High-level Specification Tools for Parallel Application Development†

Edwin de Jong[3], Edwin M. Paalvast[1], Henk J. Sips[2], Maarten R. van Steen[1]
[1] TNO Inst. of Appl. Comp. Sc., POB 6032, NL-2600 JA Delft
[2] Delft Univ. of Technology, Dept. of Appl. Physics, POB 5046, NL-2600 GA, Delft
[3] Univ.of Leiden, Dept. of Comp. Sc., POB 9512, NL-2300 RA, Leiden

Abstract

Development of parallel applications has been seriously hampered due to the intricacy of the underlying programming model in which hardware architectural features have to be taken into account in order to ensure efficiency. The standard approach towards parallel program development has, in addition, resulted in applications which are difficult to port to different parallel computers. The parTool project tackles these problems by aiming at a parallel programming system in which high-level specification tools play a prominent role. This paper outlines two components of the parTool system: the transaction-based programming language Vista, and the data-parallel language Booster.

Introduction

Parallel processing, for years being an academic curiosity, has now been brought to the fore front of commercial innovation. Parallel machines are readily available to the users community, promising a significant speed up over the use of a single processor to solve a problem. However, the additional complexity inherent to parallel processing generally makes it hard to achieve this objective as a parallel software developer is confronted with a programming model in which aspects such as processor-memory pairing, network topologies, and ways of synchronization and communication have a significant influence on the software design process.

In this paper we present an overview of the approach followed by the *parTool* project in developing a parallel programming system. The key feature of *parTool* is a separation of algorithm specifications and the allocation of hardware resources to data and computations. Algorithms are formulated at an abstract level in a specification language having its own ideal virtual machine, thus preserving the parallelism inherent to the algorithm. Mapping the algorithm onto a specific target machine, i.e., allocating resources to data and computations, is done by adding annotations to the description of the algorithm.

Using separate mapping information, programs can be generated for a variety of parallel machines including shared-memory- and distributed-memory machines. As each machine may require a different resource allocation strategy porting a program from one machine to another is done by merely changing the mapping annotations. The principle is illustrated in Figure 1.

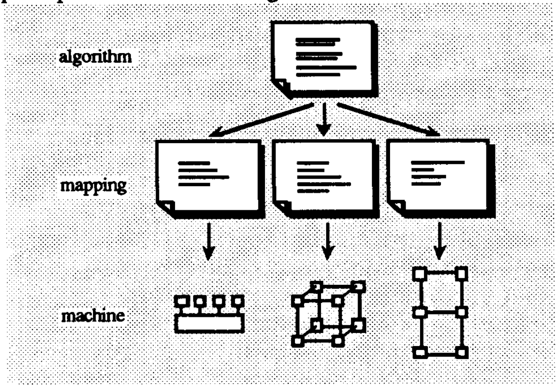

Figure 1. Separation of concerns in parallel program development.

In the following two sections we discuss two prominent components of *parTool*: the transaction-based programming language Vista, and the data-parallel language Booster. The last section concludes with some final remarks and indications for further research.

Transaction-based programming in Vista

The Vista language provides a transaction-based model of programming, which makes it particularly suited for symbolic computations. Transaction-based parallel programming is founded on *generative communication* as originally coined in the Linda programming language [2,4]. Using generative communication, data is located in a *global* yet logically distributed space and is addressed by

† This reaserach is sponsored by SPIN

content rather than by explicit naming.

Conventionally, a parallel program consists of a number of sequential processes, that concurrently retrieve and insert data. In Vista, generative communication is extended with *transaction-based programming*. This means that a program consists of a collection of concurrently executing transactions, i.e., query-action pairs, that iteratively update the data without mutual interference.

Transaction-based programming in combination with generative communication yields highly parallel programs, and by absence of explicit communication patterns, it considerably eases the parallel programming task as shown in [11].

Data representation. Fundamental in writing Vista programs is the form in which data is represented. In Vista, data is represented by a global space of *tuples*. A tuple is simply a sequence of values, e.g. ("pi", 3.14) or ("Mary", female, 34). The first tuple consists of the string "pi" followed by the numeric value 3.14. Likewise, the second tuple consists of the string "Mary", the literal *female*, and the numeric value 34. To manage the diversity of tuples, every tuple is associated with a *descriptor*. For instance, the following descriptor *constant* can be used as a type definition of the tuple ("pi", 3.14).

```
descriptor constant {
   string name;
   float value;
}
```

Transactions. In transaction-based programming tuples are inserted and deleted by a set of transactions. In general, a transaction consists of a *query* part and an *action* part. The query part makes a selection of tuples, while the action part deletes and inserts tuples.

