Dynamic Adaptive Scheduling in an Implementation of a Data Parallel Language*

Edward A. Kornkven
Department of Computer Science
University Of Illinois
Urbana, IL 61801
email: kornkven@cs.uiuc.edu
tel: (217)-333-5827
fax: (217)-333-3501

Laxmikant V. Kalé
Department of Computer Science
University Of Illinois
Urbana, IL 61801
email: kale@cs.uiuc.edu
tel: (217)-244-0094
fax: (217)-333-3501

Abstract

In the execution of a parallel program, it is desirable for all processors dedicated to the program to be kept fully utilized. However, a program that employs a lot of message-passing might spend a considerable amount of time waiting for messages to arrive. In order to mitigate this efficiency loss, instead of blocking execution for every message, we would rather overlap that communication time with other computation. This paper presents an approach to accomplishing this overlap in a systematic manner when compiling a data parallel language targeted for MIMD computers.

1 Introduction

Data parallel languages have proven to be efficient platforms for programming in a wide range of application domains. Parallelism is explicit and clearly understood in these languages. From the user's point of view, the execution model is straightforward, unambiguous, and is an "easy" way to attain a high degree of parallelism. It is now also clear that the underlying hardware need not have the same execution model and in fact, better performance might be attainable on MIMD machines than on the original home of data parallel languages, the SIMD computers. This is due in large measure to the ability of MIMD processors to operate independently, on different parts of the program. This decoupling of processors offers the potential for increasing processor utilization by not forcing one processor to wait on another when it could be doing useful work.

Even on a MIMD machine however, there are pitfalls that would inhibit the realization of maximal utilization for a given program because the coupling of processors, while eliminated in the hardware, may be reintroduced by the program. That is, if the program forces unnecessary waiting (e.g. unnecessary synchronization or waiting for messages), the hardware will still be under-utilized.

For a programmer, it can be difficult or impossible to recognize in the program just where synchronization is necessary or where opportunities for overlapping communication and computation are. The programmer might attempt to reorder program statements in order to overlap communication with computation, but in general these are influenced by run-time conditions and cannot be completely predicted statically. In this paper, we will present our approach to implementing a subset of data parallel Fortran in which these difficulties are overcome for the compiler writer by generating adaptive schedules at compile time that enforce all the control and data dependences correctly, while allowing the run-time system to re-order processing of different messages (and associated subcomputations) depending on the arrival order of messages at run-time. The schedule and computations are specified using Dagger notation, which is part of the Charm parallel programming system. Charm is an explicitly parallel language with a message-driven execution model. Since Charm is machine independent and has already been ported to numerous parallel machines, the code produced by the compiler is machine independent by default.

The focus of this paper is to demonstrate the feasibility of such adaptive schedules, and to introduce the static analysis and code generation that a compiler must carry out to implement a data paral-

*This research was supported in part by National Science Foundation grant CCR-91-06608

lel source language in this environment. In section 2 we give a brief introduction to the source language. In section 3 an overview of Charm and Dagger is presented. In section 4 we discuss the compilation problem in more detail and section 5 illustrates the techniques used to compile various structures of the source language. Finally, we discuss related research in section 8 and summarize our results in section 9.

2 The DP Source Language

The source language that we wish to compile is a subset of High Performance Fortran (HPF) [Hig92] which we call DP. Like other languages in its class, its distinguishing features include the ability of source statements to operate on entire arrays or sections of arrays at a time, and a large set of intrinsic functions for manipulating those arrays. An assignment statement will assign a scalar value to a scalar variable, an array value to a array variable, or a scalar value to a array variable. An assignment to a scalar in which an array expression appears on the right-hand side requires the use of a reduction function and is discussed below.

Similarly, intrinsic functions may take scalar arguments, array arguments, or both. As an example of an intrinsic function that operates on arrays, consider the assignment

$$k = SUM(A)$$

where k is a scalar variable and A is an array. The SUM function is a *reduction* function–i.e., it "reduces" an array to a scalar value by performing some commutative and associative combining operation on all the elements of the array. In the case of SUM, the sum of all the array elements is returned.

A DP fragment for a Jacobi iteration of an approximation of Poisson's equation is shown in Figure 1. Variables A, $Anew$, and W are two-dimensional arrays while c is a scalar. $CSHIFT$ is an intrinsic function. There are two assignment statements–they are both array assignments.

$$A = Anew$$
$$Anew = (1/(2+2*c))* \ \&$$
$$(CSHIFT(A,1,-1)+CSHIFT(A,1,1)+ \ \&$$
$$c*(CSHIFT(A,2,-1)+CSHIFT(A,2,1))-W)$$

Figure 1: DP source statements for Jacobi iteration

In the HPF execution model (which we follow), parallelism is attained by distributing array elements across processors with each processor computing its own section of the array. The distribution method may be suggested by directives in the user program. For this program, let us assume that the array is partitioned by blocks, where each processor is allocated a "sub-rectangle" of the arrays A and $Anew$. We will avoid details of declaring arrays and distributing them onto processors. It will be assumed in the examples discussed here that all arrays are the same size and shape, and that corresponding elements are stored on the same processor. The reader is referred to [Hig92] for further details of HPF.

Some statements will be able to execute entirely locally on the processors–the first assignment statement in Figure 1 is an example. Generally, however, data will have to be shared between processors at some time during execution of the program, necessitating interprocessor communication. For example, in Figure 1 the calls to the instrinsic $CSHIFT$ serve to reference neighboring elements from above, below, left, and right respectively. These function calls will force interprocessor communication for those elements whose neighbors don't lie on their home processor.

Let us be more precise about the code a compiler might generate for each processor for the example in Figure 1. In outline form, the steps involved in executing this statement are as follows:

1. Send left boundary values (of the local array section) to the processor on the left. Likewise for the right, bottom, and top boundary values.

2. When the top message arrives, shift the array and add it into $Anew$.

3. When the bottom message arrives, shift the array and add it into $Anew$.

4. When both the right and left messages arrive, shift them, add them, and multiply by c, accumulating this product in $Anew$.

5. When all messages have arrived, complete the calculation.

The computations of complex source language features such as the $CSHIFT$ intrinsic function can often be best expressed by intermediate-level instructions. We will use a sort of intermediate pseudocode to enable us to discuss the code that the compiler has to work with while still avoiding low-level details that are beyond the scope of this paper. That intermediate-level pseudocode for our example appears in Figure 2. As a matter of convention, we will capitalize the names of arrays, make the names of built-in features all capitals, and leave scalars in lower case.

