Complete Parallelization of Computations: Integration of Data Partitioning and Functional Parallelism for Dynamic Data Structures

J.C. Browne, Dwip Banerjee[1]
Department of Computer Sciences
University of Texas
Austin, Texas

Abstract: This paper presents a parallel programming system which i) supports complete parallelization of array oriented computations through a coherent integration of data partitioning parallelization and functional decomposition based parallelization and ii) implements a declarative representation of operations over distributed dynamic arrays. The conceptual framework for this integration is a generalized dependence graph model of parallel computation. The properties of the programming system are illustrated by two examples: a red/black parallel solver for linear systems and an h-adaptive finite element algorithm.

1. Problem Statement: Complete parallelization of adaptive computation systems

Partitioning of arrays into segments for SPMD parallelism [Hil 86, Fox 91, Hir 91]is the natural mode of parallelization for the major fraction of the computational work in many engineering and scientific computations. But after the array computations are parallelized, there usually remains a substantial amount of computational work which cannot be parallelized using data partitioning. Functional decomposition [Chan 92, Car90] is the natural mode for parallelizing these typically control flow dominated computations. Yet, if the parallelization is to be scalable to high degrees of parallelism (and avoid Amdahl's dilemma) then both the array oriented and the control flow oriented segments of the computation must be fully parallelized.

There have been few efforts made to define and implement clearly integrated representations for data partitioning and functional decomposition based parallelism [Cha 94]. The common practice for integration of data partitioning and functional decomposition mode of parallelism is to embed data partitioning and communication/synchronization primitives into a procedural sequential language [Cha 91b, Hir 91, RSW 90]. This embedding of orthogonal constructs into languages which are already cluttered by historical accretion of constructs leads to languages which are both difficult to use and hard to compile.

This paper presents one attempt to define a smooth integration of data partitioning and functional decomposition modes of parallel structuring declaratively and at a level of abstraction which supports both ease of expression and portability.

The second aspect of integration of data partitioning and functional decomposition modes of parallel structuring addressed in this paper is the extension of data partitioning to parallelization of operations over distributed dynamic arrays. The emerging generations of high performance parallel computations will be largely based on adaptive algorithms which lead to operations over dynamic arrays. It is awkward to express parallel operations over dynamic arrays in languages such as Fortran or C. Programs for parallelized adaptive algorithms in such languages often have 80% or 90% of their code volume implementing data management for dynamic data structures. The parallel programming system defined and illustrated in this paper integrates declarative expansion and contraction operators for arrays with the data management facilities it provides for data partitioning.

2. Approach

The generalized dependence model of parallel computation [Bro85] provides a natural framework for integration of data partitioning and functional decomposition modes of parallelization. The nodes of the dataflow graph can be viewed as functions which accept the input data from the arcs and place their output on the arcs. The nodes *fire* i.e. *are enabled* when some prespecified condition on the state of the input arcs is satisfied and once completed they propagate the data according to specific routing rules. The Figure 1a) below expresses the concept pictorially as well as textually for a specific system (CODE 2.0)

The conceptual approach to adding data partitioning to a dataflow graph is to associate partitioning, merging, contract and expand operations with arcs which carry arrays. Figure 2a) is a fragment of a generic dataflow graph illustrating data partitioning. The semantics associated with the construct of Figure 2a) are:

- the array A is partitioned according to the specification given by the arguments of the **partition**() operator

---

[1] *http://www.cs.utexas.edu/users/{browne,dwip}*

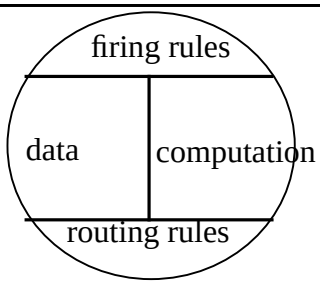

Figure 1. a) Computation

```
input_ports { int X;}
output_ports { int Y;}
vars { int v;}
firing_rules {  X -> v => }
comp { val = invert(v); }
routing_rules {   TRUE => Y<- val; }
```

Figure 1. b)Text inside a computation

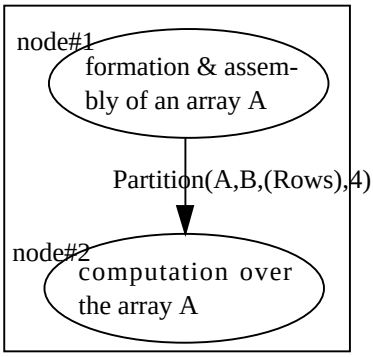

Figure 2. a) Dataflow Graph with partitioning operator on the arc

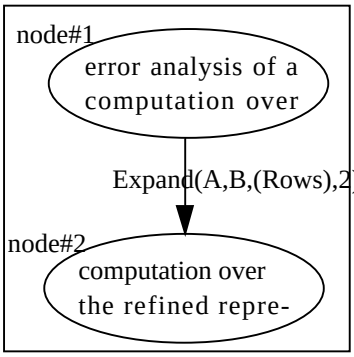

Figure 1. b) Dataflo w graph with array Expand operator

- creation and distribution (if necessary) of *n* replicas of the computations of node#2
- defining of the appropriate array partitions to each replicated copy of the computation
- execution of the computation at each replica INCLUDING all necessary communication and synchronization required for dealing with overlapping partitions.

Expand/contract operators are also similarly represented and defined. Figure 2b) is a fragment of a generic data-flow graph with an array expansion. The semantics of the construct are:

- the array placed on the arc by the node is expanded according to the specifications of the **expand()** operation
- the structure of the global array is updated to reflect the modification caused by the expand operation.
- the expanded array partition is delivered to node#2 to continue the computations of the operations
- the operations of node #2 are executed including any necessary communication and synchronization

3. The CODE 2.0 Programming Environment

The CODE 2.0 parallel programming system is a high level parallel programming environment which currently implements a task/dependence model of parallel computation. Parallel programs are composed as dependence graphs where the tasks are atomic units of computation and the arcs are dependence relations among tasks. Interactions are restricted to task instantiation and task termination times. This model has a number of advantages, including: (i) parallel structuring is cleanly separated from the sequential computations which are being composed to construct the parallel applications and (ii) an abstract declarative specification of communication and synchronization. This task/dependence model of computation, with its abstract specification of parallel structure and its separation of parallel structuring from sequential computation, provides a framework for integration of data partitioning and functional decomposition modes of parallelizations.

