

Exploiting Shape in Parallel Programming

C. Barry Jay David G. Clarke Jenny J. Edwards

School of Computing Sciences,
University of Technology, Sydney,
PO Box 123 Broadway, NSW,
2007, Australia.
Phone: +61 2 330 1848
Facsimile: +61 2 330 1807
E-mail: {cbj,clad,jenny}@socs.uts.edu.au

Abstract

Shape theory is a new approach to data types and programming based on the separation of a data type into its “shape” and “data” parts. Shape is common in parallel computing. This paper identifies areas where the explicit use of shape reduces the burden of programming a parallel computer, via the implementation of Cholesky decomposition.

keywords: shape, parallel programming, error detection, Cholesky decomposition

Abstract

Shape theory is a new approach to data types and programming based on the separation of a data type into its “shape” and “data” parts. Shape is common in parallel computing. This paper identifies areas where the explicit use of shape reduces the burden of programming a parallel computer, via the implementation of Cholesky decomposition.

1 Introduction

Shape theory [15, 11, 12] consists of shapely data types, based on the separation of shape and data, shapely operations, and shape analysis. Shape theory can be used to:

- describe complicated data structures and provide mechanisms for debugging them (in both sequential and parallel settings)
- give a semantic description for the operations of packing and unpacking data to and from messages when performing calculations in a MIMD parallel message passing environment
- capture the notions of scattering data across parallel processors, and gathering the results.

Shape analysis gives constraints on the shapes of inputs to various operations, and provides a new kind of error checking. This paper describes these applications of shape theory with particular examples from the parallel implementation of a common linear algebra algorithm, Cholesky decomposition [21]. However, these concepts can be extended quite easily to other parallel algorithms.

1.1 Problems of Parallel Programming

Before attempting to define shape, let us consider some of the concerns of the parallel programmer, which could be simplified by shape theory.

These include: task partitioning; data partitioning; message passing; data partitioning; and synchronisation. (There is also the complexity of managing many communicating parallel tasks. However, a typical MIMD program is written in a SPMD style where most processors are executing approximately the same code, so this is not as critical [23].) In performing these tasks, the programmer must ensure that every array access is in bounds, both locally (i.e., for the arrays in a processor’s memory) and globally; the number of *sends* and *receives* of messages match up; individual sends and receives match; each particular data structure is the correct size; the correct amount of data is packed into and unpacked from messages; all the data structures are distributed and reassembled correctly; and that all code optimisations applied [20, 1, 9, 31] are

done in a semantically correct way. Beyond such questions of correctness, there is the pervasive issue of efficiency, in which the goal is to minimise communication costs while maintaining a proper load balance. Many of these are issues of “shape”.

Parallelising compilers handle most of these issues automatically, as in [31, 1]. However, for many scientific applications, these compilers are unable to utilise the specific features of individual algorithms, and a knowledgeable programmer can produce better parallel code than a parallelising compiler [6].

Many researchers [16, 10, 27, 24] argue that using functional programming languages to program massively parallel computers addresses the abovementioned issues. Unfortunately, these languages do not have the features required by some scientific users. Other researchers suggest using a more synchronised and restricted programming model. One possibility is using HPF which works in a SPMD fashion using a shared memory model [19]. However, the programmer must rely on the compiler to produce good code. Another possibility is a data parallel language [3, 8, 28]. One further possibility for easing the difficulties of parallel programming is *skeletons*. Skeletons provide efficient implementations of common computational patterns (second order functions). However, work on skeletons is still in its infancy [4].

Ultimately, however, many programmers of distributed memory MIMD computers opt for using Fortran or C with some message passing library [19, 18], both to use the vast amount of existing scientific routines in these languages [5], and to get the best performance from parallel machines. Unfortunately, these low level languages currently lack some of the features that are now available in higher level sequential programming languages, such as strong static type systems and the associated type safety.

Generally, there are few type structures or verification rules to assist with the problems of programming high performance computers. Shape theory aims to provide the required support: types, verification, and analysis. This paper investigates the explicit use of shape to alleviate these problems.

1.2 Shape Theory

Shape theory (or just *shape*) refers collectively to *shape polymorphism* and *shape analysis*, and their underlying basis of *shapely types* [15, 12]. A data type is shapely if it can be separated into two components, one representing its shape, with “holes” for the data, the other representing the data to be stored in those holes. Shape theory aims to exploit this separation of shape and data.

