Partitioning Regular Applications for Cache-Coherent Multiprocessors

Karen A. Tomko and Santosh G. Abraham

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
Email: karent@eecs.umich.edu, sga@eecs.umich.edu
Phone: (313)936-2917
Fax: (313)763-4617

ABSTRACT

In all massively parallel systems (MPPs), whether message-passing or shared-address space, the memory is physically distributed for scalability and the latency of accessing remote data is orders of magnitude higher than the processor cycle time. Therefore, the programmer/compiler must not only identify parallelism but also specify the distribution of data among the processor memories in order to obtain reasonable efficiency. Shared-address MPPs provide an easier paradigm for programmers than message passing systems since the communication is automatically handled by the hardware and/or operating system. However, it is just as important to optimize the communication in shared-address systems if high performance is to be achieved. Since communication is implied by the data layout and data reference pattern of the application, the data layout scheme and data access pattern must be controlled by the compiler in order to optimize communication. Machine specific parameters, such as cache size and cache line size, describing the memory hierarchy of the shared-address space machine must be used to tailor the optimization of the application to the memory hierarchy of the MPP.

This report focuses on a partitioning methodology to optimize application performance on cache-coherent multiprocessors. We give an algorithm for choosing block-cyclic partitions for scientific programs with regular data structures such as dense linear algebra applications and PDE solvers. We provide algorithms to compute the cache state on exiting a parallel region given the cache state on entry; and methods to compute the overall cache-coherency traffic and choose block-cyclic parameters to optimize cache-coherency traffic. Our approach is demonstrated on two applications. We show that the optimal partition computed by our algorithm matches the experimentally observed optimum and we show the effect of cache line size on partition performance.

1 Introduction

In this report, we present a compile-time data distribution algorithm for shared address space multiprocessors to minimize cache coherency traffic. The algorithm partitions regular programs having linear array subscript expressions into block cyclic partitions such that cache coherency traffic is minimized. Applications that perform dense linear algebra computations or solve partial differential equations using relaxation steps on regular grids are examples of regular programs. The partition algorithm is a five step algorithm which uses compiler techniques from data flow analysis to determine the cache state for each loop nest in the program. Changes in cache state are used to calculate the coherency traffic for the program. Def/use analysis is performed to determine the array sections that are referenced in each loop nest. The array sections are summarized using *cyclic regular section descriptors* (CRSDs) which are an extension of an array summary technique used by the Rice compiler group. Unlike previous descriptors, CRSDs are sufficiently powerful to represent array subset accesses induced by block-cyclic partitions.

We provide a data flow algorithm which generates the *def* and *use* data sets for distributed array data. We characterize the state of each processor's cache by classifying data as being in one of three states: shared, invalid, exclusive. Then we calculate the input and output cache footprints of each state for all of the parallel loop nests in a program. The size of the difference between output and input cache footprints is used to estimate the cache-coherency traffic and hence the communication cost. We also show how the cost function can be optimized by choosing the distribution parameters. We present experimental results from the KSR1 on two sample parallel programs which show that we are correctly estimating the cache-coherency traffic and choosing the best partitions for the sample programs.

This report is organized into the following sections. First we discuss related work and describe our machine and program models. Then we give an overview or our algorithm, background material, and the details of our algorithm. We follow with experimental results and finally a summary of our contributions.

2 Related Work

The methods described in this report draw primarily on two aspects of parallel processing research: performance analysis and compilation of applications with regular data structures.

2.1 Performance Analysis

Our work on partitioning regular applications uses simple performance models to guide the choice of data and work load partitioning of applications. Many researchers have developed tools for performance analysis of shared memory MPPs. We describe their work below, and the limitations of the methodologies for our problem.

Fahringer and Zima [14] have developed a performance prediction tool (PPPT) which statically computes many parameters that can be used to calculate parallel program performance. Among the parameters that they calculate are the amount of data transferred between processors and number of uniprocessor cache misses. The amount of data transferred is determined using polytopes to represent array sections in the program and calculating the area of the polytopes. Their work assumes that a block data distribution is specified by the programmer and that communication directives have been inserted (by their compiler VFCS). Like our work the authors calculate the amount of data transferred between processors. However, PPPT supports only block data distributions, a subset of the block-cyclic partitions that we support.

Balasundaram *et al.* [6] statically evaluate the performance of different partitioning schemes for

programs executed on distributed memory multiprocessors. They assume that a rectangular block data partitioning is specified and that communication primitives have been inserted in the code. A *training set* is used to characterize the relative performance of communication and computation on a particular machine. Similarly, we evaluate the performance of different partitioning schemes for an application program, and then determine which scheme is optimal by our criterion. We support a larger space of partitioning schemes than Balasundaram *et al.* who do not handle block-cyclic partitions.

Atapattu and Gannon [5] have created a performance prediction tool for the Alliant FX/8, which has a single shared memory and shared cache. The tool performs assembly level analysis of vector and scalar code for the Alliant. The shared memory is modeled with a simple queuing model. The miss rate is supplied by the user but can be calculated using the reference window work described in [15, 13]. Our work performs source level analysis of parallelized code for the KSR1. Like the Alliant FX/8, the KSR1 is a shared memory machine; unlike the FX/8, processors in the KSR1 have private caches. We estimate coherency cache miss rate based on program analysis.

Larus *et al.* [24] propose the check-in, check-out (CICO) performance model for cache-coherent shared-memory parallel computers. Similar to our work the authors estimate the cost of coherency traffic. However, they rely on user annotations not program analysis to determine the state of a cache block. Their coherency model is similar to ours, they model three states (idle, shared, and exclusive) and they do not model contention or conflict misses.

2.2 Compilation of Regular Applications

There are several compiler projects for regular parallel applications, some of the most important projects are FORTRAN D [20, 21], High Performance FORTRAN [19], Vienna FORTRAN [33], SUIF [32, 4], Crystal [25, 12], and PARADIGM [30, 16]. These groups have done work in data partitioning, interprocedural analysis and message generation which we draw on in our research. In addition to these projects there are some smaller groups that have also done work relevant to our own.

Automatic data partitioning algorithms which minimize coherency traffic have also been developed by Hudak and Abraham [22, 1] and by Agarwal, Kranz and Natarajan [2]. Hudak and Abraham have developed automatic partitioning techniques for regular data-parallel loops with array accesses that have unit-coefficient linear subscripts. Agarwal, Kranz and Natarajan generate optimal block partitions for cache-coherent multiprocessors. They generalize the program model to handle any array index expressions that are affine functions of loop indices. The data footprints for array references are calculated and combined to determine the cache usage. A partition is chosen which minimizes that footprint in the cache. An approximation is used to combine data footprints for references having different strides. Like [2], we support array index expressions that are affine functions of loop indices, in addition we support block-cyclic data partitions which they do not handle.

Heuristic techniques for automatic data partitioning have been developed as part of the PARADIGM compiler. The compiler calculates constraints and cost functions for the loops in the program which are combined into a quality measure that is used to choose a data distribution. Once the data distribution has been specified another module generates the send and receive communication sets for a message passing machine. The heuristics appear to perform well. However, PARADIGM is a message-passing compiler and it does not necessarily generate partitions which minimize coherency traffic. We have developed a partitioning algorithm with the goal of minimizing coherency traffic. Unlike, PARADIGM, our model is not heuristic, it is optimal under our programming and machine model.

Our algorithm for generating block-cyclic data partitions use array summary methods similar to those developed for interprocedural analysis. Array summary methods are methods to represent the set of array elements accessed within a region of code. The following array summary techniques have been proposed:

Regions by Triolet *et al* [28], Linearization by Burke and Cytron [9], Atom Images by Li and Yew [26], and Data Access Descriptors by Balasundaram and Kennedy [7]. Our *cyclic regular section descriptor* is an extension to *regular section descriptors* used at Rice University for interprocedural analysis [10, 18] as part of the Fortran D project. We propose adding an additional parameter to the RSD notation in order to represent the subarray assigned to processor when a block cyclic data distribution is used. We use CRSDs for analysis of communication cost due to distributed arrays on a cache coherent multiprocessor. The PARADIGM research group uses a summary notation similar to ours for generating message send and receive sets for block-cyclic data distributions.