We present here an overview of CODE 2.0's basic model of computation, mostly by means of a simple example program. CODE 2.0 programs consist of a set of graph instances that interact by means of Call nodes. Graph instances in CODE 2.0 play the role of subroutines in conventional programming languages. The number and type of the instances is determined at runtime, but each graph is instantiated from one of a fixed set of graph templates. Figure 3a) and 3b) shows two important node types. When the user draws a graph in the programming model, he is creating a template, not an instance. Instances are created at runtime when they are referenced from a Call node. This concept of dynamic instantiation from fixed templates is ubiquitous in CODE 2.0; almost all model objects work in this way. Once the graph is drawn it can then be translated for different architectures to produce executables which can be run on that machine.

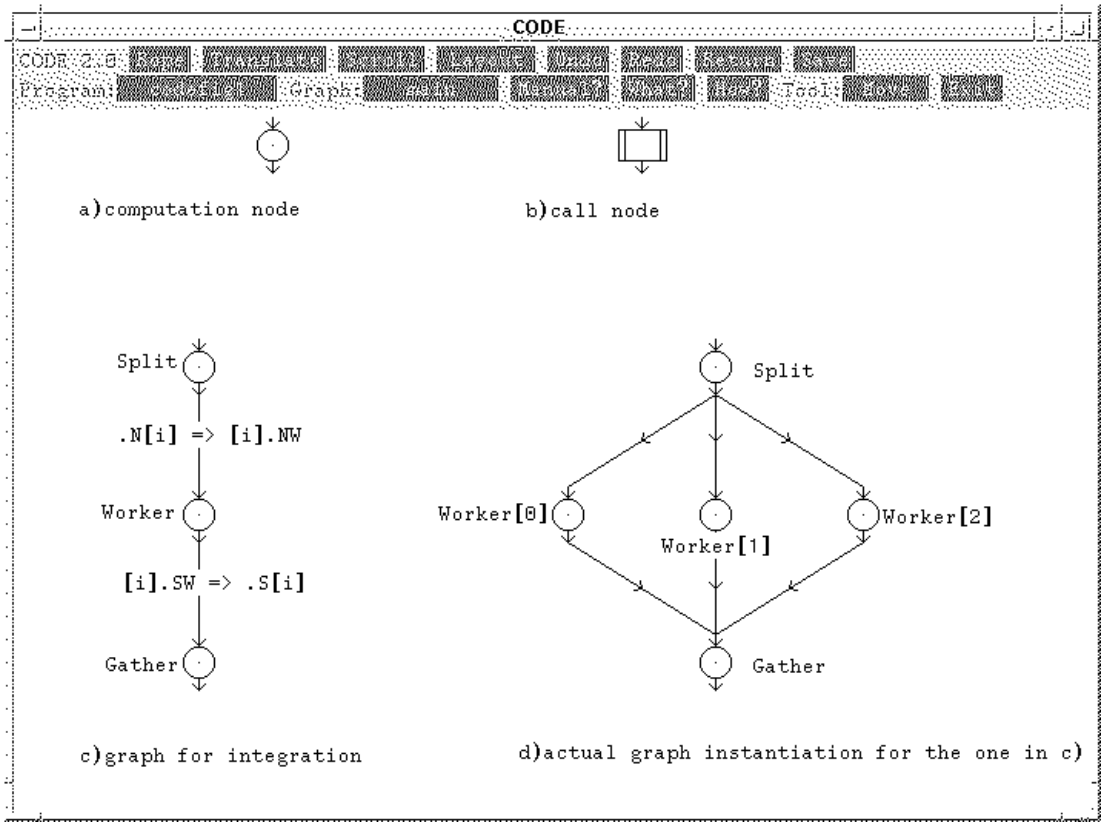

Figure 3. Code icons and a simple graph

Graphs consist of nodes and arcs. Arcs represent channels for the flow of data from one node to another. They serve as unbounded (subject to system memory constraints) FIFO buffers. There are several types of nodes. The most widely used nodes - the computation node and the call node are shown in Figure 3a) & 3b). As alluded to previously, Call nodes

specify arc connections to other graph instances. Interface nodes specify points at which a calling graph can connect to another graph. Unit of computation nodes (UCs) represent basic sequential computations. Hence, they are the fundamental elements from which parallel programs are assembled. They consume data from incoming arcs, perform a computation, and place data onto outgoing arcs for other nodes to consume.

UC nodes have several attributes. The node's computation is expressed as a call to a subroutine in a conventional sequential language. Firing rules specify the conditions under which this computation is allowed to execute. UC nodes also have input and output ports for the communication of data to and from the node. Arcs are bound to specific ports. Hence, multiple arcs may enter or leave a node without ambiguity.

Figure 1b) illustrates a computation node.This node has an input port X and output port Y. The node may execute when there is a value on incoming the incoming arc bound to X. The value is placed into variable v and inverted by the node's computation. Finally, the value is placed onto the outgoing arc bound to Y.

Multiple arcs may connect two nodes. Hence, CODE graphs are really multigraphs. An extended example is depicted in Figure 3c) in which numerical integration is performed in parallel using Simpson's rule. The node Split partitions the interval which are then sent to specific instances of the Worker node (#0 or #1 or #2) by means of arc topology specification. It is this specification (shown above as.N[i] => [i].N) which creates the requisite number of the Worker nodes and thus the parallelism. Once done the workers send their part of the result to the assembler node (Gather) which sums it up. The fan-in of the parallelism is performed on the arcs of the Worker node. Thus while the CODE graph is represented in Figure 3c) the actual instantiation with the parallel structuring is depicted in Figure 3d). The node computation for the Worker is illustrated in Figure 4.

```
input_ports { int NW;}
output_ports { real SW;}
vars { int i; real a; real b; real val;
       int n; real h; }
firing_rules { NW -> n => }
comp {
  i = NodeIndex[0];
  h = (B-A)/NPROC;
  val = simp(A+i*h, A+(i+1)*h, n); }
routing_rules { TRUE => SW <- val; }
```

Figure 4. Node specification for the Worker task in Figure 2

4. CODE 2 Extensions for Dynamic Data Partitioning

CODE2.0 previously instantiated a functional decomposition mode of parallelization. This section specifies the means for incorporating dynamic data partitioning into the CODE2.0 environment. The extensions which implement dynamic data partitioning are now described in detail.

