Yale University
Department of Computer Science

The Automated Crystal Runtime System: A Framework

Joel H. Saltz, Ravi Mirchandaney, Roger M. Smith,
David M. Nicol and Kay Crowley

YALEU/DCS/TR-588
January 1988

This work has been supported in part by the U.S. Office of Naval Research under Grant N00014-86-K-0564

The Automated Crystal Runtime System: A Framework *

Joel H. Saltz[§†] *Ravi Mirchandaney*[§] *Roger M. Smith*[§]

David M. Nicol[‡†] *Kay Crowley*[§]

January 1988

[§]Department of Computer Science
Yale University
New Haven, CT., 06520
saltz-joel@yale.arpa

[‡]Department of Computer Science
College of William and Mary
Williamsburg, VA 23185
nicol@icase.arpa

Abstract

There exists substantial data level parallelism in scientific problems. The Crystal/ACRE runtime system is an attempt to obtain efficient parallel implementations for scientific computations, particularly those where the data dependencies are manifest only at runtime. This can preclude compiler based detection of certain types of parallelism. The automated system is structured as follows: An appropriate level of granularity is first selected for the computations. A directed acyclic graph representation of the program is generated on which various aggregation techniques may be employed in order to generate efficient schedules. These schedules are then mapped onto the target machine. We describe some initial results from experiments conducted on the Intel Hypercube and the Encore Multimax that indicate the usefulness of our approach.

Using the runtime system, it will be relatively easy to program different applications and study the performance implications of the various parameters. When the performance data is available, we would like to develop mathematical models that describe the relationships between the various important parameters in the system.

*This work was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0564.

[0]† Supported by NASA Grant NAS1-18107 while consulting at ICASE, NASA Langely Research Center, Hampton, VA 23185

1 Introduction

There is substantial data level parallelism that can be exploited when one seeks to solve problems in a large number of application areas. Many algorithms have been run with good speedups on parallel machines with vastly differing architectures. The list of algorithms that have been implemented on multiprocessors include:

- iterative and direct methods for solving linear systems of equations
- explicit, implicit and operator splitting methods for solving partial differential equations
- image and signal processing algorithms
- graph and combinatorial algorithms

Unfortunately, experience has shown that a very substantial programming effort is required to obtain correct solutions and good performance on many currently available architectures. Parallel programs are often much less general and flexible than sequential programs. The complexity involved in partitioning the data and computation causes programmers to tend to take advantage of problem specific information in mapping and scheduling work. In order to do this, programmers need to have good understanding of the architecture and the problem. Furthermore, when problem partitioning must be explicitly specified, the number of lines of code required to solve a problem using current methods often increases dramatically on a parallel machine.

High performance multiprocessor architectures differ both in the number of processors, and in the delay costs for synchronization and communication. In order to obtain good performance on a given architecture for a given problem, an appropriate choice of granularity is essential. Hand tailoring programs to obtain good performance on a given architecture is a laborious, error prone task that is often of only transient value due to changes in hardware or changes in the implemented algorithm.

In this document we outline the design of the *Automated Crystal Runtime Environment*, or simply ACRE. ACRE allows for the high level specification and control of computational granularity along with a robust load balancing mechanism for automated scheduling and mapping. While the system is being developed to complement the advantages exhibited by Crystal, our design is only loosely coupled to the language Crystal. We are endowing ACRE with a C interface designed to allow wide use of its features. The rest of this document is organized as follows: In the remainder of Section 1, we illustrate the need for a runtime system. Section 2 includes the motivation for automating the main features of the runtime system. In Section 3, we outline ACRE's organization, and provide an example of the C interface being designed. Section 4 comprises a description of the various issues concerning mapping and scheduling

of computations. In Section 5, we substantiate the promise of our methodology with empirical data, and summarize the paper in Section 6.

Throughout this paper, we illustrate our approach with functions that are used in conjugate gradient type algorithms for the solution of linear sytems which are preconditioned by incomplete factorization (we refer to these as PCG algorithms). The problem itself is important and the algorithms are reasonably representative of a large body of scientific applications. The two main areas where a runtime system can provide performance benefits are the following:

- Automatic detection of parallelism that cannnot be determined by a compiler due to data dependencies that become manifest only at runtime.
- Partitioning and mapping of the computation in a manner that is able to take advantage of the multiprocessor architecture.

We illustrate the above issues with example C programs.