Message generation while not directly applicable to cache-coherent multiprocessors is also of interest to us. Message generation requires determining exactly what data must be communicated between processors and identifying the sending and receiving processors. Similarly, we need to determine the amount of data that must be communicated between processors for a given set of partition parameters. Gupta, *et al.* [17] provide closed form solutions for the generation of communication sets for distributed-memory machines. They use a virtual processor approach to find the communication sets for block-cyclic distributions. The communication sets are efficiently generated at run time for a user (or compiler) specified data distribution. This solution to message generation requires that the block-cyclic parameters be completely specified to enumerate the communication requirements of the partition. Therefore we cannot use the method directly for generating the parameterized cost function used by our optimization procedure. However, we may be able to incorporate a virtual processor approach in our algorithm.

3 Machine Model

We use a shared address space MIMD multiprocessor as our machine model and assume a hardware-coherent memory hierarchy consisting of local cache, possibly a local memory, and remote memory where the latency to access remote memory is on the order of hundreds of processor cycles. Remote memory consists of the memory or cache storage local to other processors and communication between processors occurs when remote memory is accessed. This model covers a wide spectrum of machine types including distributed shared memory architectures such as the Stanford DASH, bus based multiprocessors such as the Sequent Symmetry and SGI Challenge Series, and cache only memory architectures (COMA) like the Kendall Square Research KSR1 and KSR2.

Hardware-coherent multiprocessors require a mechanism to maintain the status of data within the local cache of a processor. For a given unit of storage in the cache, valid/invalid and exclusive/shared information must be recorded. The valid/invalid field specifies whether the local cache contains a valid copy of the data for a given data item. If the data is valid, the exclusive/shared field specifies whether the local processor owns the data in exclusive mode or shared mode. A data item will be in shared mode if it is read but not updated. Several processors may have a copy of the same data item in shared mode. The data must be owned in exclusive mode in order to be updated. A state diagram of the coherency protocol for the KSR1 is given in Figure 3.

3.1 KSR1 Architecture

The Kendall Square Research KSR1 was used to evaluate our methods. We give a brief description of the architecture here, much of which has been taken from [31, 8]. The KSR1 is characterized by a hierarchical ring interconnection network and cache-only memory architecture. Each cell, consisting of a 20 megahertz processor, a 512 kilobyte subcache, and a 32 megabyte local cache, is connected to a unidirectional pipelined slotted ring. Up to 32 processors may be connected to each ring and multiple rings may be connected in a hierarchy of rings. We ran our experiments on a 64 processor two ring

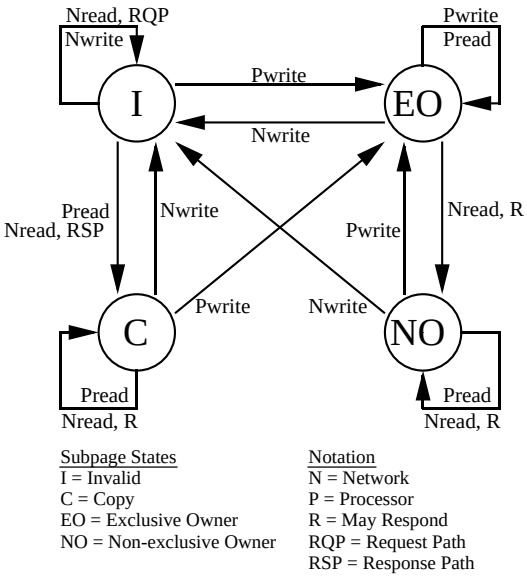

Figure 1: KSR1 Local Cache Coherency Protocol
Reproduced from [31]

system. The KSR architecture is diagramed in Figure 3.1.

The size of a local cache subblock, called a subpage, is 128 bytes and serves as the unit of transfer between processors. Communication requests by any processor proceed around the ring in the direction of ring communication. Such requests are viewed by all processors as they pass by, enabling the hardware cache management system to maintain memory coherency. Given a P processor system, when a processor i makes a read request on the ring, the first processor encountered which has a valid copy of the subpage, processor j, will respond by placing a copy of the subpage on the ring. Every processor encountered between j and i which contains an invalid copy of the subpage can optionally update its copy automatically as the request passes on its return path to the requesting processor i (referred to as automatic update). Likewise, if a processor needs to write to a shared copy of a subpage, it must send a transaction around the ring requesting that each processor with a copy of the subpage in its local cache mark the subpage as invalid.

A shared-address space multiprocessor such as the KSR1 provides a simple programming paradigm to the user. There is no need to explicitly assign data to processors and determine when data should be sent or received from another processor. However, because remote data accesses may take an order of magnitude longer than local data accesses, high performance can only be achieved when remote accesses are minimal. On the KSR1 remote accesses take 135-175 processor cycles within a single ring of the system (Table 3.1). Remote accesses occur implicitly whenever a new or an invalid data item is referenced by the processor and whenever a shared data item is written by the processor.

3.2 Program Model

We assume a data parallel programming model with the parallelism expressed as DOALL loops. The DOALL loops may not contain any cross processor dependencies because the dependencies would introduce nondeterministic program behavior for a cache coherent multiprocessor. The input program may be

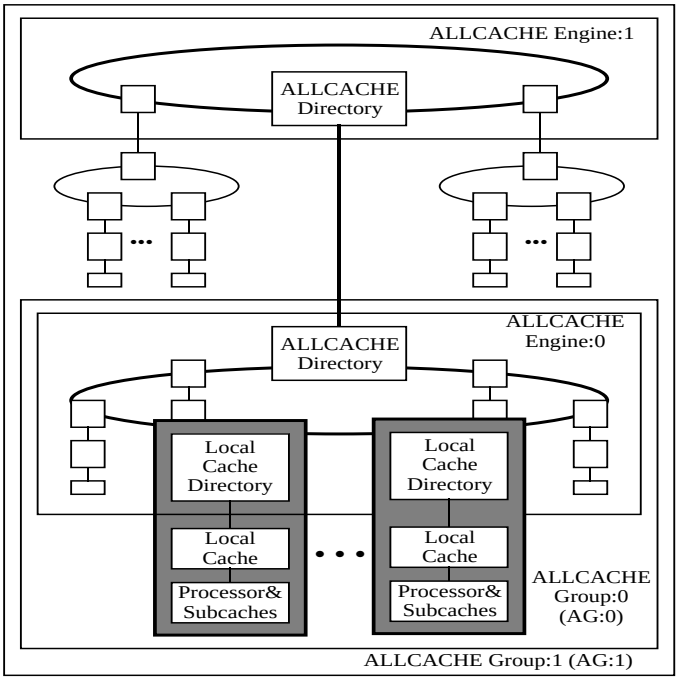

Figure 2: KSR1 Communication Hierarchy
Reproduced from [31]

Memory Component	**Memory Size (Mbytes)**	**Memory Access Read (Cycles)**
Each Subcache	0.25	2 (1 per clock)
Local Cache (existing page)	32.0	
(allocated block)		23.4
(unallocated block)		49.2
Remote Cache	32.0 each	
(allocated page AE:0)	(1024 total)	135-175
(allocated page AE:1)	(34816 total)	470-600

Table 1: Memory Size and Read Access Latencies of the KSR1
Reproduced from [31]

```
program JACOBI

real a(n, m), b(n, m)
align a(i, j) with b(i, j)

do k = 1, 1000
    doall j = 2, m − 1
        doall i = 2, n − 1
            a(i, j) = (b(i − 1, j) + b(i + 1, j) + b(i, j − 1) + b(i, j + 1))/4
        enddoall
    enddoall
    doall j = 2, m − 1
        doall i = 2, n − 1
            b(i, j) = a(i, j)
        enddoall
    enddoall
enddo
end
```

Figure 3: Jacobi Example

comprised of DO loops, DOALL loops and assignment statements. Any ordering of these statements are allowed with the following restriction: any DOALL loops occurring in the same subtree of the program control flow tree are perfectly nested with respect to each other. This restriction is imposed so that the program is either running sequentially or running in parallel and is running on the entire processor set when running in parallel.

We use CRSDs to summarize the array sections accessed within the DOALL loop nest. The following assumptions regarding the array subscript expressions and loop bounds within the DOALL loop nest ensure that we can represent the array accesses precisely with CRSDs. Within a DOALL loop nest the expressions for the loop bounds are of the form $b \cdot k + c$ where k is the index of the innermost sequential DO loop enclosing the DOALL loop nest and b, c are invariant with respect to the innermost sequential DO loop and any enclosing DOALL loops. Array subscript expressions are of the form $a \cdot i + b \cdot k + c$ where i is a DOALL loop index variable, a is an expression which is invariant with respect to the innermost sequential DO loop and any enclosing DOALL loops, and b, c, k are as described above.