4.1 Integration of Functional and Data Partitioning Parallelism

The CODE 2.0 parallel programming environment provides a framework for integrating data partitioning and functional partitioning in a clean and simple way. The essence of data decomposition is that parallel structure is specified as a replication of the same operation on partitions of regular data structures (Single Program Multiple Data or SPMD mode of parallelism).

Data partitioning is realizable in the current CODE 2.0 by the user explicitly creating arrays which are partitions of other arrays and writing an appropriate routing rule. This process can become quite complex when the partitions include overlaps, etc.

Explicit declarative representation of data partitioning is introduced into CODE 2.0 by defining an appropriate set of operators on arrays primitive in the representation and applying the operators to the data structures which are assigned to an arc by a topology specification placed on the arc. The primary operators are: ***partition, merge, expand, contract.*** (There are some further operators which are not discussed herein [Ban 94].)

In the current version of CODE 2.0 parallelism is introduced by creating multiple output ports for a node and using the arc topology specification to connect each port to a different instance of the node which is to execute in parallel. The replicated node (the sink node for the arc that carries the data being partitioned) has partitions of data structures assigned to each replication. The application of the partitioning operator extends this method of creation of replicated nodes. The partition operator, when applied to an arc, partitions the data and automatically replicates the arc and the sink node to conform to the number of partitions specified in the partition operation. Figure 5 shows the partitioning operator applied to a CODE 2.0 graph. The routing rule here simply assigns the arrays to the arc and the partition operator completes the creation of the instances of N_2. There is, in this simple example, a hint of the convenience induced by the integration of the two modes of specification.

Dynamic arrays are implemented through the use of expansion and contraction of data structures to model adaptive algorithms where the underlying representations change structure at runtime. For example, in a finite element mesh used to model a domain, it is often necessary to focus on some regions of interest and refine the mesh at those regions. We have provided several versions of the **Expand** and **Contract** operators to handle these cases efficiently. When applied in conjunction with the **Partition** and **Merge** operators they provide a convenient tool to extend data partitioning to dynamic distributed arrays.

4.2 The operators

While effective data partitioning was one concern the other side of the challenge is represented by algorithmic dynamicity. Many modern algorithms employ evolving data structures that undergo transformation with time and information availability. It is critical to be able to represent these structural modifications in a coherent scheme that also allows partitioning. An example of this arises in the modelling of air flow over an aeroplane's wings. If the wing is represented as a mesh, we will be more interested in the action on the edges rather than at the center. This will mean that the mesh requires more resolution at the edges than the center. We are thus dealing with a non-uniform mesh whose interest region is the more refined parts (the edge). As more information gets available during the modelling and we get some idea about the airflow on the wing the data is used to further refine the mesh. It is almost a self-modifying procedure till a state of dynamic stability is reached. These are the kinds of computations targeted by the integration of functional and data partitioning parallelism. They involve partitioning of the data (mesh here) along with other kinds of data manipulation (in this case the refining of the mesh at some regions and the coarsening of the mesh at others). This is a challenge because the requirements can often be conflicting. For example, most data structures supporting partitioning efficiently will probably be incapable of dealing with the dynamic self modification of the substructures of data and its accompanying needs.

We have introduced some new abstractions in the CODE2.0 framework to deal with dynamic data partitioning. The data partitioning/manipulating operators are associated with the arcs that carry the data. We chose the arcs as the site for these primitives is because arcs form the link between a data and its transformation. The operators may be viewed as transforming (or partitioning) the data on the arc between invocations of computations that use the original structure and the transformed structure. For practical reasons too it is best for the manipulation to take place on the arcs because it is easier for the compiler to do the implementation as well as the checking since it can use information about the data structure from both the node from which it emanates and also from the incident node.

On the implementation side we have devised a new array management scheme where an extendible hash table is used to implement arrays using space filling curves to generate keys [Fag 79, Kor 90, Sal 93]. Partitioning is performed by splitting the table while expansion/contraction of data is handled extending/shrinking the hash table as required.

4.2.1 Partition

Partitioning a data structure into substructures is the singlemost frequent operation in the extraction of data parallelism (note promise of embedded conforming communication/synchronization operations). The operator **Partition** divides an array into its constituent (possibly overlapping) subblocks. Its syntax is

Partition(*A,B, part_spec, size_spec, overlap_spec)*

- *A* is the array being partitioned
- *B* is the name of the array of partitions (each of which is an array) being created
- *part_spec* contains the partitioning directive along each dimension
- *size_spec* defines the size information along each dimension for the partitions
- *overlap_spec* is the specification of the overlap constraints for the partitions

The partition operation specification is associated with the arc carrying the array and implicitly connects a node (the data structure to be partitioned) to a nodearray (inputting the partitions generated). The first array (A in this case) is in the scope of the source node while the created partitions (B) are in the scope of the sink nodearray. In other words each created partition is called B and is in the scope of the node to which it is routed. Figure5 demonstrates visually the effect of Partition on an arc connecting two nodes N_1 and N_2 .

The **Partition** operator returns an *n*-tuple where *n* is the number of partitions determined by the partitioning directive (<*part_dir*> above) and each partition is a subblock of the original array determined by <*part_dir*>. The number of partitions is a runtime determined parameter (called the creation parameter in CODE 2.0 [Bro 89, New 92, New 93]). These members are arranged in a grid specified by the partitioning dimension. Normal array indexing notation can be used to refer to individual blocks of the partition within the computation at the nodes. Overlapping partitions can be generated by specifying the *overlap-info* part in the requisite way. Moreover, it is possible to refer to individual elements in the partition (subblocks) by using the indices of the partition. For example to refer to the fifth element in the partition of the 2D array *A* decomposed into 8 blocks along rows with adjacant blocks sharing the boundary rows, one would use

Partition(A, A, (BLOCK, *), (8, *),(OVERLAP=1, *)) [5]

and to refer to the top left corner element generated by the partitioning of a two-dimensional array without boundary sharing, one can use

Partition(A, A, (BLOCK, CYCLIC), (8, 10),(*, *)) [0][0]

In simple cases, partitioning by rows or columns or blocks will suffice but users can combine these specifications to generate complicated partitioning patterns. The number of dimensions along which the partitioning is performed determines the dimension of the partition grid generated. In the following examples, 16 partitions are formed in a 4×4 grid in the first case and a 4×2 grid is formed in the second case.

