Using Domain-Specific, Abstract Parallelism

Ira Baxter and Elaine Kant
Schlumberger Laboratory for Computer Science
8311 North RR 620
Austin, Texas, 78720-0015
baxter@slcs.slb.com, kant@slcs.slb.com

Abstract

Discovery of potential parallelism in low level code is difficult, especially in the absence of problem domain knowledge. An alternative is to explicitly represent maximal potential parallelism in abstract program components. A transformation system refines a program composed of such components into a concrete program. We discuss an experimental system in which we are installing such facilities. An example refinement sequence is provided.

1 Introduction

Compiling problem-domain independent program representations for parallel architectures is often difficult because of the need to infer opportunities for parallelism. Because safe inference of parallelism must be conservative, the inferred parallelism is often considerably less than that actually available in the applications. This problem leads to a demand for tools such as E/SP [SMD+89], ParaScope [BKK+89] and MIMDizer [Cor90], which identify points of potential, but unverifiable, parallelism and query the programmer to determine a less conservative version of the truth. Inference and query-the-programmer are both methods for rediscovering the parallelism. All of this would be unnecessary if the knowledge of what was parallel at the time of program construction were not lost.

An alternative approach we are pursuing is to capture the inherent parallelism (actually, absence of execution ordering constraints) in an abstract program in a domain-specific fashion. Then a transformation system would refine not only the program but also the parallelism information into the concrete program. In this fashion both the expense of the conservative inference and the need to query the programmer are minimized.

In this paper, we give a short example of an abstract domain-specific component whose full parallelism is "refined away" (rather than rediscovered) until it is usable on a particular target machine. We also briefly motivate the need for non-tree-structured internal representations.

2 Problem Domain

SINAPSE [KDMW90] [KDMW91] is an experimental tool to synthesize mathematical modeling programs for a variety of similar applications. These have, to date, been primarily acoustic wave propagation problems, typically used to validate geophysical models for oil exploration.

SINAPSE accepts specifications of typically 20 to 50 lines, and produces C, Fortran, or Connection Machine Fortran programs that solve the differential equations related to the problem domain by using a finite differencing method. Resulting programs are typically 500 to 1500 lines in size; lines are often very dense.

A number of programs generated by SINAPSE have produced useful scientific results for Schlumberger modelers after some post-generation hand optimization. The work described here is part of research aimed at automating that optimization.

Synthesizing modeling programs requires knowledge of the wave propagation problem domain, knowledge about solution techniques for problems in that domain, general programming knowledge, and control knowledge to sequence the synthesis process. This class of program provides many opportunities for data-parallel computation [HS86]; consequently, knowledge of potential parallelism and when to use it is also useful.

3 Synthesis Process

User-specified algorithm schemas are refined by repeatedly replacing schema components with lower-level schemas or parameter values. These component replacements are taken from knowledge bases selected by, or computed directly from, the specification. Rather than being the initial abstract program, the specification simply directs the choice of schemas and parameter values.

Algorithm schemas are stated in terms of a high-level programming language called "algSinapse," which includes assignments, conventional control constructs, array and scalar computations, references to parameters, and references to other algorithm schemas.

Generic programming knowledge as well as application domain knowledge is needed to produce efficient programs. Much of the programming knowledge is in the form of algorithm refinements that expand constructs such as parallel enumeration or matrix multiplication into built-in constructs or a combination of loops and scalar operations depending on the target architecture and language. Rather than having a runtime library of special-case methods (e.g., different matrix multiplications for diagonal arrays), SINAPSE derives the special methods directly. This is accomplished by substituting representations, determined by explicitly represented properties of interest (e.g., DIAGONAL-ARRAY or SYMMETRIC), for references to values, and simplifying away unneeded operations and combining similar terms. This avoids the need to rewrite such libraries for each new target language. The approach is made feasible by the use of Mathematica [Wol91], a symbolic manipulation language as an implementation platform. There are also a number of optimizing transformations.

One of the problems of refining abstract schemas into real programs are inefficiencies introduced because of necessarily conservative analysis of the original schemas. These come about simply because schemas, while optimized maximally on an individual basis, may be more optimizable when combined.

One can resolve this problem in a number of ways, of which SINAPSE currently uses two:

- General purpose optimization techniques, and
- Special case algorithm schemas.

SINAPSE has an optimizer which moves static computations outside of loops. Abstract computations are often placed inside a loop in an originating schema simply because domain knowledge tells us they are usually loop-index dependent. Only when the expression is actually instantiated can we determine the actual loop dependency. If domain knowledge tells us that some expression is always loop independent, then it can be encoded outside the loop in the schema.

The optimizer simply moves blocks of code earlier into the computation as long as this is consistent with the data-flow constraints. This often moves code outside of loops. The moved code is placed in parallel with the earliest statement it can precede. Thus, a free side effect of running the code motioner is the conversion of unnecessary sequencing constructs into parallel execution constructs. A special mechanism detects when expressions dependent only on loop indices can be moved outside the loop. The values of these expressions will be cached in a array. More specifically, storage for the array is allocated, code to fill the array is generated outside the loop, and the cached values from the array are referenced inside the loop. We plan to add a common-subexpression eliminator.

Considerable payoff also occurs when the problem or target domain dictates certain properties of the code; one can then optimize a schema in advance of supplying it to SINAPSE, thereby avoiding the expense of dynamic optimization at program synthesis time. A price is paid for this: manual encoding of such optimized schemas at synthesizer-construction time, and conditioning the instantiation of the special case schemas on the domain property.