We can illustrate this computation by a dependence graph as shown in Figure 3. The rectangular nodes represent computations while the circular nodes represent points at which messages are received. Arcs from circles to rectangles represent a dependence on a message, arcs from rectangles to circles represent the readiness to accept a message, and arcs from rectangles to other rectangles indicate dependences between computations.

Notice that some of these steps depend on other steps but also that some are independent of others. Note also that it will be difficult or impossible to establish at compile time a schedule for these steps that always allows a ready step to execute without waiting on another step. For example, we probably will not be able to predict in general whether the message from a left neighbor will arrive before the message from a right neighbor. Even if the behavior were the same for all executions of the program, it might even be that the order in which these two messages arrives is different for different processors. Clearly, then, ordering these statements for optimal overlap is difficult and not amenable to a completely static approach. An alternative dynamic approach is the topic of this paper.

3 Dagger and Charm

Charm [Kal90, Cha92] is a machine-independent runtime system which runs on a variety of shared-memory and distributed-memory parallel computers and supports an explicitly parallel C-like language. Charm supports the dynamic creation, manipulation, and scheduling of small tasks called *chares*. Chares may create other chares or send messages to entry points of existing chares, enabling those chares to be scheduled for execution. Charm supports an *asynchronous* programming style. Chares are designed to execute for a relatively short time and then suspend, waiting for another message. In the meantime, other messages that have arrived are de-queued and execution continues, possibly in another chare. A special kind of chare, the *branch office* chare, is automatically replicated on each processor. Branch office chares are useful for describing a computation that is similar on all processors–such as the array processing in DP. They are also useful for implementing distributed data structures and for defining interfaces between parallel modules.

Dagger [Gur93] runs on top of Charm and is a tool for performing computations that can be represented by directed graphs. Dagger facilitates the expression of the synchronization of threads of execution by specifying the condition under which a thread executes in terms of the other threads that must complete first. In this paper, we describe the use of Dagger to schedule statements and messages of an executing program–Dagger is the target language of our compiler. At compile time, the conditions under which a statement may execute are described but the order in which they will actually execute depends on which of those conditions are met first during a particular execution of the program. Using Charm constructs, Dagger can in effect "execute" a dependence graph. This results in a dataflow-flavored approach which enables statements to be executed at their earliest opportunity.

Dagger allows one to define a special form of chare in which the code is segmented into `when` blocks. A particular `when` block is tagged with dependences that must be satisfied in order to enable the `when` block to be initiated for execution by the Charm run-time system. These dependences assume one of two main forms in Dagger:

1. An *entry*–the target of message that will be sent to the chare by some PE, or
2. A *condition variable*–a special variable that Dagger watches.

A function called `Ready` is used to set the condition variables, which are initialized to be "not ready".

Syntactically, `when` blocks have the form

$$\texttt{when}\ cv_1, cv_2, cv_3, \ldots, cv_m, e_1, e_2, e_3, \ldots, e_n\texttt{:}\\ \{\ \textit{code to be executed}\ \}$$

When all condition variables cv_i are ready, and for each controlling entry e_j an `expect` statement has been executed and a message directed at e_j has been received, the `when` block is ready to execute. When it actually executes depends on which ready `when` block Dagger next selects for execution.

As an example, we show an outline of the Dagger code for the Jacobi iteration example in Figure 4. In Dagger, a chare begins in its `init` block. For the sake of brevity, some syntactic details not germane to this discussion have been omitted. For the same reason, we show the array computations in their intermediate pseudocode form. Note in this example the rather straightforward correspondence between the dependence graph (Figure 3) and the Dagger code. The remainder of this paper deals with deriving the Dagger code from the source (or intermediate) code.

4 The Problem

There are, of course, many facets of the general problem of code generation for parallel computers. Our concern here is how to derive the schedule of instructions that will be executed. Our goal is to leverage the asynchronous, message-driven features of Charm/Dagger to enable statements to execute as soon as they are ready to be executed.

When we say that a program statement is "ready" to execute, we mean that (a) there is no data upon which that statement depends that is not yet available, and (b) any control flow conditions imposed on the statement have been met. This means that any statements that compute these antecedent data must have already completed execution. In commonly-used procedural languages, the ordering of source statements also imposes dependences between statements, but this ordering is unnecessary unless the statements share data. For example, in the program fragment

$$A = B + C$$
$$X = Y + Z$$

variable A doesn't necessarily have to be computed before variable X even though that is the order specified by a top-to-bottom reading of the source code (assuming there are no aliases that prevent it). Removing these artificial dependences and allowing code to execute in a different order from that specified by the source program is the basis of many optimizations for all kinds of architectures (not just parallel machines). Of course, most statements will depend on at least one other statement and such dependence relationships must be preserved. But suppose we have the following source statements:

$$k = SUM(X)$$
$$A = SIN(X)$$

Each processor will execute code equivalent to the following intermediate pseudocode:

$tmp = ComputeLocalSUM(X)$
Send tmp to global SUM
Receive back global SUM $\rightarrow k$
$A \leftarrow SIN(X)$

The third statement, the *Receive*, will cause execution to block if the message is not available. But in that case, we could be executing the expensive SIN function on the array X instead of waiting idly–each processor will be unnecessarily underutilized. The simple solution is to change the order in which we execute these statements–move the calculation of A before the *Receive*. This will give the *Receive* a longer time to finish. But of course, how do we know in general that such a change will be enough? Should we move another statement before the *Receive*?

Consider another example that uses two intrinsics, MAX and SORT:

$$k = MAX(A)$$
$$B = SORT(A) + D$$
$$C = A/k$$

These statements are represented by the intermediate pseudocode shown in Figure 5. As we consider how we might reorder these statements to maintain full utilization we see that the situation is more complicated. It helps to examine the data dependence graph for these instructions (Figure 6). We could start the $SORT$ first and overlap its communication with the local MAX computation. We could next send $tmp1$, but then what? We are expecting replies to two messages, each of which head up independent threads of computation. However, in general we are unable to determine at compile time which message will be ready first and therefore which statement to execute next. This is because at compile time we may not know what the load of each processor at run-time will be, exactly which instructions will execute for each processor, what the problem size will be, perhaps the number of processors or even what kind of processors will execute the program, etc. Therefore, it is essential that our scheduling method be able to generate dynamic schedules that adapt to actual execution-time conditions. We propose calculating dependences between statements at compile time which, when satisfied at execution time, enable the dependent statement to execute.