Shape polymorphism allows a common syntax and algorithm for different instances of polymorphic functions over different shapely data types. For example, the operation of mapping a function over all the elements of a list, or over all the elements stored in a tree is an instance of a shape polymorphic function.

Shape Analysis [13] deals with error detection and optimisation¹ based on the shapes of values. There are two phases to shape analysis, *static* and *dynamic* shape analysis, occurring at compile-time and run-time, respectively.

The rest of the paper is organised as follows. Section 2 gives a description of the version of Cholesky decomposition implemented. Section 3 describes shapely data types, and gives a computational interpretation of them. Section 4 discusses shapely operations. Section 5 describes shape analysis indicating its application to parallel computing. Section 6 contains the conclusions, followed by future directions in Section 7.

2 Cholesky Decomposition

Cholesky decomposition takes a symmetric positive definite matrix \mathbf{A} and produces a lower triangular matrix \mathbf{L} such that $\mathbf{L}\mathbf{L}^T = \mathbf{A}$.

The scheme described here for performing Cholesky decomposition is called *push from behind* and can be found in [7] (and the references cited therein). The algorithm takes an n by n lower triangular matrix (indexed $0, \dots, n - 1$) and proceeds as in figure 1(a).

The implementation of this algorithm is as follows. The matrix is partitioned into blocks on the host, and these are distributed across the processors. The algorithm is then performed on and with these blocks, the blocks are then redistributed among processors so that further computations can then be performed on the new blocks in *local* memory. The process continues until a result is obtained. Finally, there is a phase which involves gathering the final distributed data and restoring the resulting matrix. The selection of the block size is done to maximise efficiency by considering factors such as cache size, load balancing, and the amount message passing. Breaking the data into blocks requires all the loops in the program to be transformed to iterate over these blocks. This is essentially the difference between algorithms 1(a) and 1(b).

Although the algorithm in figure 1(a) parallelises trivially, the granularity of computation is too small to make it efficient. This is solved by splitting the matrix into blocks and giving the recurrence in terms of blocks. This recurrence is given in figure 1(b) where each of the primitive operations in figure 1(a) is replaced by a block operation. This algorithm works for an n by n block lower triangular matrix (indexed $0, \dots, n - 1$).

The rows of the matrix are mapped to processors using *reflection* mapping [7]. This is described as:

Divide the rows into kp groups of rows, k is an integer and p is the number of processors. Groups $0, \dots, n - 1$ are distributed over nodes $0, \dots, p - 1$,

¹The word “optimisation” will always refer to the optimisations performed by compilers, not its mathematical sense.

for columns $j = 0, \dots, n - 1$

$$l_{jj} = \sqrt{l_{jj}}$$

$$l_{ij} = \frac{l_{ij}}{l_{jj}} \quad (i > j)$$

for $k = j + 1, \dots, n - 1$

$$l_{kk} = l_{kk} - l_{kj}^2$$

$$l_{ik} = l_{ik} - l_{ij}l_{kj} \quad (i > k)$$

(a)

for block column $j = 0, \dots, n - 1$

L_{jj} = Cholesky decomposition of L_{jj}

Broadcast L_{jj}

$$L_{ij} = L_{ij}(L_{jj}^T)^{-1} \quad (i > j)$$

Broadcast L_{ij}

for $k = j + 1, \dots, n - 1$

$$L_{kk} = L_{kk} - L_{kj}L_{kj}^T$$

$$L_{ik} = L_{ik} - L_{ij}L_{kj}^T \quad (i > k)$$

(b)

Figure 1: Sequential and Parallel Block Cholesky Decomposition

groups $p, \dots, 2p - 1$ are distributed over nodes $p - 1, \dots, 0$, etc. If the rows do not divide evenly then the extra rows can either be shared among processors or, preferably, stored in the first block.

One advantage of this scheme is that it distributes the workload on processors more evenly than a simple block or even block-cyclic mapping scheme would. Note that it is difficult to express such a mapping scheme in a parallel language, such as HPF, C*, Nesl, or Data-Parallel C [19, 29, 2, 8].

The algorithm was implemented on the Fujitsu-AP1000, a distributed memory MIMD machine, in C using the message passing library supplied with that machine.

3 Shapely Data Types

A shapely data type is a data type that can be separated into two components, the shape and the data. In [12], the data type is defined as the shape/data pairs having the condition that the number of holes in the shape equals the number of data elements.

