Yale University
Department of Computer Science

Run-time Scheduling and Execution of Loops on Message Passing Machines

Externally Distributed as ICASE report 89-7

Kay Crowley, Joel Saltz, Ravi Mirchandaney, Harry Berryman

YALEU/DCS/TR-657
January 1989

This work has been supported in part by the U.S. Office of Naval Research under Grant N00014-86-K-0310, authors Saltz and Mirchandaney were also supported by NASA contract NAS1-18605 while they were in residence at ICASE, NASA Langley Research Center, Hampton VA

Run-time Scheduling and Execution of Loops on Message Passing Machines*

Kay Crowley *Joel Saltz* † *Ravi Mirchandaney* †
Harry Berryman
Department of Computer Science
Yale University
New Haven, CT 06520

January 27, 1989

Abstract

Sparse system solvers and general purpose codes for solving partial differential equations are examples of the many types of problems whose irregularity can result in poor performance on distributed memory machines. Often, the data structures used in these problems are very flexible. Crucial details concerning loop dependences are encoded in these structures rather than being explicitly represented in the program. Good methods for parallelizing and partitioning these types of problems require one to assign computations in rather arbitrary ways.

Naive implementations of programs on distributed memory machines requiring general loop partitions can be extremely inefficient. Instead, the scheduling mechanism needs to capture the data reference patterns of the loops in order to partition the problem. Once this partition is obtained, a pre-processing step is required. First, the indices assigned to each processor must be locally numbered. Next, it is necessary to precompute what information is needed by each processor at various points in the computation. The precomputed information is then used to generate an execution template designed to carry out the computation, communication and partitioning of data, in an optimized manner.

In this paper, we present the design of a general preprocessor and schedule executer, the structures of which do not vary, even though the details of the computation and of the type of information are problem dependent. We draw the following conclusions from this work: it should be possible to solve a whole variety

*This work was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0310

†Additional support for these authors was provided under NASA contract NAS1-18605 while they were in residence at ICASE, NASA Langley Research Center, Hampton VA

of sparse and adaptive problems using a single mechanism for the setup phase of the problem. These mechanisms can then be incorporated into automated compilers for distributed memory machines as well as explicit language extensions. We also present results that illustrate the performance benefits that accrue from effective preprocessing.

1 Introduction

1.1 Overview

The focus of this paper is to explore methods of handling loop structures arising from problems such as sparse system solvers whose irregularity can result in poor performance on distributed memory machines. In these problems, crucial details concerning loop dependences are encoded in data structures rather than being explicitly represented in the program.

A distributed memory machine is a multiprocessor in which each processor has highly preferential access to its own local memory. Accesses to storage in other processors is substantially slower. Typically, there is no hardware support for accessing individual storage locations in remote processors. Each processor directly reads or writes only to its own memory locations. Remote memory fetches must be carried out in a rather roundabout manner. Processor A obtains the contents of a given memory location which is not on A by sending a message to processor B associated with the memory location. Processor B must be programmed to anticipate such a request, to satisfy the request and to return a responding message containing the information requested. In most architectures the sending or receiving of a message is accompanied by a communication startup time that is substantially higher than either the time of a computation or the time required to send or receive an incremental word of data.

Distributed memory machines have very substantial advantages in their abilities to deliver computational power at extremely high rates. When the work in scientific programs is partitioned in an appropriate manner, excellent performance can often be attained on machines or networks of multiprocessors with relatively rudimentary communication capabilities.

Unfortunately, general purpose modern codes for solving partial differential equations and sparse linear systems are examples of the many types of problems whose irregularity can result in poor performance on distributed memory machines.

One can use a shared memory model to execute a nest of loops on a distributed memory machine [1],[3]. In all but very coarse grained problems, the results will be very unsatisfactory unless mechanisms are used: 1) to anticipate what data will be needed in which processor and 2) to cluster communications to reduce the number of startups. This type of optimization is typically performed manually in problems with regular communication patterns that do not vary when the program is executed with different input parameters.

Attention must frequently be paid to how one is to partition work in a nest of loops in order to achieve a good load balance along with reasonable communication costs. When it is necessary to partition an array among processors in an irregular fashion, the location of an array element in a local storage may not have a simple relationship with any global numbering of the array elements. If these transformations between global and local coordinates are resolved during the loop execution, a substantial performance penalty will generally be paid.

The methods to be described here involve execution time preprocessing and in essence require a prior symbolic execution of a loop structure. The usefulness of these methods is clearly for inner loops of programs. Without them performance penalties would have to be paid during every loop iteration rather than being paid for once during an initial optimization.

In this paper, we present an outline of the infrastructure required to perform the optimizations outlined above on a class of loops we term *start-time* schedulable. A nest of loops is *start-time* schedulable if all data dependences are resolved before the program begins execution and if these dependences do not change during the course of the computation. This includes codes in which dependences are completely specified at compile-time. We also present ideas on how the use of the infrastructure we describe might be automated. Experimental data is included that quantifies the performance obtainable using the methods to be discussed here.

One long term aim of this research is to produce software that is able to interface with any program written on a distributed memory machine through what will appear to the user to be a subroutine call. The subroutines in question will be formed from standard templates we develop. These templates will be customized in a way that optimizes inner loop performance for loop structures specified by the user. The current research involves identification of techniques to achieve high performance for certain important classes of loops. Follow on research would include the development of tools that allow easy customization of the templates we provide.