In the next section we explain how these adaptive schedules can be generated and expressed in the Dagger notation. Then, we try to improve the generated code. One such improvement is to attempt to reduce the overhead of scheduling statements by reducing the number of times we return control to Dagger.

One final note: the reader should bear in mind as we discuss "processor utilization", "overlapping computation", "scheduling", etc., that we are concerned here with the behavior *within a single processor*. Due to the nature of the programming model that we are compiling for (i.e., data parallelism), any solution to the overlap problem obtained on one processor will by definition be a solution on the others.

5 Our Approach

In the following sections, we take progressively more complicated cases of DP code in turn and show how Dagger code will be generated.

5.1 Straight-Line Code

We first examine the translation of DP programs without loops and conditionals. This will enable us to introduce the method without bogging down in the details that arise for branching.

5.1.1 Analysis Required

Let us consider again the Jacobi iteration example from Figure 1. As previously discussed, we wish to reorder instructions in an attempt to overlap waiting for the completion of communication with other independent computations. Our task is to do that reordering, or more precisely, to enable that reordering to happen at run-time. We will first create a data dependence graph (DDG) for the program to assist in our analysis. Each rectangle node in the DDG will translate to a `when` block in the Dagger program. A `when` block is labelled by condition variables and entries corresponding to other DDG nodes upon which the node depends.

Initially, let each node of the DDG correspond to one statement in the intermediate code of the parsed source program. The initial DDG corresponding to the example code is shown in Figure 8. As explained in section 2, the circle nodes in the graph represent entries which receive messages and the rectangular nodes represent computations that don't involve receiving messages. These circle nodes are of special interest because they indicate a statement that depends on a run-time event which cannot be predicted by the compiler (the receipt of a message).

Overhead will be incurred in this programming model when Dagger schedules `when` blocks for execution. To reduce overhead, we would like to simplify the DDG by merging statements into larger blocks. Larger DDG nodes result in more user work being done per scheduling event. Our node merging is constrained by a couple of factors. First, we don't merge circle nodes because, as previously stated, the receipt of a message does not depend on the satisfaction of data dependences but on the completion of message-passing the time of which is unknown to the compiler. Second, we cannot permit an invalid ordering of the statements (i.e. one that violates our dependences). Since this is straight-line code, we have no control dependences to worry about and we know that the DDG is acyclic. We might be tempted to merge nodes as much as possible. However, when a pair of nodes are merged, the new node must have as its incoming arcs the union of the incoming arcs of the original nodes– likewise for the outgoing arcs. This means that if we merge a node with a successor node, that node will now depend on everything on which its successor depended. These are new dependences that we have introduced and may be unnecessary for that statement. Note however, that a node which depends on only one other node may be safely merged with that node. Therefore, our node-merging algorithm will moderate the goal of maximal merging with the goal of not introducing unnecessary dependences by not merging nodes that have more than one predecessor. The node-merging algorithm is given in Figure 7 and the DDG after merging is applied was given in Figure 3.

The names used in the node-merging algorithm have the following meaning:

v The node being examined for merging.

visit_count Number of times this node has been visited in the graph traversal (initially 0 for each node).

child List of children of a node.

n_predecessors The number of predecessors of a node.

new_block A routine that creates a new node in the new DDG, inserts the statement corresponding to the argument into the block, and updates the pointers according to the pointers of the argument.

merge_into_pred A routine that merges the argument node into the block of the predecessor node. If we are prioritizing statements for execution, those priorities will have to be used here.

all_predecessors_same A function that returns TRUE if all the predecessors of a node have the same block in the new DDG being built by the merge algorithm, FALSE otherwise.

Note that after node merging, a new node depends only on statements upon which the *first* statement in the block depended. Also, any statement that was dependent on any of the original statements becomes dependent on the new block.

One other refinement we can make to the DDG is to remove *redundant* dependences. A dependence arc is redundant when it specifies a dependence that

