

A *sharing expression* describes a program's interactions with sharing areas, built by composing the basic actions which may be applied to areas. Each sharing expression is a term in a modified process algebra (based on the system PA of Baeten and Weijland (1990)) using events instead of communication as the basic elements.

Events may be built into larger expressions using sequential composition, alternative composition and parallel composition, denoted by the functions ;, + and || respectively*. So the term `rd(a) ; (wr(a)+(wr(a)||wr(b)))` describes the sharing behaviour of a function which first reads elements from sharing area `a` and then *either* updates elements in `a` *or* updates elements in `a` and `b` in parallel. In PA, as in most process algebras, parallel composition is viewed as non-deterministic interleaving.

* PA uses . (dot) rather than ; (semi-colon) to denote sequential composition.

We may define a set of equations on the terms created from the events and combining operators. If x, y, z represent arbitrary terms, then:

$$\begin{array}{lll} x+y = y+x & (x+y)+z = x+(y+z) & x+x = x \\ (x+y);z = x;z+y;z & (x;y);z = x;(y;z) & x \| y = y \| x \end{array}$$

New sharing areas are introduced using the υ operator: the expression $\upsilon a.x$ (where x is an arbitrary term) introduces a new sharing area a for use in x (called the *scope* of a). For simplicity we assume that no sharing area name is ever re-used.

A sharing area a is said to *occur* in a term x if x contains an event $rd(a)$ or $wr(a)$. Events in a occurring inside the scope of a υa operator appear *bound*; any other occurrences appear *free*. We define using structural induction a function FA which computes the set of sharing areas appearing free in a term:

$$\begin{array}{lll} FA(rd(a)) = a & FA(x;y) = FA(x) \cup FA(y) & FA(x+y) = FA(x) \cup FA(y) \\ FA(wr(a)) = a & FA(x \| y) = FA(x) \cup FA(y) & FA(\upsilon a.x) = FA(x) \setminus \{a\} \end{array}$$

We then use FA to define side conditions for equations relating to the υ operator, controlling the scope of sharing areas in terms:

$$\begin{array}{llll} \upsilon a.(x;y) = x;\upsilon a.y & a \notin FA(x) & \upsilon a.(x;y) = (\upsilon a.x);y & a \notin FA(y) \\ \upsilon a.(\upsilon b.x) = \upsilon b.(\upsilon a.x) & & \upsilon a.x = x & a \notin FA(x) \\ (\upsilon a.x)+(\upsilon b.y) = \upsilon a.\upsilon b.(x+y) & & & \\ & a \notin FA(y) \wedge b \notin FA(x) & & \end{array}$$

The first two equations state that terms which do not actually use a sharing area may be moved into or out of its scope. The fourth axiom states that unused sharing areas may be deleted, or conversely that new areas may be introduced freely. The final axiom allows sharing areas to be factored into or out of alternative compositions.

Renaming Operators

PA defines a small set of renaming operators, allowing actions to be changed into other actions. We may usefully import this idea into sharing theory.

A *renaming function* is a function which maps actions to actions. The renaming function $r(f)$ maps every action f to some other action g (which may be the same as f). A renaming function ρ_r applies r to a term, and is defined by a straight-forward structural induction. So in the term $x = \upsilon a.\upsilon b.(rd(a);wr(b))$ if we define $r = id\{wr(b) \mapsto wr(a)\}$ then $\rho_r(x) = \upsilon a.\upsilon b.(rd(a);wr(a))$ changes every $wr(b)$ action into a $wr(a)$ action.

Note that ρ_r changes actions, not sharing areas. We may however use it to define another operator which renames sharing areas in a term. Let x be a term. Let S be a sub-set of the sharing areas occurring in x, and let S' be a set of new sharing areas not occurring in x. Let s be a *sharing area renaming function* mapping elements of S to elements of S'. Now create a renaming function such that for all $a \in S$ and $b=s(a)$ we have $r(rd(a)) = rd(b)$, $r(wr(a)) = wr(b)$ and r is the identity on all other actions. We now define a *sharing area renaming operator* α_s which renames sharing areas and their events. We first introduce each new sharing area in S' using the υ

operator, then rename according to the renaming function induced by $\mathtt{s}$. Using our example term $\mathtt{x}$ from above, if we set $\mathtt{s} = \mathtt{id\{b \mapsto c\}}$ then

$$
\begin{aligned}
\alpha_s(\mathtt{x}) &= \alpha_s(\nu\mathtt{a}.\nu\mathtt{b}.(\mathtt{rd(a);wr(b)})) & & \\
&= \nu\mathtt{c}.\nu\mathtt{a}.\nu\mathtt{b}.(\rho_r(\mathtt{rd(a);wr(b)})) & & \text{for } \mathtt{c} \notin \mathtt{FA(x)} \\
&= \nu\mathtt{c}.\nu\mathtt{a}.\nu\mathtt{b}.(\mathtt{rd(a);wr(c)}) & & \\
&= \nu\mathtt{c}.\nu\mathtt{a}.(\mathtt{rd(a);wr(c)}) & & \text{eliminating unused area } \mathtt{b}
\end{aligned}
$$

The usefulness of this operator will become apparent in the example (section 5).