4 A Weak Representation for Parallelism

SINAPSE currently represents abstract programs as tree schemas containing various control constructs representing explicit classes of parallelism:

- $seq[s_1, s_2, \ldots, s_n]$
  Sequencing of state-changing constructs s_i
- $doSeq[s, j, lb, ub]$
  Iteration of statement s requiring sequential execution with index j in range $lb \ldots ub$
- $par[s_1, s_2, \ldots, s_n]$
  Arbitrary execution ordering of state-changing constructs s_i
- $doPar[s, [j_1, lb_1, ub_1], [j_2, lb_2, ub_2], \ldots, [j_n, lb_n, ub_n]]$
  Parallel execution of (possibly compound) statement s instantiated with simultaneous assignment of loop indices j_i

doPar provides much of the opportunity to generate data-parallel programs for the Connection Machine 2 (CM2), written in CM Fortran 90. As an example, the following construct:

$$doPar[A[j] = B[j] * k - C[j], [j, 1, size(A)]]$$

is converted into the Fortran 90 array statement:

$$A[1 : size(A)] = B[1 : size(A)] * k - C[1 : size(A)]$$

When the rank of a target array does not match the rank of a source, then the Fortran 90 intrinsic function:

$$\mathtt{SPREAD}(value, axisNumber)$$

is generated to expand the source array along necessary axes.

SINAPSE algorithm schemas also allow the expression of computations on entire arrays, which pass through virtually unchanged to CM Fortran. SINAPSE replaces entire-array operations with *doPar* equivalents when the target is sequential Fortran 77. It is then trivial to generate corresponding sequential code for any *par* and *doPar* constructs.

An explicit concession to data parallelism used in our representation is a variant of *doPar*:

$$makeArray[f(j_1, j_2, \ldots, j_n), [j_1, lb_1, ub_1], [j_2, lb_2, ub_2], \ldots, [j_n, lb_n, ub_n]]$$

which constructs a rank n array for which each element value is defined by the function f, usually instantiated as an expression over the index variables.

While this seems to work well for pure data-parallel constructs (for SIMD target machines such as the CM2), this representation is too weak to represent more general parallelism. Consider four computations A, B, C, and D, with the requirements that A occur before C, and that B occur before C and D; the present primitives can at best express only overly-constrained versions of the requirements, thus losing the ability to explicitly represent the potential parallelism. The partial order in which finite-difference equations must be evaluated is one such example. In general, tree-structured representations cannot capture partial orders (without resorting to some kind of context-sensitivity).

5 Proposed Representation

We are considering using a variant of "Unified Computation Graphs" (UCGs) [WBS+91] to represent programs. Such graphs are based on simple data-flow graphs, with the addition of shared data, control-flow arcs (similar ɔ program dependence graphs [FOW87]) and "exclusion constraints" between nodes. Exclusion constraints ɪ revent two or more parallel activities from simultaneous execution, and are usually associated with access ៶ ɔ overlapping parts of a shared data structure.

In Figure 1, we show computations as bubbles, data- and control-flow as solid arrows, and exclusion dependencies as a dashed arc with no arrows. Primitive computations in bubbles are represented as trees.

UCGs assume global shared data among computations, while pure data-flow assumes no shared data. We have added a representation for data shared among particular nodes, and show the storage shared by two computations with an enclosing dashed arc. Computations not sharing data with others are not necessarily functional; each may still have internal state. Parallel-prefix operations may be represented either by reduction operators over data aggregates, such as the Fortran 90 `SUM` operation, or explicitly via n-ary trees on explicitly represented operands. We currently do not represent pipeline parallelism or multiple simultaneous activations of each operator [AG82].

We mix notations by writing (sub)UCGs isomorphic to purely parallel (respectively sequential) constructs as their textual *par* (respectively *seq*) equivalent.

5.1 Refinements on UCGs

A refinement is a type of transformation that introduces detail (i.e., removes possible models). Several generic refinements of standard UCGs are possible:

R_{comp} Refine a computation into a sub-UCG (Figure 2).

R_{flow} Refine a data-flow carrying a complex data structure into multiple data-flows carrying parts of that structure (composing this with the previous action produces data-parallel computations when data-flow components are homogeneous).

$R_{abstract}$ Group parallelizable computations into a single computation.

R_{merge} Merge a set of parallel computations into a single computation.

$R_{serialize}$ Sequentialize a pair of parallel activities by adding control-flow arc.

$R_{sequence}$ Refine an exclusion constraint arc into a control-flow arc going in either direction.

New refinements possible because of the enriched representation:

R_{store} Refine data-flow between nodes into control-flow plus shared storage.

$R_{coalesce}$ Coalesce several shared storage regions into one.

6 Example

In this section, we sketch the refinement of a domain-specific component into good CM2 code, given our proposed representation.

We present an equivalent Fortran 77 code fragment first, abstracted from a real modeling program, to ensure that the reader initially sees what a conventional compiler sees. A conventional compiler must determine which of these steps can be executed in parallel, knowing nothing of the intent.

```
      DO 100, ix=K+1,K+N
         DO 100, iy=K+1,K+N
100   padarray(ix,iy)=mu(ix-K,iy-K)
      <...lots of unrelated code...>
      DO 128 iy=1,N
         DO 128 ix=1,K
128   padarray(ix,iy) = padarray(K+1,iy)
      DO 129 iy=1,N
         DO 129 ix=N-K+1,N
129   padarray(ix,iy) = padarray(N-K,iy)
      DO 132 ix=1,N
         DO 130 iy=1,K
130      padarray(ix,iy) = padarray(ix,K+1)
         DO 131 iy=N-K+1,N
131      padarray(ix,iy) = padarray(ix,N-K)
```