This model allows multiple array variables, triangular or trapezoidal iteration spaces, nonperfect nesting of DO loops and sequential DO loops within DOALL loop nests. Figure 3 shows an example of a program that fits our model. Our program model encompasses a wider class of loops than presented by Chen and Hu in [12] or Abraham and Hudak [23], who assume that all loops are perfectly nested and that DO loops can not occur within DOALL nests. It is also a wider model than presented by Agarwal *et al.*, who consider only a single loop nest and assume that the iteration space is rectangular.

4 Partition Generation

As described in the Section 3 the cache state for our machine model has three components: data owned in exclusive mode, data owned in shared mode and data in invalid mode. The cache state for a processor can be represented with three sets ($outX$, $outS$ and $outI$) and coherency cache traffic can

PARTITION

1) Generate control flow tree (CFT) for program
2) Prune sequential branches and mark parallel nodes of CFT
3) Determine def/use sets for parallel nodes in CFT
4) Determine cache state and cost function for CFT
5) Optimize cost function generated in step (4)

Figure 4: Outline of Partitioning Algorithm

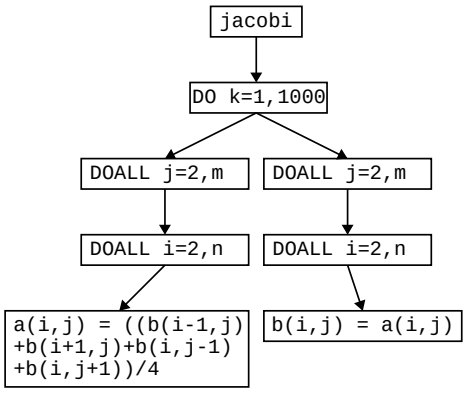

Figure 5: Control Flow Tree for Jacobi

be calculated based on the changes in the cache state variables. Our partitioning algorithm calculates the cache state for each statement and loop nest in the program and calculates a coherency traffic cost function which is optimized to determine the appropriate partition for the application. An outline of our algorithm is given in Figure 4

In order to describe the details of our algorithm we must define some background material and notation used in this section. First we describe the control flow tree representation of a program. Then we describe definition and use sets for the statements of a program as well as cyclic regular section descriptors used to summarize array regions in the definition and use sets. Next we describe the state transition equations for the cache state variables. Finally we give the details of our algorithm.

4.1 Control Flow Trees

The *control flow tree* (CFT) represents the control hierarchy of the program and contains four types of nodes: DO loop nodes, DOALL loop nodes, assignment nodes and a program node. Loop nodes may have any number of children which represent the body of the loop. The children may be loop nodes or assignment nodes where the leftmost child is the first statement in the loop body and the rightmost child is the last statement in the body. An assignment node represents an assignment statement and contains array subscript expressions for the left hand and right hand sides of the assignment. The program node is the root of the control tree and like a loop node may have any number of children. The children may be loop nodes or assignment nodes and the nodes are in sequential program order from left most child to right most child. The control flow tree for the Jacobi example is given in figure 5.

The data structure for each node of the control tree contains several fields. The fields which define the

structure of the control flow tree are: *type* which is set to one of DO, DOALL, ASSIGNMENT or PROGRAM; *leftchild* which points to the first node in the list of children nodes, *rightchild* which points to the last node in the list of children nodes, and *next* which points to the right sibling of a node. *Parallel* is a flag which is set to TRUE if a node is executed in parallel or FALSE if it is executed sequentially. Loop nodes have the fields *index*, *lb*, and *ub* which are the loop index variable, loop lower bound and loop upper bound respectively. In addition each node has the fields def, DEF, *use*, *outI*, *outS*, *outX* and *cost* which are used to determine cache state and calculate coherency traffic cost. These fields are described in more detail below.

4.2 Definition/Use Analysis

Definition and use analysis is standard in today's compilers. A value is *defined* when it is written, (i.e. whenever it appears in the lhs of an assignment statement) and used whenever it is read (i.e. whenever it appears in the rhs of an assignment statement). Definition and use analysis (Def/use analysis) can be applied either to variables or live values. For cache optimization we are concerned with the memory locations themselves so our def/use analysis is on variables.

In our partitioning algorithm, array definition and array use information is stored in three variables def, use and DEF for each node of the control tree. For a given node in the tree, $def[node]$ is a list of CRSDs, one CRSD for each array variable defined in the subtree rooted at *node*, and $use[node]$ is a list of CRSDs for arrays read by the subtree rooted at *node*. The variables def and *use* represent the array sections referenced by a single processor. The variable DEF represents the data that is defined on any of the processors and is the union of def for all processors.

4.3 Cyclic Regular Section Descriptors

We use *cyclic regular section descriptors* (CRSDs) to summarize the def/use and cache state array sections used by our algorithm. CRSDs are a generalization of RSDs defined by Havlak and Kennedy in [18]. RSDs have the same format and meaning as the triplet notation used in FORTRAN 90. An RSD is comprised of three fields for each dimension of an array: a lower bound, an upper bound, and a stride. As such, RSDs can precisely represent arrays that are block distributed or cyclically distributed across processors. However, it is not possible to represent the more general block cyclic distribution with a single RSD. A cyclic distribution assigns every pth element to the same processor. A block cyclic distribution divides the data space in to blocks and assigns every pth block to the same processor. We have extended the regular section descriptor defined in [18] to include a repeat field in order to represent block cyclic data distributions.

The CRSD is represented by a 4-tuple for each dimension i of the array, $[lb_i : ub_i : s_i : r_i]$, where lb_i is the lower bound, ub_i is the upper bound, s_i is the stride and r_i is the repeat value for dimension i. If r_i is not specified then it defaults to a value of 1. A CRSD with $r_i = 1$ is equivalent to an RSD of the form $[lb_i : ub_i : s_i]$. Figure 6 diagrams some block cyclic regular sections. The CRSDs corresponding to the figure are A=[1:20:5:2], B=[2:20:5:2], C=[2:20:5:1], D=[1:20:5:3], and E=[1:20:5:1]. C and E can be represented as the RSDs C=[2:20:5] and E=[1:20:5].

In our use of CRSDs we need the following operators: intersection ($\cap$), union ($\cup$), difference ($-$), and sizeof. The operands for the binary operators are restricted to CRSDs with the same stride. This corresponds to having one cyclic partition for all arrays in the program. CRSDs are not closed under intersection, union or difference. These binary operations applied to any two CRSDs may produce a list of CRSDs. In this report we use the term CSRD to refer to a single descriptor or a list of descriptors representing array sections of a single variable.

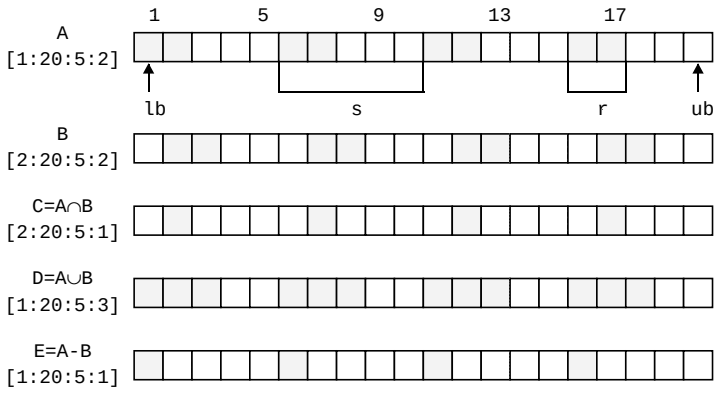

Figure 6: Cyclic Regular Sections

4.4 CFT Generation and Initialization

The first step in the partition algorithm is the generation of the control flow tree (CFT) for the program. CFT generation is performed in most compilers during parsing so we do not describe it here [3]. The second step of the algorithm performs two functions, pruning sequential branches and marking nodes as parallel or not parallel. We prune sequential branches of the tree because we assume that the computationally significant portions of the code are parallelized and because the amount of data that must be gathered and then scattered for sequential regions does not vary with the shape of the partition. Nodes are marked as parallel or not parallel for use by the def/use and cache state algorithms. A node in the CFT is parallel if it is a DOALL node or has an ancestor that is a DOALL node. A recursive descent of the call tree can be used to mark the parallel nodes. The nodes marked as parallel by this method are shaded in the Figure 8.

