Structured parallel programming: parallel abstract data types

John Darlington  Imperial College, London, UK

Abstract

A key requirement for any parallel programming system is the ability to support abstraction in order to allow programmers to work with familiar and natural concepts from their area of application without having to be too concerned with the low level complexities of parallel programming. However, some control over execution patterns and resource allocation is still required to ensure efficient execution. In this paper we show how these two, apparently conflicting, requirements can be reconciled in the structured parallel programming framework developed in the earlier paper. In particular we show how using the Structured Co-ordination Language (SCL) and its particular Fortran instantiation, Structured Parallel Fortran (SPF), we can systematically define user-oriented Parallel Abstract Data Types (PADTs) which provide a set of higher level co-ordination forms with which a user can construct his or her application. The important point is that parallelism is encapsulated in these operators which are implemented using the lower level co-ordination forms developed in the previous paper which provide explicit control over parallel behaviour. Knowledge available in the abstract structures can be utilised in these implementations to achieve efficient execution. We illustrate this technique by the development and implementation of a PADT to support parallel computation over an unstructured mesh.

Keywords: parallel programming; abstract data types

1  Introduction

Irregular computation underlies many important applications in science and engineering, such as computational fluid dynamics and particle simulation. A common characteristic of these problems is that they compute over irregular data structures, such as trees and graphs, rather than regular data structures which can be represented directly as arrays. Such applications pose a serious difficulty for most data parallel programming languages, such as HPF [2], which have usually been designed with regular applications in mind. This paper explores the opportunities of building parallel abstract data types (PADTs) for writing irregular data parallel applications in the Structured Parallel Fortran (SPF) language introduced in the accompanying paper. The philosophy of this approach is to build the PADTs on top of the existing language constructs rather than to extend the language. The advantage of this approach is that the kernel language remains small, thus enhancing portability and simplifying the language design and implementation by reducing redundancy. The use of PADTs allows problems to be expressed at a level of abstraction closer to the application programmer than is possible using proposed language extensions [1].

2  Abstraction in conventional programming languages

The need for abstraction arises from the contrast between the structures that occur naturally in several important application areas and the concrete, execution-oriented, structures found in many conventional programming languages. For example many scientific and engineering applications involve discretisations or meshings of physical spaces which have their own natural structures and characteristic operations with which any scientist would be familiar. In contrast Fortran 77 provides one fundamental data
structure, the array. There are well known techniques for implementing scientific structures using arrays but they all involve some coding or representation of the abstract structure in the concrete representation. Crucially this coding is implicit and all knowledge of the natural structure is lost in its programming language representation. This has two, undesirable, consequences. Firstly, the programmer’s task is made more difficult as he or she has to deal with complex and unnatural structures and secondly, much information that is naturally associated with the abstract structure is lost in the lower level representation. Thus parallelisation, optimisation or resource allocation decisions that could take advantage of this information are more difficult to make. This gap is not so apparent for parallelising regular domain decomposition problems where the array is a natural encoding of a regular mesh in Cartesian space, but for more sophisticated irregular and adaptive problems the parallel programming and the operations necessary on this coding become more complex and opaque.

It would be advantageous, therefore, to be able to define high-level, problem-oriented, data structures and have users code directly using them. This capability would pave the way to the development of user-oriented application development environments that would allow domain specialists to develop parallel applications using familiar concepts and not have to be concerned with the programming details of their implementation. The problem, as discussed in the preceding paper, is one of efficiency. Given the conventional programming language approach to building applications some direct control of execution patterns and resource allocation is necessary to achieve efficient execution. The approach adopted in SPF, in contrast, separates the specification of the intent of an application from its efficient implementation. Thus this intent can be specified at a suitably abstract level using pre-defined, abstract, co-ordination forms or skeletons and the efficient execution achieved by tailored implementations of these building blocks in lower-level, execution oriented, data structures.

3 Layered abstractions in SPF

The realisation of abstraction in SPF relies on the capacity of the language for extension; the definition, within the language, of higher-level structures and the realisation of these abstractions by tailored representations and implementations in lower-level, machine-oriented, structures that are invisible to an end-user.