1.2 Background

There are several research efforts under development which address the programmability of distributed machines. DINO[8] is a C language extension which has been designed for programming numerical algorithms on distributed machines. The constructs provided by DINO allow the programmer to specify mappings of data structures onto a virtual parallel machine. Accesses to local data are as usual but an access to a remote data object results in automatic insertion of a message. Thus, for very regular problems whose patterns of data accesses are known at compile-time, DINO relieves the programmer of specifying the *send-receive* pairs for the parallel program. Callahan and Kennedy[3] describe another approach to programming distributed memory machines. User annotations to a high-level program declare whether a data-structure is logically shared between the virtual processors and a one-one mapping between elements of the structure and virtual processors is provided. Clustering of elements onto a single physical processor is specified separately. The authors suggest a long list of possible optimizations that will be attempted; many of these are analogous to those carried out for shared-memory and vectorizing compilers. Koelbel et. al. describe BLAZE[5] which is a language extension to Pascal. Here a user program may describe a distribution of shared arrays among the machine's processors. The main loop construct discussed in this paper is the *forall* loop, distributed using strip-mining. The Linda system[1] provides an associative addressing scheme by which a reference to variables can be resolved at run-time; this in essence provides a shared name space for distributed memory machines.

In section 2 we describe in some detail the kinds of loop structures to which our methods apply and we provide an overview of the optimizations we investigate. In section 3 we outline the design of the methods we have explored. In section 4 we present experimental results on the Intel iPSC/2 that quantify the benefits that can be obtained through the use of our optimizations. We also present data that quantifies the costs that would be incurred were one to take a naive approach towards dealing with loops of the type we consider. Finally, in 5 we draw conclusions about the role of the methods we have been investigating. We also draw conclusions about some of the issues involved in mapping work from irregular or sparse problems onto distributed memory machines.

2 Prescheduling Sparse Loop Structures

In section 2.1, we present a set of programs that carry out a simple regular computation. These examples demonstrate why even simple programs coded using loops that involve flexible data structures can cause performance problems on distributed memory machines. In section 2.2, we describe the added issues that arise when loops with somewhat more complex dependency structures are considered. In section 2.3 we give an overview of the way we carry out the optimizations presented here.

2.1 Sparse Matrix Vector Multiply

To provide a context for what follows, we will present two programs that carry out a sequence of sparse matrix vector multiplies (Jacobi iterations) and what might be done to execute these programs on a distributed memory machine. In Section 4 we will present experimental results illustrating the differences in performance that arise

```
do iter=1,num
   do i=1,n
 do j=1,n
   x(i,j) = a(i,j)*xold(i+1,j) + b(i,j)*xold(i-1,j) +
c(i,j)*xold(i,j-1) + d(i,j)*xold(i,j+1)
 end do
   end do

   do i=1,n
do j=1,n
   xold(i,j) = x(i,j)
end do
   end do
end do
```

Figure 1: Jacobi Iteration

from the optimizations discussed here. The first version of the program is depicted in Figure 1. Such a problem should be partitioned in a manner that 1) distributes load between the processors roughly equally 2) limits the number of communication startups and 3) limits the size of the messages that need to be communicated between processors. Depending on the time required for communication startups, the typical strategy is to partition all arrays by strips or rectangular blocks [7]. Values of variables along each side of the periphery of the strips or blocks can be exchanged.

The program in Figure 1 performs a sequence of Jacobi sweeps or point relaxations over an n by n square. In this program, the variable values at domain point i,j are represented by `x(i,j)` and `xold(i,j)`. The values of `a(i,j)`, `b(i,j)`, `c(i,j)` and `d(i,j)` are used each iteration for the calculation of `x(i,j)`. Because of the regular geometry of this program, the programmer can easily identify variables whose values must be sent to other processors, and values which must be received from other processors. When this program executes, a single message can be formed from all variable values corresponding to the side of a rectangle. If we partition the program in Figure 1 between P processors using a vertical strip decomposition, Figure 2 gives the pseudocode for the corresponding message passing program. We use a FORTRAN 8x type notation in our pseudocode for depicting sending or receiving subarrays of floating point numbers.

In a distributed memory machine, array references must refer to memory locations on a particular processor. In Figure 2, we see that it can be straightforward to translate loops so that all references are to a processors local memory.

Consider Figure 3 which depicts a program that is somewhat more general than the

```
do iter=1,num

   do i=1,n/P
 do j=1,n
   x(i,j) = a(i,j)*oldx(i+1,j) + b(i,j)*oldx(i-1,j) +
c(i,j)*xold(i,j-1) + d(i,j)*xold(i,j+1)
 end do
   end do

   do i=1,n
do j=1,n
   xold(i,j) = x(i,j)
end do
   end do

   send xold(1;) to proc p-1
   send xold(n/P;) to proc p+1
   receive from proc p-1, put in xold(0;)
   receive from proc p+1, put in xold(n/P+1;)

end do
```

Figure 2: Message Passing Jacobi

```
S1  do iter=1,num

S2 do i=1,n**2
  do j=low(i),high(i)
S3  x(i) = x(i) + a(j)*xold(column(j))
 end do
   end do

S4  do i=1,n**2
   xold(i) = x(i)
end do
end do
```