Partition(A,B,(BLOCK, BLOCK),(4,4),(OVERLAP=1, OVERLAP=2))

Partition(A,B,(BLOCK, CYCLIC),(4,2),(OVERLAP=1, OVERLAP=1))

A one-dimensional array can be partitioned into scattered blocks of data sharing the boundary element, in the following manner:

Partition(A,A,(BLOCK_CYCLIC(8)),(4),(OVERLAP=1))

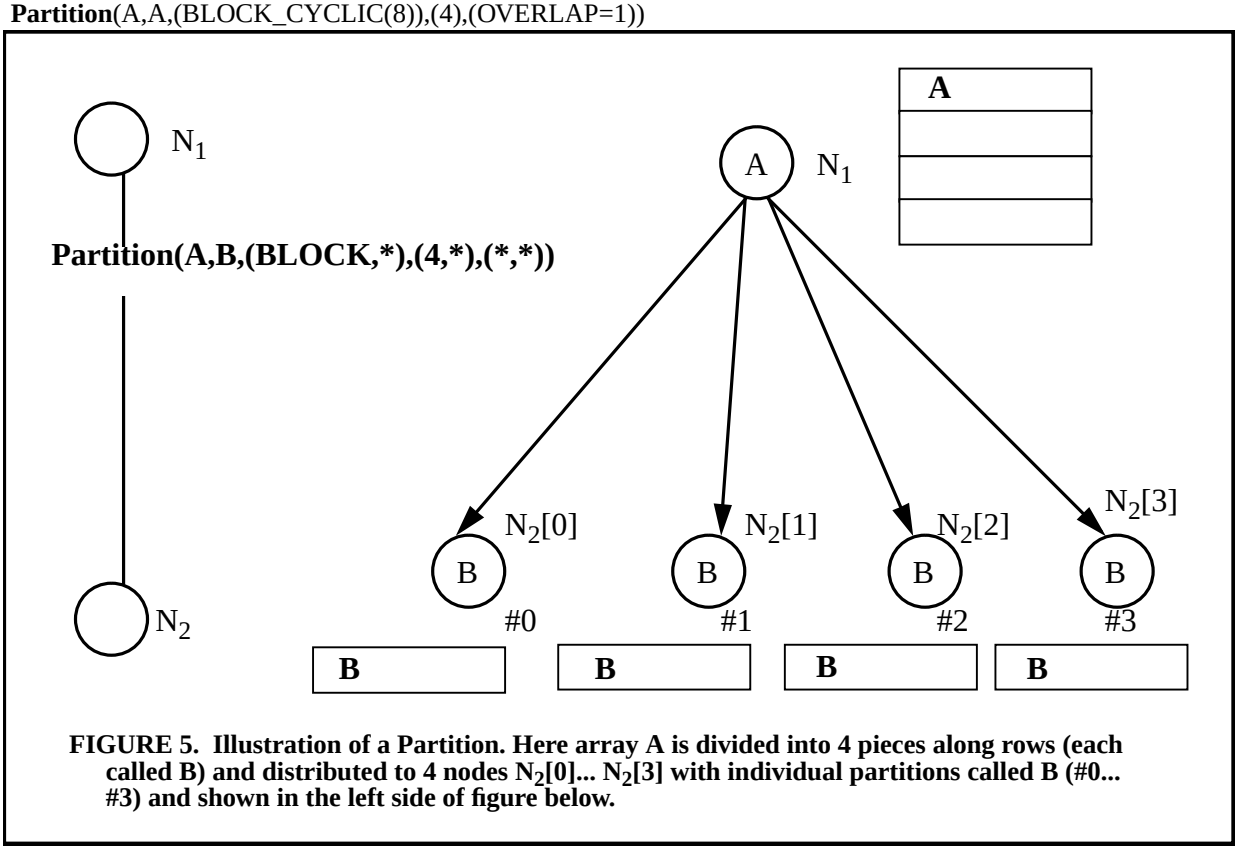

FIGURE 5. Illustration of a Partition. Here array A is divided into 4 pieces along rows (each called B) and distributed to 4 nodes $N_2[0]$... $N_2[3]$ with individual partitions called B (#0... #3) and shown in the left side of figure below.

4.2.2 Merge

This has just the inverse functionality of **Partition**. It is used to merge conforming arrays into another array. This is an useful operation particularly at synchronization points in a program and gives a global view of data when required. Moreover, it facilitates *data redistribution* when needed. When several data structures are merged, the merging is done according the lexicographic ordering (in terms of the index) of the nodes that are participating in the merge. The inverse topology of Partition can be used to merge the data. The syntax is:

Merge(*A,B, merge_spec, size_spec, overlap_spec*)

- *A* is the name of the structures to be merged
- *B* is the merged (whole) structure.
- *merge_spec* gives information on how the structures are to be merged
- *size_spec* specifies the number of structures being merged along each dimension
- *overlap_spec* specifies the overlapping nature of the structures being combined so that the merged structure will retain the proper semantics

Example:

Merge(B, A, (BLOCK, BLOCK), (4, 4), (OVERLAP=1, OVERLAP=1))

can be used to get back the data array A which was distributed as

Partition(B, A, (BLOCK, BLOCK), (4, 4), (OVERLAP=1, OVERLAP=1)

We have also introduced some auxiliary operators in addition to the above, mainly for convenience reasons. One such operator is **Strip_Update**(*A,B, part_spec, size_spec, overlap_spec)* which has very similar syntax as partition/merge. It is used for doing effective communication while updating overlapping data parts shared between partitions. It makes call to a communication library when invoked. [Ban 94] describes these operators in substantial detail.

4.2.3 Expand & Contract

In many applications it is necessary to have nonuniform grids with varying levels of fineness at different places. There are two commands to facilitate the generation of non-uniform grids and map them to a dynamic array.

Expand(A, B, *#dim, g, i*)

Contract(A, B, *#dim, g, i*)

- A is the array to be expanded/contracted
- B is the expanded/contracted array
- *#dim* is the dimension of A
- *g* is the growth factor (no. of times it is to expanded/contracted along that dimension)
- *i* is the index of dimension along which the operation is done.