From the semantics of shape, we can derive the computational characterisation of a shapely data type as:

- a data type constructor, F , which takes a type, A and returns another type, FA
- a type of the shapes of F , represented as $F1$
- a function to extract the shape, $\# : FA \rightarrow F1$
- a function to extract the data from a data type, $\delta : FA \rightarrow LA$, where L represents lists

- a partial function for reconstructing the data type, given a shape and a data list, `reconstruct` : $F1 \times LA \hookrightarrow FA$, and
- a set of constraints that determine when the data type is valid, `valid` : $F1 \times LA \rightarrow \text{Bool}$. (In the semantics these constraints determine F .)

Thus a shapely data type is given by a 6-tuple, $(F, F1, \#, \delta, \text{reconstruct}, \text{valid})$. The following relationships between these operations hold:

- `reconstruct` $\circ \langle \#, \delta \rangle = \text{id}$.
- if `valid` then $\langle \#, \delta \rangle \circ \text{reconstruct} = \text{id}$.

That is, pulling apart a data structure and reconstructing it gives the same data structure, and, given a *valid* shape/data pair, reconstructing the data type, followed by pulling it apart gives the same shape/data pair. The predicate `valid` and the function `reconstruct` are intimately related. The domain of the partial function `reconstruct` is determined by `valid`.

Supplying the 6-tuple for each shapely data type provides a safe and modular way of passing complex data structures in messages (see section 3.2), and provides a helpful new debugging aid (see section 5).

3.1 Examples of Shapely Data Types

The following data structures are used in the implementation of Cholesky: lists, matrices, lower triangular matrices, row of blocks, and lower triangular block matrices. The shape of these data structures and the constraints on the shape and the amount of data are discussed below. All of the checking functions described are easily implemented.

Lists Lists are a primitive data type (in the semantics of shape and in functional programming languages). As such, there are no constraints on them, and the `reconstruct` function is the identity for lists.

Matrices The shape of a matrix is a pair of numbers representing the number of rows and columns. The constraint on matrices is that the number of entries in the matrix is the product of the number of rows and columns.

Lower Triangular Matrices These have shape given by a single number, the number of rows/columns, n . The constraint on a lower triangular matrix is that the number of entries is equal to $\frac{1}{2}n(n + 1)$. Reconstruction of lower triangular matrices is done by the routine `ltmatrix_rebuild()` in section 3.2.

Row of blocks This is a more interesting shapely data type (figure 2). It consists of a list of blocks, zero or more being rectangular matrices followed by a lower triangular matrix. There is the constraint that the number of rows in each of the matrices is the same.

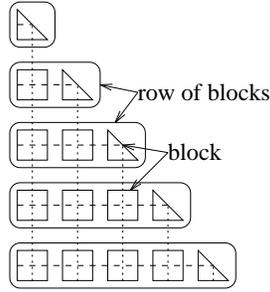


Figure 2: Lower Triangular Block Matrices

Lower Triangular Block Matrices A lower triangular block matrix (figure 2) consists of a list of rows of blocks with some constraints on the number of elements in each row, and on the number of columns in each block. The shape of the lower triangular block matrix is given by a list of numbers, where the length of this list is equal to the number of block row/columns and the elements of the list are the size of the diagonal blocks. The constraints for a lower triangular block matrix are:

- correct number of blocks in each row. One in the first row, two in the second row, etc.
- each row is a valid row of blocks data type
- the number of columns in the blocks in each column are the same (and are the same as that specified in the corresponding element of the shape)
- the number of entries is correct. This is given by the formula $\frac{1}{2}n(n + 1)$ where n is the sum of the numbers in the shape.

3.2 Shape/Data Separation and Data Type Reconstruction

The ability to separate a data type into its shape and its data, and then reconstruct it, has useful applications in parallel computing. The first application is in message passing. Complicated data structures, such as those involving pointers, can be passed efficiently in messages if the shape of the data structure can be stored in the message, along with the data, and a method for rebuilding is present. Secondly, shape/data separation also allows vector operations to be used on more complex data structures. The data can be extracted from a data type, an efficient vector operation can be applied to the data, and then the reconstruction function used to rebuild the result. Finally, the shape/data separation may also have uses in data distribution, e.g., see Nesl [2].

In the parallel implementation of Cholesky decomposition, the data structure representing lower triangular matrices provides a perfect example of a shapely data type.