Figure 3: Sparse Mesh Jacobi

program in Figure 1 but for the appropriate array initializations represents the same problem as the program in Figure 1. In the following figures we use a modification of a standard sparse matrix data structure where the non zeros in matrix `A` are stored in a one dimensional array `a`. Non zero elements are taken from consecutive rows of `A` and are assigned to `a` beginning with the leftmost column of each row of `A`. For each row i, `low(i)` and `high(i)` represent the locations in array `a` of the left and rightmost non-zero columns of the row in matrix `A`. The column of `A` corresponding to element j of `a` is given by `column(j)`. Rather than sweeping over a two dimensional array, we sweep over a one dimensional array where dependences are given by the integer array *column*. A problem with the same pattern of dependences as seen in Figure 1 could be specified by the program in Figure 3. This form of the code is important because sparse matrix solvers use this type of formulation.

In Figure 3 we use the arrays `a` and `column` to designate which array elements will be needed to compute the right hand side in statement S3. Unless we know how arrays `low` and `high` have been initialized, we do not know which elements of *column* and *a* will be needed on each processor. In a naive implementation of the algorithm, we would have to partition *column* and *a* in some regular manner and would have to fetch the array values when they are needed, possibly generating high performance penalties. A naive multiprocessor implementation of the code in Figure 3 also requires that a fetch from a remote memory be performed whenever $xold(column(j))$ in program statement S3 specifies a memory location not assigned to the processor on which the code executes. The experimental results in Section 4 quantify how costly this kind of implementation can be on a message passing machine.

We can systematically partition a problem to obtain a good balance between communication costs and load balance. In Figure 4, we show an example of a partitioned

```
do iter=1,num

S1   doall pe=1,num_processors
S2 do i=1,nlocal(pe)
S3 next = schedule(i,pe)
  do j=low(next),high(next)
S4  x(next) = x(next) + a(j)*xold(column(j))
  end do
   end do
 end doall

S5  do i=1,nlocal
   next = schedule(i)
   xold(next) = x(next)
end do
end do
```

Figure 4: Transformed Sparse Mesh Jacobi

version of the program shown in Figure 3. Each iteration of the doall loop S1 is assigned to a unique processor *pe*. Each processor *pe* loops over the indices assigned to it (statement S2), the indices assigned to *pe* are specified in statement S3 by the subarray *schedule(;pe)*. In this illustration, global names are still given to all array references and loop indices.

If we wished to store the required elements of arrays `a` and `column` in the processors where these elements are used, some sort of scheme would be required to associate the global index numbers with locations in local storage. To fetch values of `xold` in the right hand side of statement S4, we would have to (1) determine the processor p responsible for producing a variable value (2) determine the local identity of the variable on p and (3) instruct processor p to send the variable value.

Optimizations that can be performed on this partitioned program are: (1) data to be transmitted between processors can be formed into longer packets to amortize startup costs, (2) communications to be carried out by each processor can be prescheduled so that each processor knows when to send and when to receive values of specific variable, and (3) the global index numbers used to access elements of `x, old, a` and `column` can be translated into local index numbers that represent storage locations in each processor.

```
   do iter=1,n
 doall i=1,m
    .
    .
..   = x(*,iter-*)
    .
    .
   x(i,iter) = ..

  end do
end do
```

Figure 5: Sequence of Doall Loops

2.2 Prescheduling Loop Structures

We can perform optimizations of the type described in Section 2.1 in more general cases where we have a sequence of *doall* loops between which there are dependency relations. For instance in Figure 5 we have a sequential outer loop S1 whose loop body contains a doall loop S2. S2 contains an expression with variables that may have been written to during earlier iterations of S1.

Sets of recursion equations are a particularly important example of a type of problem that are programmed as a sequence of doall loops. These equations are frequently specified in a sparse format. Consider for example the program for solving a lower triangular system in Figure 6. In that program, we must assume that the outer loop S1 has to be executed in a sequential fashion. Sets of iterations of S1 (Figure 6) that can be concurrently executed can be identified by performing a topological sort on the dependency graph relating the left hand side of S1 to the right hand side. This sort is performed by examining the integer array `column`. In this way the sequential construct in Figure 6 can be transformed into a parallel construct consisting of a sequence of doall loops. Each doall loop represents a concurrently executable set of indices from S1 of figure 6.

It is frequently possible to cluster work carried out in solving recursion equations so that adequate parallelism is preserved but so that a reduction is obtained in the number of computational phases and hence the number of communication startups. For example, consider what is required for efficiently solving a set of explicitly defined recursion equations

$$y_{i,j} = a_{i,j}y_{i,j-1} + b_{i,j}y_{i-1,j} \tag{1}$$

on an X by Y point square (the recursion equations are subject to some suitable bound-

```
S1 do i=1,n**2
  do j=low(i),high(i)
S2  x(i) = x(i) + a(j)*x(column(j))
 end do
   end do
```

Figure 6: Sparse Mesh Lower Triangular Solve

ary conditions). We can concurrently solve for variables along antidiagonals, i.e. variables $y_{i,j}$ satisfying $i + j = k$ for positive k. The computation will be divided into $X + Y - 1$ phases. Assume we instead partition the domain into a grid with X/m point horizontal strips and X/n point vertical strips. The work in each of the resulting X/m by X/n point rectangles is scheduled as a single unit. This modified computation will require only $X/m + X/n - 1$ phases.