Effect Analysis

We define a function $\mathtt{AE}$ to extract the free sharing areas in which the term causes events. The function generates two sets, containing the areas in which read events occur and the areas in which write events occur:

$$
\begin{array}{ll}
\mathtt{AE(rd(a))} = (\{\mathtt{a}\},\emptyset) & \mathtt{AE(wr(a))} = (\emptyset,\{\mathtt{a}\}) \\
\mathtt{AE(x+y)} = \mathtt{AE(x)+AE(y)} & \mathtt{AE(x;y)} = \mathtt{AE(x)+AE(y)} \\
\mathtt{AE(x\|y)} = \mathtt{AE(x)+AE(y)} & \mathtt{AE}(\nu\mathtt{a.x}) = \mathtt{AE(x)}-(\{\mathtt{a}\},\{\mathtt{a}\})
\end{array}
$$

(where + and - respectively denote pointwise set union and set difference on pairs). We use two functions $\mathtt{AE}_{\mathtt{rd}}$ and $\mathtt{AE}_{\mathtt{wr}}$ to project the first and second sets, so $\mathtt{AE}_{\mathtt{rd}}(\mathtt{x})$ is the set of sharing areas in which $\mathtt{x}$ causes read events.

3 EXTENSIONS

The core theory can express sharing in an important class of algorithms, but lacks some of the features commonly encountered in parallel and distributed applications. We shall now extend it in two directions, to encompass the decomposition and copying of data structures. These extensions are completely modular, in that they may be added individually or together to the core theory to generate a more expressive system.

Sub-Areas

Expressing algorithms as manipulations on large shared types can make applications more analysable. However, direct implementation of a large value as a single object may lead to contention and centralisation which would damage performance. Programmers must often decompose a data structure (such as a grid) into sub-parts which are then distributed and processed in parallel. There will in general be many different decomposition strategies for a value, each semantically equivalent but with different performance profiles. We would like to capture the decomposition of a value without losing the fact that the sub-parts combine to form a single, larger whole.

Disjointness, Containment and Decomposition

Two sharing areas $\mathtt{a}$ and $\mathtt{b}$ are *disjoint* (denoted $\mathtt{a}\oplus\mathtt{b}$) do not share elements in common. In the core theory, all sharing areas are mutually disjoint. We may relax this restriction and allow two areas share some elements. Of particular interest is the case where all the elements of a sharing area b are also elements of an area a, so that b identifies a sub-part of a. We denote this by $b \sqsubset a$, and say that a *contains* b (or that b is a *sub-area* of a).

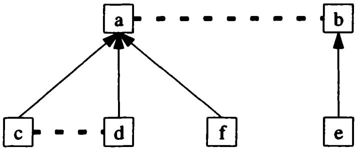

Figure 1 Sub-areas form a tree under the containment relation.

Introducing sub-areas into the theory of sharing requires another operator. If a is a sharing area, the term $\Delta_a\{a_1, a_2, \ldots, a_n\}.x$ introduces the set of disjoint sub-areas $a_1, a_2, \ldots, a_n$ of a into x. Each a_i is a sub-area of a, so $a_i \sqsubset a$. There may be elements of a not contained in any a_i. Both the υ and Δ operators introduce sharing areas. However, υ generates new (shared) state whereas Δ simply partitions existing state. So the term

$$\upsilon a.\upsilon b.(x;(\Delta_a\{c,d\}.\Delta_b e.(y;(\Delta_a f.z))))$$

describes a set of sharing areas which form a forest of trees (figure **1**, where the arrows denote containment of one area within another and the dotted lines denote areas which are explicitly known to be disjoint). We may also assert that if $a \oplus b$, $c \sqsubset a$ and $e \sqsubset b$ then $c \oplus e$ (and similarly for d); but it is not necessarily the case that$c \oplus f$. Extending the definition ofFA to encompass this new term

$$FA(\Delta_a D.x) = FA(x) \backslash D$$

for some set of sub-areas D, we may define some additional equations for the extended theory of sharing with sub-areas:

$$\Delta_a D.(x;y) = (\Delta_a D.x);y \qquad D \cap FA(y) = \varnothing$$

$$\Delta_a D.(x;y) = x;(\Delta_a D.y) \qquad D \cap FA(x) = \varnothing$$

$$\Delta_a D.(\Delta_b E.x) = \Delta_b E.(\Delta_a D.x)$$

$$\Delta_a \varnothing.x = x$$

$$\Delta_a(D \cup d).x = \Delta_a D.x \qquad d \notin FA(x)$$

$$(\Delta_a D.x) + (\Delta_b E.y) = \Delta_a D.\Delta_b E.(x+y) \qquad D \cap FA(y) = \varnothing \wedge E \cap FA(x) = \varnothing$$

Note that there is no equation of the form $\Delta_a D.\Delta_a E.x = \Delta_a(D \cup E).x$ as disjointness of sub-areas is not implied across separateΔ terms.