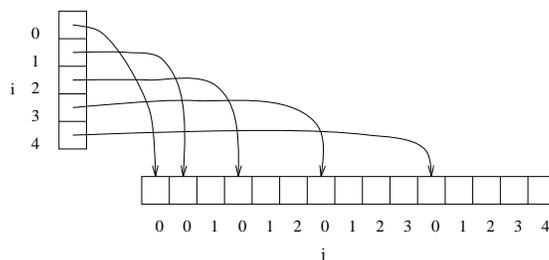


Figure 3: Lower Triangular Matrices in C

The elements of the matrix are stored by rows in an array. An array of pointers is constructed so that the i th element of the pointer array points to the start of the data for the i th row of the data². A 5×5 lower triangular matrix is depicted in figure 3.

This data type cannot be sent in a message as is, as doing so would invalidate the pointer array. The problem is solved by packing the shape and data into a message, and then rebuilding the data type upon receipt of the message. In the implementation, the function `send_row()` applies shape/data separation. This function packs a row of a matrix (the data) and its shape (a number) into a message and sends it from the host to a node:

```
void send_row(int dest, int size, float *data)
{
    int arity = size * (size + 1) / 2;          /* amount of data */
    int m_size = sizeof(int)+arity*sizeof(float); /* size of message */
    char *msg = (char *) malloc(m_size);
                /* start of data part in message */
    float *out = (float *) ((int *)msg + 1);

    memcpy(msg, size, sizeof(int));             /* copy size */
    memcpy(out, data, arity * sizeof(float));   /* copy data */
    l_aseg(dest, TID, ROWDAT, msg, size);      /* send message */
}
```

Upon receipt of a message, the following code for reconstructing the lower triangular matrix data type is executed. (This code has been adapted from the code in [21].) The reconstruction involves building the list of pointers to access the data in rows.

```
float **ltrimatrix_rebuild(float *data, int size)
{
```

²The duality between arrays and pointers in C allows the element m_{ij} to be accessed using the syntax `m[i][j]`.

```

int i;

float **m = (float **) (malloc(size * sizeof(float*)));

m[0] = data;
for (i = 1; i < size; i++) {
    m[i] = m[i-1] + i;
}
return m;
}

```

This code implements the appropriate `reconstruct` operation.

Employing the shape data separation in the systematic way suggested here will ease the implementation of parallel programs and their debugging. The routines for packing and unpacking messages can be written independently of the rest of the code, and can be stored in a library of shapely data types.

4 Shapely Operations

Some operations allow the shape of the output to be calculated from the shape of the input, without knowing the value of the input data. These are called *shapely operations*. They are given by operations $f : FA \rightarrow GB$ between shapely types, for which there is an operation $u : F1 \rightarrow G1$, called the *output shape function*, such that the following diagram commutes

$$\begin{array}{ccc}
 FA & \xrightarrow{f} & GB \\
 \# \downarrow & & \downarrow \# \\
 F1 & \xrightarrow{u} & G1 .
 \end{array}$$

That is, the shape $f;\#$ of the output is given by $\#;u$ and so is determined by the shape of the input.

If a program is composed of shapely operations, then all of the intermediate shapes that occur, as well as the shape of the result, can be determined from the shape of the input. This information can be used for shape analysis (section 6) before looking at the data, i.e., before any computation has begun.

4.1 Scattering and Gathering

Two common shapely operations in parallel computing are scattering and gathering of data to and from the host processor. *Scattering* is the process of partitioning a data structure and distributing it across processors. *Gathering* is its inverse operation.

The routine to do the Cholesky scatter breaks the matrix into blocks and uses the reflection mapping to distribute these across processors. All the calculations in this procedure are based on the three shape parameters input to the program (the size of the matrix, the number of processors and the number of blocks per processor). As scattering is a shapely operation, the output shape function can be used to check that the correct amount of data is delivered to each processor.

In the gathering operation the result matrix is rebuilt on the host using blocks returned from the nodes. A key characteristic of the gather operation is that the order in which the blocks are received from the nodes is nondeterministic. This implies that the shape of an individual message cannot be known statically. However, with a limited amount of extra information it is easy to check that each message received has the right shape. Once we have determined where a block comes in the original matrix it can be stored there. We can statically determine the number of blocks that are expected based on the shape of the matrix, and use this number to assist with debugging.

4.2 Global and Local Views

Many data types, particularly those used in numerical computing (e.g., a matrix), can be indexed. Indexing of shapely data types [14] captures interesting operations common in parallel computing.