There are different kinds of expansion that can be allowed and there are strong reasons to incorporate more than one in the language, so that the problems of expansion in different domains can be handled in a flexible and succinct manner. Among the possibilities are expansion by rows, columns, strips (row-column band at the end of the structure) and other expansion schemes as may be necessary in practice in a given domain (e.g. the expansion/contractions caused by adaptive mesh refinement in finite element code). We are actively involved in the use of dynamic structures to handle the runtime expansion/contraction (in place) of data structures while maintaining desirable properties like contiguity of structurally adjacant elements, reuse of structures etc. [Ros 71].

For adaptive finite element methods, the ability to refine and coarsen an element is of fundamental importance. Here elements are grouped into subdomains and each element (matrix) contributes to the overall subdomain matrix in a manner determined by the subdomain construction. The best results are obtained if it is possible to maintain contiguity for both the element level matrices as well as the subdomain matrices. We have extended the definition and implementation of expand/contract to encompass new cases that can handle this problem efficiently.

Example: **Expand**(A, A,2,2,1) causes the 2D array A to be expanded twofold along rows (so the number of rows is doubled) and the expanded array is called A also.

Contract(A,A,2,2,2) causes the 2D array A to be shrunk to half its size columnwise

An important and desirable feature of the operators is they can be composed meaningfully. For example, merging of the partitions created by a *partition* operator causes the original structure to reappear. Selective refinement/coarsening of data can be obtained by applying expand/contract on partitioned structures. We have also developed a rich theory for these operators for sophisticated data structures (hierarchical matrices) and extended the composability theory to the same [see Ban94].

We will now try to illustrate the use of these operators through examples. The first example deals with a red-black checkerboard algorithm performing a Jacobi iteration and the next one with a more involved finite element algorithm.

5. Examples

5.1 Red-Black checkerboard algorithm

The basic ideas of data decomposition in the (data partitioning) extended version of CODE 2.0 are illustrated with the following example. It implements a red-black successive over relaxation technique on a simple grid [Fox 88a]. The grid is partitioned by blocks along rows and columns and the sub-arrays created are assigned to separate processes which perform their computation on their section of the data. The border data is shared in such a manner that there is just one writer and a reader for that data (a strip in this case). At the end of each iteration the adjacant processes need to communicate their shared boundary values, with the owner of the strips sending the updated values to the readers. Thereafter, the processes repeat the entire cycle till some desired criterion is satisfied.

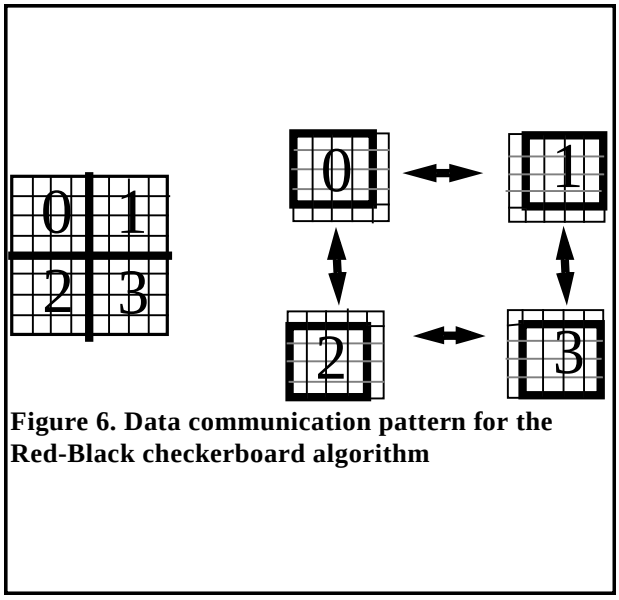

Figure 6. Data communication pattern for the Red-Black checkerboard algorithm

Figure 6 shows the case when the computation is performed with four processors. The data exchange pattern is indicated with arrows. The data partition is generated by the following statement: **Partition**(A,B,(BLOCK,BLOCK),(2,2),(OVERLAP=1, OVERLAP=1)). In this case the partitioning is performed by blocks along rows and columns as well. We also have the capability of specifying partitions in a cyclic (round-robin) fashion along any dimension of an array.

The communication is performed by the use of the **Strip-Update**(A,B,(BLOCK,BLOCK),(2,2),(OVERLAP=1, OVERLAP=1)) statement which simply updates the overlapping parts in an efficient manner [see Ban 94 for more details on this operation]. This allows for an efficient communication generation by specifying that a guard region of width one is shared between adjacant processors which need to be updated whenever this command is executed. Each strip has at most one writer. Finally the entire array is displayed as output and the whole array is produced through the **Merge(**A,B,(BLOCK,BLOCK),(2,2),(OVERLAP=1, OVERLAP=1)) statement. Figure 7 displays the CODE 2.0 program for this example.

The results of running the program on a shared memory Sequent Symmetry multiprocessor are displayed in Figure 8. In this case the checkerboard algorithm was applied on a 200×200 matrix with each processor sharing a guard region of