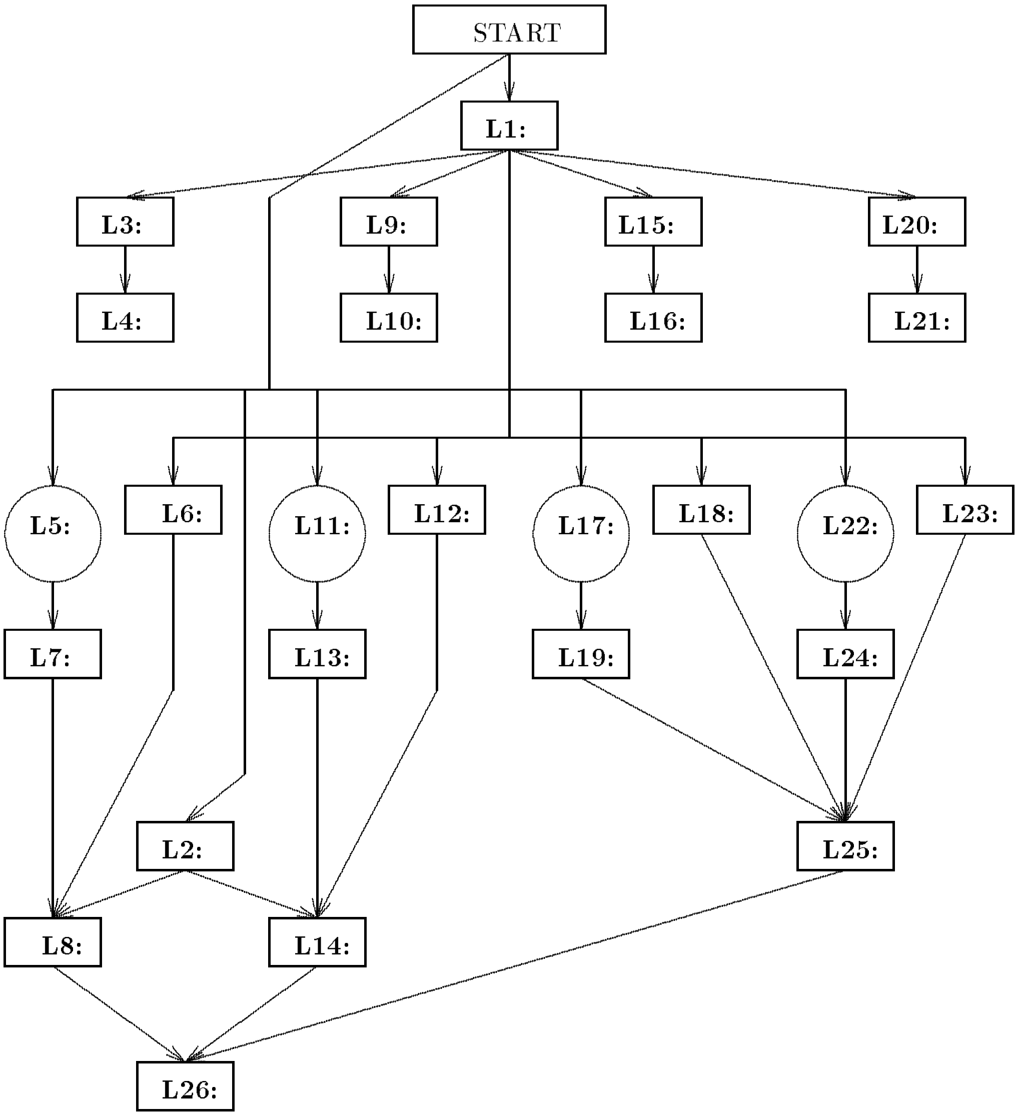

Figure 8: The labelled DDG before node merging.

is already a necessary condition of the correct execution of the program. Specifically, suppose we have a DDG with nodes a, b and c. Let $a\ \delta\ b$ mean that statement b is dependent on statement a and let $a\ \delta^*\, b$ mean that there is a sequence of dependences $a\ \delta\ x\ \delta\ y\ \ldots\ \delta\ z\ \delta\ b$. Then if $a\ \delta^*\, b$ and $b\ \delta^*\, c$, if there is also an arc $a\ \delta\ c$, we may remove it. Again, the rationale for this is that if b must follow a and c must follow b, it is redundant to say that c follows a.

One situation in which redundant dependence arcs appear in the DDG is illustrated in Figure 9. In this example, the arc from **S1** to **S3** is redundant because we already know that **S1** δ^* **S3** because **S1** δ **S2** δ **S3**. However, careful implementation of our node merging algorithm can eliminate this extra arc and all three nodes can be merged into one.

Node merging presents a trade-off to consider in our code generation. Larger blocks reduce overhead by reducing the number of times that Dagger has to select and initiate tasks. On the other hand, smaller blocks (i.e. single statements) might present more opportunities for initiating other tasks upon the block's completion thereby keeping processor utilization high. However, we can safely make blocks large without negatively affecting processor utilization, at least not directly. This is true for the following reasons. When we merge a node with its predecessor, we are essentially removing a "degree of freedom" by forcing that node to execute immediately after the predecessor node instead of possibly later. Note though, that one still has the option during the merger of reordering the statements in the merged node by placing a higher priority statement (e.g. a statement that initiates communication) first, assuming there are no dependences being violated by such a reordering. Since we don't merge circle nodes with their predecessors, this merging cannot cause the node to block–Charm/Dagger guarantees that once execution on a `when` block is initiated, it will run to completion. So the processor must be completely busy while executing the merged block.

5.1.2 Translation to Dagger

Given the new DDG, generation of Dagger code for this segment involves the following steps:

1. Label the blocks in the new DDG to be used in generating `when` block condition variable names.

2. Visit each rectangle node in the new DDG and produce a `when` block for it, using the dependences from the DDG and their labels from the first step to create the conditions that will trigger the `when` block. If a rectangle node in the DDG depends on rectangle nodes (computations) $b_1, b_2, \ldots, b_m$ and circle nodes (messages) $c_1, c_2, \ldots, c_n$, then create a `when` block that depends on condition variables $b_1, b_2, \ldots, b_m$ and messages $c_1, c_2, \ldots, c_n$. Circle nodes have no `when` blocks associated with them–they only indicate that a message must be received.

3. For each rectangle node with an arc to another rectangle node, emit a `Ready` statement at the end of the block to indicate its completion. If a rectangle node has an arc to a circle node, emit an `expect` statement corresponding to the message of that circle node.

4. Into the `when` block, insert the code from the corresponding DDG node.

An outline of the Dagger code for this segment was shown in Figure 4.

6 Branches and Loops

In this section we will discuss the translation of programs containing IF and loop statements. We will focus our attention on loops since an IF statement can be viewed as a special case of a loop which iterates zero or one times. (An IF-THEN-ELSE can also be emulated by two simple IFs.) The DP language supports only well-structured IFs and loops–arbitrary GOTOs are not allowed. For the purposes of this discussion, assume that we are dealing only with Fortran's IF and DO WHILE statements.

6.1 Issues Raised by IFs and Loops

In the presence of IF statements and loops, analysis is complicated by the fact that the statements that will generate the values used by other statements are not known at compile time. For example, suppose we have the Jacobi iteration code inside a loop as shown below.

$Anew = init\_val$
$DO\ WHILE\ (not\_done(Anew, A, epsilon))$
$A = Anew$
$Anew = (1/(2+2*c)) *\ \&$
$(CSHIFT(A,1,-1) + CSHIFT(A,1,1) +\ \&$
$c*(CSHIFT(A,2,-1) + CSHIFT(A,2,1)) - W)$
$END\ DO$
$D = Anew - A$

When computing D, which definition of $Anew$ should be used? That depends on whether the loop branch is ever taken–if it is, then the value of A from the last iteration should be used; otherwise the value of *Anew* will still contain *init_val*. Consequently, we have to consider all possible sources of dependences and make sure that in our Dagger program they will trigger the dependent `when` blocks under the proper conditions. Additionally, loops complicate our code generation process in a number of other ways:

1. Iteration requires each block inside the loop to execute repeatedly, but under the control of the loop condition. This has a number of implications: (a) every block in a loop must be made to depend on the loop condition in some way, (b) we must be able to re-execute `when` blocks (unlike in a straight-line code, in which each block is executed at most once), and (c) any communication done inside a loop will be done repeatedly–therefore it is possible that a processor could find more than one message for an entry (from different iterations) in its message queue and must be able to process those messages in the proper sequence.

2. The loop condition may be a function of one or more of the variables defined in different blocks in the loop body. So, the dependence of the loop condition on specific blocks in the loop must be considered.

3. In the presence of *loop-carried dependences*, extra care must be taken to assure that the correct values are used for each iteration. A loop-carried dependence arises when a statement in a loop defines a variable that is used in a subsequent iteration.

4. Variables that are both defined and used inside the loop body but are used before they are defined obtain values *either* from the definition before entry to the loop (on the first iteration) *or* from the previous loop iteration (on subsequent iterations). Therefore, we need a mechanism for selecting the appropriate one.