Generalisation

It often impedes the understanding of an application if the partitioning of data structures is made too explicit, and this is also true of sharing theory: the use of sub-areas can make analysis unnecessarily difficult. The *generalisation transform* allows us to abstract away from different sub-area decompositions where necessary.

If $\mathtt{a}$, $\mathtt{a_1}$ and $\mathtt{a_2}$ are sharing areas such that $\mathtt{a_1, a_2 \subset a}$, we say that $\mathtt{a}$ is a *generalised sharing area* of $\mathtt{a_1}$ and $\mathtt{a_2}$. ($\mathtt{a}$ may itself be a sub-area of another, larger area.) Let $\mathtt{S}$ be a set of mutually disjoint sharing areas, and let $\mathtt{r}$ be a renaming function. The generalisation transform $\Gamma_{\mathtt{r}}{}^{\mathtt{S}}$ is defined by another structural induction, the only interesting case of which is

$$\Gamma_r{}^S(\Delta_a D.x) = \begin{cases} \Delta_a D.\Gamma_r{}^S(x) & \text{if } \neg\exists c \in S.a \subset c \\ \Gamma_{r'}{}^S(x) \text{ where } r' = r\{rd(a_i) \mapsto rd(c), wr(a_i) \mapsto wr(c) | a_i \in D\} & \text{if } \exists c \in S.a \subset c \end{cases}$$

In a $\Delta_{\mathtt{a}}$ term, if there is an area in $\mathtt{S}$ which is a generalisation of $\mathtt{a}$ (there can be at most one such area) then the events in the scope of the term are re-written so as to be events in the generalised area.

The generalisation transform can generalise a selection of sharing areas to abstract away from any decomposition, or can eliminate decomposition entirely. Many terms may have the same generalisation, so generalisation may be seen as a form of refinement: two terms $\mathtt{y}$ and $\mathtt{y'}$ such that $\Gamma_{\mathtt{id}}{}^{\mathtt{S}}(\mathtt{y}) = \Gamma_{\mathtt{id}}{}^{\mathtt{S}}(\mathtt{y'})$ for some $\mathtt{S}$ can be considered to be "the same" *modulo* different decompositions.

Copy Events

Suppose we have a sharing area representing some commonly-read state in an application. A useful optimisation might be to pre-copy the data to sites which use it, forming a local cached copy at each site. This would reduce the number of accesses to a single point in the system.

To capture this notion within sharing theory, we introduce a new set of actions. For every pair $\mathtt{a,b}$ of sharing areas we define an event $\mathtt{cp(a,b)}$ denoting the atomic "snapshot" of all the elements of $\mathtt{a}$ into $\mathtt{b}$. Immediately after a copy event $\mathtt{b}$ is identical to $\mathtt{a}$. The effect of performing some action which usesa is the same as performing the same action using an identical copy of a:

$$\mathtt{\upsilon a.x = \upsilon a.\upsilon b.(cp(a,b);\alpha_{\{a\mapsto b\}}(x))} \qquad \mathtt{b \notin FA(x)}$$

In the presence of sub-areas we refine the definition slightly to prevent pathological cases such as copying the contents of an area into one of its sub-areas. We do this by restricting the existence of $\mathtt{cp(a,b)}$ events to those cases where $\mathtt{a \oplus b}$. Having done this, we may define the behaviour of copy actions on sub-areas as

$$\mathtt{\Delta_a(D \cup d).x = \Delta_a(D \cup d).\upsilon c.(cp(d,c);\alpha_{\{d\mapsto c\}}(x))} \qquad \mathtt{c \notin FA(x)}$$

where $\mathtt{d}$ is some sub-area of $\mathtt{a}$.

4 APPLICATIONS

The theory we have presented above allows us to capture the sharing behaviour of a wide class of programs. The programs may then be analysed for side effects, possible conflicts or non-determinism, and potentialoptimisations.

Program Analysis

A function or procedure within a program gives rise to a sharing expression which captures its dependence and influence on shared state. One may define a mapping from a language to sharing theory, and then manipulate the sharing expression to draw conclusions about the code. Our aim is to also reverse this mapping, to use sharing theory as the basis of a transformation system – to date we have concentrated on the analysis phase.

Conflicts and Synchronisation

Suppose we have a program in which an object is being updated in parallel by two functions. We represent the object by a sharing area a, and the two functions x and y as having behaviour given by rd(a);wr(a);...;wr(a). The overall behaviour of this system is given by the expression $z = \upsilon a.(x\|y)$. If we compute AE for x and y, we discover that $a \in AE_{wr}$ for both. This indicates that the parallel term has a potential conflict as different interleavings may introduce write events in different orders, making the program non-deterministic.