4.5 Calculating Def/Use

The DEF_USE_ANALYSIS algorithm traverses the CFT from leaves to root. For each node marked as parallel it generates $def[node]$, $use[node]$ and $DEF[node]$. The algorithm is given in Figure 7. At the leaves of the tree the nodes are all assignment nodes, and the def,use, and DEF are simply set to the array references within the assignment statement. The routine MAKE_CRSD() converts the array subscript expressions of the program into CRSD notation. For example, the call MAKE_CRSD($a(i,j)$) returns $a[i:i:1:1, j:j:1:1]$.

For each do loop, the def/use sets of all the children nodes are unioned together to form one CRSD which is then expanded to summarize the array references for all iterations of the loop. The routine EXPAND_CRSD() modifies the def/use CRSDs by updating the lower bound, upper bound, stride and repeat values of the CRSD in the dimension indexed by the index variable of the do loop with the appropriate loop bounds. The routine takes the parameters: CRSD, index, lower bound, upper bound, stride and repeat. For example, the call EXPAND_CRSD($a[i:i:1:1, j:j:1:1], i, 1, n, 1, 1$) returns $a[1:n:1:1, j:j:1:1]$. The expanded CRSD summarizes all array references for the subtree of the CFT which is rooted at the loop node.

Similarly, for each doall loop node the def/use sets of the children are combined and then expanded to summarize the array references for the entire loop. However, since the loop is parallelized, the CRSD only represents the iterations assigned to a single processor. Partition parameters along with the loop bounds are used to specify the iterations assigned to a processor under a general block cyclic partition. The

DEF_USE_ANALYSIS

1) Perform a bottom up traversal of CFT
2) Generate def/use sets for each parallel node of CFT

DEF_USE_SETS ($node$)

if ($type[node]$ = ASSIGNMENT) **then**
$def[node] \leftarrow$ MAKE_CRSD($lhs[node]$)
$DEF[node] \leftarrow$ MAKE_CRSD($lhs[node]$)
$use[node] \leftarrow$ MAKE_CRSD($rhs[node]$)

else if ($type[node]$ = DO) **then**
$def[node] \leftarrow \bigcup_{children} def[child]$
$DEF[node] \leftarrow \bigcup_{children} DEF[child]$
$use[node] \leftarrow \bigcup_{children} use[child]$
$def[node] \leftarrow$ EXPAND_CRSD ($def[node], index[node],$
$lb[node], ub[node],$ STRIDE_ONE, REPEAT_ONE)
$DEF[node] \leftarrow$ EXPAND_CRSD ($DEF[node], index[node],$
$lb[node], ub[node],$ STRIDE_ONE, REPEAT_ONE)
$use[node] \leftarrow$ EXPAND_CRSD ($use[node], index[node],$
$lb[node], ub[node],$ STRIDE_ONE, REPEAT_ONE)

else if ($type[node]$ = DOALL) **then**
$def[node] \leftarrow \bigcup_{children} def[child]$
$DEF[node] \leftarrow \bigcup_{children} DEF[child]$
$use[node] \leftarrow \bigcup_{children} use[child]$
$def[node] \leftarrow$ EXPAND_CRSD ($def[node], index[node], pid_{dim} \cdot block\_size_{dim} + lb[node],$
$ub[node], p\_dim \cdot block\_size_{dim}, block\_size_{dim}$)
$DEF[node] \leftarrow$ EXPAND_CRSD ($DEF[node], index[node],$
$lb[node], ub[node], stride\_one, repeat\_one$)
$use[node] \leftarrow$ EXPAND_CRSD ($use[node], index[node], pid_{dim} \cdot block\_size_{dim} + lb[node],$
$ub[node], p\_dim \cdot block\_size_{dim}, block\_size_{dim}$)

Figure 7: Def/Use Analysis Algorithm

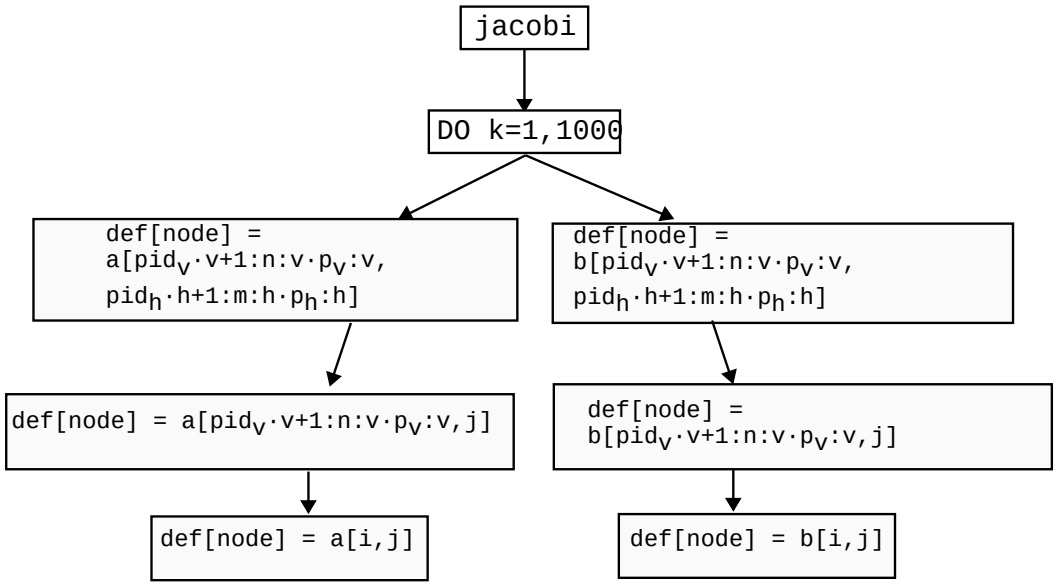

Figure 8: Control Flow Tree for Jacobi annotated with $def[node]$

partition parameters used are: p_v, p_h - the number of processors in the vertical and horizontal dimension respectively; v, h - the block sizes in the vertical and horizontal dimensions respectively; pid_v = (processor id) / p_v; pid_h = (processor id) mod p_h. Note that def/use sets are parameterized with the processor ids (pid_v,pid_h); Though p processors generate p different def/use sets, all of the sets are represented by a single CRSD parameterized by pid. Also note that we use the two-dimensional block-cyclic notation throughout this paper in order to minimize notational complexity. In Figure 8, $def[node]$ of the innermost DOALL loops have been expanded in the dimension accessed by index variable i. The definition sets for the other nodes of the Jacobi call tree are also given.

4.6 Calculating Cache State and Coherency Traffic Cost

The CACHE_STATE_ANALYSIS algorithm traverses the CFT in a depth first manner stopping at the highest DOALL node in each branch. For each DOALL node it calculates the cache state and coherency cost as a function of the output cache state of the previous node and the def/use information of the DOALL node. As the algorithm traverses back up the tree it calculates the communication cost for the subtree rooted at *node* by summing the cost of the children and summing the cost for all iterations of the loops.

4.6.1 Cache State Flow Equations

We represent the cache state for a processor with three variables ($outX$, $outS$ and $outI$) at each node in the control tree, where i is the processor id parameter. $outX[node]$ is a list of CRSDs summarizing the array data elements that are owned in exclusive mode after executing the current node of the control tree. Likewise, $outS[node]$ and $outI[node]$ are lists of CRSDs that summarize the array data owned in shared mode and in invalid mode respectively after executing the current node. Any particular data element is either in exclusive mode, shared mode, or is invalid in the cache; therefore $outX$, $outS$ and $outI$ are disjoint array sections. Figure 9 diagrams the possible transitions between cache states. The data flow

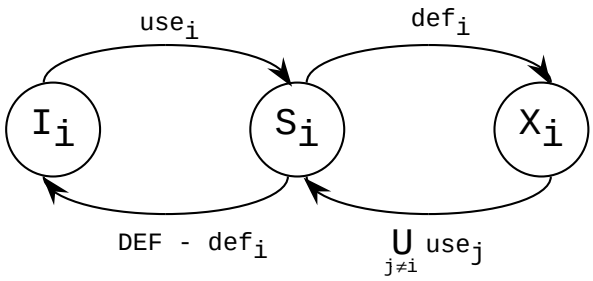

Figure 9: Cache State Transition Diagram for Processor i

formulas for calculating cache state are given by equations (4.1) through (4.5).

$$recv_i = (def \cup use) - (inS \cup inX) \tag{4.1}$$

$$RECV = \bigcup_{pid=0}^{p-1} recv \tag{4.2}$$

$$outI = inI \cup (DEF \cap inS) - (use \cup def) \tag{4.3}$$

$$outS = (inS \cup recv \cup (RECV \cap inX)) - DEF \tag{4.4}$$

$$outX = (inX - RECV) \cup def \tag{4.5}$$

For each processor, $recv$ is the set of data that is required but not available locally. The union of def and use gives the data required. The union of inS and inX represents the data already in the cache. $RECV$ is the union of the receive data for all processors.