The definition and realisation of an abstract structure in SPF has the following five distinct components:

- **Introduction of the parallel abstract data type.** A PADT consists of the name of a naturally occurring type and set of parallel operators over this type.
- **Specification of the meaning of the PADT.** A PADT and its associated operators can be given a precise meaning independent of any implementation by specifying the semantics of the operators on some mathematical representation of the type.
- **Representing the PADT in a concrete structure.** The PADT can be represented in some concrete structure, typically an array or configuration of arrays. This encoding can be specified formally as a mapping between the mathematical presentation of the PADT and its representation in SPF.
- **Implementation of the PADT operators.** Given the above encoding of the PADT structure implementations can be written for its operators in the concrete structures in SPF. The formal correctness of these implementations can be established by showing that they satisfy the specification of the operators under the encoding used.
It is intended that the above procedure is carried out by the developers of the PADT. An end user would just be presented with the pre-defined operators of the PADT and other appropriate co-ordination forms. An application would be constructed by the composition of these forms. This would then be executed by expanding the PADT operators to their structured parallel implementation using the concrete representation. In addition a set of application-oriented transformation rules could be available to assist the optimisation of programs constructed in this way and knowledge or information associated with the abstract structure could be built into the tailored implementations of the PADT operators.

A key point is that there is no explicit control over parallelism at the PADT level. In contrast it is encapsulated in the implementation of the PADT operators and expressed using the lower level explicit parallel co-ordination forms developed in the earlier paper. Crucially, however, knowledge inherent in the PADT can be utilised to provide efficient parallel implementations without requiring user intervention.

We have followed this procedure to build a set of PADT’s for scientific and engineering computation on top of the kernel SPF described in the previous paper. In particular we have implemented PADT’s for unstructured mesh applications and adaptive particle simulations. In the rest of this paper we present the development and use of a PADT for unstructured meshes.

4 An unstructured triangular mesh application

The example we shall study in this paper is based on an irregular triangular mesh. The example abstracts the code structure often found in applications such as Euler solvers which model fluid flow over physical structures. The physical space being modelled is discretised into a triangular mesh in the two-dimensional case and a tetrahedral mesh in the three-dimensional case. A small example of a triangular mesh for two-dimensional problem is shown in Figure 1. In abstract terms this structure is a graph, a collection of vertices with associated values connected by directed edges. One of the most common computations performed over a triangular mesh is a local neighbourhood computation. In a local neighbourhood computation the value of each vertex is updated by the sum of the weighted values of the vertex’s neighbouring vertices. In general the function used to calculate the weight of a neighbouring vertex will depend on whether the vertex is connected by an in-coming or out-going edge.

Specifically in the fragment of the Euler solver we are studying, there are two values, representing flow variables, associated with each vertex. These two variables will be referred to as X and Y. The computation performed by the application is exactly that of the local neighbourhood operation. The function used to calculate the weight of a neighbouring vertex connected by an out-going edge is \( F(x_1, x_j) \) where \( x_1 \) is the X value of the vertex being updated and \( x_j \) is the X value of the connecting vertex. The
function used to calculate the weight of a neighbouring vertex connected by an in-coming
edge is $G(X_j, X_i)$ where $X_i$ is the $X$ value of the vertex being updated and $X_j$ is the $X$
value of the neighbouring vertex. The overall computation is described by the following
pseudo code:

DO I = 1, NUM_VERTICES
   Y value of Ith vertex
   = Y value of Ith vertex
   + $\sum G(X_i, X_j)$
   where $X_j$ is the $X$ value a vertex connected by an out-going edge
   + $\sum G(X_j, X_i)$
   where $X_j$ is the $X$ value a vertex connected by an in-coming edge

10 CONTINUE

4.1 A triangular meshes PADT

A triangular mesh of type TMesh is a set of vertices and a set of edges. A formal
definition of the triangular mesh PADT using set notation and a functional programming
syntax is:

TMesh key value :: ( Vertices key value, Edges key)
Vertices key value :: { (key, value) }
Edges key :: { (key, key) }

To manipulate an instance of a triangular mesh a number of operators over the
triangular mesh are required. Typical operations over a triangular mesh include
TMesh_map and TMesh_LNO. TMesh_map, changes the value of every vertex in the mesh by
applying a given function to every vertex.