If the data structure has some form of indexing in its original and scattered forms, then the different indexing schemes provide two different views of a data structure. The *global view* is the indexing scheme for the data structure before it has been scattered. A *local view* on a processor is the indexing scheme for the part of the scattered data structure present on that processor. In parallel programming, it is important to be able to convert between the different views, this is achieved using *index transformations*.

In Cholesky decomposition the input data is a lower triangular matrix. The global view of this is a lower triangular block matrix, where indexing this gives the blocks. Each processor receives a number of rows of this matrix. This means that each local view is a list of row blocks. The (i, j) th element in the local view does not normally correspond to the (i, j) th element in the global view. We must, therefore, convert between local and global indexing schemes to access the correct elements as required.

For processor p , the following mapping between indices occurred:

$$\begin{aligned} \text{owner}(i, j) &= (j - p) \bmod (2 \text{ NPROCS}) = 0 \vee (j + p + 1) \bmod (2 \text{ NPROCS}) = 0 \\ \text{local}(i, j) &= \text{if } \text{owner}(i, j) \text{ then } (i, \lfloor j / \text{NPROCS} \rfloor) \text{ else } - \end{aligned}$$

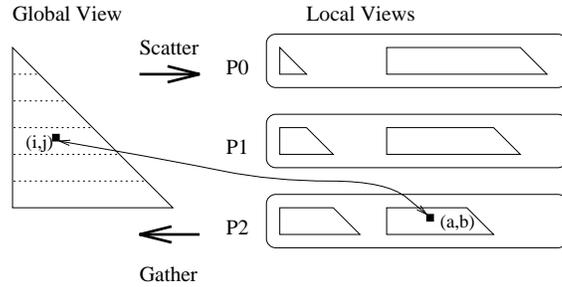


Figure 4: Scattering and Gathering, Global and Local Views

$$\text{global}(i, j) = (i, \text{if } j \bmod 2 = 0 \text{ then } 2j \text{ NPROCS} + p \text{ else } 2j \text{ NPROCS} - p - 1)$$

These were used in the parallel implementation of Cholesky decomposition in three kinds of situations relating to the different views:

- The main loop of Cholesky decomposition iterates sequentially over the columns of the matrix (using the global view). The diagonal element is used to update all the other elements in a column. The function `owner(i, i)` is used by each processor to determine whether that processor has the diagonal element, (i, i) . The result is calculated using index transformations.
- To access an element (such as (i, i) above) a processor needs to perform an index transformation from the global indices to the local indices, using the function `local(i, i)`. This is a partial operation because not all processors have access to all elements.
- One of the update operations applies to all rows/elements of a matrix/column that are below the main diagonal. This means that we could iterate through all the local elements, convert the coordinates to global ones, check that it is below the main diagonal, and then perform the required operation. The function `global(i, j)` is used to perform the task.

The relationships between scattering, gathering, and global and local views are depicted in figure 4. The local view illustrates the reflection mapping.

5 Shape Analysis

Shape analysis is concerned with using shape to detect errors (called *shape checking*), and for optimisation. Some operations have constraints on the shapes of their input, e.g., the multiplication of two matrices requires that the number of columns in the first matrix is the same as the number of rows in the second matrix. Shape checking ensures

that these operations are used on consistent data. Computationally, we have a partial function $f : FA \hookrightarrow GB$, constraints on the shape of the domain, $\text{con} : F1 \rightarrow \text{Bool}$, and, when f is a shapely operation, the output shape function $u : F1 \rightarrow G1$. Thus, a shapely operation is given by a triple (f, con, u) and a non-shapely operation is given by a pair (f, con) .

The constraints on certain operations provide a method for detecting the shape errors in programs. Shapely operations give a method for propagating such constraints throughout a program. We can then check the system of constraints for validity at compile time, or as soon as the shapes of the input data are known, for early error detection at run time.

It is not necessary for the programmer to supply all the shape constraints. The prototype shape analyser, LISA [26], has been developed for a parallel programming language based on Nesl [2]. It generates constraints automatically for programs and performs appropriate simplification, giving shape errors when an inconsistency is detected. Shape analysers, such as LISA, use existing tools and techniques to simplify generated constraints [22, 17, 25, 30].