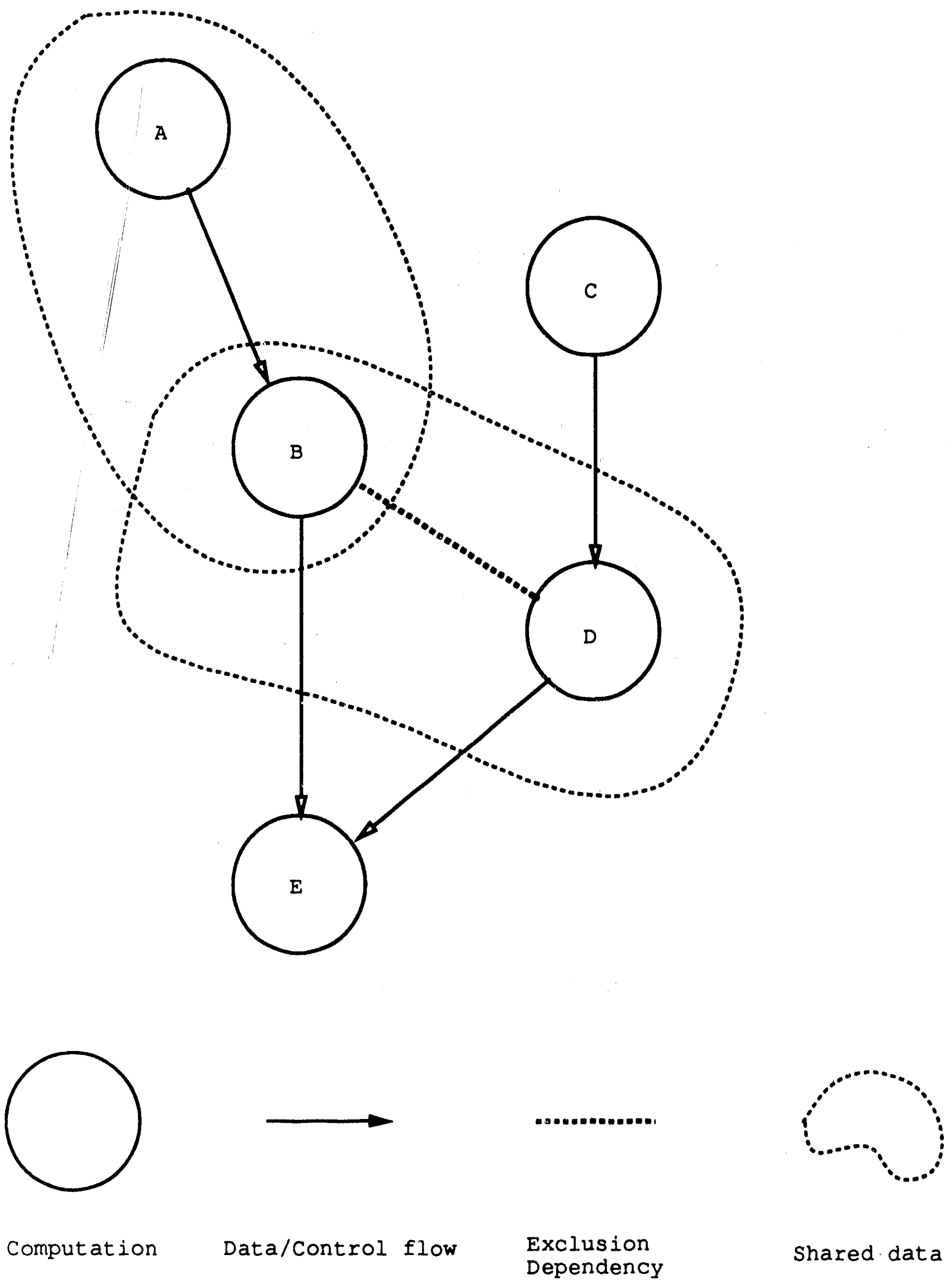


Figure 1: Representation of Parallelism

This code actually pads an $N \times N$ array (**mu**), producing an $(N + 2 * K) \times (N + 2 * K)$ array (**padarray**) with a K-wide "taper" region along all edges. This is a common operation in modeling codes on variables representing properties of space when taper boundary conditions are used [IO81]. The intent is to fill the taper boundary areas with copies of the nearest edge of the original array; a conceptual view of this operation is provided in Figure 3. The resulting array has nine regions. The taper edges are filled with copies of the corresponding edge of the original array; symmetry leads one to the conclusion that the corner regions of the padded array must be filled with values from the closest corner of the array. Each region is defined as the set of elements selected by the cross-product of a particular range of indices; the upper, left hand corner has range $[i, 1, K], [j, 1, K]$, etc.

We give an abstract program schema defining the domain-specific notion Pad(**array**), using case analysis to determine in which region an element resides:

```
Pad[originalarray,K] isSchema
N:=AxisSize(originalarray);
makeArray[
  case[
    1<=i<=K and 1<=j<=K: originalarray[1,1]; (* upper left corner *)
    1<=i<=K and K+1<=j<=K+N: originalarray[1,j-K+1]; (* North side *)
    1<=i<=K and K+N+1<=j<=N+2*K: originalarray[1,N]; (* upper right corner *)
    K+1<=i<=K+N and 1<=j<=K: originalarray[i-K+1,1]; (* West side *)
    K+N+1<=i<=N+2*K and 1<=j<=K: originalarray[N,1]; (* lower left corner *)
    K+N+1<=i<=N+2*K and K<=j<=K+N: originalarray[N,j-K+1]; (* South side *)
    K+N+1<=i<N and K+N+1<=j<=N+2*K: originalarray[N,N]; (* lower right *)
    K+1<=i<=K+N and K+N+1<=j<=N+2*K: originalarray[i-K+1,N]; (* East side *)
    K+1<=i<=K+N and K+1<=j<=K+N: originalarray[i-K+1,j-K+1]; (* middle *)
   ], (* case *)
      [i,1,N+2*K],[j,1,N+2*K]]
```

This schema provides for maximum (data) parallelism; every element can be computed independently, and thus the entire computation takes only $O(1)$ time on an appropriate architecture. It could be used by SINAPSE whenever padding is required; no rediscovery of parallelism is required.