Work involved in solving recursion equations specified in a sparse format must be scheduled so that parallelism can be exploited. In order to take advantage of efficiencies that can be gained by clustering blocks of variables, operations assigned to a given processor executed during a particular phase must be scheduled in a specific order.

Figure 7 demonstrates how the computations for 1 are carried out. Note that a single triangular solve requires the solution of a sequence of doall loops (statements S1 and S2); within each doall loop a set of row substitutions are scheduled (statement S3). As mentioned above, carrying out the operations in the order given by the array schedule in statement S3 may be essential. Methods for clustering work in sparse programs that solve recursion equations are discussed in detail in [9].

2.3 Overview of Optimizations

First we must specify how the data structures appearing on the left hand side of expressions will be partitioned. For problems having sparse representations, partitions can be calculated using the integer data structures that represent dependency graphs (such as the array `column` in the examples shown here) Alternately partitioning information of variables appearing on the left hand side of expressions can be specified on an a-priori basis by a user with a good understanding of the computational behavior exhibited by an application. In our experiments described in Section 4 and in [6] we have used both methods. A system is being designed that symbolically transforms annotated programs to produce code that directly encodes dependency graphs. These graphs are encoded as integer arrays that can be directly interpreted by a preprocessor. Descriptions of problem partitioning methods can be found in [2], [4], [7].

```
S1 do phase=1,num_phases

S2   doall pe=1,num_processors
S3do j= 1, npoints(phase,pe)
S4 next = schedule(phase,pe,j)
   do k=low(next),high(next)
S5   x(next) = x(next) + a(k)*x(column(k))
   end do
  end do
 end doall

   end do
```

Figure 7: Transformed Lower Triangular Solve

When a partition is calculated or input, a preprocessing step is performed. The preprocessor precomputes the information needed to be sent to each processor at each point in the computation and locally numbers the indices assigned to each processor. This precomputed information is then used in a schedule execution template designed to carry out (in an optimized manner) the computation, communications on the partitioned data. In the examples displayed in this paper, the scheduled and partitioned indices are represented in array `schedule`, (Figures 4 and 7). The structure of the schedule executer and preprocessor do not vary, even though the details of the computation and the type of information are problem dependent.

The reason we assume iterative codes is because there is a cost associated with this preprocessing stage. For fine-grained computations, this cost will need to be amortized over several iterations. The start-time requirement is related; were the elements needed by an index of computation to change between iterations, the information computed by the preprocessor will be invalid after that iteration.

3 System Design

3.1 Overview

A *preprocessor* was written to precompute the communication patterns needed in the problems we solve. In addition, the preprocessor generates a request list for each processor detailing what data it will need to complete its computation and where this data can be found. The preprocessor also reorders the input data so it can easily be distributed to the processors. In the *executer*, each processor simply uses its request list and communication data to determine what communication and computation is needed during a computational phase. The entire computation is solved over a fixed number of steps. Each step or phase corresponds to one of the doall loops presented in Figure 5.

Consider what is involved in an implementation of an *executer* that doesn't utilize preprocessing to preschedule the transmission of needed data. Every time a processor p discovers that it doesn't have some data it needs it must go through the following steps: (1) determine what processor p' has that particular data; (2) send a "request to receive" message to p'; and (3) receive the requested data from p'. The executer could be slightly smarter and have p package all requests for p' during a computational phase into a single message. In addition to having much higher computational overhead, this implementation still necessitates twice as much computation as does the version of executer using preprocessed data. In a message passing machine, where the communication costs are extremely high, such an implementation is very inefficient (unless little or no inter-processor communication is needed).

Due to the static nature of the dependency relations between different input data elements in the problems we study (i.e. problems are *start-time* schedulable) these expensive "house-keeping" computations can be performed prior to the actual computation.

3.2 A Simple Schedule Executer

The executer performs computations in a streamlined manner. In this discussion, we will use as our running example the program in Figure 7. Elements of arrays that never appear on the left hand side of expressions are mapped to locations in local storage. Array a is an example of a data structure that would be treated in this manner. Data structure elements that appear on left hand sides of expressions can be organized in the order in which they are first written. An integer array is used to specify the *dependency list* ; this specifies the location in local storage of array elements of x that will be used on the right hand side of the expression. A distinction is made between elements of x stored on the local processor and elements that have to be obtained from other processors. Elements of x that must be obtained from other processors are stored in temporary buffers that are filled when data is obtained from other processors. In Figure 8, we treat the arrays `datalist, dependents, local` as stacks, the function `pop` returns the next element on the stack.

In Figure 8, we present a schematic version of an executer that corresponds to Figure 7. The example conveys how we construct an executer; the actual executer is programmed so that fewer array increments and references are required. Integer array `numelements` in statement S1 gives the number of indices assigned to each computational phase. In this version of the executer, we take advantage of the fact that elements of array x are written to in a consecutive manner. Array `numdependents` in statement S2 gives the number of dependents corresponding to the current local index. In the context of the lower triangular solve this is the number of non zeros appearing in a given row. The element of x required may either have been computed by the current processor and resides in its local solution vector or it has been computed by another processor and was transmitted to the current processor at the end of the last phase. Logical array `local` contains this information. If the index has been computed on the local processor, the local index number is stored in integer array `dependents` (statement S3). If the index has to be received from another processor, `dependents` reflects the location of the array element in temporary storage (statement S4). `datalist` stores the indices of the elements of a. Elements of a are never found on the left hand side of any expression in the executer so that it is possible to initially assign elements of a required in each processor to locations in local memory.