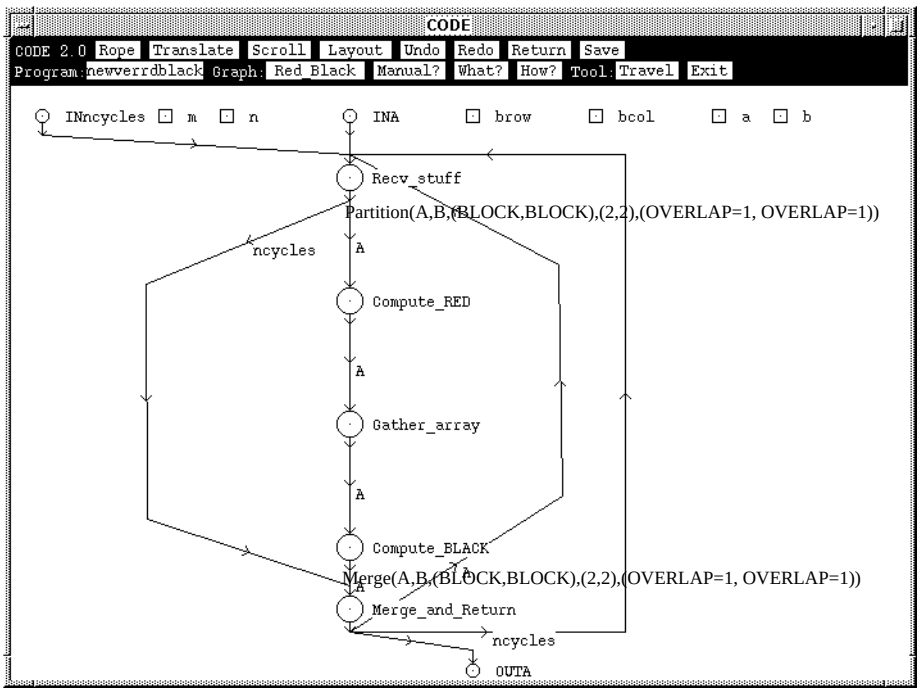

Figure 7. CODE graph to perform the Red-Black problem

width one. The graph indicates a fairly good speedup for this midsize problem. We expect the performance to get even better as the problem size gets bigger and the granularity of computation increases.

The main benefits illustrated by this example is the simplicity of the CODE graph program specification. The challenge of writing complicated routing rules and demanding arc mappings was replaced by the invocation of a single data partitioning primitive on the arc that did the partitioning and routing.

5.2 Finite Element Example

If we consider a linear system of the form

$$Ax = b$$

such that A is positive definite i.e. $(Ax, x) > 0 \ \forall x \neq 0$

Letting $A = A_1 + A_2$, the solution algorithm can be given as

$(\mu I + A_1)\, x^{n+\frac{1}{2}} = (\mu I - A_2) x^{n}$

$(\mu I + A_2)\, x^{n+1} = \mu I - A_1)\, x^{n+\frac{1}{2}}$ where $\forall \mu > 0$ and A described as above it can be proven that the scheme converges.

If we consider a domain as in Figure 9, we can represent the domain $\Omega = \Omega_1 + \Omega_2$ where Ω_1 is the union of all the lightly colored strips and Ω_2 is the union of all the shaded strips.

Let A_1 be the operator associated with Ω_1 and A_2 the operator associated with Ω_2. Obviously the variables in two different strips have no interaction. So, A_1 is a block diagonal matrix with one block associated with each strip. The only assumption here is that the strips belonging to each subdomain are disjoint which is achieved by separating strips associated with one subdomain from strips belonging to another subdomain. The geometry or the size of the strips concerned is inconsequential.

Since A_1 and A_2 are both block diagonals, the operations associated with each stage of the finite element solution process over their individual domains can be carried out in parallel on different processors without any interface or communication problems having to be solved. The only communication requirement is that updates are made for neighboring processors after matrix vector products and triangular solves are made for shared degrees of freedom.

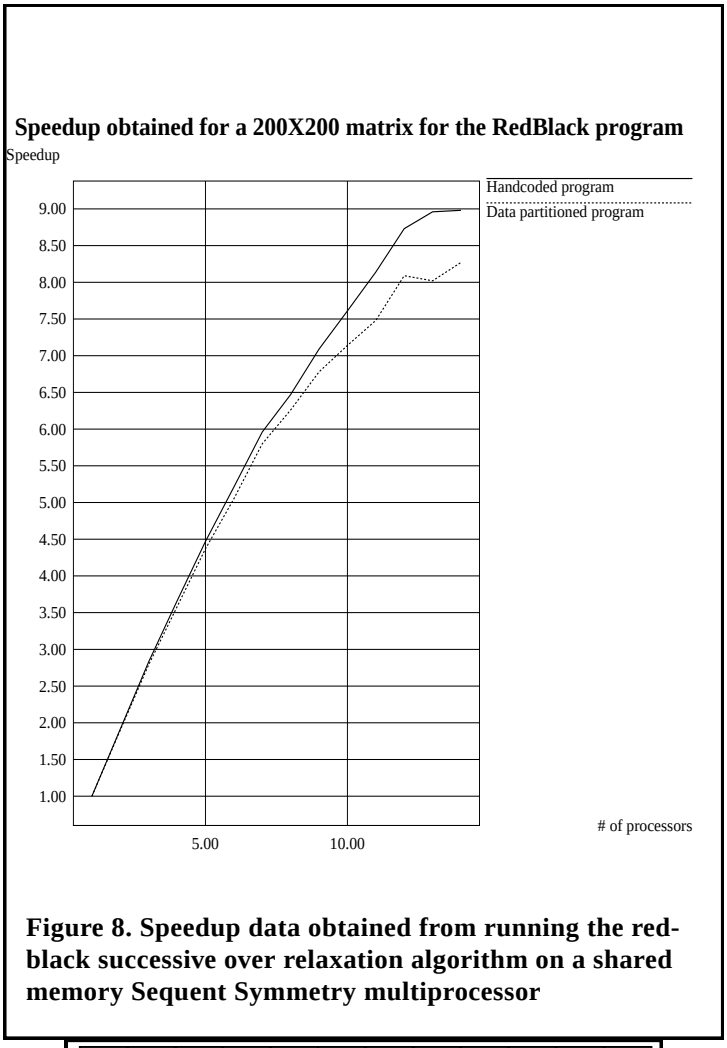

Figure 8. Speedup data obtained from running the red-black successive over relaxation algorithm on a shared memory Sequent Symmetry multiprocessor

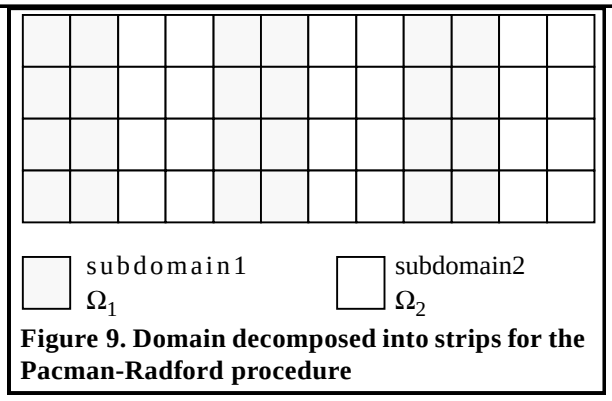

Figure 9. Domain decomposed into strips for the Pacman-Radford procedure

We now describe the algorithm. Let A_1 -> A_{red} and A_2->A_{black}.