5. Similarly, variables defined inside the IF/loop body must forward (signal the "readiness" of) their values to the statements that use those variables that follow the loop, but only if the IF branch/loop is taken. If it isn't, the defined values of those variables before the IF/loop must be used. We must select the appropriate values.

6. In a loop, we must ensure that only a definition from the *last* iteration is forwarded to subsequent uses outside the loop. This implies that the conditions that trigger the uses of those definitions must include the termination of the loop.

To illustrate, let's return to the previous example. A and $Anew$ are defined inside the loop. First, the statement which computes D must be enabled by *either* the defining statement in the loop or the statement that defines the variable if the loop is not executed (i.e., its defining statement before the loop). In terms of Dagger, the statement $D = Anew - A$ will appear in a `when` block that depends on condition variables that trigger the `when` block after either (a) the definitions of A and $Anew$ in the loop, or (b) the previous definitions of A and $Anew$ if the loop is not taken. Second, if the loop is taken (i.e. in case (a)), the A and $Anew$ definitions inside the loop must not trigger the $D = Anew - A$ statement until after the last iteration of the loop. Also, suppose that for some arbitrary processor P, a neighboring processor Q sends a message to P (say, its left boundary for a left CSHIFT). It could happen that before P processes this message, it sends its right boundary to Q (for the right CSHIFT), Q receives and processes all of its messages, and begins the next iteration. Q then sends another message to P before P has processed the first. P must be able to match each message with the proper iteration.

We first consider how to make all the blocks of the loop body execute just the right number of times, each variable using the correct value for every iteration. A Dagger program is completely data-driven–we have `when` blocks that are activated under certain conditions which correspond to the computation of data values. So we must construct a looping mechanism using this framework. We must also have a way for multiple independent execution paths to be coordinated by a single control mechanism (i.e. the loop condition) while honoring the loop-carried dependences. We take a conservative approach to satisfying these requirements by simply synchronizing all threads through the loop after each loop iteration.[1] This nullifies loop-carried dependences but may miss some opportunities for overlap. In future work, we will let loop iterations execute as independently as possible by identifying paths of execution that can execute independently and then providing extra support for allowing, if possible, different paths to execute different iterations. This is sufficient to coordinate control within a processor, but we must still deal with the problem of multiple messages arriving from

[1] Notice that we are *not* synchronizing across all the processors. We are only coordinating the `when` blocks within a loop, on each processor separately.

different loop iterations described above. We employ a Dagger mechanism called *reference numbers*.

A reference number is a tag that may be attached to a condition variable or message to enable Dagger to group a set of condition variables and messages together. This enables Dagger to distinguish between different instances of a computation (e.g. different iterations of a loop). To use a `when` block that depends on a set of such messages and condition variables, condition variables and entries are declared with the `MATCH` option which tells Dagger to match reference numbers when checking for fulfillment of a `when` block's dependences. A `when` block is not activated for execution until (a) a `Ready` statement (using a reference number) is executed for each of its condition variables cv_i, (b) for each of its entries e_i an `expect` for the same reference number has been executed, and (c) for each of its entries e_i, a corresponding message with the same reference number has been received. The reader is referred to [Gur93] for more details.

The DDG as previously described is nearly sufficient to support the translation of IF and loop statements. We augment the DDG with a new node type (the diamond) to represent IF and loop tests–these statements require some special additional code to be generated. The node-merging algorithm must be modified to handle the loops and IF statements of DP. The main point of this revision is to ensure that we don't merge nodes in different IF/loop nests.

6.2 Translating IFs and Loops

We turn now to the translation of IFs and loops. To begin with, carry out the following analyses:

1. During parsing, the *nest* of each statement is computed. We define the nest of a statement as the sequence of all loops and IFs, ordered from the outermost to the innermost, which contain the statement. We can then name each loop and IF statement and associate a *nesting label* with each statement. The nesting label of a statement is obtained by concatenating the names of the loops and IFs of its nest, again, from outermost to innermost. For example, the nesting label of a statement directly nested in a loop named l_2 which is itself enclosed in a loop named l_1 is $l_1\_l_2$.

2. Construct the DDG, ignoring all loops and IFs. I.e., as far as our dependence analyzer is concerned, the program consists entirely of straight-line code.

3. Do the node-merging on the DDG, but precluding merging two nodes which have different nests.

After completion of these steps, we will have a new DDG of merged blocks, each block containing a list of the statements it contains and marked by its nesting level and an identifier for the block. If a block is not nested inside a control structure, we can generate Dagger code for it as previously shown for straight-line code. When the nesting information reveals a loop, we need to know the following:

1. What statements (nodes) comprise the loop.
2. The variables that are defined within the loop.
3. The variables that are used within the loop.
4. The variables that are both used and defined in the loop and are used before they are defined in the loop body (and therefore need to be forwarded to the next iteration of the loop).

Given this information, we can generate code for loops. First, we must select between multiple alternative definitions for a use of a variable. Since DP has restricted control structures, we know that such selections need only be made in certain situations:

1. When a variable being used in a statement might be defined in an earlier IF or loop.
2. When a variable being used in a statement inside a loop might be defined by a previous iteration of the loop.

Consider the code and DDG in Figure 10 as an example of the first case. The statement $A = X$ may execute either (a) immediately after $X = Y$ executes, or (b) immediately after we discover that the branch will not be taken *and* $X = 1$ has executed. We express this in Dagger by the code in Figure 11. The compiler emits a `Ready` statement for each block in the body of the IF for the case that the branch is not taken. In similar fashion, when a variable is defined inside a loop and might be used inside the loop before it is defined, we must select between the definition before the loop (which is used in the first iteration) and the definition that forwards its value to the next iteration of the loop (which is used in subsequent iterations).

As previously mentioned, by synchronizing at the end of each iteration our DDG will not have loop-carried dependences. To accomplish this synchronization, each `when` block corresponding to a block nested in a loop will include code to call `Ready` for a condition variable that indicates completion of the block. Execution of the loop test is then made dependent on the completion of those blocks.

Another complication to be worked out can be illustrated by taking the example in Figure 10 and changing the IF to a WHILE as shown below.