However, the computation would still be unnecessarily inefficient on a SIMD machine such as the CM2, for which a data parallel operation requires all processing elements (PEs) to perform the same instruction and then synchronize. Each PE must synchronously execute the entire **case** statement body. Assuming one machine instruction for each operand and operator, each case requires about 15 instructions, so the nine cases require about 135 instruction times.

We can lower this cost by eliminating runtime evaluation of the case bounds. The cases conveniently define SIMD-compatible partitions of the instruction streams. SINAPSE assumes that a case construct precisely covers its cases, with no overlap; thus all case clauses may be executed in parallel:

```
seq[ N:=AxisSize(originalarray);
     allocate(padarray,N+2*k,N+2*k); (* creates storage for padarray *)
     par[
       doPar[padarray[i,j]:=originalarray[i-K+1,j-K+1],
                           [i,K+1,K+N],[j,K+1,K+N]]; (* middle *)
       doPar[padarray[i,j]:=originalarray[i-K+1,1],
                           [i,K+1,K+N],[j,1,K]]; (* West side *)
       doPar[padarray[i,j]:=originalarray[i-K+1,N],
                           [i,K+1,K+N],[j,K+N+1,N+2*K]]; (* East side *)
       doPar[padarray[i,j]:=originalarray[1,1],
                           [i,1,K],[j,1,K]]; (* upper left corner *)
       doPar[padarray[i,j]:=originalarray[1,j-K+1],
                           [i,1,K],[j,K+1,K+N]]; (* North side *)
```

```
        doPar[padarray[i,j]:=originalarray[1,N],
                            [i,1,K],[j,K+N+1,N+2*K]]; (* upper right corner *)
        doPar[padarray[i,j]:=originalarray[N,1],
                            [i,K+N+1,N+2*K],[j,1,K]]; (* lower left corner *)
        doPar[padarray[i,j]:=originalarray[N,j-K+1],
                            [i,K+N+1,N+2*K],[j,K,K+N]]; (* South side *)
        doPar[padarray[i,j]:=originalarray[N,N],
                            [i,K+N+1, and K+N+1,N+2*K]]; (* lower right *)
      ]; (* par *)
    padarray] (* seq *)
```

Each of these cases now maps directly onto a Fortran 90 array primitive, and each would execute in just a few instructions on a CM2. However, the CM2 has only one set of data-parallel processors, so each "parallel" case competes for the data-parallel processor resource. Static resolution of this resource contention requires serializing access to the set of data-parallel processors, and consequently 9 units of time are actually taken. This can be reduced to 5 (as in the original hand-coded fragment) by combining steps. Consider Figure 4; in stage 1, after copying the original array (1 unit), we expand the copied array along the X-axis (1 unit in both directions); in stage 2, we expand the expanded array along the Y-axis (1 unit in both directions).

To make progress towards this reduction in effort, we apply the following refinements:

- Group ($R_{abstract}$) some parallel activities, with the intention of merging them (R_{merge}), and
- Order ($R_{serialize}$) some parallel activities, to eventually ensure that certain properties are present when needed.

The result is shown in Figure 5.

The activities so grouped can be combined into a single data-parallel primitive. This is because after copying to the center, and filling east and west edges, we have entire edge rows ready to replicate vertically, as shown in Stage 2 of Figure 4. Consequently we can rewrite the three steps:

```
  par[
      doPar[padarray[i,j]:=originalarray[1,1],
                          [i,1,K],[j,1,K]]; (* upper left corner *)
      doPar[padarray[i,j]:=originalarray[1,j-K+1],
                          [i,1,K],[j,K+1,K+N]]; (* North side *)
      doPar[padarray[i,j]:=originalarray[1,N],
                          [i,1,K],[j,K+N+1,N+2*K]]; (* upper right corner *)
     ] (* par *)
```

as the single step:

```
   doPar[padarray[i,j]:=padarray[K,j],
                       [i,1,K],[j,1,N+2*K]]; (* upper edge *)
```

We similarly optimize the code for filling the lower edge.

On the CM2, copying from one array to another is cheap only if the copied array has the same size and alignment in memory as the target. When the source is smaller than the destination, changing the alignment, the communication costs are high (roughly 100 times slower than the aligned case!). We can do little about the cost of copying **originalarray**. However, we need not suffer as great a cost when filling the east and west edges; we can take advantage of the fact that an aligned copy of the east edge of the original array is present in the target array, and copy that instead. This optimization requires that we add additional computation-ordering constraints ($R_{serialize}$) to ensure that copy-original-to-center occurs before filling the east or west edges. Having accomplished that, we can rewrite:

```
doPar[padarray[i,j]:=originalarray[i-K+1,1],
                    [i,K+1,K+N],[j,1,K]]; (* West side *)
```

as:

```
doPar[padarray[i,j]:=padarray[i,K+1],
                    [i,K+1,K+N],[j,1,K]]; (* West side *)
```

Again, we can do the same for the east side, producing a computation in the form shown by Figure 6.