1. domain is partitioned into p connected components with each partition having its own A_1 and A_2
2. all the element matrices A_e are assembled into A_{red} and A_{black} in parallel
3. b = b_{red} + b_{black}
4. pick a μ>0 and an initial guess to solution $x_b^0$and x_r^0
5. form the matrices $(\mu I + A_{red})$, $(\mu I + A_{black})$, $(\mu I - A_{red})$ and $(\mu I - A_{black})$
6. do LU-factorization of $(\mu I + A_{red})$ and $(\mu I + A_{black})$
7. repeat the following till stopping criterion is satisfied:

* in parallel $r_b = b_{black} - (\mu I - A_{black})\, x_b^k$
* transfer interface data for connected partitions
* parallel solve $(\mu I - A_{red}) x_r^k = r_b$
* transfer interface data
* repeat the four previous steps with 'black' for 'red' and vice versa
* if $\| x_b^{k+1} - x_b^k \| + \| x_r^{k+1} - x_r^k \| < T$ then stop.

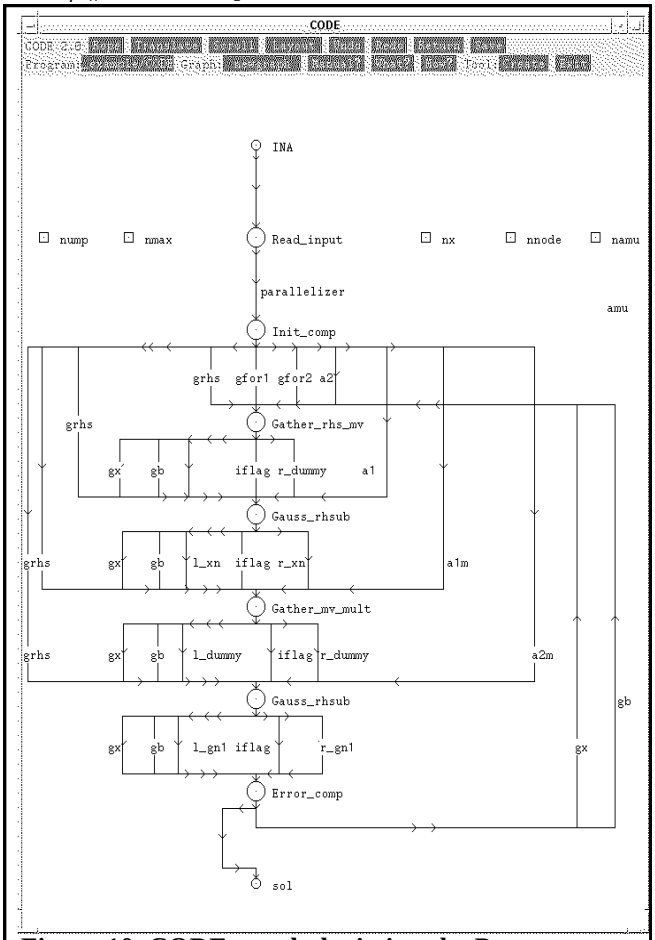

Figure 10. CODE graph depicting the Pacman-Radford solution method

Figure 10 above shows the CODE graph that performs the Peaceman-Rachford procedure [Pat 93]. The arrays are partitioned by blocks along with right hand sides and also the solution array. Thereafter each process does its own solution and exchanges data along the interface with other processes thereby propagating their own contributions in a way. The pro-

gram was translated and run on a PVM backend (distributed memory) and the resultant timings are shown in Figure 11 in the form of speedups obtained.

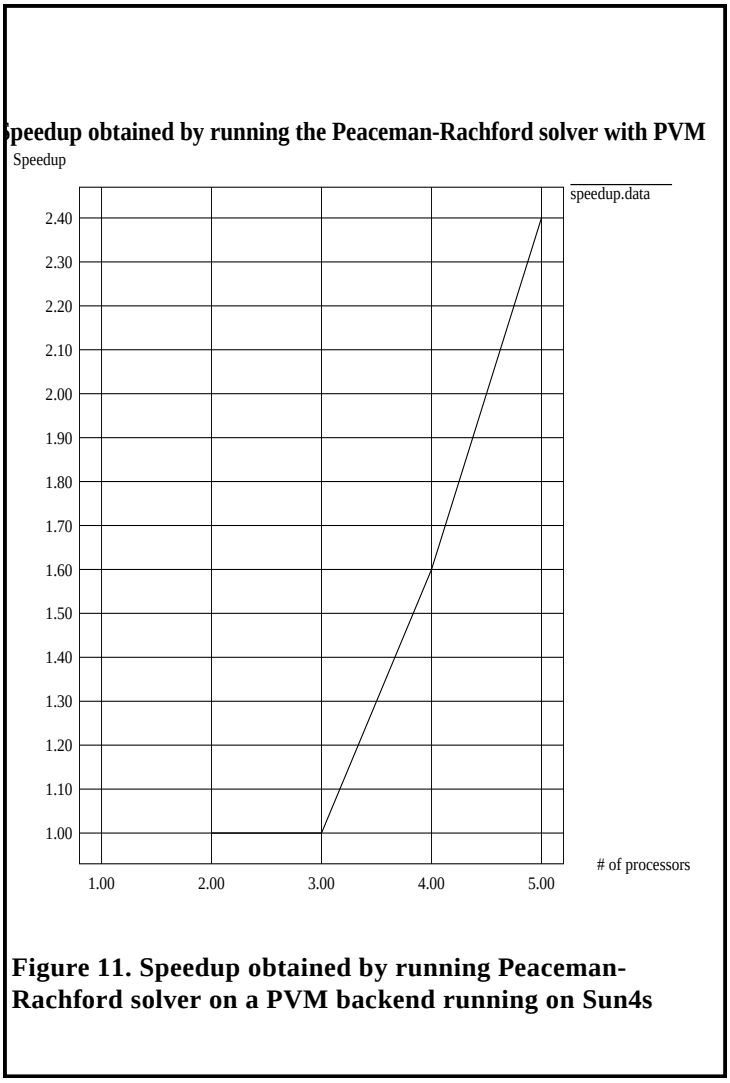

Figure 11. Speedup obtained by running Peaceman-Rachford solver on a PVM backend running on Sun4s

The above program was run for matrices of size 37X37. This proved to be too small a granularity to obtain good speedup because of the communication overhead of the pvm backend. There is a ring communication going on at four places in the graph (nodes Gather_rhs_mv, Gauss_rhsub, Gather_mv_mult and Gauss_rhsub#2) where each of the parallel nodes talks to its left neighbor and also its right neighbor (nodes at the end being the exception). This is the means by which overlapping values are exchanged and interface data is propagated. But it also a synchronization bottleneck and entails a tremendous communication overhead since it is done for each iteration. This is the main reason for not obtaining a more impressive speedup. For larger matrices, when the computation dominates, there will be further improvement in performance.