The query part of a transaction selects a number of tuples that satisfy some condition. For instance, the value for π can be retrieved by the query

```
select constant(c,v) where (c == "pi");
```

The query searches for a tuple of type *constant*, where the first field contains the value "pi." If a matching tuple is found, the value of the second field is returned in the variable *v*. In general, it is possible that more than one tuple can be found that matches the query. In that case, a selection is made at random.

The action part of a transaction deletes and inserts tuples. By default, every tuple selected in the query part is deleted by the action part. The default can be overridden by preceding a tuple in the query part with the marker '?'. The following query, for instance, does not delete the selected value for π. Instead, it merely copies the value into variable *v*.

```
select ?constant(c,v) where (c == "pi");
```

Tuples that are to be inserted must be specified in the action part explicitly. As an example, consider the following query-action pair.

```
select constant(c,v)
     where (c == "pi" && v <= 3.14);
insert constant(c, 3.1416);
```

The query-action pair selects a value for π less than or equal to 3.14 and replaces it by the more accurate value 3.1416.

A transaction consists of a query-action pair and the required variable declarations. A complete transaction for replacing the value of π can be specified as follows.

```
transaction replace { string c; float v;
   select constant(c,v)
        where (c == "pi" && v <= 3.14);
   insert constant(c, 3.1416);
}
```

Programs. A *program* is constructed from a number of transactions and a description of the initial tuples. The tuple space is partitioned into a collection of *stores*, where each store contains zero or more tuples. Program execution commences from the initial tuples and proceeds by selecting non-deterministically a transaction in the program which query part can be satisfied. The transaction is then executed by making a selection of tuples such that the query succeeds and subsequently deleting and inserting tuples in accordance to the action part. Execution continues while there are transactions which query part can be satisfied. Otherwise, program execution terminates.

To illustrate, we consider a program consisting of the previous transaction *replace* and the initial tuple ("pi", 3.14). In the program, the initial tuple is specified to reside at a store *constants* by the declaration

```
store constants { constant("pi", 3.14); }
```

The declaration defines a store initially consisting of the tuple ("pi", 3.14). The program executes transaction *replace* exactly once, after which it terminates leaving the tuple ("pi", 3.1416) in store *constants*.

Parallelism in a program is obtained by allowing multiple transactions to execute at once. An important assumption is that transactions execute without mutual interference. This means that multiple transactions executing at once have exactly the same effect as executing the transactions in an arbitrary sequence.

A programming example

To illustrate, we consider a transaction-based parallel program for solving the single-source shortest path problem. The problem description is as follows.

Consider a directed graph consisting of vertices numbered 0,...,n–1. With every edge a non-negative weight is associated, given by a weight function W defined on each pair of vertices. We assume that $W(u,v)$ is infinite if there is no edge from vertex u to vertex v. Given a source vertex s, the single-source shortest path problem consists of determining the length of a shortest path from s to v, for every vertex v in the graph.

The following program solves the single-source shortest path problem for a graph with source vertex s.

```
descriptor vertex {
   int id;
   float length;
}

store graph {
   for (int v = 0; v < n; v++)
      vertex(v, W(s,v));
}

transaction find[n] (int u, v) {
   float x, y;
   select { ?vertex(u,x); vertex(v,y); }
      where ( y > x + W(u,v));
   insert vertex(v, x + W(u,v));
}
```

The vertices are represented by tuples of type *vertex* residing at a store named *graph*. Initially the store contains, for every vertex v, a tuple (v, $W(s,v)$). That is, for every vertex v, there is a tuple which *length*-field equals the weight of an edge from the source s to v; at termination of the program, the *length*-field will equal the length of a shortest path from s to v.

The shortest paths in the graph are determined by transaction *find*. Intuitively, the query acquires vertices u and v, such that a path from s to v via u yields a shorter length than the path to v found so far. If the query succeeds, the transaction deletes the tuple (v, y) and replaces it by the tuple (v, $x + W(u,v)$). The program terminates if the query part of transaction *find* no longer succeeds. It is easily seen that this situation corresponds to a state where all shortest paths have been found.

Parallelism is obtained by instantiating multiple transactions. In the program, an array of transaction *find* is declared, consisting of as many transactions as there are vertices in the graph. Since transactions are executed concurrently without mutual interference, this allows each vertex in the graph to be examined in parallel.

Mapping Programs to Machines

Transaction-based parallel programs are formulated at a convenient level of abstraction. This allows the program engineer to focus on the specification of a programming solution in which the complex details and particularities of a multiprocessor system can be neglected.