We say that two terms *interact* if the events caused by one may affect the behaviour of the other in terms of the results of read events. Two terms *interfere* if they interact or if they cause write events in a common area. Non-interacting terms cannot directly affect each others' behaviour as they do not update any state accessed by the other. Interference is a stronger condition which also encompasses terms which, while not necessarily affecting each other's actions, may still generate non-deterministic final effects if composed in parallel. Both interaction and interference are properties which may simply be determined by examination of the terms involved.

In some cases interference is observed "spuriously" because sharing theory works at the level of sharing areas, not elements within those areas. For example if the sharing area represents a large grid and the interfering functions are updating different parts of it, then there is no problem. If this property is captured by means of sub-areas, the interference is removed. Structured algorithm or type design can help make this information available.

For other cases, interference constrains an application to ensure that the atomicity of events is maintained, using locking *et cetera*. The converse is also true: in the absence of interference, no concurrency control is needed. This means that the theory can detect cases in which concurrency control may be "switched off" to avoid overheads.

Caching and Copying

Many algorithms make many more read accesses to shared data than write accesses, and it may be advantageous to create cached copies of the shared state local to each process rather than have all processes share a single copy. Sharing theory may be used to detect situations in which caching may be applied. The basic technique is to observe parts of a term in which a sharing area incurs only read events in parallel, and then create new copies of the area local to each parallel process.

For example, let x and y describe functions which repeatedly read elements from a sharing area a in order to update an area b, so that no write events are caused in a. We may transform this expression to introduce private copies ofa:

$$\begin{aligned}
X &= \upsilon a.\upsilon b.(x\|y)\\
&= \upsilon a.\upsilon b.((\upsilon c.x)\|(\upsilon d.y))\\
&= \upsilon a.\upsilon b.((\upsilon c.(cp(a,c);\alpha_{\{a\mapsto c\}}(x)))\|(\upsilon d.(cp(a,d);\alpha_{\{a\mapsto d\}}(y))))
\end{aligned}$$

In a distributed system such caching may be highly advantageous. Rather than access a single copy of some shared data, possibly involving network access, it is possible to generate a term which uses local copies of the data and is provably equivalent to the shared-data case. The technique is particularly effective in the presence of sub-areas, where we may generate local copies of only those parts of a piece of shared state which are actually needed in each partial computation.

However, not all such opportunities for caching and replication will be equally advantageous. There is a hidden assumption – not always made explicit in work on transformation – that accessing local copies is far less expensive than accessing a shared copy and justifies the copying overhead. In systems which make highly infrequent access to shared state it may not be worth performing this optimisation. Deciding between these two situations is an interesting problem.

Example: Parallel Solution of the 2-D Wave Equation

To demonstrate these techniques we shall analyse a program calculating the numerical solution of a partial differential equation. The application models the motion of a wave in a fluid medium – for example a pressure wave in a gas. In two dimensions this equation has the discrete form

$$C[i,j] = B[i,j] - A[i,j] + \frac{1}{4}(B[i+1,j] + B[i-1,j] + B[i,j+1] + B[i,j-1])$$

where three grids *A*, *B* and *C* are used to hold values of the simulation at different time steps. Grid *B* holds the values of points at time *t*; grid *A* at time *t-1*; and grid *C* holds the new values for time *t+1*. The new value of a point *(i, j)* is computed as a function of its past value and those of its immediate four neighbours. Let us assign sharing areas a, b and c to represent the grids *A*, *B* and *C* respectively. These sharing areas contain many elements, one for each point in the grids.

Simple Sharing Analysis

To compute the new value of a point we apply a function NewValue which accesses areas a and b in order to compute a value with which to update areac. This gives rise to the sharing expression:

```
newvalue   = (rd(b) ‖rd(a) ‖rd(b) ‖rd(b) ‖rd(b) ‖rd(b));wr(c)
```

Calculation of a single time-step involves applying NewValue to each point in the space in parallel:

```
calc = newvalue‖newvalue‖...‖newvalue
```

where each instance has the same sharing behaviour, but updates a different point in c.

Every instance of newvalue in calc interferes with every other instance, as each is updating c. However, as we know from the structure of the calculation that each instance updates a different point, this interference is spurious. We may make the structure explicit by decomposing c into sub-areas $c_1, c_2, \ldots, c_n$ such that each sub-area contains a single point. If we then apply each instance of newvalue to a different sub-area:

$$\begin{aligned}\texttt{calc'} &= \Delta_c\{c_1, c_2, \ldots, c_n\}.\\ &\quad \alpha_{(c\mapsto c1)}(\texttt{newvalue}) \parallel \alpha_{(c\mapsto c2)}(\texttt{newvalue}) \parallel \ldots \parallel \alpha_{(c\mapsto cn)}(\texttt{newvalue})\end{aligned}$$

the spurious interference has disappeared – at the price of an extremely complex sub-area structure. However, note that `calc` = Γ_r^s`(calc')`: `calc'` is simply a refinement of `calc` using a particular decomposition strategy, and we can easily generalise it to retrieve the simpler form.