$X = 1$
$i = 0$
$WHILE\ i < n$
$\quad X = X + Y$
$\quad i = i + 1$
$ENDWHILE$
$A = X$

We are tempted to translate this as in Figure 12. The problem with this translation is that the condition variable that triggers the scheduling of the last statement (`Block_2`) is set each time through the loop. This means that block `3` (which is waiting for the condition variable `Block_2` to be set) will be triggered after the first iteration whereas it should be triggered only after all iterations of the loop have completed. Our solution to this difficulty is to give an alias to each block that is nested inside an IF or loop. A condition variable with that name is used to signal the completion of that block during execution of the loop but once the loop exits, the "usual" name is used. This way, no code outside the loop needs to be aware that the definition is even in a loop. Similarly, we don't want blocks before an IF or loop to trigger blocks inside that IF or loop until the condition of the IF/loop has been tested. A similar renaming technique will handle that situation also.

7 Algorithm Outline

The following is the top-level view of our algorithm for code generation to Dagger. It is assumed that the program has been parsed and the usual information from that phase is available (e.g. parse tree and symbol table).

1. Compute the nest (inside IF or loop structures) for each statement.
2. Construct DDG_0, the initial data dependence graph, ignoring control structures.
3. Perform node-merging on DDG_0 to yield DDG, our working data dependence graph.
4. Starting with all children of the START node, traverse the DDG, generating code for each node as detailed below.

We begin by naming the nodes of the DDG. We will name the nodes of the DDG by some sequential numbering–this name we will call the *actual* name of the node. If the node is a loop or IF condition its actual name is the nesting label described in the last section. Each node that is nested in a loop or IF nest has one *pseudo name* for each "level" at which it is nested. This pseudo name is formed by concatenating its actual name with the nesting label for that level. The Dagger program will consist of `when` blocks which correspond to nodes of the DDG. The dependences are expressed by condition variables and entries in the `when` statement. The names of the condition variables will be chosen to correspond to appropriate DDG node pseudo names.

Next we consider the details of code generation for a single DDG node. First, note that circle nodes only indicate that a message is to be received–no `when` block corresponding to a circle node is produced. The following subsections examine different situations.

Suppose we have a DP DO WHILE loop L as follows

$DO\ WHILE\ (e(p_1, p_2, p_3, \ldots, p_k))$
$\quad D_1; D_2; D_3; \ldots; D_l;$
$ENDWHILE$

containing blocks $D_1, D_2, D_3, \ldots, D_l$. We generate the following when blocks for this statement. First, generate the `when` block to do the loop test.

`when` $node(p_1), node(p_2), node(p_3), \ldots, node(p_k)$:
$\quad$ if $(e(p_1, p_2, p_3, \ldots, p_k))$
$\quad\quad$ Ready$(take_loop_< loop\ name >)$;
$\quad$ else Ready$(exit_loop_< loop\ name >)$;

where $node(p_i)$ is the DDG node that defined variable p_i.

Next, generate the `when` block for the end of the loop. Recall that we make sure that all the blocks for an iteration have completed before starting another.

`when` $end\_body\_< loop\_name >$:
$\quad$ if $(e(p_1, p_2, p_3, \ldots, p_m))$
$\quad\quad$ Ready$(take_loop_< loop\ name >)$;
$\quad$ else Ready$(exit_loop_< loop\ name >)$;

Then, emit a `when` block to exit the loop. When we exit the loop, we must trigger blocks that depend on blocks $D_1, D_2, D_3, \ldots, D_l$.

`when` $exit\_loop\_< loop\_name >$:
$\quad$ Ready$(\gamma(D_i, L))$;

where $\gamma(D_i, L)$ returns the pseudo name for block D_i in the loop or IF that encloses L. (Recall that if L isn't nested in a loop or IF, $\gamma(D_i, L)$ (the pseudo name of D_i) will be its actual name.)

Finally, a `when` block is emitted to signal the end of the execution of the loop body.

`when` $\gamma(D_1, D_1), \gamma(D_2, D_2), \ldots, \gamma(D_l, D_l)$:
  Ready(*end_body*$<$ *loop_name* $>$);

We now turn our attention to the code that is generated for the blocks that are nested inside loops. Let D_i be one of the blocks within the loop L for some i. If D_i is a loop/IF, the same procedure being described here is applied recursively. Otherwise D_i is a basic block containing code to be executed. Let

$B_1, B_2, B_3, \ldots, B_n, C_1, C_2, C_3, \ldots, C_m$

be the nodes that D_i depends on where each B_i occurs outside L and prior to L in the textual order of the program and where each of C_i occurs within L prior to D_i. The compiler will emit the following `when` blocks that deal with D_i.

`when` *take_loop_*$<$ *loop_name* $>$,
  $B_1, B_2, B_3, \ldots, B_n, C_1, C_2, C_3, \ldots, C_m$:
    Ready(*start_*D_i);

`when` *take_loop_*$<$ *loop_name* $>$,
  *end_body_*$<$ *loop_name* $>$,
  $C_1, C_2, C_3, \ldots, C_m$:
    Ready(*start_*D_i);

`when` *start_*D_i:
  $<$ *code in* D_i $>$
  Ready($\gamma(D_i, D_i)$);

The first `when` block triggers D_i on the first iteration through the loop. The second `when` block does not trigger on the first iteration because of the *end_body_*$<$ *loop_name* $>$. It triggers D_i in the second and subsequent iterations when the blocks within the loop that it depends on (C_i) are completed. The B_is are "consumed" when the first `when` block is fired and so it cannot be triggered on subsequent iterations.

To handle the problem arising from the possibility of multiple messages from different iterations being available at the same time, we attach a reference number to each condition variable and message entry. This is done by maintaining a single counter called *refnum* which is incremented every time any loop body is iterated (when `Ready`(end_body_$<$ *loop_name* $>$) is executed). Every `Ready` and `expect` statement is also modified to include the current reference number as a parameter.

IF statements are handled analogously. The details are not included here due to lack of space.

8 Related Research

Numerous efforts have been undertaken recently to compile languages based on data parallelism to MIMD machines. Many of these efforts explicitly address the problem of generating efficient communication. Typically, this problem is attacked on a couple of orthogonal fronts: (1) generating efficient communication (often by attempting to minimize the amount of communication performed), and (2) attempting to schedule communication so that it overlaps with computation. In our compiler, we will profit by improvements in the former category but our work deals only with the latter. What distinguishes our work is that the order in which these tasks are performed is determined at run-time according to our adaptive schedule. We know of no other machine-independent run-time system that offers the asynchronous, message-driven virtual machine to which we compile.