A final equivalence simplifies a set of parallel computations, each of which is connected to all of their descendants, into a simple sequence:

```
seq[  N:=AxisSize(originalarray);
      allocate(padarray,N+2*k,N+2*k); (* creates storage for padarray *)
      doPar[padarray[i,j]:=originalarray[i-K+1,j-K+1],
                           [i,K+1,K+N],[j,K+1,K+N]]; (* middle *)
      par[doPar[padarray[i,j]:=padarray[i,K+1],
                               [i,K+1,K+N],[j,1,K]]; (* West side *)
          doPar[padarray[i,j]:=padarray[i,K+N],
                               [i,K+1,K+N],[j,K+N+1,N+2*K]]; (* East side *)
         ];
      par[doPar[padarray[i,j]:=padarray[K+1,j],
                          [i,1,K],[j,1,N+2*K]]; (* upper edge *)
          doPar[padarray[i,j]:=padarray[K+N,j],
                           [i,K+N+1,N+2*K],[j,1,N+2*K]]; (* lower edge *)
         ];
      padarray] (* seq *)
```

At the present time, SINAPSE represents the **Pad** component in essentially this form, rather than refining it from a more abstract description.

At final code-generation time, we generate code in any order consistent with the partial order over the computations and produce the following CM2 Fortran 90 code:

```
C     copy original array
      padarray(K+1:K+N,K+1:K+N)=mu(1:N,1:N)
C     Fill West edge
      padarray(K+1:K+N,1:K)=SPREAD(padarray(K+1:K+N,K+1),2)
C     Fill East edge
      padarray(K+1:K+N,K+N+1:N+2*K)=SPREAD(padarray(K+1:K+N,K+N,2)
C     Fill upper boundary
      padarray(1:K,1:N)=SPREAD(padarray(K+1,1:N),1)
C     Fill lower boundary
      padarray(K+N+1:N+2*K,1:N)=SPREAD(padarray(K+N,1:N),1)
```

It is interesting to compare this to the original hand-generated code. At no point must we rediscover the parallelism from a complex, highly optimized target language code, as done by conventional compilers. The same efficiency has been achieved from an abstract specification that could be used to generate code for multiple architectures and languages. On a MIMD machine with low communication costs, we could assign one processor per result-array element, and specialize the **case** statement for that processor at synthesis time, providing effectively unit time execution of the padding operation. For a high-communication cost MIMD machine, we might refine the original component into 9 parallel tasks with no shared storage, and a final assembly step.

Since we avoid the discovery process, we also avoid the requirement to harness the often necessary problem domain knowledge to aid this process. Conventional compilers do not have this knowledge, and thus the application programmer must be somehow brought into the process, making compilation partly manual.

7 Lessons

For the CM2, a good strategy for generating code seems to be:

- Represent computations with functional code fragments expressing data parallelism over regions (this should allow us to target other parallel architectures as well).
- Convert the functional fragments to side-effecting fragments over the same regions.
- Reduce operation count by merging parallel data-parallel operations on adjacent regions.
- Reduce communications cost by aligning data.

For data-parallel architectures with regular communication topologies, misaligned data in operations can be very expensive. A model of the communication costs would help to focus attention of the synthesis system on points needing optimization. The actual optimization can be accomplished by copying to an aligned array (creating a singly-assigned temporary variable) and allowing code motion to move the copy step to a point where the copy is only evaluated once.

8 Conclusions

We have found tree-structured representations of parallelism to be overly constraining, and are moving towards representations including partial orders. Such representations will allow us to both directly encode domain-specific components with maximal parallelism, and hence enable us to perform optimization and resource assignment based on the potential parallelism.

SINAPSE is also being enhanced in several other areas. More detailed knowledge about problem domains such as 3D ultrasonic wave propagation is being added. Solution techniques such as finite-element methods as alternatives to finite differencing are contemplated. We are currently adding programming knowledge about multiple target languages. We hope to eventually generate production quality modeling programs for parallel machines.

References

[AG82] Arvind and Kim P. Goestelow. The U-Interpreter. *Computer*, pages 42–49, February 1982.

[BKK+89] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley, and Jaspal Subhlok. The ParaScope Editor: An Interactive Parallel Programming Tool. In *Proceedings of Supercomputing '89*, pages 540–550. ACM Press, November 1989. ACM Order Number 415892.

[Cor90] Pacific-Sierra Research Corporation. *The MIMDizer User's Guide*. Pacific-Sierra Research Corporation, 12340 Santa Monica Blvd, Los Angeles, CA 90025, 1990.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[HS86] W. Daniel Hillis and Guy L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1184, December 1986.

[IO81] Moshe Israeli and Steven Orszag. Approximation of Radiation Boundary Conditions. *Journal of Computational Physics*, 41:115–135, 1981.

[KDMW90] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Automated Synthesis of Finite Difference Programs. In *Symbolic Computations and Their Impact on Mechanics, PVP-Volume 205.* The American Society of Mechanical Engineers 1990, New York, NY, 1990. ISBN 0-791800598-0.

[KDMW91] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Scientific Programming by Automated Synthesis. In M. Lowry and R. McCartney, editors, *Automating Software Design.* AAAI Press, 1991. To appear.

[SMD+89] K. Sridharan, M. McShea, C. Denton, B.Eventoff, J. C. Browne, P. Newton, M. Ellis, G. Grossbard, T. Wise, and D. Clemmer. An Environment for Parallel Structuring of Fortran Programs. In E.C. Plachy and P.M. Kogge, editors, *Proceedings of 1989 International Conference on Parallel Processing*, pages 98–106, 215 Wagner Building, University Park, PA 16802, August 1989. The Penn State Press.