Another possible strategy is to divide the decomposed parts of `c` into (say) four sets for distribution onto four processors:

$$
\begin{aligned}
\mathtt{calcd} = {} & \Delta_c\{\mathtt{p,q,r,s}\}.\\
& \quad (\Delta_p\{\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n\}.\\
& \qquad\qquad (\alpha_{\{c\mapsto p_1\}}(\mathtt{newvalue}) \,\|\, \alpha_{\{c\mapsto p_2\}}(\mathtt{newvalue}) \,\|\, \ldots))\\
& \quad \| \, (\Delta_q\{\mathtt{q}_1,\mathtt{q}_2,\ldots,\mathtt{q}_m\}.\\
& \qquad\qquad (\alpha_{\{c\mapsto q_1\}}(\mathtt{newvalue}) \,\|\, \alpha_{\{c\mapsto q_2\}}(\mathtt{newvalue}) \,\|\, \ldots))\\
& \quad \| \ldots
\end{aligned}
$$

In this case we see that the "processor" terms are non-interfering, and the instances of `newvalue` within each term are also non-interfering. Furthermore we may generalise the decomposition of the processor terms to obtain a term describing the events at each processor, and then further generalise to obtain `calc`.

Copying and Storage Re-use

Inspecting the discrete form of the wave equation, we see that the grids *A*, *B* and *C* are used cyclically: the values at time*t* become those at time*t-1* on the next cycle of computation. Abstractly

- the system calculates the values of*C* using those of*A* and *B*;
- it then creates three new grids*A'*, *B'* and *C'*;
- it copies the values of*B* into*A'* and *C* into*B'*; and
- it then performs the next cycle of calculation using the new grids.

This behaviour is captured by the expression

```
υa'.υb'.υc'.((cp(b,a')‖cp(c,b'))...)
```

and we may use this simple description of the system's behaviour and derive a new expression which performs "pointer swapping" and re-uses the existing storage without copying.

Let us consider the full definition of two cycles of computation, using the basic definition of `calc` for simplicity. The calculation has the behaviour:

$$
\begin{aligned}
&\mathtt{wave} = \upsilon\mathtt{a}.\upsilon\mathtt{b}.\upsilon\mathtt{c}.(\mathtt{setup;twostep})\\
&\mathtt{setup} = \mathtt{wr(a)} \,\|\, \mathtt{wr(b)}\\
&\mathtt{twostep} = \mathtt{calc;calc2}\\
&\mathtt{calc2} = \upsilon\mathtt{a'}.\upsilon\mathtt{b'}.\upsilon\mathtt{c'}.(\mathtt{cp(b,a');cp(c,b')};\alpha_{\{a\mapsto a',b\mapsto b',c\mapsto c'\}}(\mathtt{calc}))
\end{aligned}
$$

(where `setup` initialises the grids *A* and *B* with the initial state of the system). So the computation initialises the grids, calculates the first iteration, creates new grids and initialises them by copying, and then repeats the calculation. Theoptimised version replaces`twostep` with:

$$
\mathtt{twostep'} = \mathtt{calc};\alpha_{\{a\mapsto b,b\mapsto c,c\mapsto a\}}(\mathtt{calc})
$$

where the grids from the first step are re-assigned in the second.

Optimising wave involves converting twostep into twostep'. The only observation which we need to make involves the use of a copy event as the final action on a sharing area. If a copy is made of an area, and the original is never accessed again, then one may equally make use of the original area instead of the copy. This is expressed by the equation:

$$\nu\texttt{a.(x;}\nu\texttt{b.(cp(a,b).y))} = \nu\texttt{a.(x;}\alpha_{\{\texttt{b}\mapsto\texttt{a}\}}\texttt{(y))} \qquad \texttt{a}\in\texttt{FA(y)}$$

Using this equation, we may transformtwostep as follows:

$$\begin{aligned}
\texttt{twostep} &= \texttt{calc;calc2}\\
&= \texttt{calc;}\nu\texttt{a'.}\nu\texttt{b'.}\nu\texttt{c'.(cp(b,a');cp(c,b');}\\
&\qquad\qquad \alpha_{\{\texttt{a}\mapsto\texttt{a'},\texttt{b}\mapsto\texttt{b'},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc))}\\
&= \texttt{calc;}\nu\texttt{a'.}\nu\texttt{c'.(cp(b,a');}\nu\texttt{b'.(cp(c,b');}\\
&\qquad\qquad \alpha_{\{\texttt{a}\mapsto\texttt{a'},\texttt{b}\mapsto\texttt{b'},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc)))}\\
&= \texttt{calc;}\nu\texttt{a'.}\nu\texttt{c'.(cp(b,a');}\alpha_{\{\texttt{b'}\mapsto\texttt{c}\}}\texttt{(}\alpha_{\{\texttt{a}\mapsto\texttt{a'},\texttt{b}\mapsto\texttt{b'},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc)))}\\
&= \texttt{calc;}\nu\texttt{c'.}\nu\texttt{a'.(cp(b,a');}\alpha_{\{\texttt{b'}\mapsto\texttt{c}\}}\texttt{(}\alpha_{\{\texttt{a}\mapsto\texttt{a'},\texttt{b}\mapsto\texttt{b'},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc)))}\\
&= \texttt{calc;}\nu\texttt{c'.(}\alpha_{\{\texttt{a'}\mapsto\texttt{b}\}}\texttt{(}\alpha_{\{\texttt{b'}\mapsto\texttt{c}\}}\texttt{(}\alpha_{\{\texttt{a}\mapsto\texttt{a'},\texttt{b}\mapsto\texttt{b'},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc))))}\\
&= \texttt{calc;}\nu\texttt{c'.(}\alpha_{\{\texttt{a}\mapsto\texttt{b},\texttt{b}\mapsto\texttt{c},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc))}
\end{aligned}$$

which eliminates the intermediate grids *A'* and *B'*, but still generates a new grid *C'* for each cycle. We may re-use the grid *A* instead of creating *C'*, but the system has no way of determining this: if we indicate it, by using an additional copy action, then

$$\begin{aligned}
\texttt{twostep} &= \texttt{calc;}\nu\texttt{c'.(cp(a,c');}\alpha_{\{\texttt{a}\mapsto\texttt{b},\texttt{b}\mapsto\texttt{c},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc))}\\
&= \texttt{calc;}\alpha_{\{\texttt{c'}\mapsto\texttt{a}\}}\texttt{(}\alpha_{\{\texttt{a}\mapsto\texttt{b},\texttt{b}\mapsto\texttt{c},\texttt{c}\mapsto\texttt{c'}\}}\texttt{(calc))}\\
&= \texttt{calc;}\alpha_{\{\texttt{a}\mapsto\texttt{b},\texttt{b}\mapsto\texttt{c},\texttt{c}\mapsto\texttt{a}\}}\texttt{(calc)}
\end{aligned}$$

and we have achieved our aim.

Halos and Caching

The method described above is a generic technique, applicable to any system based around iterative time-series methods. The most common implementation of such divides the grids into disjoint regions which are mapped onto different processors for calculation in parallel, often with "halos" of data dependencies between regions.

Let us define a distributed memory two-step solver on four processors using a quadrant partitioning. Computing a point in a particular region of C may involve access to points in the corresponding and neighbouring regions of A and B. The distributed update function for a particular region, $\texttt{newvaluer}_\texttt{n}$, is thus defined by

$$\begin{aligned}
\texttt{newvaluer}_\texttt{n} &= \texttt{(rd(b}_\texttt{n}\texttt{)}\,\|\,\texttt{rd(a}_\texttt{n}\texttt{)}\,\|\,\texttt{rdhalo}_\texttt{n}\texttt{);wr(c}_\texttt{n}\texttt{)}\\
\texttt{rdhalo}_\texttt{n} &= \texttt{(rd(b}_\texttt{n}\texttt{)+rd(b}_{\texttt{n-left}}\texttt{))}\,\|\,\texttt{(rd(b}_\texttt{n}\texttt{)+rd(b}_{\texttt{n-right}}\texttt{))}\,\|\\
&\qquad \texttt{(rd(b}_\texttt{n}\texttt{)+rd(b}_{\texttt{n-up}}\texttt{))}\,\|\,\texttt{(rd(b}_\texttt{n}\texttt{)+rd(b}_{\texttt{n-down}}\texttt{))}
\end{aligned}$$

where $\mathtt{b}_{\mathtt{n-left}}$ is the left-neighbouring region of $\mathtt{b}_n$ and so forth, depending on the indexing scheme chosen. For a single cycle within a region, we apply $\mathtt{newvaluer}_n$ to all the points within the region:

$$\mathtt{calcr}_n = \mathtt{newvaluer}_n \| \mathtt{newvaluer}_n \| \ldots$$

For the four-processor decomposition, we would perform a single cycle of the computation by applying$\mathtt{calcr}_n$ to all four sets of sub-areas. The full two-cycle computation is given by:

$$\begin{aligned}
\mathtt{waver} &= \upsilon\mathtt{a}.\upsilon\mathtt{b}.\upsilon\mathtt{c}.\mathtt{(setup;twostepr)}\\
\mathtt{twostepr} &= \mathtt{step1;step2}\\
\mathtt{step1} &= \Delta_{\mathtt{a}}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b}}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c}}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\mathtt{calcr}\\
\mathtt{step2} &= \upsilon\mathtt{a'}.\upsilon\mathtt{b'}.\upsilon\mathtt{c'}.\mathtt{(cp(b,a');cp(c,b');}\\
&\qquad \Delta_{\mathtt{a'}}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b'}}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c'}}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\mathtt{calcr)}\\
\mathtt{calcr} &= \mathtt{calcr}_1\|\mathtt{calcr}_2\|\mathtt{calcr}_3\|\mathtt{calcr}_4
\end{aligned}$$