Eventually, however, the program will have to run on a specific system, preferably as efficiently as possible. Dependent on the type of machine, this means that the following issues must be addressed.

1. An efficient schedule must be devised for the transactions in the program (for every type of machine).
2. The schedule must be partitioned among the available processors (for multiprocessor systems).
3. The tuples in the program must be distributed among the available memories (for distributed-memory machines).

Schedules identify a particular order in which transactions are executed. To illustrate, we return to the program in the previous section for which we propose to adopt a depth-first search strategy, which is specified by the following schedule.

```
schedule examine (int u) {
   forall (int v = 0; v < n; v++)
      if ( W(u,v) < ∞ )
         if ( find[v](u,v) ) examine(v);
}
```

Schedule *examine*(u) looks at all outgoing edges (u, v) of a given vertex u. If a shorter path is found to vertex v, by executing transaction *find*[v], then vertex v is examined next. Note that all outgoing edges can be examined in parallel. If no more edges remain to be examined, the schedule terminates. A complete schedule of the program is given by *examine*(s), which starts at the source vertex.

Clearly, the above schedule can be executed on uniprocessor systems. Since modern parallel systems based on shared memory almost all provide automatic load-balancing in hardware, the schedule can be used on this type of machine as well.

For multiprocessor systems based on distributed memory, we must also provide a partitioning of the schedule across the available processors. Assuming a system consisting of p processors, where each processor has exclusive access to a local memory, this can be specified as follows.

```
distribution graph {
   float x;
   for (int v = 0; v < n; v++)
      vertex(v,x) @ (v % p);
}

schedule examine (int u) {
   forall (int v = 0; v < n; v++)
      if ( W(u,v) < ∞ )
         if ( ?find[v](u,v) @ (v % p) )
              examine(v);
}
```

First, we assign each tuple in store *graph* to a dedicated processor. For simplicity, we use a static wrap-around assignment of the vertices to the available memories. In general, the assignment need not be static.

The schedule is partitioned among the available processors by assigning the execution of each transaction to a specific processor. In the example, the schedule is partitioned such that execution of a transaction is dispatched to the processor that owns the vertex to be examined by the transaction.

Data parallelism in Booster

The notion of transactions in Vista makes the language particularly suited for symbolic computations. To facilitate the specification of numerical computations as well, we developed the Booster language as part of the *parTool* project. The language offers its users a high level of abstraction in programming numerical algorithms while maintaining the possibility to generate efficient parallel programs in C or FORTRAN.

Shapes and Views. In a conventional programming language (such as FORTRAN), the array is used as the basic data structure for storing related data elements. These elements can be accessed by use of indices to the array. However, it is not possible in these languages to reason about or manipulate indices themselves. This ability is particularly relevant to parallel programming where it is often important to identify sets of index values that refer to data upon which computations can be executed in parallel. The importance of this has already been recognized in a language like Actus [10].

In Booster, these observations have resulted in a strict distinction between data- and index-domain of a program. The *data-domain* consists of several possible data types, just as in conventional languages. The *index-domain* consists of non-negative integer values. On the index-domain ordered index sets can be defined, and operations can be performed on these sets independent of the data-elements that the index values in question refer to.

There are two concepts in Booster to reflect the two domains. The first is the *shape*, Booster's equivalent of a traditional array. Unlike arrays, shapes need not necessarily be rectangular. Shapes serve, from the algorithm designer's point of view, as the basic placeholders for the algorithm's data.

The second concept is that of the *view*. A view enables the manipulation of the index set of a shape without affecting the actual data in the shape. By changing the original index set one can still access the elements of the shape, but it is as though we now "view" the shape's data-elements through a different window.

The declaration of shapes is fairly straightforward, e.g.

```
SHAPE A (3#10) OF REAL;
```

declares a matrix A of 3 by 10 elements. The index set for this shape is the ordered set {(0,0), (0,1), ... (2,8), (2,9)}.

Content Statements. *Content statements* allow the manipulation of the data stored in shapes:

```
A := 2.5;
A[1,8] := 3.1416;
```

In the first content statement, all elements of A are initialized to 2.5. In the second statement, the value 3.1416 is stored in the element of A with index value (1,8). Content statements are Booster's equivalent of the standard assignment statement in procedural languages. Apart from standard scalar operators, Booster supports their multi-dimensional equivalents. Multi-dimensional operators are defined strictly ***element-wise***, that is the operator is applied to the elements which have the same *ordinal* number[1] and all such operations are applied simultaneously.