Several factors/goals influenced the design of this module but the most important were that (1) irregular problems be solved efficiently, and (2) executer be flexible so that its functionality could easily be expanded.

The first of these goals was acheived by using a preprocessor to precompute communication patterns, to perform local to global index transformations and to organize local processor data. We accomplished the second goal by minimizing the amount of function-specific code and presenting the input data in a form that is independent of the function being computed. The computation required to execute the triangular solve is found in statement S5. We present in brackets the statements required to modify this executer so that it will compute a sequence of sparse matrix vector multiplies. In this case, the number of elements in a phase in S1 does not vary with the phase. The same memory locations are accessed during each computational phase so the `local`, `datalist` and `dependents` stacks are reinitialized each computational phase.

The `CommunicateData()` routine constructs packets of data and sends packets to and receives packets from other processors. The preprocessor generated communication data is used to determine how to construct these data packets and where packets are sent and received.

3.3 A Preprocessor

The preprocessor requires an input *schedule*. The schedule designates the indices that are to be executed by each processor during every phase of computation. Three main tasks are performed by the preprocessor:

1. Determination of the communication pattern needed by each node;
2. Global to local address translation of the input data;
3. Reorganization of the data to allow efficient execution and generalization of the executer.

```
C
C  Evaluate, phase by phase, x(local)



 call initialize(...)
 do phase=0,NumPhases

 [MVM:i_lhs = 1; initialize(datalist,local,dependents)   ]

S1do i=1,numelements(phase)
   i_lhs = i_lhs+1
S2  do j=0,numdependents(phase,i)
   C  evaluate request element, it may be local or
   C  may have been computed by another processor
   C  and sent here by CommunicateData()
   C
  if (pop(local).eq.LOCAL) then

S3   reqvalue = x(pop(dependents))
  else
S4   reqvalue = recvPacket(pop(dependents))(pop(dependents))
  endif
end do
S5  x(i_lhs) = x(i_lhs) - reqvalue*a(pop(datalist))

[MVM:xnew(i_lhs) = x(i_lhs) + reqvalue*a(pop(datalist)) ]

  end do
[MVM: do i=1,numelements(phase)
x(i) = xnew(i)
  end do  ]
 end do
  CommunicateData();
```

Figure 8: A Schedule Executer

These three tasks are now discussed briefly, further details and an outline of a parallel version of the preprocessor are presented in the Appendix.

A sequence of send and receive communication calls are precomputed. The send and receive communication calls must be paired; everytime a processor *p1* sends data to another processor *p2*, *p2* must know to receive data from *p1* and processor *p2* must also be aware of the format *p1* has imposed on the data it is sending.

To amortize message passing startup costs in distributed memory machines, we use the schedule and the DAG to find out what information a processor p must receive from each of the machine's other processors during each computational phase. The list describing the information p must send each phaseeach phase to each of the other processors can be found directly from the list describing the information that must be received. This list can also be found by using the transpose of the computational DAG.

Programs in a distributed memory machine must always refer to locations in local memory. The programmer should be able to write loops referring to indices in a single shared index set. Global to local mapping is done by having the preprocessor determine the local address for each index element i. The local address of an element can be determined by examining the *schedule* since the schedule defines the order of execution. Once local addresses for all indices i are found, this information is used to: (1) rename the indices the list of index i's dependents (e.g. array dependents in Figure 8), and (2) translate the schedule designating what is to be sent and received at any given computational phase to refer to locations in local memory.

3.4 A more general example

The preprocessor and schedule executer can be customized to execute an arbitrary number of functions where each of these functions may be executed different number of times over subsets of the index space. Recall that thus far we have only considered code blocks with a single computation; Figure 9 depicts a slightly more general code. Functions `F_1` and `F_2` are executed over different parts of the index set. Further, there is a conditional in the inner loop of the code.

Because the code is assumed to be start-time schedulable, the dependences associated with each left hand side index can be determined once `array1` and `array2` have been initialized. Similarly, the conditional need not be evaluated for every iteration of the above code; an execution schedule can be created by the preprocessor so that the inner loop function `F_2` is computed for only those values of indices that satisfy the conditional. The schedule executor contains a list of indices which are used to evaluate functions `F_1` and `F_2` as many times as given by the values of the loop bounds and conditional values which may be computed at start-time.

In Figure 10, we depict a template for a schedule used by the executer. This template

```
  do i=1,N
 do j=low(i),high(i)
S1: a(j) = F_1(array1(j))
do k=1,n
if(array1(j) < array2(k))
S2:a(k) = a(j)*F_2(array(2))
 end if
end do
 end do
  end do
```

Figure 9: A more general computation

```
  struct st_logical_sched
 {
 int num_phases;
 int *functions;
 int *function_runs_this_phase;
 int *indices;
  }
```

Figure 10: Representation of Schedules

specifies the number of phases for a given code block, the list of functions, the number of times each function is computed and the list of indices in order of their function applications. The schedule executer is customized based upon the number of such functions within a given block of code. Thus, when there are two functions to be evaluated in a code block, the executer contains the code for these functions. The lists of left and right hand side indices are initialized and used as shown in Figure 8.