There are two extreme approaches to shape analysis. Firstly, all the constraints are manually entered by the programmer. Alternatively, all constraints could be generated by the compiler, as in LISA. Regardless of the method for generating constraints, they will never be complete, due to the usual computability reasons. Evaluation of the trade-offs between the different approaches — computability, complexity, extent of error checking, and whether the onus should be on the compiler or the programmer — is a topic for future research.

5.1 Other Forms of Shape Checking

Other forms of shape checking can be performed.

Correctly Implemented Data Types Some data types are difficult to implement. Constraints on the shapes of these data types can be defined as a simple check for the validity of the implementation. An important use of this is when data is distributed across the processors of a parallel computer. In this case, data would start on the host processor. It would then be partitioned and sent to the node processors. Each node will then execute the function $\text{valid} : F1 \times LA \rightarrow \text{Bool}$ to determine whether or not it received the correct data.

Correct Indexing One area of shape checking is correct indexing or bounds checking. Every access to a matrix must be within the bounds for which the matrix is defined. Some languages (e.g. Modula-2, Pascal) can give a run-time error for out-of-bound access (compiler-dependent), whereas other languages such as C [18] often give errors

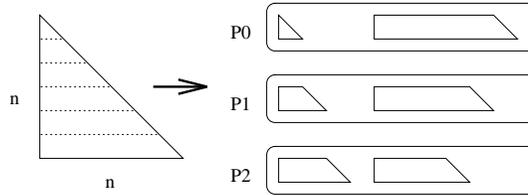


Figure 5: Partitioning and distribution of a lower triangular matrix across 3 processors using the reflection mapping scheme

only when a memory violation occurs. If arrays are defined so that the bounds on their dimensions are enforced then shape analysis will aid in the detection of indexing errors at compile time.

Indexing is more complicated in parallel computing. For example, a matrix may be distributed across processors and stored in the local memory of processors. Sometimes the matrix may need to be accessed using global indexing (i.e., indices from the original matrix), or using local indexing (i.e., indices into the local memory of a processor). Both sets of indexing must be correct. The constraints on this type of indexing are that (1) a local index must be in the bounds of the local array, and (2) the global index must be in the bounds of the global matrix, and the corresponding distributed element must be in the local memory of the processor.

Correctly Implemented Shapely Operations The compiler can insert code to test that the shape of the output of a shapely operation is the same as the shape given by the output shape function, u . Typically, the function to compute the output shapes will be a lot simpler to implement than the function that does the computation.

5.2 Examples of Shape Checking

Appart from the constraints on the shapely data types given in section 3, there were other places where shape analysis was used.

Shapely Operations Partitioning and distributing a lower triangular matrix is one of the most complicated operations in the implementation. Its input is a lower triangular matrix of size n . The output is a list of list of row blocks. Figure 5 illustrates this operation.

When each processor receives its parts of the lower triangular matrix, the output shape function can be executed to check that the data received has the correct shape, i.e., that all the data has been received.

Constraints in the Implementation Most of the constraints on the shapes stemmed from three parameters. They are the number of node processors, the size of the input matrix and the number of row blocks of the matrix per processor.

```
#define NPROCS 64, SIZE 128, BLK 2
```

These parameters were used in the following constraints:

- the size of the input lower triangular matrix was `SIZE`
- when the input matrix was partitioned into a list to be distributed across processors, the number of elements in the list was `NPROCS`
- the number of row blocks that were allocated to each processor was `BLK`.

Correct Dimensions and Indexing There are four routines to perform the factorisation steps of the Cholesky algorithm. These are quite similar (standard sequential matrix code) so just one is given.

The routine, `update(a,b,c)` (figure 6) performs the operation $a \leftarrow a - bc^T$, for matrices a , b , and c . We can apply shape analysis [25] to the routine and derive a set of constraints that the inputs must obey.

Let r_a, c_a, r_b, c_b, r_c and c_c represent the number of rows and columns of the inputs. Constraints produced³ are given in figure 6. The constraints on line 2 are constraints from the definition of the operation. These can be supplied by the programmer. On line 11 the local variables `r`, `col`, and `cc` are initialised to be the number of rows in a , the number of columns in a , and the number of columns in b . The constraints adjacent to line 11 reflect those assignments. The constraint on line 13 gives the initialisation of the shape of the matrix `temp` as (r, col) . The loop surrounding the statement in line 16 has bounds `r` and `col`. This suggest that the shape of the matrix, `temp`, should have dimensions of at least that size to allow correct indexing. More complicated examples of this are in lines 25 and 33. At compile-time these constraints simplify to $r_a = r_b \wedge c_a = r_c \wedge c_b = c_c$, i.e., there are no shape errors in the code. The shape of the inputs must satisfy this constraint at run-time to be valid. Note that these constraints were generated and solved by hand.