View Statements. Index sets are manipulated in Booster by *view statements*. They come in four flavours. The most simple view statement defines the identity view:

```
V <- A;
```

where V is called a *view identifier*. By the statement V<-A, the view identifier V is bound to the index set of the shape A. After the view statement in the code given above, the two content statements below will have the same effect.

```
A[0,0] := A[10,0];
V[0,0] := V[10,0];
```

The following view statement defines a *selection* view:

```
V <- A[0,3:8];
```

The *index expression* 3 : 8 selects the subset or *range* of indices (0,3) through (0,8) of A and binds them to the view identifier V. The element V[0,0] actually refers to A[0,3], etc: *renumbering* of the index sets after a view statement causes all index sets to start from zero, just as the original index set does. A itself is never affected by any view statement.

The second type of view defines a *permutation* view:

```
V[0,i] <- A[0,9-i];
```

In this statement, i denotes a *free variable*, which is bound to the index set of the structure it is applied to. Above, we access, through V, the elements of the first row of A in reverse order.

Free variables can be used for even more powerful purposes, as is illustrated by the third type of view: the *di-*

1 The ordinal number of an element is the sequence number derived from ordering the indices lexicographically.

mension changing view. An example of a *dimension increasing* view is:

```
V{2#5}[i,j] <- A[0,(5*i)+j];
```

The (one-dimensional) first row of A can be accessed through the two-dimensional view identifier V. The relation between the index sets of V and A is defined by the functional relations of the free variables i and j. For all practical purposes, the identifier V now becomes a two dimensional structure. The fact that the index set of this structure refers, *via* a view identifier, to a one-dimensional structure is completely transparent to the "user" of V.

Dimension decreasing views are relatively simple:

```
V <- A[1,_];
```

Here the second row of A is selected and assigned to the view identifier V. The underscore character stands for selecting all elements in that dimension.

Finally, Booster has *content selection views*, in which value dependent selections can be taken on structures.

LU Factorization

To illustrate, we take the well-kown LU factorization algorithm, without pivoting. The algorithm takes a previously declared non-singular *nxn* matrix *A* as input. The algorithm eliminates in successive steps each column, until an upper triangular matrix results. First, we need a *function* for matrix multiplication:

```
FUNCTION xmtrx PRIORITY 7 (A,B) -> (C);
  A,B,C (n#n) OF REAL;
BEGIN
  C[i,j] := REDUCE(+,A[i,_]*B[_,j]);
END;
```

The function is defined according to the element-wise semantics of Booster. REDUCE is the built-in reduction function. The body of the LU-factorization program can now be specified as follows:

```
H <- A;
WHILE size(H) > 1 DO
  R <- H[1:upb,1:upb];
  R := R - H[1:upb,0] xmtrx
           H[0,1:upb]/H[0,0];
  H <- R;
END;
```

In each step, four selections are involved: the pivot element H[0,0], the pivot row H[0,1:upb], the pivot column H[1:upb,0], and the remainder R (see Figure 2). In the initial step, the index set of B is set to the view identifier H. The remaining part of the algorithm is coded as a conditional loop with three statements. The first view statement defines the remainder region and the succeeding content statement performs the actual operation, using the four regions. In the second view statement, the view identifier H is redefined as the remainder R. The algorithm terminates of the size of the view H is equal to 1.

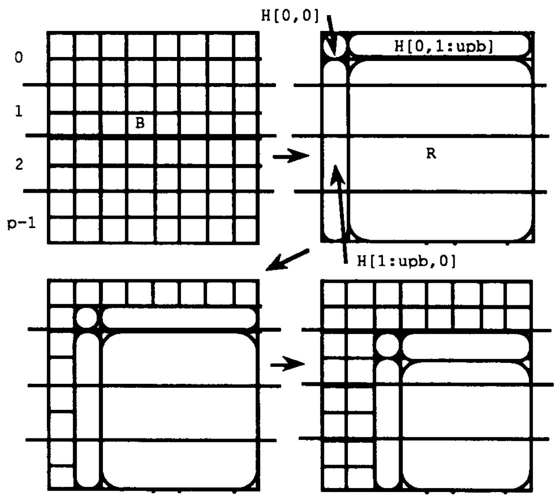

Figure 2. LU-decomposition.

Resource Allocation for Parallel Machines

To discuss Booster's method of resource allocation, let us return to the concepts of shapes and views. In shared-memory processors, the resources needing allocation of processing responsibilities are the processors. In distributed-memory processors also data has to be allocated to the distributed memory. In Booster, both allocations are specified through *computation* and *data organization* annotations. To this aim, the same mechanism is used as provided for index manipulation in the construction of algorithms, namely the *view*. This principle is illustrated in Figure 3.