The preprocessor and the executer have been extended to solve the numerical phase of sparse incomplete LU factorization. In this problem, we compute elements of the factored matrix only at certain prespecified matrix locations. In the versions we have implemented, the preprocessor precomputes the precise pattern of computation. Conditionals that determine whether a matrix element is to be computed (based on the known location of non-zeros) are only seen by the preprocessor.

The schedule executer is customized based upon the number of such functions within a given block of code, as in Figure 9. Thus, when there are two functions to be evaluated in a code block, the executer contains the code for these functions. The lists of left and right hand side indices are initialized and used as shown in Figure 9, with the code within the executer being driven by the contents of the structure shown in Figure 10.

The software we plan will ultimately be able to interface with any program written on a distributed memory machine through what will appear to the user to be a subroutine call. The subroutines in question will be formed from standard templates we develop. These templates will be customized in a way that optimizes inner loop performance for loop structures specified by the user. User annotations could be used to specify the parts of a code that comprise the actual functions or computations. Thus, statements `S1` and `S2` may be encapsulated within an annotation such as `begin_node end_node`. For a more detailed description of this idea, the reader is referred to [6].

4 Analysis of Executor Performance

4.1 Overview of Executor Performance

In order to obtain an initial estimate of the efficiency of the executor on the Intel iPSC/2, we carried out a sequence of sparse matrix vector multiplications using matrices generated from a square mesh and a five point template. We expect this problem to parallelize well; the experiments are carried out not to demonstrate this obvious fact but to quantify the overhead caused by the extra non-floating point operations performed by the executor. In particular, we need to distinguish between inefficiencies due to the executor itself versus architecture and mapping related overheads such as communication delays and load imbalance. We performed a number of inter-related experiments to extract these factors.

Table 1: Matrix Vector Multiply 100 by 100 Mesh, 5pt. template

Processors	total time (ms)	computation time (ms)	executor overhead (ms)
1	409	409	37
2	207	205	19
4	107	103	10
8	55	52	6
16	31	26	3
32	18	14	2

Single processor timings from an optimized sequential program were compared with the parallel code run on a single processor. The optimized sequential code required $T_{sequential} = 0.372$ seconds while the parallel code on a single processor required 0.409 seconds. The overhead for using the executor in this case is approximately 9 percent.

We partitioned the loop indices evenly into blocks, assigning consecutive blocks of indices to physically adjacent processors of an iPSC/2. In Table 1 we depict the total time required to solve the problem on varying numbers of processors. A separate estimate of computation time T_{comp} was obtained by eliminating the communication calls. Because indices are partitioned evenly between processors in this very uniform problem, we expect the load to be almost perfectly balanced. We form an estimate of the overhead *due to extra operations performed by the executor*

$T_{comp} - \frac{T_{sequential}}{P}$

and depict this in Table 1. In may be seen that this overhead is roughly 10 percent of the parallel execution time.

To demonstrate that the executor is capable of achieving high efficiencies in an absolute sense, we reduce the relative contribution of communication costs by increasing the size of the problem. We compared timings from matrix vector multiplications using matrices generated from square meshes of sizes 100, 150 and 200 using a five point template. The parallel efficiencies for 32 processors were 0.65 , 0.75 and 0.81 for problems arising from 100, 150 and 200 point meshes respectively. As usual, we define parallel efficiency as the ratio between the execution time for the optimized sequential program divided by the product of the number of processors and the execution time of the multiprocessor code.

Table 2: Comparison with Shared Memory: Matrix Vector Multiply

Processors	Linda time (seconds)	executor time (seconds)	time given 100 percent efficiency (seconds)
8	17.936	0.055	0.047
16	12.679	0.031	0.023
32	11.724	0.018	0.012

4.2 Comparison of Executor Performance with that of a Shared Memory Simulator

The Linda system[1] provides an associative addressing scheme by which a reference to variables can be resolved at run-time, this in essence provides a shared name space for distributed memory machines. We used the Linda system to estimate what efficiencies one might expect if one were to fetch various required array values on a demand basis with no preprocessing. We performed the same matrix vector multiplication experiments described in section 4.1 using a matrix generated from a 100 by 100 mesh.

Referring to Figure 3, each element of array `xold` was fetched from the storage location assigned to it by the Linda system when a need for that element was encountered. All elements of `a` and `column` corresponding to a given row were stored contiguously and fetched as a single unit. Table 2 depicts the timings obtained using the Linda code along with a repetition of the timings obtained using the executor. The striking difference in timings can be understood when one considers that using the iPSC/2 as a shared memory machine requires having to pay several milliseconds per data fetch. It is important to emphasize that optimizations similar to the ones discussed here could be fruitfully employed in conjunction with the Linda system so that Linda's apparent shared memory would be used much more sparingly.

4.3 Distributed Memory Model Problem Analysis

In section 4.4, we will experimentally examine the performance of the executor in the more challenging problem of solving sparse triangular systems. To properly interpret the results we obtain, we need to first examine load balance communication cost trade-offs in the context of solving the recursion equations in Section 2.2, equation 1. This dependency pattern is seen in a lower triangular system generated by the zero fill factorization of the matrix arising from a X by Y point rectangular mesh with a five point template (this system might arise in preconditioned Krylov space iterative linear system solvers). We will utilize P processors and partition the domain into n horizontal strips where each strip is divided into m blocks.