The new set of data in the invalid set, $outI$, is derived as follows. Data that is invalid will stay invalid if it is not used, this gives us the term inI. Data that is currently owned but is defined by another processor will become invalid giving the term $(DEF \cap inS)$. Note that we are using a data parallel model thus exclusively owned data will never be defined by another processor and never become invalid. The final term, $(use \cup def)$, is necessary because any data being referenced by the current processor must be removed from the invalid set.

The new data set for shared data, $outS$, is computed as the sum of four terms. The first term, inS is included because any data that is in shared state will stay in shared state if it is not written or invalidated. The shared state also includes new data that is received, $recv$, and data that is initially exclusively owned but read by another processor, $(RECV \cap inX)$. Finally we must exclude from the shared data set any elements written by the local processor or any remote processors, DEF.

Similarly, the exclusively owned set, $outX$, is derived by taking the union of data already exclusively owned and not read by other processors, $inX - RECV$, with data that is written, def.

4.6.2 Cache State Algorithm

Figure 10 gives the CACHE_STATE_ANALYSIS algorithm which determines $outI$, $outS$, $outX$, and cost for each node of the CFT. The algorithm performs a forward traversal of the control flow tree treating

the parallel nodes as leaves. For each DOALL node the def/use information is used to calculate the cache state for the node using the data flow equations above (4.1)-(4.5). The node cost is calculated by taking the difference between the input and output cache state. There are three cost coefficients C_S, C_X and C_I, one for each cache state. They represent the cost incurred on a transition to the specified cache state. The The SIZEOF() routine calculates the size in number of array elements of the CRSDs representing the difference between input and output states.

For a program node the cost function, $cost[node]$, is the sum of the difference between the input state and the output state of the left most child plus the difference in output state between neighboring children from left to right. The cache state for each child is calculated by recursive calls to CACHE_STATE. The final cache state is equal to the cache state of the right most child.

Cost and cache state for a sequential DO loop is calculated similarly. The cost of the first iteration is calculated as described above, then the children are each visited once more to determine the steady state cost for the loop, this cost is summed for all remaining iterations of the loop. The output cache state for a sequential DO loop is equal to the output state of its right most child on the last iteration of the DO loop. The routine SUBSTITUTE($expr1$, $expr2$, $expr3$) replaces occurrences of $exp2$ in $expr1$ with $expr3$ and is used to replace an expression in a CRSD or subtree of the CFT with another expression. We use SUBSTITUTE() to modify the index variable expressions when calculating cache state and cost for DO loops.

The operation of the CACHE_STATE_ANALYSIS algorithm for the Jacobi example is diagramed in Figure 11. The cache state for one block of the block cyclic partition is shown and the CRSDs for $outX$ are given.

4.6.3 Correctness of Algorithm

The CACHE_STATE algorithm makes the assumption that the cost function computed for the second iteration of a DO loop is representative of the rest of the iterations. This is true if the cost is specified by a linear function of the iteration variable of the do loop, k. To prove the cost is linear in k we need to show: 1) the original subscript expressions are linear in k, 2) CRSD generation preserves linearity in k, 3) CRSD combining operations ($\cap$, $\cup$, $-$) preserve linearity in k, and 4) SIZEOF() produces a product of linear functions in k.

The original subscript expressions are linear in k. This is true because of the assumptions of our program model. Array subscript expressions are of the form $a \cdot i + b \cdot k + c$ where i is a DOALL loop index variable, k is the index of the innermost sequential DO loop enclosing the DOALL, and a, b, and c are an expressions which are invariant with respect to the innermost sequential DO loop and any enclosing DOALL loops.

CRSD generation preserves linearity in k. CRSDs are generated in two steps. First the CRSD upper and lower bounds are set to the subscript expressions for an array reference. So lb and ub are of the form $a \cdot i + b \cdot k + c$. Then the loop bounds of the loops that enclose the array reference are substituted into the CRSD for the loop index variable i. Since loop bounds are of the form $\acute{b} \cdot k + \acute{c}$. The resulting expression looks like $a \cdot (\acute{b} \cdot k + \acute{c}) + b \cdot k + c$ which can be rewritten as $(a\acute{b} + b)k + (\acute{c} + c)$. Thus the lower and upper bound of the CRSD are still linear in k.

CRSD combining operations preserve linearity in k. Let $C = AopB$ where op is one of the binary operators $\cap$, $\cup$, $-$. The resultant lower bound, lb_c, of C is either $\max(lb_a, lb_b)$ or $\min(lb_a, lb_b)$. Likewise

CACHE_STATE_ANALYSIS

1) Perform a depth first traversal of CFT, treat DOALL node as leaf
2) Generate cache state and cost for each node of the CFT visited

CACHE_STATE $(node, inI, inS, inX)$

if $(type[node] =$ DOALL$)$ **then**
$recv[node] \leftarrow (def[node] \cup use[node]) - (inS \cup inX)$
$RECV[node] \leftarrow \bigcup_i recv[node]$
$outI[node] \leftarrow inI \cup (DEF[node] \cap inS) - (use[node] \cup def[node])$
$outS[node] \leftarrow (inS \cup recv[node] \cup (RECV[node] \cap inX)) - DEF[node]$
$outX[node] \leftarrow (inX - RECV[node]) \cup def[node]$
$cost[node] \leftarrow C_S \cdot SIZEOF(outS[node] - inS) + C_X \cdot SIZEOF(outX[node] - inX)$
$+C_I \cdot SIZEOF(outI[node] - inI)$

else if $(type[node] =$ PROGRAM$)$ **then**
$cost[node] \leftarrow 0$
$child \leftarrow leftchild[node]$
while $(child \neq nil)$ **do**
CACHE_STATE $(child, inI, inS, inX)$
$cost[node] \leftarrow cost[node] + cost[child]$
$inI \leftarrow outI[child]$, $inS \leftarrow outS[child]$, $inX \leftarrow outX[child]$
$child \leftarrow next[child]$

else if $(type[node] =$ DO$)$ **then**
$cost[node] \leftarrow 0$
$child \leftarrow leftchild[node]$
while $(child \neq nil)$ **do**
CACHE_STATE $(child, inI, inS, inX)$
SUBSTITUTE $(cost[child], index[node], lb[node])$
$cost[node] \leftarrow cost[node] + cost[child]$
$inI \leftarrow outI[child]$, $inS \leftarrow outS[child]$, $inX \leftarrow outX[child]$
$child \leftarrow next[child]$
$child \leftarrow leftchild[node]$
while $(child \neq nil)$ **do**
SUBSTITUTE $(child, index[node], index[node] + 1)$
CACHE_STATE $(child, inI, inS, inX)$
$cost[node] \leftarrow cost[node] + \sum_{i=lb[node]+1}^{ub[node]}$SUBSTITUTE$(cost[child], index[node] + 1, i)$
$inI \leftarrow outI[child]$, $inS \leftarrow outS[child]$, $inX \leftarrow outX[child]$
$child \leftarrow next[child]$
$outI[node] \leftarrow$ SUBSTITUTE$(outI[rightchild[node]], index[node] + 1, ub[node])$
$outS[node] \leftarrow$ SUBSTITUTE$(outS[rightchild[node]], index[node] + 1, ub[node])$
$outX[node] \leftarrow$ SUBSTITUTE$(outX[rightchild[node]], index[node] + 1, ub[node])$