We have not yet identified the halos explicitly. This is a useful thing to do, as it defines exactly which parts of a sub-grid are needed in computations on other sub-grids, which in turn allows optimisation of the sharing. The grid represented by $\mathtt{b}$ has been divided into four sub-grids $\mathtt{b}_1$–$\mathtt{b}_4$. Each of these sub-grids is further divided into three parts: vertical halo, horizontal halo, and non-halo elements. Note that these divisions are *not* disjoint, as the two halo areas share an element in the corner. We identify the two halo regions within an area $\mathtt{b}_n$ by $\mathtt{b}_{nv}$ and $\mathtt{b}_{nh}$ for the vertical and horizontal halo areas respectively. Using this decomposition$\mathtt{newvaluer}_n$ may be re-written as

$$\begin{aligned}
\mathtt{newvaluer}_n{}' &= \mathtt{(rd(b}_n\mathtt{)}\|\mathtt{rd(a}_n\mathtt{)}\|\mathtt{rdhalo}_n\mathtt{');wr(c}_n\mathtt{)}\\
\mathtt{rdhalo}_n{}' &= \mathtt{(rd(b}_n\mathtt{)+rd(b}_{\mathtt{nh-left}}\mathtt{))}\|\mathtt{(rd(b}_n\mathtt{)+rd(b}_{\mathtt{nh-right}}\mathtt{))}\|\\
&\quad\ \mathtt{(rd(b}_n\mathtt{)+rd(b}_{\mathtt{nv-up}}\mathtt{))}\|\mathtt{(rd(b}_n\mathtt{)+rd(b}_{\mathtt{nv-down}}\mathtt{))}
\end{aligned}$$

where $\mathtt{b}_{\mathtt{nh-left}}$ denotes the horizontal halo of $\mathtt{b}_n$'s left neighbour and so forth. These expressions give rise to an expression $\mathtt{calcr'}$ for the full calculation. The expression $\mathtt{step1}$ may be re-written to use halos:

$$\begin{aligned}
\mathtt{step1'} &= \Delta_{\mathtt{a}}\{\mathtt{a}_1,\mathtt{a}_2,\mathtt{a}_3,\mathtt{a}_4\}.\Delta_{\mathtt{b}}\{\mathtt{b}_1,\mathtt{b}_2,\mathtt{b}_3,\mathtt{b}_4\}.\Delta_{\mathtt{c}}\{\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4\}.\\
&\quad \Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{h}}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{v}}.\ldots.\mathtt{calcr'}
\end{aligned}$$

We may show that $\mathtt{step1'}$ only ever accesses sub-areas of each $\mathtt{b}_n$, not the full areas. Furthermore, no process ever causes write events within any $\mathtt{b}_n$ in the scope of $\mathtt{step1'}$. We may therefore introduce copy actions to move the halo regions used by each sub-calculation into a local area. To make the derivation simpler to read, we shall abstract from $\mathtt{step1'}$ the terms which relate to the calculation of area$\mathtt{c}_1$, and transform them:

$$\begin{aligned}
\mathtt{step1'} &= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{h}}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{v}}.\mathtt{(newvalue}_1\|\ldots\mathtt{)}\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{h}}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{v}}.\mathtt{(}\upsilon\mathtt{h.(cp(b}_{2\mathtt{h}}\mathtt{,h);}\alpha_{\{\mathtt{b2h}\mapsto\mathtt{h}\}}\mathtt{(newvalue}_1\mathtt{)))}\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{h}}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{v}}.\ \mathtt{(}\upsilon\mathtt{h.(cp(b}_{2\mathtt{h}}\mathtt{,h);}\upsilon\mathtt{v.(cp(b}_{3\mathtt{v}}\mathtt{,v);}\\
&\qquad \alpha_{\{\mathtt{b3h}\mapsto\mathtt{h},\ \mathtt{b3v}\mapsto\mathtt{v}\}}\mathtt{(newvalue}_1\|\ldots\mathtt{))))}\\
&= \ldots\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{h}}.\Delta_{\mathtt{b}_1}\mathtt{b}_{1\mathtt{v}}.\mathtt{(}\upsilon\mathtt{h.(cp(b}_{2\mathtt{h}}\mathtt{,h);}\upsilon\mathtt{v.(cp(b}_{3\mathtt{v}}\mathtt{,v);}\\
&\qquad \alpha_{\{\mathtt{b2h}\mapsto\mathtt{h},\ \mathtt{b3v}\mapsto\mathtt{v}\}}\mathtt{(newvalue}_1\mathtt{))))}\|\ldots\mathtt{)}
\end{aligned}$$

The `newvalue`$_n$ terms make use of the correct halos, copied into new areas used only by them. This means that the calculation of `step1'` may be re-written so that each region calculation first copies its halo into a local sharing area before using it, and does not access any other regions in the course of its computation. This models pre-fetch copying of data into local memory. Once more, the transformations are only effective because of a precise knowledge of the update behaviour of the underlying function`NewValue`.