5.3 The Shapely Computational Paradigm

Automation of shape analysis requires a new computational paradigm, as proposed in [13]. In addition to compilation of sequential code, there are four phases to computation: static shape analysis, dynamic shape analysis, dynamic optimisation, and execution. The relationship between these is given in figure 7.

³Assuming array accesses are all intended to be in bounds.

```

(1) void update(struct BLOCK *a, struct BLOCK *b, struct BLOCK *c)
(2) /* does   a ← a - b * cT */           {ra = rb, ca = rc, cb = cc}
(3) {
(4)   float **temp;
(5)   int i,j,k;
(6)   int r,col,cc;
(7)   float **am, **bm, **cm;
(8)
(9)   /* initialize locals */
(10)  am = a->m; bm = b->m; cm = c->m;
(11)  r = a->r; col = a->c; cc = b->c;           {r=ra, col=ca, cc=cb}
(12)  /* temp = new zero matrix */
(13)  temp = matrix(r,col);                   {rtemp = r, ctemp = col}
(14)  for (i = 0; i < r; i++) {
(15)    for (j = 0; j < col; j++) {
(16)      temp[i][j] = 0;                     {rtemp ≥ r, ctemp ≥ col}
(17)    }
(18)  }
(19)
(20)  /* perform temp ← b * cT */
(21)  for (i = 0; i < r; i++) {
(22)    for (j = 0; j < col; j++) {
(23)      for (k = 0; k < cc; k++) {
(24)        temp[i][j] += bm[i][k] * cm[j][k]; /* remember cT */
(25)        {rtemp ≥ r, ctemp ≥ col, rb ≥ r, cb ≥ cc, rc ≥ col, cc ≥ cc}
(26)      }
(27)    }
(28)  }
(29)
(30)  /* perform a ← a - temp */
(31)  for (i = 0; i < r; i++) {
(32)    for (j = 0; j < col; j++) {
(33)      am[i][j] -= temp[i][j]; {ra ≥ r, ca ≥ col, rtemp ≥ r, ctemp ≥ col}
(34)    }
(35)  }
(36) }

```

Figure 6: Shape analysis of the routine `update()`

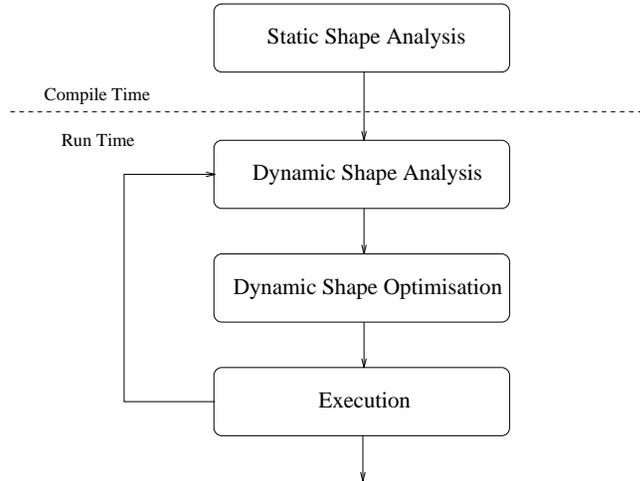


Figure 7: The Shape Computational Paradigm

The static shape analysis phase takes the program source and derives constraints from it (for example, as discussed in section 5.1). The constraints are simplified, where possible. Inconsistent constraints indicate that there is an error in the code. Satisfiable constraints require run-time checks to be inserted, whereas consistent constraints require no run-time checks in the code. Any remaining constraints are then delayed to be checked during the dynamic shape checking phase.

At run-time, the constraints on the shapes of the inputs are checked. An error is given if these constraints are not satisfied. Computation then proceeds through to the execution phase⁴. At some point there will be a constraint that needs to be checked, possibly because the shape of some variable becomes known. This point in the execution is called a *reshaping point*. Control then returns to the dynamic shape analysis phase and constraints on the new shape are then checked.

The motivation of the computational paradigm is to provide an environment where the constraints generated by static shape analysis are checked as early as possible to avoid wasting time with long erroneous computations.