TMesh_map :: (value1 -> value2) -> TMesh key value1 -> TMesh key value2
TMesh_map f (V, E) = ( { (k, f x) | (k, x) in V }, E )

The local neighbourhood operator TMesh_LNO updates the value of each vertex in a mesh
by the sum of the weights of its neighbouring vertices’ values. This operator has been
abstracted by generalising the functions for calculating the weights of the neighbouring
vertices and the functions for summing the weighted values together. The weight of a
neighbour $J$ connected by an in-coming edge to a vertex $I$ is computed by $inWeight(J, I)$. Similarly the weight of a neighbour $J$ connected by an out-going edge to a vertex $I$ is
computed by $outWeight(I, J)$. The contribution from edges flowing into or out of a
vertex are summed using function sum. The operator can be defined as:

TMesh_LNO :: (value -> value -> value) -> (value -> value -> value) ->
   (value -> value -> value) ->
   TMesh key value -> TMesh key value
TMesh_LNO sum inWeight outWeight (V, E) =
   ( { (k, update k x) | (k, x) in V }, E )
   where
   update k x = $\sum x \text{ sum } ((\text{inCont } k x) \cup (\text{outCont } k x))$
   inCont k x = { inWeight(V(k'), x) |
               (k', k) in E }
   outCont k x = { outWeight(x, V(k')) |
                 (k, k') in E }

4.2 Expressing the application using the PADT

The operators described for the triangular mesh PADT are sufficient for expressing the
Euler solver application. Assuming that the value of a vertex is represented by a pair of
variables, where the first element is \( X \) and the second element is \( Y \), the application can be expressed as:

\[
\text{EulerFragment mesh = TMesh_LNO(ADD_Y, G_X, F_X, mesh)}
\]

Where \( \text{ADD}_Y, \text{G}_X \) and \( \text{F}_X \) are functions in a base language (e.g. Fortran 77).

Notice that there is no indication of the underlying parallelism, other than through the use of PADTs. However, the appropriate method for partitioning a triangular mesh will usually depend on the particular application. To provide the programmer with a means for specifying the partitioning several distribution schemes will be provided. Note, however, that the distribution schemes are described at the abstract level over the abstract data type rather than the underlying implementation. For example, a scheme commonly used for a triangular mesh is orthogonal recursive bi-section (ORB) [3]. The application with partitioning scheme specified is:

\[
\text{EulerFragment mesh = TMesh_LNO(ADD_Y, G_X, F_X, mesh, ORB)}
\]

### 4.3 Concrete encoding of the triangular mesh PADT

A triangular mesh can be represented concretely in Fortran 77 by two arrays, an array \( \text{Vertices} \) containing the value of the vertices and an array \( \text{Edges} \) containing the edges. In this representation, an edge is a pair of indices. The first index is the location of the value of the vertex at the start of the edge. The second index is the location of the value of the vertex at the end of the edge. This is shown in Figure 2.

In the concrete representation described in this paper a triangular mesh will be partitioned by vertices. One might assume that the concrete parallel representation of the triangular mesh can be obtained by simply distributing the elements of the \( \text{Vertices} \) array across the processors using some translation of the distribution scheme. However, this would be incorrect as can be seen when considering the patterns of computation over the abstract data structure. In \( \text{TMesh_LNO} \) updating the value of a vertex requires the values of all the vertex's neighbouring vertices. In general, some of the neighbouring vertices of the vertices in a partition will belong to another partition. The solution adopted here is to copy the value of these neighbouring non-local vertices onto the
Figure 3: A partition of the triangular mesh and its halo.

Figure 4: The SPP(X) representation of a partition and its halo.

partition. These duplicated vertices are known as the halo. An example is shown in Figure 3.