5 RELATED WORK

Research on process algebra has traditionally focused on calculi using communication between processes – indeed, our system is the only process algebra of which we are aware which addresses shared data. We see sharing theory as a possible complement to the usual algebras in the specification of shared memory computations.

Parallelising compilers make use of ,many of the optimisations we have identified. We believe that our theory allows many of the techniques of parallelisation, data dependence analysis (Zima, 1991) and interference analysis (Lucassen, 1988) to be cast in a new and more tractable framework. Furthermore, the theory gives insights into the design of types and operations which may eliminate much complex analysis by making the sharing behaviour of functions available directly to the compiler. In many ways this is closely related to algorithmic skeletons and bulk data types (Bird, 1986)(Skillicorn, 1991)(Skillicorn, 1995) with the important addition of being applicable to mutable data types.

Another related issue is that of weak memory coherence (Frank, 1992)(Li, 1989) caching, and bulk synchrony (McColl, 1994). We may use copy events to model the action of systems where "shared" state is not updated synchronously across a system, although the correspondence is far from exact and needs further investigation.

6 CONCLUSIONS AND FUTURE WORK

We have described the development of a theory of sharing in distributed systems, using a modified process algebra which allows shared pieces of state to be defined and manipulated. The core theory can describe programs performing read and write access to unitary pieces of shared data. Extensions allow shared data to be decomposed and atomic copies to be made. The theory can easily detect common synchronisation problems, and can be used to transform systems which use local caches of read-only data.

Our approach is to define high-level shared abstract data types whose definitions capture the most common programming idioms, including their sharing behaviour. We see sharing theory as applicable in three ways:

- in specifying the sharing behaviour of operations of types;
- in analysing new operations for unwanted interactions or potential bottlenecks, to ensure the type is scalable; and
- in defining new operations.

The first application uses the theory as a concise description of a function's interactions and side effects, and is close to the traditional uses of process algebra in specification. The second helps ensure that any types included in a distributed applications library are indeed scalable. The third –

rather more speculative – allows the programmer to define new operations and associate sharing expressions with them (or derive them automatically), making new functions "first class citizens".

Our immediate plans for the future include investigating performance models to guide analysis, for example to differentiate between possible and advantageous opportunities for caching. This will lead to methods to aid the design of types suitable for portable distributed programming, and the development of automated tool support for the analysis and manipulation of sharing expressions.

7 REFERENCES

Baeten, J.C.M. and Weijland, W.P. (1990) Process algebra. Cambridge University Press.

Bird, R. (1986) An introduction to the theory of lists, in *Logics for Programming and Calculi of Discrete Design.*

Frank, S. (1992) Virtual memory to ALLCACHE memory, in *Proceedings of the Virtual Shared Memory Symposium*, Centre for Novel Computing, University of Manchester

Li, K. and Hudak, P. (1989) Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, **7**, 243–271.

Lucassen, J.M. and Gifford, D.K. (1988) Polymorphic effect systems, in *Proceedings of the 15th ACM Symposium on Principles of Programming Languages.*

McColl, W.F. (1994) BSP programming, in *DIMACS series in Discrete Mathematics and Theoretical Computer Science.*

Milner, R. (1986) A calculus of communicating systems. Technical report ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, University of Edinburgh.

Skillicorn, D.B. (1991) Models for practical parallel computation. *International Journal of Parallel Programming*, **20**, 133–158.

Skillicorn, D.B. (1995) Categorical data types, in *Abstract Machine Models for Highly Parallel Computing* (ed. J.R. Davy and P.M. Dew), Oxford Science Publishers.

Zima, H. and Chapman, B. (1991) Supercompilers for parallel and vector computers. ACM Press.

8 BIOGRAPHIES

Simon Dobson received a DPhil in Computer Science from the University of York in 1993, with a thesis on programming models for highly scalable computers. He joined the Rutherford Appleton Laboratory as a research fellow in 1992 to pursue his interests in languages and architectures for parallel and distributed systems, and has worked on a variety of projects involving advanced compilation techniques, system architectures, formal methods and hypermedia.

Chris Wadsworth started his research career in the Programming Research Group at Oxford University and worked at Syracuse and Edinburgh before moving to the Rutherford Appleton Laboratory in 1981 where he heads the Parallel and Distributed Systems Group. He is well known for his seminal contributions to lambda calculus, lazy evaluation and denotational semantics, and was a joint developer of the LCF theoremprover and the ML language.