[WBS+91] John Werth, James C. Browne, Steve Sobek, T. J. Lee, Peter Newton, and Ravi Jain. The Interaction of the Formal and the Practical in Parallel Programming Environment Development: CODE. Technical Report TR-91-09, Department of Computer Science, University of Texas at Austin, April 1991.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addision-Wesley Publishing Company, Inc., Reading, Massachusetts, 1991. Second Edition.

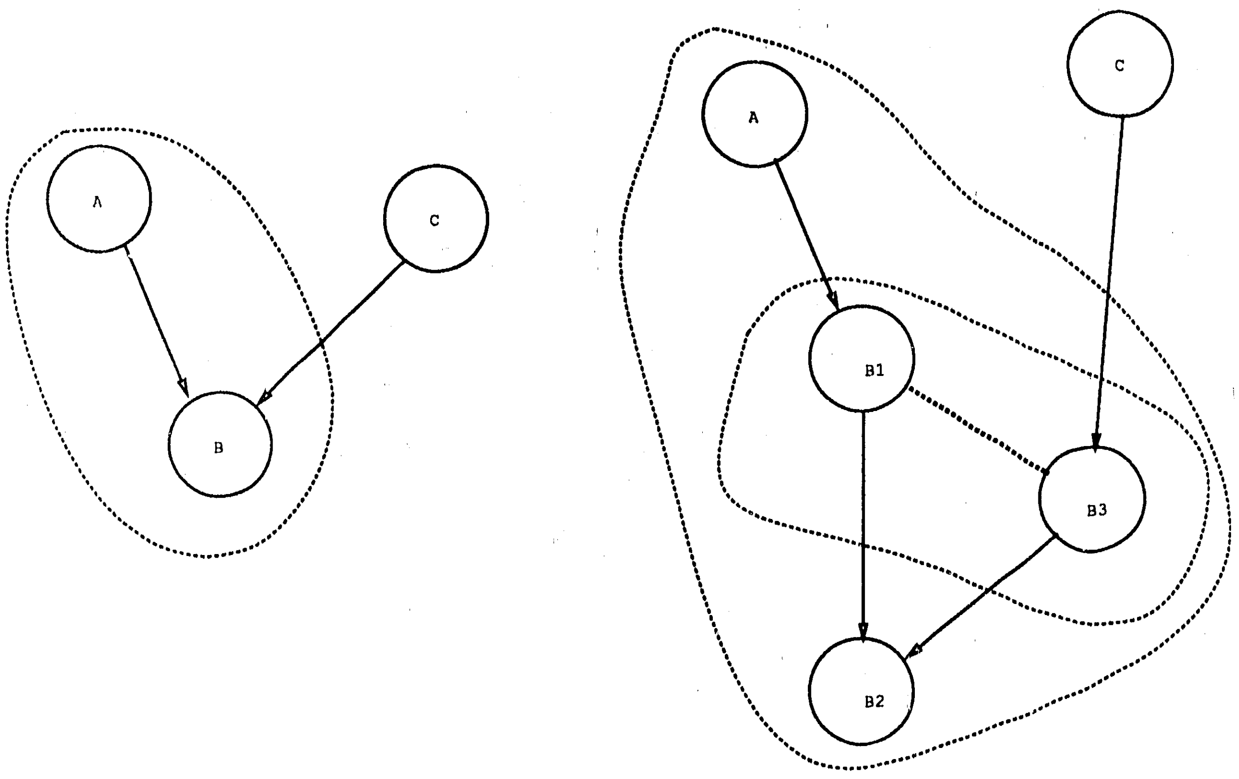


Figure 2: Refining node B

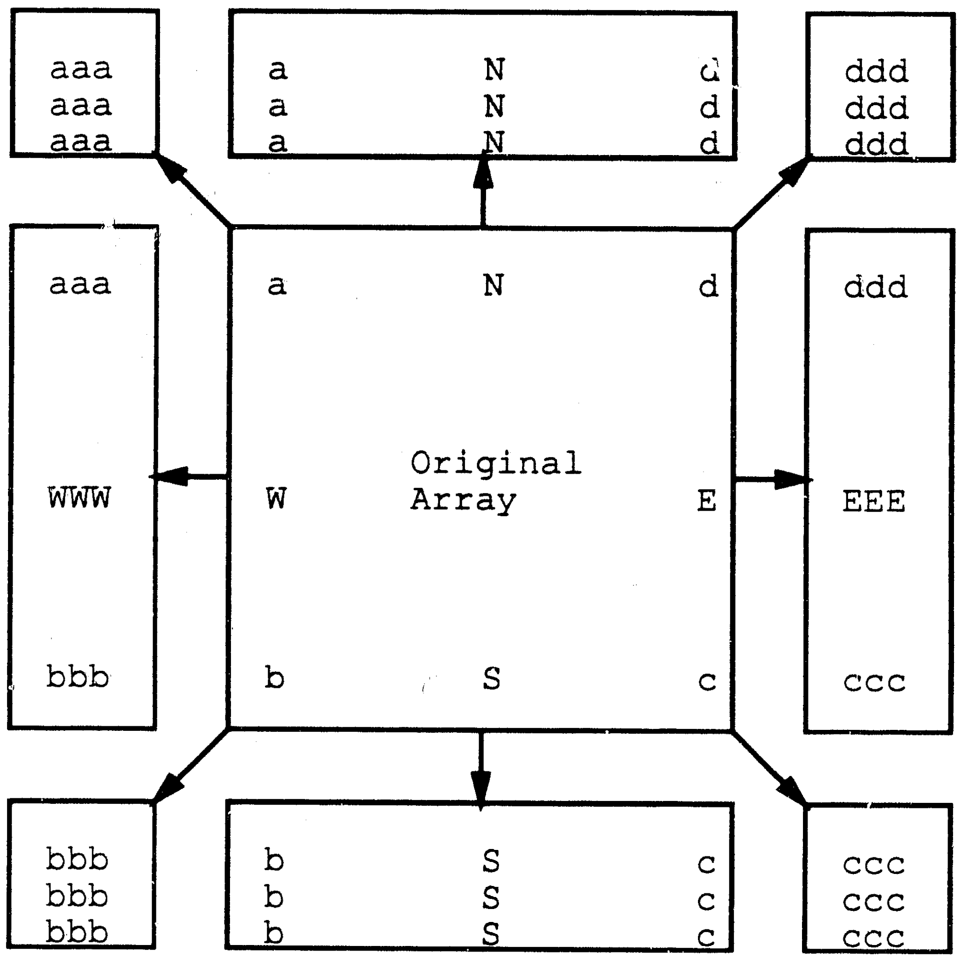
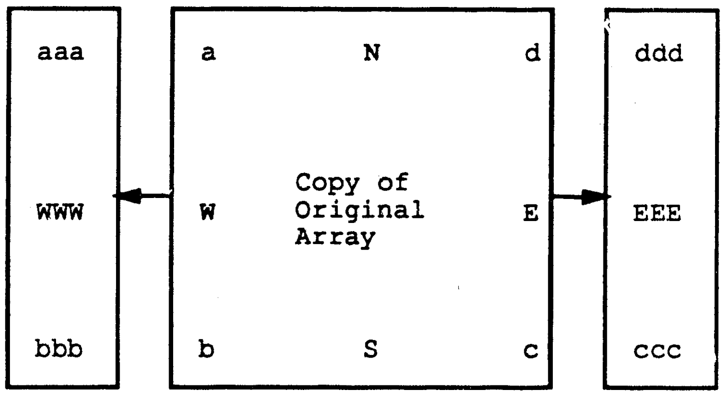
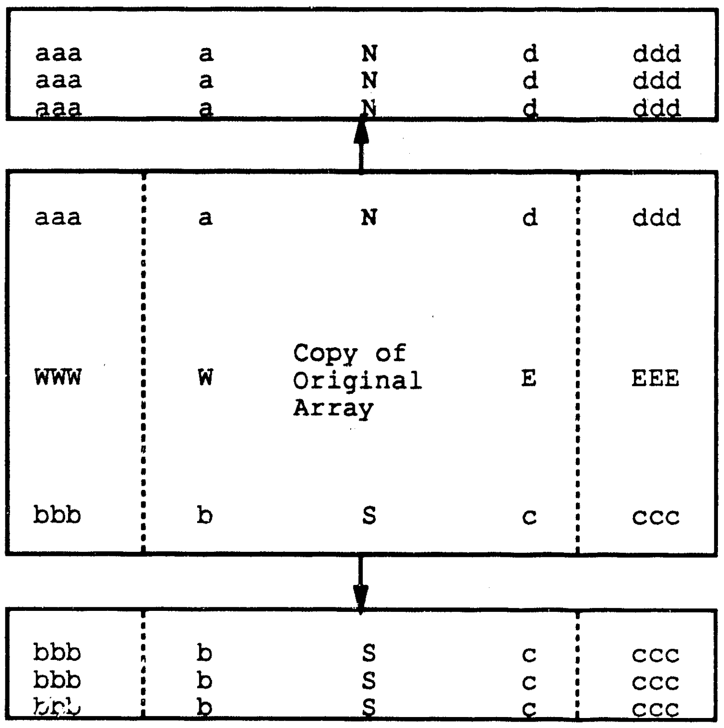