With very general assumptions, we show below that it is optimal to make the vertical strip size Y/n as large as possible and to decrease the horizontal block size X/m until the increased communication time becomes larger than the benefit of decreased idle time. We assume for convenience that m and n are multiples of P, and let S be the time required to perform the sequential computation. We define T_C to be the time taken to perform the computation in a block for a given m, n and S, and assume that $T_C = S/mn$.

Estimated total time without communications can be expressed as the sum of the time that would be required were the computation evenly distributed between processors in the absence of any load imbalance plus the time wasted due to load imbalances:

$$\frac{T_C mn}{P} + \frac{T_C min(m,n)(P-1)}{P}$$

where T_C = calculation time per block.

We now derive and expand upon the expression representing the time wasted due to load imbalances. The number of phases is equal to $m + n - 1$. Assuming that m and n are multiples of P, the term for idle time can be derived by noting that during any phase $j \leq min(m,n) - 1$ when j is not a multiple of P, there are $P - j \ mod \ P$ processors idle When j is a multiple of P, no processors are idle. Thus the sum of the processor idle time for $j \leq min(m,n) - 1$ is:

$$\frac{T_C min(m,n) \sum_{l=1}^{P}(l-1)}{P * P} = \frac{T_C min(m,n)(P-1)}{2P}$$

Through similar reasoning, the sum of the processor idle time for the last $min(m,n) - 1$ phases is the same. During the intermediate phases, the load is balanced with $min(m,n)$ blocks assigned to each processor. Thus the total idle time is:

$$\frac{T_C min(m,n)(P-1)}{P}$$

If there are no communication costs, $T_C = \frac{S}{mn}$, where S = sequential time. Then the total estimated time equals

$$\frac{S}{P} + \frac{S(P-1)}{max(m,n)P} \tag{2}$$

Thus in the absence of communication costs, all terms involve m and n in a symmetric manner.

First we show that in the presence of communication costs, we should choose $m \geq n$. We calculate the size of the largest message that must be sent between two processors during each phase. We assume that the time required for communication is equal to the sum of the times required each phase to send the largest messages. This tacitly assumes that the system is essentially synchronous, that computation and communication occur in alternating non overlapping periods of time.

The time required for communication can be safely assumed to be an increasing function of message size. For phases 1 through $min(m, n) - 1$, the maximum number of data values sent by any processor is $\lceil p/P \rceil * B_S$, where p = phase number, $B_S = X/m$, and X = horizontal dimension of the matrix. For phases $min(m, n)$ through $m + n - min(m, n)$, the maximum number of data values sent by any processor is $\lceil min(m, n)/P \rceil * B_S$, and for phases $m + n - min(m, n) + 1$ through $m + n - 1$ a maximum of $\lceil (m + n - p)/P \rceil * B_S$ data values. If B_S were held fixed, the time required for communication would be symmetric in m and n. Since B_S is a decreasing function of m, it is always advantageous from the standpoint of communication cost to choose $m \geq n$. Since equation 2 is also symmetric in m and n, the minimum total time always occurs when $m \geq n$.

To minimize all terms involving n, we should chose n to be as small as possible, *i.e.* P. For $m \geq n$, equation 2 has no dependence on n. For any given m, the communication cost does not increase with decreasing n. If dependency graph G_1 has m by n_1 points and dependency graph G_0 has m by n_0 points, with $n_1 \leq n_0$, G_1 can be embedded in G_0. Since the communication cost per block ($B_S = X/m$) is dependent only on m, G_1 need have a communication requirement no greater than G_0. We thus conclude that the vertical blocks chosen should be as large as possible.

4.4 Executer Performance

We will show that we can account for the execution time in solving sparse triangular systems by estimating the time lost due to both load imbalance and communication delays. As mentioned earlier the input schedule dictates: (1) how the problem is partitioned; and (2) the computational granularity. Computational granularity is a crucial determinant of performance in message passing multiprocessors which possess relatively high communication latencies. It is important to aggregate or clump work in a way that leads to a controlled trade off between load imbalance and communication requirements. Optimal partitioning (i.e. mapping of the unaggregated problem) cannot be acheived without taking into account the geometrical relationship between index elements. Aggregation methods used to generate input schedules arising from sparse forms of a wide variety of recursion equations are described in [9]. An aggregation strategy that can be used for the sparse versions of the recursion equations was discussed in Section 4.3. The granularity of parallelism is parameterized using the parameters m and n introduced in Section 4.3. Recall that as m and n increase, the size of the scheduled computational grains decreases.

In Table 3 we depict the sequential time, parallel time, time required by the parallel program run on one processor, estimated communication time, and estimated communication free time. on 32 nodes of an Intel iPSC/2. The communication time estimate is obtained by running problems in which computation is deleted but communication patterns are maintained. The estimated communication free time indicates the computation time that would be expected in the absence of communication delays, this is given by equation 2; for this purpose we used the processor parallel time as the sequential time. The problems solved were on a 192 by 192 point domain and on a 576 by 576 point domain. It was not possible to obtain sequential times or one processor parallel times for the larger problem; the times shown were extrapolated from the 192 by 192 point problem.