6 Conclusions

Shape was prominent in the implementation of parallel Cholesky decomposition and used in a substantial amount of the code. Shape theory is useful in describing data structures, in describing the process of packing and unpacking messages, in giving transformations between data structures. Shape theory also provides a basis for spec-

⁴There is also a phase for dynamic optimisation, but the details of this are beyond the scope of this paper.

ifying constraints on the shapes of the inputs. These constraints can be checked in the compiler or be delayed until run-time, either way providing better information for debugging. Shape theory, and tools and techniques based on it, will make parallel programming easier.

The concepts applied here can be adapted easily to any dense linear algebra with little change. However, it should also be easy to apply these ideas in more general parallel computing.

7 Future Work

We are committed to developing the shapely computational paradigm, and the theoretical tools necessary to support it. The issues include:

- definition of language constructs for describing shapely types
- the determination of the level of programmer intervention in the generation of constraints
- optimisations available for shapely operations
- the use of shape theory to keep track of the number of messages sent
- the formalisation and evaluation of the computational paradigm
- frameworks for reasoning about/transforming/simplifying constraints
- the consistent incorporation of data distribution, views and indexing schemes into the theory.

Acknowledgements

We would like to thank the ANU-Fujitsu CAP project for the use of their machine, the Fujitsu-AP1000. We would also like to thank members of the Algorithms and Languages Group for their constant support and constructive criticism.

References

- [1] David F. Bacon, Susan L. Graham, and Oliver S. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

- [2] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.
- [3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.
- [4] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe*, June 1993.
- [5] Jack Dongarra and David Walker. Libraries for linear algebra. In Gary Sabot, editor, *High Performance Computing*, chapter 4, pages 94–134. Addison-Wesley, 1995.
- [6] Kelvin K. Droegemeier, Ming Xue, Kenneth Johnson, Matthew O’Keefe, Aaron Sawdey, Gary W. Sabot, Skef Wholey, Kim Mills, and Neng-Tan Lin. Weather prediction: A scalable storm model. In Gary W. Sabot, editor, *High Performance Computing*, chapter 3, pages 45–92. Addison-Wesley, 1995.
- [7] Jenny Edwards and J. A. Tomlin. Parallel Cholesky Factorisation. In *Proceedings of the Fifth Australian Supercomputing Conference*, 1992.
- [8] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.
- [9] Efthymios C. Housos, Chih Chung Huang, and Jun-Min Liu. Parallel Algorithms for the AT&T KORBX System. *AT&T Technical Journal*, 68(2):37–47, May/June 1989.
- [10] P. Hudak. Para-functional programming in Haskell. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 159–196. ACM Press Frontier Series, 1991.
- [11] C.B. Jay. Matrices, Monads and the Fast Fourier Transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80, 1994.
- [12] C.B. Jay. A Semantics for Shape. *Science of Computer Programming*, in press, 1995.
- [13] C.B. Jay. Shape analysis for parallel computing. In *Parallel Computing Workshop ’95 at Fujitsu Parallel Computing Centre, Imperial College*, 1995.
- [14] C.B. Jay. Data categories. *Computing: The Australian Theory Seminar*, to appear, 1996.

- [15] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science, pages 302–316. Springer-Verlag, 1994.
- [16] S. L. Peyton Jones. Parallel implementations of functional languages. *The Computer Journal*, 32(2):175–186, 1989.
- [17] Wayne Kelley, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wannacott. *The Omega Calculator, version 0.8*. University of Maryland, August 1994.
- [18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.
- [19] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Massachusetts, 1994.
- [20] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, 1994.
- [21] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.
- [22] W. Pugh and D. Wonnacott. Eliminating false dependencies using the Omega Test. *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 140–151, June 1992.
- [23] Gary W. Sabot, editor. *High Performance Computing*. Addison-Wesley, 1995.
- [24] Wolfgang Schreiner. Parallel functional programming: An annotated bibliography. Available from author (schreine@risc.uni-linz.ac.at), December 1993.
- [25] Milan Sekanina. Personal Communication.
- [26] Milan Sekanina. Shape analysis of a Nesl-like language. In preparation.
- [27] S.K. Skedzielewski. Sisal. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–158. ACM Press Frontier Series, 1991.
- [28] David Skillicorn. *Foundations of Parallel Programming*. Number 6 in Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

- [29] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, May 1993.
- [30] Stephan Wolfram. *Mathematica*. Addison-Wesley, 1988.
- [31] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Machines*. ACM Press, New York, NY., 1990.