Figure 10: Cache State Analysis Algorithm

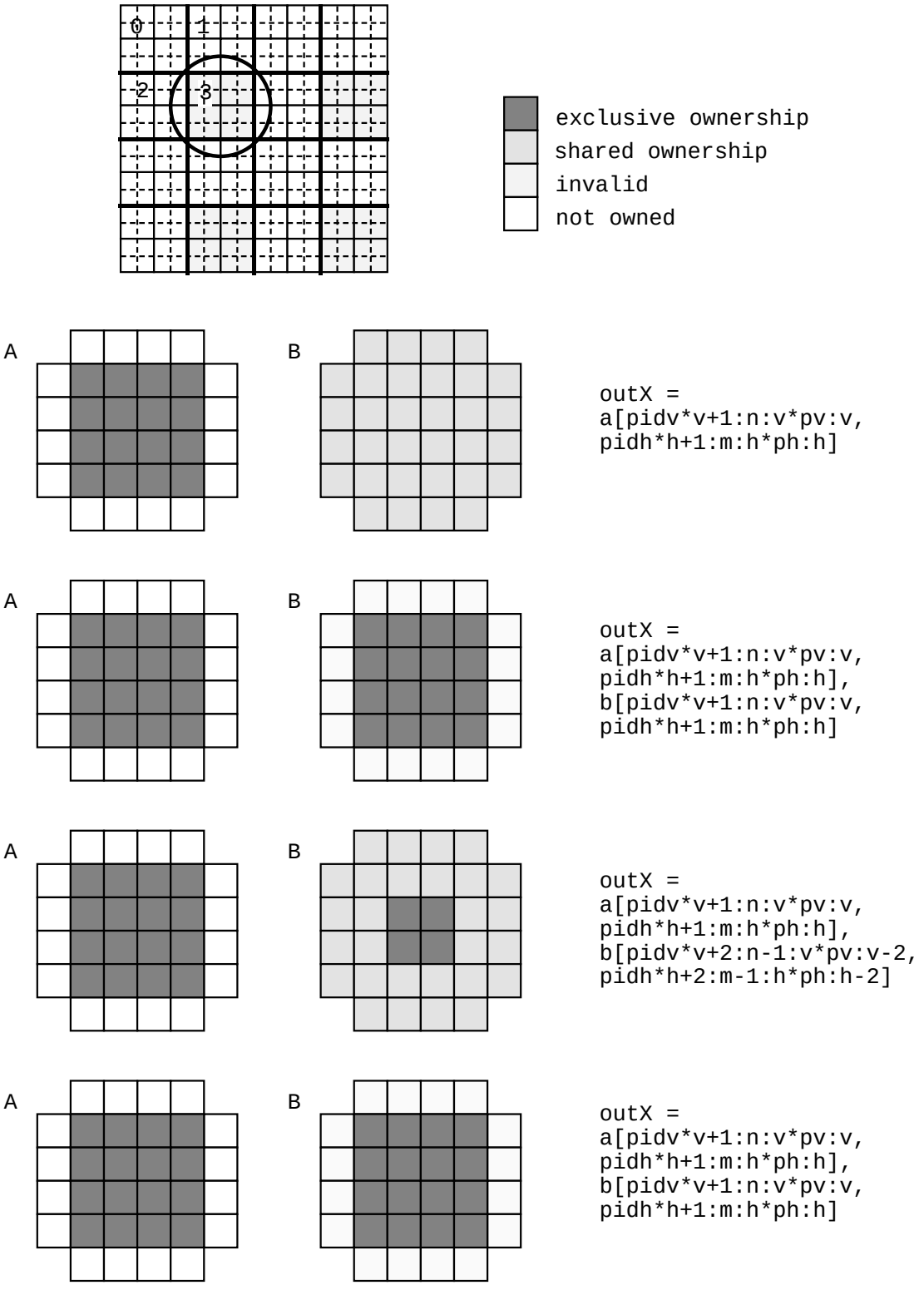

Figure 11: Output Cache State for Jacobi from CACHE_STATE() Algorithm

ub_c is either $\max(ub_a, ub_b)$ or $\min(ub_a, ub_b)$. A max or min function of two linear functions is either one of those linear functions if the two lines do not intersect or it is a piecewise linear function if the two lines do intersect. In the first case linearity in k is preserved. If at compiler time, k is the only non constant value in the linear expressions for the upper and lower bounds then we can preserve linearity by splitting loops in the program at the intersection points and treating them as separate loops. A similar technique is used by Carr and Kennedy in [11]. Unfortunately, if there are unknows other that k in the bounds expressions then our technique may not work because we may be unable to perform CRSD combining operations.

SIZEOF() produces a product of linear functions in k. SIZEOF() can be estimated by the equation $\prod_{i=1}^{n} \frac{r_i}{s_i}(ub_i - lb_i + 1)$ where n is the number of dimensions in the CRSD. If the conditions in 1), 2), and 3) are met then ub and lb are linear in k. The stride s_i and repeat r_i are static for a given partition and are thus constant in k. Therefor the SIZEOF() function is a product of factors that are linear in k.

4.7 Cost Function Optimization

The first four steps of the PARTITION algorithm produces a symbolic cost function representing interprocessor communication. In step five of the algorithm, the cost function is optimized by choosing partition parameters that minimize the function. The partition parameters are p_v, p_h - the number of processors in the vertical and horizontal dimension respectively and v, h - the block sizes in the vertical and horizontal dimensions. Our optimization process is a generalization of the work by Hudak and Abraham presented in [23].

The first stage of step five is simplification of the cost function. This includes simplifying some terms and eliminating others. We use the heuristics listed below.

1) assume that processor zero has at least as much communication as any other processor and use $pid_h = pid_v = 0$
2) substitute p for $p_v \cdot p_h$
3) remove terms which are not a function of h, v, p_v, or p_h
4) remove low order terms
5) assume that terms of the form $var - const/var$ are approximately 1

From the Jacobi example the cost term due to acquiring shared border elements for exclusive ownership is $cost = C_X \cdot \sum_2^{1000}[\frac{n}{v \cdot p_v}\frac{m}{p_h} + \frac{(n-v+1)}{v \cdot p_v}\frac{m}{p_h} + \frac{(n-2)(v-2)}{v \cdot p_v}\frac{m}{h \cdot p_h} + \frac{(n-2)(v-2)}{v \cdot p_v}\frac{(m-h+1)}{h \cdot p_h}]$. Using the heuristics listed above this equation simplifies to $cost = C_X \cdot 999[\frac{2n \cdot m}{v \cdot p} + \frac{2n \cdot m}{h \cdot p}]$.

Once a simplified cost function is found then we find the optimal value for the parameters one at a time. If cost is a function of the processor parameters, p_h and p_v, then use the equality $p_h = p/p_v$ to represent p_h in terms of p_v. Similarly, h can be represented in terms of v using the equality $h = \frac{n \cdot m}{\tau \cdot p \cdot v}$ where τ is the number of block cyclic regions assigned to each processor. After the cost function is expressed in terms of the parameter which is being optimized, we use the first derivative test to find the extreme value of the cost function and use the second derivative test to verify that the value found is a minimum. We can determine the value of p_h from p_v and thus determine the ratio of p_v/p_h that minimizes the cost function. Similarly we can find the ratio of v/h that minimizes the cost function.

The cost function for iterations 2 through 1000 of the Jacobi example is $cost = (999)(4 \cdot C_S + 2 \cdot C_I + 2 \cdot C_X)[\frac{n \cdot m}{v \cdot p} + \frac{n \cdot m}{h \cdot p}]$. Let $f = \frac{n \cdot m}{v \cdot p} + \frac{n \cdot m}{h \cdot p}$. We can rewrite f as a function of v using $h = \frac{n \cdot m}{\tau \cdot p}$ to

get $f = \frac{n \cdot m}{v \cdot p} + \tau \cdot v$ and $f' = -\frac{n \cdot m}{v^2 \cdot p} + \tau = 0$. Solving for v we get $v = \sqrt{\frac{n \cdot m}{p \cdot \tau}}$. Solving for h gives us $h = \sqrt{\frac{n \cdot m}{p \cdot \tau}}$. Thus the aspect ratio, $v/h = 1$, which implies that a square decomposition should be used.

4.8 Extensions

We have considered two extensions to the partition generation algorithm given in this section. The first extension broadens the machine model by taking cache line size into account when determining the cost function. The second extension broadens the programming model by using a heuristic to handle conditional code constructs such as **if** statements.

Cache line size can easily be taken into account by modifying the SIZEOF function as shown below.

```
SIZEOF (A)
    size ← (r_1 / (s_1 · l))(ub_1 − lb_1 + 1)
    n ← dimensionality(A)
    for i ← 2 to n do
        size ← size · (r_i / s_i)(ub_i − lb_i + 1)
```

We assume that arrays are aligned on cache line boundaries and that the size of the first dimension of an array is a multiple of the cache line size. We also constrain $r_1 \geq l$ where l is the cache line size. This really is not a constraint at all since r_1 will always be a multiple of the cache line size in order to prevent thrashing due to false sharing of cache lines between processors. If the Jacobi example is reworked using cache line size l the cost function is $f = \frac{n \cdot m}{v \cdot p} + \frac{n \cdot m}{l \cdot h \cdot p}$. The resulting aspect ratio after optimizing this function is $v/h = l$.