[Koe91], for example, describes the generation of code for performing communication for distributed arrays in the Kali language. In that paper, mention is made of the fact that interleaving communication and computation can be done to improve efficiency. However, the Kali compiler must schedule its *receive* statements statically whereas our schedule is able to be dynamically determined because of our underlying message-driven virtual machine. Some other examples of projects in which efforts are made to overlap communication and computation include Fortran D [HKT92] and the Crystallizing Fortran Project [LC91]. All of these efforts must rely on statically scheduling program tasks.

Of course, the benefits of message-driven execution extend to the entire program (not just to communication/computation overlap) and serve to simplify the entire scheduling task. A comparable approach, known as the static *macro-dataflow* model of parallel computation, was employed in [Sar89]. This thesis describes the execution of a parallel program as the execution of units of computation called *macro-actors*. These macro-actors are similar to our **when** blocks–they are executed when they are "ready", and once initiated they run to completion. However, the run-time scheduler in this scheme is also responsible for *creating* the macro-actors once their control dependences have been met. In our case, a chare contains the entire program for a processor and the scheduler simply "awakens" it at appropriate entry points (**when** blocks). Furthermore, [Sar89] states that their scheduler is not designed to adaptively overlap communication with computation.

9 Conclusions

We have shown the feasibility of leveraging an asynchronous message-driven execution model to derive dynamic adaptive schedules of execution threads in support of a data parallel propgramming language. We have also given an algorithm that uses an annotated data dependence graph to produce such schedules in the form of code in the Dagger notation, which can be directly implemented on parallel machines. Dagger turns out to be an invaluable substrate for this problem, obviating the need to deal with many nitty-gritty details of synchronization. Running on top of Charm, it provides an easy path to a portable implementation of DP running on a variety of parallel machines.

The method we describe leaves open many optimizations that can be performed to reduce the scheduling overhead incurred. These include elimination of redundant dependences in certain `when` blocks, a reduction in the number of condition variables used, etc. Another avenue of future research involves relaxing the local synchronization done at the end of each loop body. This will allow different components of the loop body to be executing different iterations. We plan to explore these opportunities as well as carry out a full-fledged implementation of DP and its performance evaluation.

References

[Cha92] Parallel Programming Laboratory, University of Illinois Department of Computer Science, Urbana, Illinois. *The CHARM(3.0) Programming Language Manual*, December 1992.

[Gur93] A. Gursoy. Dagger: Combining the benefits of synchronous and asynchronous communication styles. Submitted to 1993 International Conference on Parallel Processing, 1993.

[Hig92] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, September 1992.

[HKT92] S. Hiranandani, K. Kennedy, , and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pages 139–176. Elsevier Science Publishers B.V., 1992.

[Kal90] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, August 1990.

[Koe91] C. Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, November 1991. Analyzes data distribution and access patterns for distributed data.

[LC91] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

```
/* Store Anew into A */
L1: CopyArr2D(A, Anew)
L2: Tmp1 ← 0

/* Up Shift and Add */
L3: CopyRow(Umsg → Array, A, u_edge)
L4: SendMsg(u_neighbor, Umsg)
L5: RecvMsg(Dtmp2)
L6: Utmp1 ← LocalShift(A,1,-1)
L7: CopyRow(Utmp1, Dtmp2, d_edge)
L8: Tmp1 ← Tmp1 + Utmp1

/* Down Shift and Add */
L9: CopyRow(Dmsg → Array, A, d_edge)
L10: SendMsg(d_neighbor, Dmsg)
L11: RecvMsg(Utmp2)
L12: Dtmp1 ← LocalShift(A,1,1)
L13: CopyRow(Dtmp1, Utmp2, u_edge)
L14: Tmp1 ← Tmp1 + Dtmp1

/* Left Shift */
L15: CopyColumn(Lmsg → Array, A, l_edge)
L16: SendMsg(l_neighbor, Lmsg)
L17: RecvMsg(Rtmp2)
L18: Ltmp1 ← LocalShift(A,2,-1)
L19: CopyColumn(Ltmp1, Rtmp2, r_edge)

/* Right Shift */
L20: CopyColumn(Rmsg → Array, A, r_edge)
L21: SendMsg(r_neighbor, Rmsg)
L22: RecvMsg(Ltmp2)
L23: Rtmp1 ← LocalShift(A,2,1)
L24: CopyColumn(Rtmp1, Ltmp2, l_edge)

/* Both left and right have arrived */
/* so scale by c and add them. */
L25: Tmp2 ← c * (Ltmp1 + Rtmp1)

/* Finish the computation of Anew */
L26: Anew ← (1/(2+2*c)) * (Tmp1 + Tmp2)
```