One of the most important and reoccurring paradigms in the domain of finite elements is to refine interesting parts of the grid where the perceived importance of the modelling lies. This gives a nonuniform texture to the grid since different areas are now modelled with various levels of fineness. When this is combined with the problem of parallelization it is quite difficult to implement the aforesaid grid in an efficient manner (with good load balancing properties). The crux of the problem is to design a data structure that will allow simultaneous expansion and contraction of different parts of the grid while retaining the overall structure at the same time. It should preferably maintain contiguity for the refined structures to the maximum extent possible. This also provides us with an opportunity to demonstrate the usefulness of the composability of our operators. For example, selective refinement of partitions can be used to realize a nonuniform mesh like the structure shown above in Figure 12 by using the following operator combination:

Expand(Partition*(mesh, ... other partition directives), ... other expansion directives))*

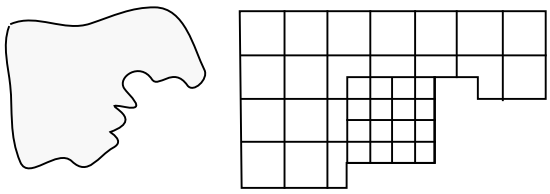

Figure 12. Selective refinement of mesh

6. Conclusion

In this paper we have tried to define a set of abstractions appropriate to the emerging generation of numerical computations. These have been tailored to reflect the new approaches that are increasingly being used in the domain of parallel processing. They include the ability to specify dynamic data partitioning along with language and architecture independence in a modular manner. We have at the same time tried to avoid the pitfalls of either forcing the user to do complex data management or trying to do everything for the user ourselves. We take the view that different application domains have different requirements and have tried to design our abstractions to meet specific needs. This domain specific approach improves performance as the user can supply useful information to the compiler and also it makes the compiler's job much easier. The results we have obtained indicates there is considerable merit to this approach and further research is being conducted to refine the abstractions to handle a wider variety of applications in an efficient manner.

7. References

[Ban 94] Dwip N Banerjee: Integration of Data Partitioning in a Visual Parallel Programming Environment, Dissertation Proposal, Department of Computer Sciences, The University of Texas at Austin

[Bro 85] J. C. Browne: Formulation and Programming of Parallel Computers: a Unified Approach, Proceedings of the International Conference on Parallel Processing, 1985, pp. 624-631

[Bro 89] J. C. Browne, M. Azam, and S. Sobek: CODE: A Unified Approach to Parallel Programming, IEEE Software, July 1989, p. 11

[Car 90] Nicholas Carriero and David Gelernter: How to Write Parallel Programs, A First Course, MIT Press, 1990

[Cha 91a] Barbara M. Chapman, Heinz Herbeck and Hans P. Zima: Automatic Support for Data Distribution, Proceedings of the Sixth Distributed Memory Computing Conference, Portland, OR, April 1991, pp 51-57

[Cha 91b] Barbara Chapman, Piyush Mehrotra and Hans Zima: Vienna Fortran - A Fortran Language Extension for Distributed Memory Multiprocessors, ICASE Report No. 91-72, September 1991

[Cha 94] Chapman et. al.: A Software Architecture for Multidisciplinary Applications Integrating Task and Data Parallelism, ICASE Report 94-18, March 1994, Langley, VA

[Chan 92] K. M. Chandy and S. Taylor: An Introduction to Parallel Programming, pp 145-157, Jones and Bartlett, Boston, 1992

[Fag 79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, " Extendible Hashing - A Fast Access Method for Dynamic Files", ACM Transactions on Database Systems, Volume 4, Number 3, (September 1979), pp. 315-344

[Fox 88a] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, S. K. Salmon and D. Walker: Solving Problems on Concurrent Processors, Volume I, published by Prentice Hall 1988

[Fox 88b] Geoffrey C. Fox: What Have We Learnt from Using Real Parallel Machines to Solve Real Problems?, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, pp 897-955

[Fox 91] Geoffrey Fox, Seems Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, Min-You Wu: Fortran D Language Specification, Department of Computer Science, Rice University, Rice COMP TR90-141, December 1990

[Hil 86] D. Hillis and G. Steele.: Data Parallel Algorithms, Communications of the ACM, 29, p. 1170

[Hir 91] Seema Hiranandani, Ken Kennedy and Chau-Wen Tseng: Compiler Support for Machine Independent Parallel Programming in Fortran D, Rice University, CRPC-TR91132, 1991

[Kor 90] Henry F. Korth and Abraham Silberschatz: Database System Concepts, McGraw Hill, 1990

[New 92] Peter Newton and J. C Browne: The CODE 2.0 Graphical Parallel Programming Language, Proceedings of the ACM International Conference on Supercomputing, July, 1992

[New 93] Peter Newton and Seema Y. Khedekar: CODE 2.0 User Manual, pp 1-66, University of Texas at Austin, March 1993

[Pat 93] Abani K. Patra: Parallel HP Adaptive Analysis, Dissertation Proposal, The University of Texas At Austin

[RSW 90] Matthew Rosing, Robert B. Schnabel and Robert P. Weaver: The DINO Parallel Programming Language, Journal of Parallel and Distributed Computing 13, 30-42 (1990)

[Ros 71] Arnold L. Rosenberg: Data Graphs and Addressing Schemes, pp 193-238, Journal of Computer and System Sciences 5, 1971

[Sal 93] Michael S. Warren and John K. Salmon: A Parallel Hashed Oct-Tree N-Body Algorithm, Proceedings of Supercomputing’93