Conditional statements are difficult to handle because branches of the conditional may have very different definition/use patterns. Therefore the output cache state of the branch can not necessarily be determined at compile time. Currently, we assume that one branch of the conditional dominates and we substitute the body of the most frequently taken branch for the conditional node in the control flow tree eliminating the conditional node. Static branch prediction methods, profiling data or user input can be used to determine the taken branch [29],[27].

5 Experimental Results

We hand optimized two programs using the techniques presented in this paper. The first program that we optimized was the Jacobi program which is given in Figure 3 and has been used as an example throughout this paper. Jacobi is representative of many computational codes that have uniform communication between neighboring processors. In addition we optimized the LU factorization linear algebra routine. We chose LU factorization because it has different analysis requirements than Jacobi. The definition/use and cache state variables are dependent on a sequential do loop index in the LU program whereas these variables are constant for each iteration of the Jacobi program.

We ran our experiments on a 32 processor KSR1. Details of the machine are given in Section 3.

Aspect Ratio v/h	v	h
0.17	32	192
0.37	48	128
0.67	64	96
1.50	96	64
2.67	128	48
6.00	192	32
24.00	384	16

Table 2: Aspect ratios used in Jacobi experiments

5.1 Jacobi

The cost analysis of the Jacobi algorithm determined that square partitions are optimal for machines with a cache line size of one and rectangular partitions with aspect ratio of $v/h = l$ are optimal for machines with a cache line size of l, where l is the number of datum per cache line. We ran several experiments on the KSR1 and give their results. To simulate various cache line sizes, l, we indexed the first dimension of the data arrays with a step value of $step = 16/l$. The subcache size on the KSR1 is 128 bytes or 16 data elements, since we only access every $step$th element the effective cache line size is $l = 16/step$.

For our experiments we used 24 processors, a 384x384 array size, and $p_v = 384/v$, $p_h = 384/h$. We varied the effective cache line size from 1 to 16. For each effective cache line size we ran seven different partitionings. The values used are given in table 2. We report the execution time per array element. The results of the experiments are summarized in Figure 12.

The analytical and experimental results correlate closely. Figure 12 give the normalized execution time as a function of the partition aspect ratio, v/h. Each curve in the figure represents a different cache line size, $l = 1$ to $l = 16$. The optimal partition aspect ratio for a cache line size of one is near one, as can be determined by the minimum point in the $ls1$ curve. As the cache line size increases, the minimum point shifts to the right indicating that a rectangular partition with $v > h$ should be used. The default partitioning scheme on the KSR1 is column partitioning which works well for this example since the array size is small. However the curve starts to flatten out between aspect ratios of 6 and 24 and column partitioning will not be optimal for larger array sizes.

5.2 LU Factorization

We used the LU factorization code given in Figure 13 for our experiments. While optimizing this code we learned a few things about our procedure.

The first was that it is easier to represent some array sections using CRSDs with a stride less than zero because the lower bound of the CRSD has two roles, specifying the lower bound of the array sections and specifying the starting position of the repeated blocks within the array section. In the case of LU factorization the array sections are dependent on k the outer loop index. When k is used as the lower bound, more that one CRSD may be needed to represent and array section because the first block will not be complete if k falls in the middle of it. If we represent the CRSD using n as the lower bound and using a negative stride we don't have this problem.

The second lesson was that partitioning can introduce conditional statements. In the LU factorization

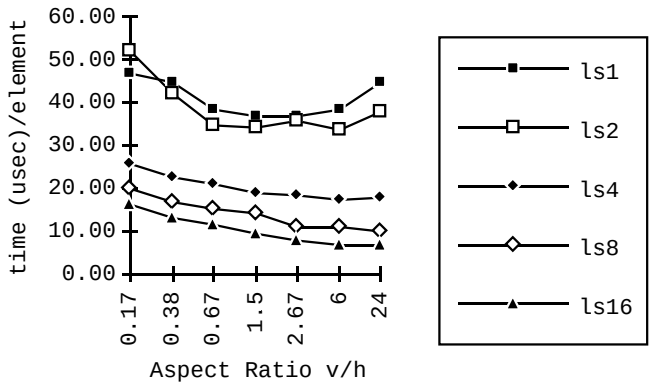

Figure 12: Normalized execution time for Jacobi using various line sizes

Aspect Ratio p_v/p_h	p_v	p_h
0.042 (col)	1	24
0.167	2	12
0.375	3	8
0.667	4	6
1.500	6	4
2.667	8	3
24.000 (row)	24	1

Table 3: Processor aspect ratios used in LU factorization experiments

the left subtree is only executed on processors owning the pivot column. Conditional execution such as this occurs in loops that do not access the entire data space. We assumed for our analysis that the left subtree is not executed.

The cache state of the LU algorithm for a given processor as a function of k is shown in Figure 14 for optimal and column partitioning strategies. The simplified cost function generated by our algorithm for LU is $f = C_s \cdot \frac{n(n-1)}{2}[\frac{1}{p_v} + \frac{1}{p_h}]$. When we differentiate and solve for p_v we find that $p_v = p_h = \sqrt{p}$ will minimize cache traffic. Using a cache line size of l the cost equation is $f = C_s \cdot \frac{n(n-1)}{2}[\frac{1}{p_v \cdot l} + \frac{1}{p_h}]$. Solving for p_v we get $p_v = \sqrt{p/l}$ and $p_h = \sqrt{p \cdot l}$. The aspect ratio is $p_v/p_h = 1/l$. Thus with large line sizes, column partitioning works best.

We used block-cyclic partitioning on 24 processors for our LU experiments with $v = l$ and $h = 1$. The array size was 384. We simulated various cache line sizes as described above. For each effective cache line size we ran seven different partitionings as given in table 3. The results of the experiments are summarized in Figure 15.

The analytical results and experimental results also match closely for this example. The optimal partition aspect ratio for a cache line size of two is near between 0.375 and 0.667 this matches closely with the predicted value of $1/l = 0.5$. As the cache line size increases the minimum point shifts to the left indicating that column cyclic partitioning should be used.

```
program LU

real a(n, n)

do k = 1, n
    doall i = k + 1, n
        a(i, k) = a(i, k)/a(k, k)
    enddoall
    doall i = k + 1, n
        doall j = k + 1, n
            a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
        enddoall
    enddoall
enddo
end
```