Figure 2: Intermediate code for Jacobi iteration

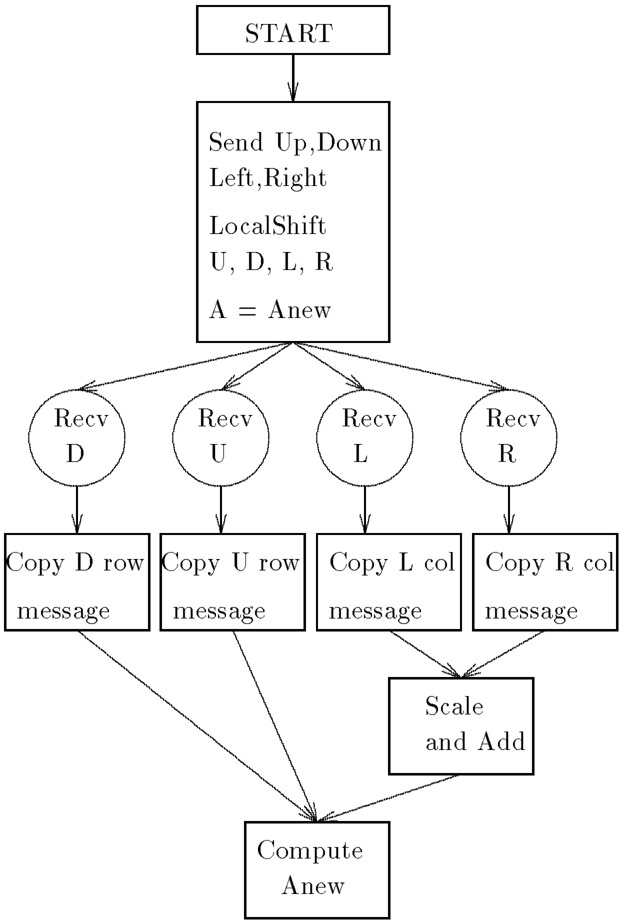

Figure 3: The Jacobi iteration dependence graph

```
Dag Chare Jacobi {
    CONDVAR both_left_and_right;
    entry  recv_up,recv_down,recv_left,recv_right;
    init: {
        /* Store Anew into A */
        CopyArrSec(A, Anew);
        Tmp1 <-- 0;
        /* Begin shifts--send boundary for each */
        CopyArrSec(Umsg-->Array, UBoundary(A));
        SendMsg(u_neighbor, Umsg);
        CopyArrSec(Dmsg-->Array, DBoundary(A));
        SendMsg(d_neighbor, Dmsg);
        CopyArrSec(Lmsg-->Array, LBoundary(A));
        SendMsg(l_neighbor, Lmsg);
        CopyArrSec(Rmsg-->Array, RBoundary(A));
        SendMsg(r_neighbor, Rmsg);
        /* Expect four shift messages */
        expect(recv_up);   expect(recv_down);
        expect(recv_left); expect(recv_right); }
    when recv_up: {
        /* Recv message, finish shift Up and add */
        RecvMsg(Dtmp);
        Utmp = LocalShift(A,up);
        CopyArrSec(Utmp, Dtmp, u_edge);
        Tmp1 <-- Tmp1 + Utmp; }
    when recv_down: {
        /* Recv message, finish shift Down and add */
        RecvMsg(Utmp);
        Dtmp = LocalShift(A,down);
        CopyArrSec(Dtmp, Utmp, d_edge);
        Tmp1 <-- Tmp1 + Dtmp; }
    when recv_left: {
        /* Recv message, finish shift Left */
        RecvMsg(Rtmp);
        Ltmp = LocalShift(A,left);
        CopyArrSec(Ltmp, Rtmp, l_edge); }
    when recv_right: {
        /* Recv message, finish shift Right */
        RecvMsg(Ltmp);
        Rtmp = LocalShift(A,right);
        CopyArrSec(Rtmp, Ltmp, r_edge); }
    when recv_left, recv_right: {
        /* Both left and right have arrived-- */
        /* scale by c and add them. */
        Tmp2 <-- c * (Ltmp + Rtmp);
        Ready(both_left_and_right); }
    when recv_up, recv_down, both_left_and_right: {
        /* Finish the computation of Anew */
        Anew <-- (1/(2+2*c)) * (Tmp1 + Tmp2); }
}
```

Figure 4: Dagger program outline for the Jacobi iteration example

$tmp1 = ComputeLocalMAX(X)$
Send tmp1 to global MAX
Receive back global MAX $\rightarrow k$
Send A to SORT routine
Receive sorted array $\rightarrow Tmp2$
$B \leftarrow Tmp2 + D$
$C \leftarrow A/k$

Figure 5: Intermediate code for code using MAX and SORT

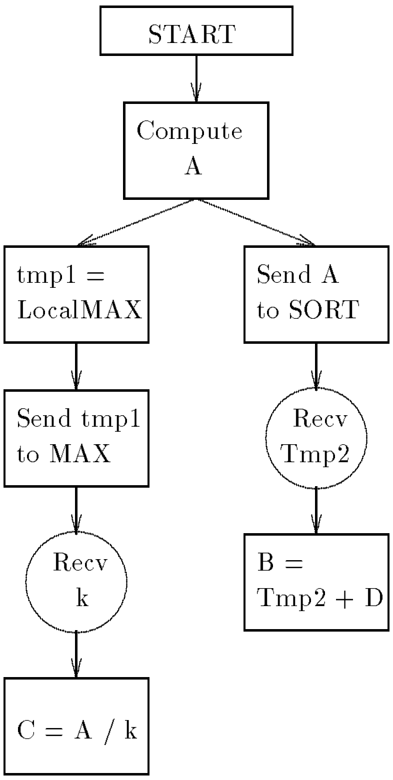

Figure 6: The data dependence graph for the MAX and SORT example

```
merge_DDG_nodes(v) {
    if (node_type(v) == CIRCLE)
        new_block(v)
    else if ((n_predecessors(v) == 1)
        if (node_type(first_pred(v)) != CIRCLE)
            merge_into_pred(v,first_pred(v))
        else
            new_block(v)
    else if (++visit_count(v) != n_pred(v))
        return
    else if (all_predecessors_same(v))
        merge_into_pred(v,first_pred(v))
    else
        new_block(v)
    for all w ϵ child(v)
        merge_DDG_nodes(w)
}
```

Figure 7: DDG node-merging for straight-line code

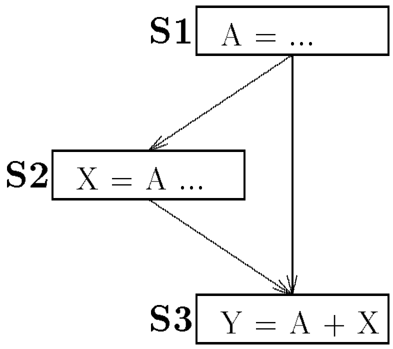

Figure 9: Redundant dependence arcs

```
X = 1
IF i < n
    X = Y
ENDIF
A = X
```

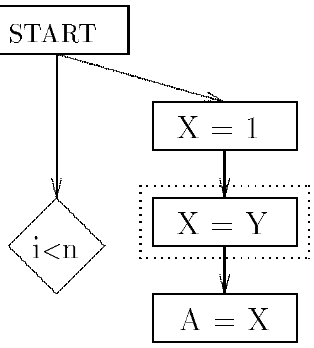

Figure 10: Code using multiple definitions of a variable with corresponding DDG

```
when Block_0 : {
    X = 1;
    if (i < n) Ready(take_branch);
    else       Ready(dont_take_branch); }
when take_branch : {
    X = Y;
    Ready(Block_2); }
when dont_take_branch : {
    Ready(Block_2); }
when Block_2 : {
    A = X;
    Ready(Block_3); }
```

Figure 11: Dagger program example for multiple definitions

```
when Block_0 : {
    X = 1;
    i = 0;
    if (i < n) Ready(take_branch);
    else       Ready(dont_take_branch); }
when take_branch : {
    X = X + Y;
    i = i + 1;
    Ready(Block_2); }
when Block_2 : {
    if (i < n) Ready(take_branch);
    else       Ready(dont_take_branch); }
when dont_take_branch : {
    Ready(Block_2); }
when Block_2 : {
    A = X;
    Ready(Block_3); }
```

Figure 12: (Incorrect) Dagger program for multiple definitions with loop