Figure 3: Padding an array by filling new boundaries with copies of edges



Stage 1



Stage 2

Figure 4: Padding operation optimized for Connection Machine

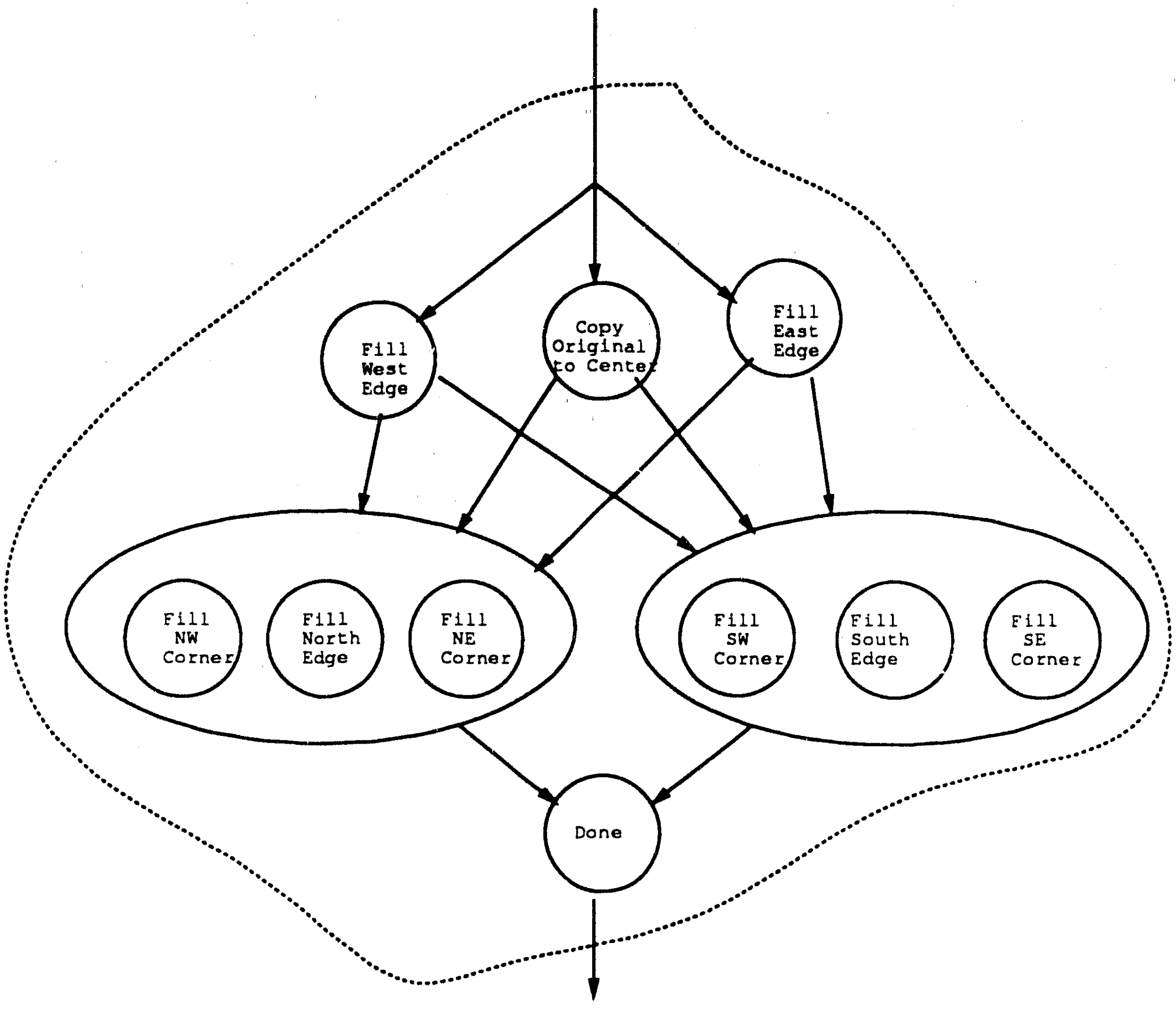


Figure 5: Refining parallel padding

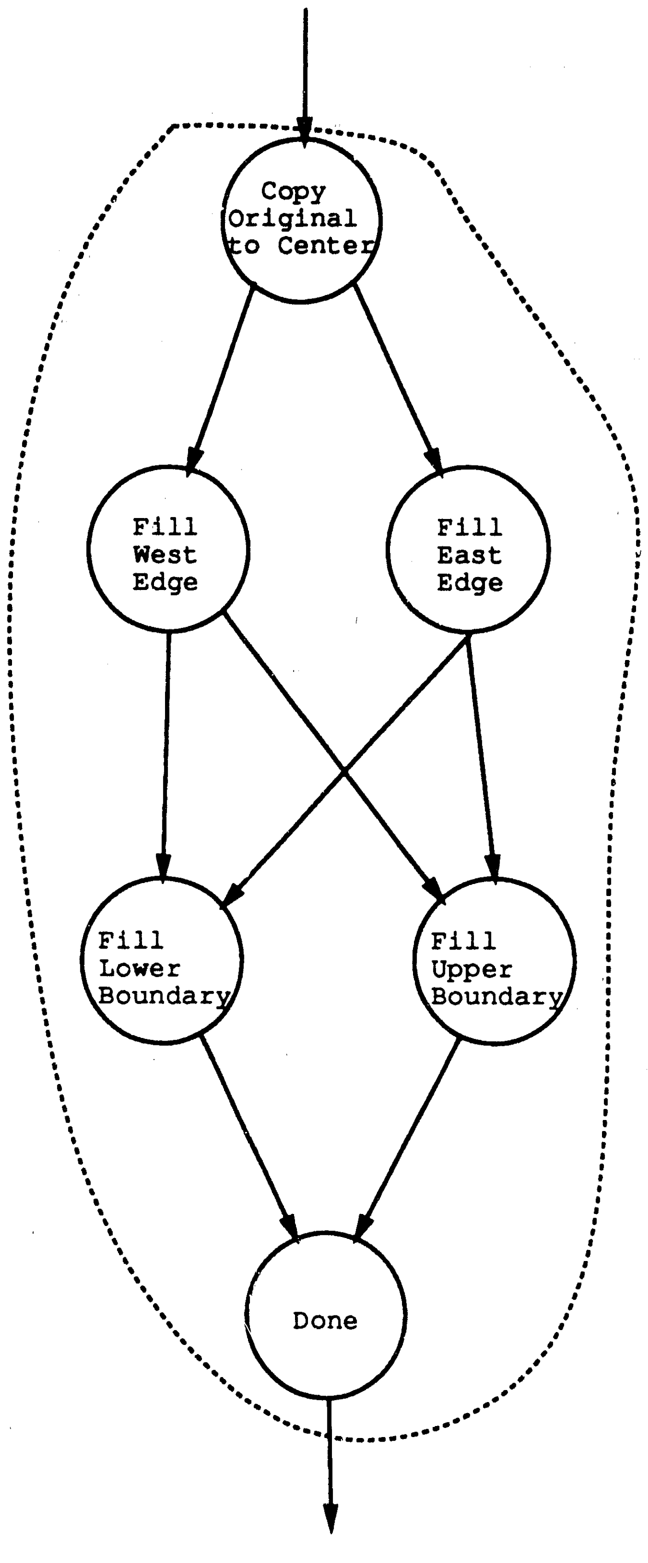


Figure 6: Final representation of padding for data parallel machine