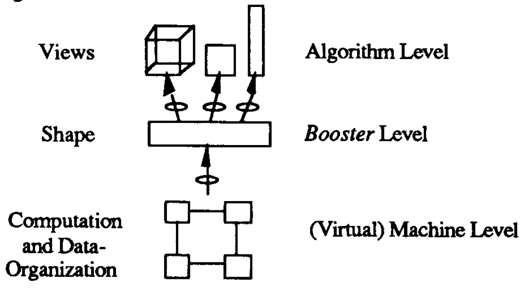

Figure 3. Computation and data organization.

Both allocations can be combined, i.e. an annotation specification defines both processing responsibilities and data assignments. The default assumption is that the processor owning the data, is also responsible for any update of that data, including the associated processing. This technique also applied in FORTRAN and C by [1,3,5,6]. In Booster, the processing and data responsibilities can be defined combined or separately, allowing complicated assignments to be made.

As an example, a two-dimensional shape is decomposed in a row-wise fashion for parallel machine with *p* processors. This is described with the following *dimension decreasing* view function:

```
VIEW FUNCTION row_decompose (R) -> (Q)
  Q (n # n); R (p # (n div p) # n);
BEGIN
  Q[i,j] <-
    R[i div (n div p),i mod (n mod p),j];
END;
```

The principle is illustrated below in Figure 4. The view function concept is used to encapsulate the view *row_decompose*.

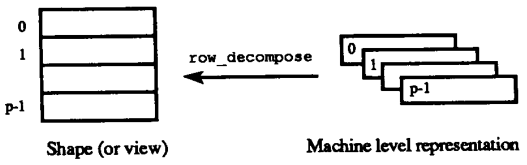

Figure 4. Row-wise decomposition.

This view function can now be used to impose a resource allocation on the LU-factorization algorithm.

```
ANNOTATION MODULE LU;
IMPORT
  R (n#n) FROM LU;
  Proc (p#(n div p)#n) FROM Processor_Model;
  BEGIN
    R <- row_decompose(Proc);
  END.
```

In the annotation module, we import the matrix B from the program module LU and a processor model, which provides a (virtual) machine image. The view *row_decompose* is now used to define both processing- and data responsibility. The decomposition is shown in Figure 4.

A more elaborate introduction to the Booster language and its annotations can be found in [8]. The translation of Booster is discussed in [7,9].

Concluding remarks

We have presented two high-level specification languages in the *parTool* system: the parallel transaction-based language Vista, and the data-parallel language Booster. Currently, a prototype Booster compiler generating C-code is operational and work is in progress to include optimizations in the system. The same compiler will be used to translate Vista programs to C and FORTRAN.

References

[1] F. André, Pazat, J.-L. and Thomas, H., "Pandore: A System to Manage Data Distribution." In *Proceedings 1990 International Conference on Supercomputing*, 1990, pp. 380-388.

[2] N. Carriero and Gelernter, D., "Linda in Context." Communications of the ACM *32*, 4 (1989), pp. 444-458.

[3] G. Fox, Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C. and Wu, M., "Fortran D Language Specification." Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, 1990.

[4] D. Gelernter, "Generative Communication in Linda." ACM Transactions on Programming Languages and Systems *7*, 1 (1985), pp. 80-112.

[5] H.M. Gerndt, "Array Distribution in SUPERB." In *Proceedings Third International Conference on Supercomputing*, 1989, pp. 164-174.

[6] K. Kennedy and Zima, H., "Virtual Shared Memory for Distributed-Memory Machines." In *Proceedings Fourth Hypercube Conference*, 1989,

[7] E.M. Paalvast, Gemund, A.J.v. and Sips, H.J., "A Method for Parallel Program Generation with an Application to the Booster language." In *Proceedings 1990 International Conference on Supercomputing*, 1990, pp. 457-469.

[8] E.M. Paalvast, Sips, H.J. and Breebaart, L.C., "Booster: a high-level language for portable parallel algorithms." Applied Numerical Mathematics 8 (1991), pp. 177-192.

[9] E.M. Paalvast, Sips, H.J. and Gemund, A.J.v., "Automatic Parallel Program Generation and Optimization from Data Decompositions." In *Proceedings 1991 International Conference on Parallel Processing*, 1991, pp. 124-131.

[10] R.H. Perrott, "A Language for Array and Vector Processors." ACM Transactions on Programming Languages *1*, 2 (1979), pp. 177-195.

[11] M.R. van Steen and de Jong, E., "Designing Highly Parallel Applications Using Database Programming Concepts." In *Proceedings First International Conference on Parallel and Distributed Information Systems*, 1991, pp. 38-45.