We note that for the problems on a 192 point square, the estimated communication free time added to the estimated communication time add up to a quantity that is close to the total time measured. Because these three quantities are derived from distinct experiments, this gives us some confidence that we are able to explain the timings observed. There is more of a discrepancy for the problem on a 576 point square. Since the one processor parallel time used is just an estimate, we expect that our estimate of communication free time here will be less accurate.

We note that when we employ a very fine grained parallelism (m and n equal to problem size) we pay a very heavy communication penalty relative to the computation time. The completion of this non-computationally intensive problem requires 383 phases, each one of which requires processors to both send and receive data. We can reduce the number of computational phases, and consequently the communication time, at the cost of increased load imbalances.

The overhead required for the operation of the executer appears to be captured well by the differences between the sequential time and the time required for the parallel code to execute on a single processor. Note that the overhead attributable to the executer, roughly 33 percent, is substantially larger here than it was for matrix vector multiply. This is understandable because the executor is having to coordinate a long sequence of distinct, rather fine grained computational phases.

While appropriate choice of computational granularity is essential for maximizing computational efficiency, the nature of the triangular solve limits the performance that can be obtained in problems that are not extremely large. The parallel efficiency obtained in the 576 by 576 point mesh was 34 percent. This is roughly half of the speed available once the overhead of the executor itself is accounted for.

It should be noted that the same types of experiments were performed on the iPSC/1 [6]. On the iPSC/1, floating point operations are much more expensive than they are on our iPSC/2. Estimates performed on the iPSC/1 indicated that the overhead introduced by the executer was substantially smaller, and the parallel efficiencies were correspondingly larger.

In Table 4 we present results from a somewhat less regular problem solved on a 32 node Intel iPSC/1 hypercube. Given appropriate clustering techniques, an extremely high degree of regularity is not essential for achieving the efficiencies possible in these

Table 3: Matrix from square meshes, 5pt. template

Mesh Size	n	m	total time (ms)	commun. time (ms)	commun. free time(ms)	seq time (ms)	1 proc parallel time (ms)
192	192	192	295	250	32	591	885
192	32	192	184	149	32	591	885
192	32	64	105	63	41	591	885
192	32	32	99	52	54	591	885
576	32	64	492	216	370	5319	7965

Table 4: Matrix from 300x300 mesh, 5pt. template reduced system

m (=) n	efficiency	total time	phases	communication time
300	0.16	3.99	598	3.45
150	0.31	2.07	299	1.57
75	0.53	1.21	149	0.88
50	0.44	1.44	99	0.65
38	0.33	1.95	75	0.45

types of problems. The matrix for this problem is obtained in a rather involved manner discussed more fully in [6]. We begin with a matrix obtained from a 300 by 300 point mesh using a five point template. A *reduced system* is obtained from this matrix by modifying the matrix in a way that increases the number of non-zeros per row and halves the number of rows.

In this problem the best efficiency was to 53%. The ability to aggregate work and control the number of communication startups plays a central role in obtaining increased efficiency.

5 Conclusion

There exist many types of problems (e.g. sparse system solvers) whose irregularity can cause problems for distributed memory machines. Good methods for parallelizing and partitioning these types of problems require one to assign computations and data in rather arbitrary ways. Efficient implementations tend to involve considerable programming effort to get good performance, making system development unnecessarily time consuming.

We have described a system that efficiently solves a variety of irregular problems that are *start-time* schedulable. This system, which consists of a preprocessor and a schedule executer, inputs a schedule which dictates/reflects how the input index data is to be partitioned. The preprocessor precomputes the information needed to be sent to each processor at each point in the computation and locally numbers the indices assigned to each processor. This precomputed information is then used in a schedule executer which is designed to carry out (in an optimized manner) the computation, communication and the partitioning of data. The structure of the schedule executer and preprocessor do not vary, even though the details of the computation and the type of information are problem dependent.

We draw the following conclusions from this work:

1. It is possible to design a system that uses a single infrastructure to solve a variety of sparse and adaptive problems.
2. It is likely that parallelizing compilers will have to customize well tuned template programs rather than generate the very complex programs needed to efficiently partition and parallelize rather simple loops.
3. The computational complexity of the preprocessor setup is relatively high. This implies that preprocessors should be used only when a loop is to be executed many times.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, August 1986.

[2] T. Allen and G. Cybenko. *Recursive Binary Partitions.* Technical Report, Tufts University Dept Computer Science, October 1987.

[3] D. Callahan and K. Kennedy. *Compiling programs for distributed-memory multiprocessors.* Technical Report TR88-74, Rice University, 1988.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[5] C. Koelbel, P. Mehrotra, and J. VanRosendale. *Semi-automatic problem partitioning for parallel computations.* Report 88-16, ICASE, 1988.

[6] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principals of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, July 1988.

[7] D. M. Nicol and J. H. Saltz. *Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors.* Report 87-39, ICASE, September 1987.

[8] M. Rosing and R. Schnabel. *An overview of DINO - A new language for Numerical Computation on Distributed Memory Multiprocessors.* Technical Report CU-CS-385-88, University of Colorado, Boulder, 1988.

[9] J. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, to appear, 1989.