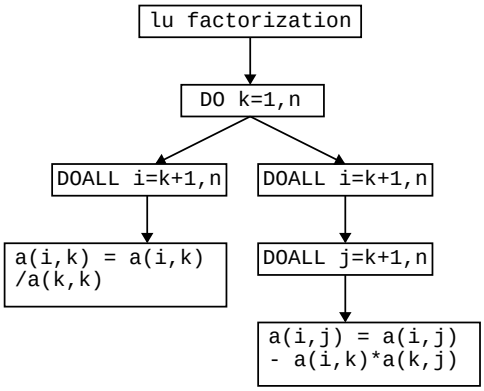

Figure 13: LU Factorization Example

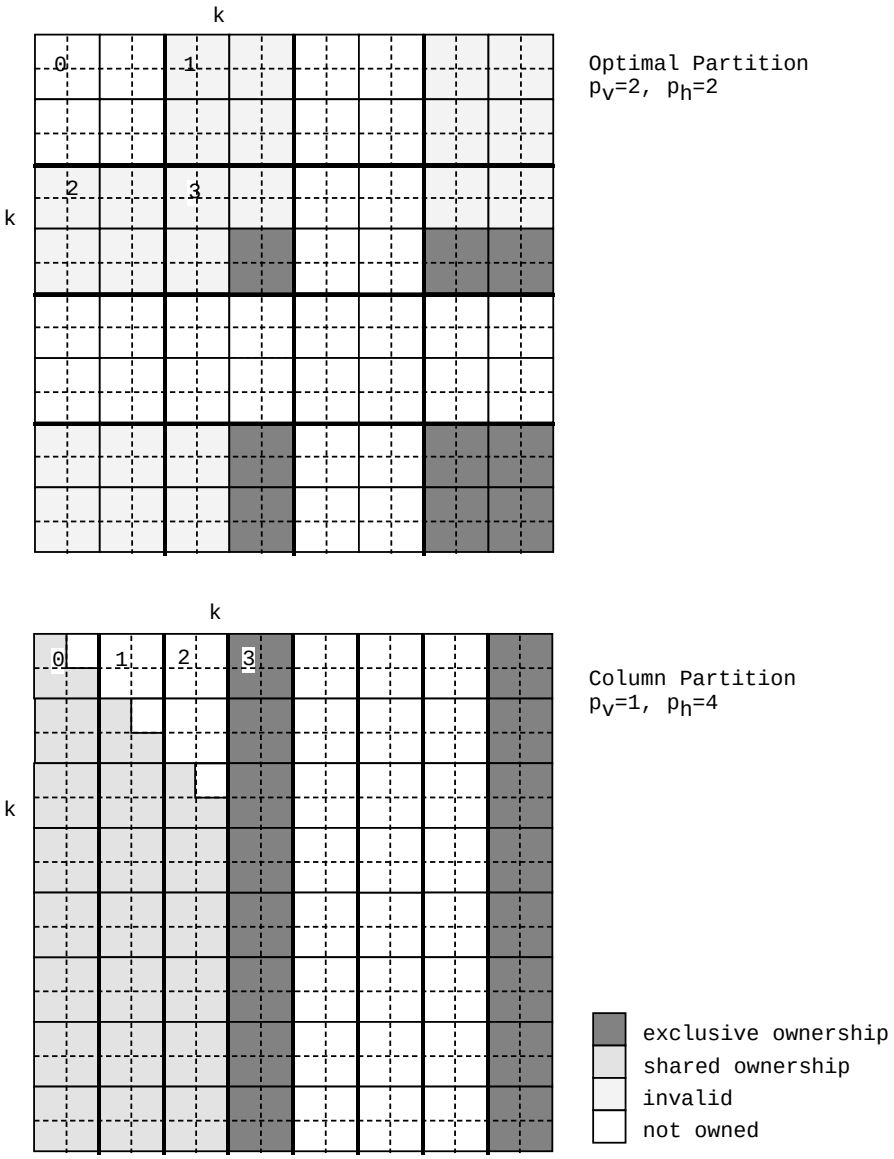

Figure 14: Cache State of LU Factorization for Iteration k

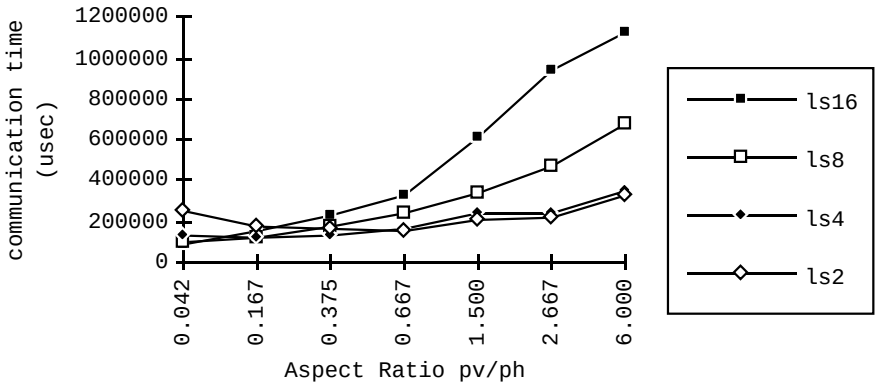

Figure 15: Execution time for LU using various line sizes

6 Conclusion

There are two main contributions in this report: algorithms to compute the cache state on exiting a parallel region given the cache state on entry; methods to compute the overall cache-coherency traffic and choose block-cyclic parameters to optimize cache-coherency traffic.

We introduce the notion of statically characterizing the state of each processor's cache in a cache-coherent multiprocessor by classifying data as shared, exclusive or invalid. The algorithms for calculating the cache state of each parallel node extend data-flow analysis techniques to the important problem of managing cache-coherency traffic. An interesting aspect of this algorithm is that, under our program model, the cache-coherency traffic for all iterations of a loop can be determined by symbolically computing the output state twice. These algorithms are also useful for performance estimation and analysis of partitioned parallel programs on cache-coherent multiprocessors. The overhead of cache-coherency traffic is not readily apparent to a programmer/user and our algorithms can be used to provide useful feedback to the programmer on the amount of cache-coherency traffic The user may use this feedback to alter the partition to improve performance. The application of these algorithms may indeed be simpler in this context because the most of the parameters involved are known.

Using our algorithms for finding the difference between CRSDs and the size of CRSDs, the cache-coherency traffic is computed at each sequential DO node in the control tree. Several simplifications and approximations are applied to the resulting expression. The block-cyclic partition parameters are chosen to minimize the traffic. The approach can naturally accommodate the effect of cache line size. The partitions produced by our approach are non-obvious and illustrate the effect of cache line size on partition performance. For instance, our approach produces a two-dimensional block-cyclic partition for the LU program in contrast to the commonly used column-based partitioning.

Though we believe that we have a promising approach for partitioning programs for multiprocessor systems, further research is necessary to firmly establish the feasibility of our approach. A major issue is whether the symbolic manipulation of CRSDs that are required in our approach can be automated. It is also not clear whether an automated system can make the right assumptions to simplify expressions.

References

[1] Santosh G. Abraham and David E. Hudak. Compile-time partitioning of iterative parallel loops to

reduce cache coherence traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[2] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops for cache-coherent multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 1, pages 2–11, 1993.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1988.

[4] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 112–125, 1993.

[5] Daya Atapattu and Dennis Gannon. Building analytical models into an interactive performance prediction tool. In *Proceedings of Supercomputing '89*, pages 521–530, 1989.

[6] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data prtitioning decisions. In *Proceedings of the 3rd ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 1991.

[7] Vasanth Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 41–53, 1989.

[8] Eric Boyd, John-David Wellman, Santosh Abraham, and Edward Davidson. Evaluating the communication performance of MPPs using synthetic sparse matrix multiplication workloads. In *Proceedings of the International Conference on Supercomputing*, pages 240–250, 1993.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of SIGPLAN '86 Symp. Compiler Construction*, pages 162–175, june 1986.

[10] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[11] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, pages 114–124, 1992.

[12] Marina Chen and Yu Hu. Optimizations for compiling iterative spatial loops to massively parallel machines. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 1–19, 1992.

[13] Christine Eisenbeis, William Jalby, Daniel Windheiser, and François Bodin. A strategy for array management in local memory. Technical Report 1262, Institut National de Recherche en Informatique et en Automatique, July 1990.

[14] Thomas Fahringer and Hans Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the International Conference on Supercomputing*, pages 207–219, 1993.

[15] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[16] Manish Gupta and Prithviraj Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the International Conference on Supercomputing*, pages 87–96, 1993.

[17] S.K.S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 301–305, 1993.

[18] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[19] High Performance Fortran Forum, CITI/CRPC, Box 1892, Rice University, Houston, TX 77251. *Draft: High Performance Fortran Language Specification*, 1993.

[20] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report Rice COMP TR91-149, Rice University, Department of Computer Science, March 1991.

[21] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the International Conference on Supercomputing*, pages 1–14, 1992.

[22] David E. Hudak and Santosh G. Abraham. Compile-time optimization of near-neighbor communication for scalable shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 15:368–381, 1992.

[23] David E. Hudak and Santosh G. Abraham. *Compiling Parallel Loops for High Performance Computers: Partitioning, Data Assignment, and Remapping*. Kluwer Academic Publishers, 1993.

[24] James Larus, Satish Chandra, and David Wood. CICO: A practical shared-memory programming performance model. Technical Report Report No. 1171, University of Wisconson-Madison, August 1993.

[25] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[26] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of ACM/SIGPLAN Symp. Parallel Programming: Experience with Applications, Languages, and Systems*, pages 85–99, july 1988.

[27] Wen mei W. Hwu, Thomas M. Conte, and Pohua P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture*, pages 224–233, 1989.

[28] R.Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of SIGPLAN '86 Symp. Compiler Construction*, pages 176–185, june 1986.

[29] James E. Smith. A study of branch prediction strategies. In *Proceedings of the Eigth Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[30] Ernesto Su, Daniel Palermo, and Prithviraj Banerjee. Automating parallelization of regular computations for distributed-memory multicomputers in the paradigm compiler. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 30–38, 1993.

[31] Daniel Windheiser, Eric Boyd, Eric Hao, Santosh Abraham, and Edward Davidson. KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *Proceedings of the International Parallel Processing Symposium*, pages 454–461, 1993.

[32] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[33] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, February 1993.