The need for haloes implies that when partitioning the sequential representation of a triangular mesh extra data structures must be introduced. A partition of the triangular mesh is represented in SPF by three arrays, the Vertices array, the Edge array and the HaloLocation array as shown in Figure 4. Notice how the halo vertices are held in the same array as the local vertices, but after the local vertices. By keeping track of the number of local vertices it is possible to differentiate between the two kinds of vertices. An extra array HaloLocation is necessary for updating the value of the halo vertices. For each halo vertex it holds the location of the vertex in the Vertices array and the identifier of the partition to which the vertex belongs.

4.4 Operations over the concrete representation

Several operations are defined in SPF for manipulating the concrete representation of the partitioned mesh. These include partitioner, halo_add and halo_update. partitioner takes a mesh-specific strategy name for partitioning the array, such as ORB, and returns a Fortran 77 function which can be used to compute the corresponding partition when used with the SCL operator partition. The operator halo_add when applied to a configuration, expands each local array with their neighbouring data. The neighbourhood relationship is determined and computed by a parameter Fortran 77 function. halo_update updates the halo values in the vertices arrays.
4.5 Structured implementation of operators of the PADT

To implement a program written using the abstract data type operators with a structured parallel language, each abstract data type operator is coded as a structured parallel program composed with a set of co-ordination forms abstracting the parallel behaviour. A SPF implementation of the TMesh_LNO is:

```spf
tMesh_LNO sum inWeight outWeight mesh distStrategy
  = gather (halo_update (map (S_LNO sum inweight outWeight) pmesh))
  where
    < dVertices, dEdges>
    = partition (partitioner distStrategy) mesh
    < dHaloVertices, dHaloLocation >
    = halo_add (findNeighbour mesh) < dVertices >
  pmesh = < dHaloVertices, dEdge, dHaloLocation>
```

```fortran
spf_subroutine S_LNO (sum, inWeight, outWeight, lX, lY, lEdge)
  real, dimension(nn), intent(inout) :: lX
  real, dimension(nn), intent(inout) :: lY
  integer, dimension(ne), intent(in) :: lEdge

do i = 1, ne
  n1 = lEdge(i,1)
  n2 = lEdge(i,2)
  lY(n1) = Sum (lX(n1), outWeight(lX(n1), lX(n2)))
  lY(n2) = Sum (lX(n1), inWeight(lX(n1), lX(n2)))
end do
```

```fortran
spf_subroutine findNeighbour (n, edges, neibOf)
  integer, intent(in) :: n
  integer, dimension(maxEdges,2), intent(in) :: edges
  integer, dimension(maxNeibs), intent(out) :: neibOfi

num-neib = 1
do i = 1, maxEdges
  if (edges(i,1) .eq. n) then
    neibOf(num-neib+1) = edges(i,2)
    num-neib = num-neib + 1
  endif
enddo
neibOf(1) = num-neib-1
```

5 Conclusion

In this paper we have briefly illustrated the capacity of SCL to support both the definition of abstract, user-oriented, data structures and their efficient implementation in a, hopefully, familiar Fortran context. We claim that this capability breaks the tension between abstraction and efficiency inherent in the programming language approach to parallel application construction and paves the way for the development high-level application environments that would enable scientists and engineers to construct applications quickly and economically while still retaining the ability to have them execute efficiently on parallel machines.

7
Acknowledgements

The ideas presented in these two papers have been developed in collaborative projects within the Advanced Languages and Architectures Section and the Fujitsu Parallel Computing Research Centre at Imperial College. We are grateful to all our colleagues there but in particular Peter Au, Moustafa Ghanem, Yi-ke Guo, Martin Kohler, Hing Wing To, Jin Yang and Wu Qian for their assistance. We thank Hing Wing To and Yi-ke Guo for their help in the preparation of these papers. This work is supported by the UK Engineering and Physical Sciences Research Council and Fujitsu Laboratories who also support the Fujitsu Parallel Computing Research Centre. We are very grateful for their continued assistance.

References