One crucial issue is the ability to extract sufficient parallelism from a computation despite ambiguities in data dependencies that may be present during compilation. Take for example the example of the sparse triangular solve as written in C and depicted in Figure 1. Imagine that we parallelize the computation by allocating all work associated with a given row to a single processor. If the matrix is dense, the solution must proceed in a strictly sequential manner, with the solution of row 0 followed by row 1 and so on as the structure of the matrix necessitates this schedule. But, in many sparse matrix computations, each row has only a few non-zero elements. For a given row R, solution values from the rows corresponding to the columns of the non-zero elements of R are needed before R may produce its solution value. This implies that the solution for row R need only wait for solutions from a few other rows. Once these solutions are available, a solution for row R can be obtained.

The structure of matrices used in many scientific problems are defined at runtime. In the example of the triangular solve, the compiler must sequentialize the solution of matrix rows. A parallelizing compiler could detect parallelism obtainable from the inner products involved in solving individual rows, however, in many cases there are very few non-zero elements in a row so that the amount of parallelism available from intra-row parallelization is not substantial. We depict the relevant experimental results in Section 5.

Another aspect of ACRE deals with partitioning and mapping of the computation in a manner that is able to take advantage of the multiprocessor architecture. In many instances, the canonical representation of a computation is specified at a very fine computation grain. As a result of this, various performance problems can occur, as described in [21]. Using the data dependencies discovered at runtime, the system uses this information to aggregate work into units which take into account the characteristics of machine architecture.

```
              Data  Structures
              ---------------
struct st_matrix
   {
    int *col;
    float *value;
   }

struct st_matrix *matrix;
double soln;


              Code Segment
              ------------

for (i=0; i < nrows; i++)
   for (j=0; j < MAXCOLS; j++)
     {
      soln[i] = b[i] - matrix[i].value[j]*soln[matrix[i].col[j]];
     }
```

Figure 1. Triangular Solve

An example of the relevance of aggregation may be seen in the code for the sparse Jacobi iteration, shown in Figure 2. One of the natural methods for partitioning the work related to this function is based upon the distribution of one or more rows onto each processor. A compiler can detect the parallelism here and distribute this work among the processors. However, data dependencies affect the frequency of communication between the processors. A compiler based mapping is unable to take advantage of this important fact because the structure of the sparse matrix (and consequently the data dependencies) is not available to the compiler. To illustrate this aspect of the problem, we depict a 6x4 mesh with two possible partitions onto a 4 processor system. A partitioning that takes into account geometry is shown in Figure 2b, with the mesh having been divided into four quadrants and each of these being assigned to a processor. The property of this partition is that the amount of data transfer between iterations grows as $N^{1/2}$, where N is the size of the mesh. Depending upon the way the mesh points are numbered, a compiler, because it does not possess any knowledge of the problem structure, might assign rows 1-6 to processor 1, 7-12 to processor 2 and so on, as shown in Figure 2c. Asymptotically, this method incurs a communication cost of order N, which is significantly worse. When the problem is less regular than the one shown, the effect of not knowing problem geometry tends to become even more significant.

The system also allows for the automated pipelining of communication and/or synchronization. Dynamic and static mapping is directed towards a virtual message passing machine where binding of work to processors occurs only at lowest level of the system. Performance considerations that determine how particular problems are to be mapped have been widely considered (see for example [11], [19], [9]). Methods of problem mapping and performance analysis that are applicable to a broader range of scientific computations are discussed in [8], [4], [7], [21].

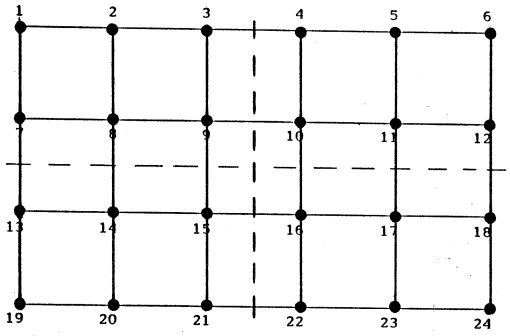
In some problems, the data dependencies are determined by functions whose parameters are other functions whose values are only determined at some unknown point during the computation. In these cases, a self-scheduled approach must be used in parallelizing the computation. An example of such a computation is the sparse LU factorization with incomplete fill, utilized in PCG algorithms. A very high level description of the data dependencies arising in this computation is shown in Figure 3.

In the above form, the pivot rows (signified by the outer index `i`) operate one at a time on all rows depicted by index `j`. `next(j)` is a function that provides the index of the pivot row that needs to modify row `j`. However, the value returned by this function could depend upon fill created by the operation of some earlier row on row `j`. Hence, it is not possible to preschedule the computation resulting from the inner loop of the above code, resulting in the need for self scheduling.

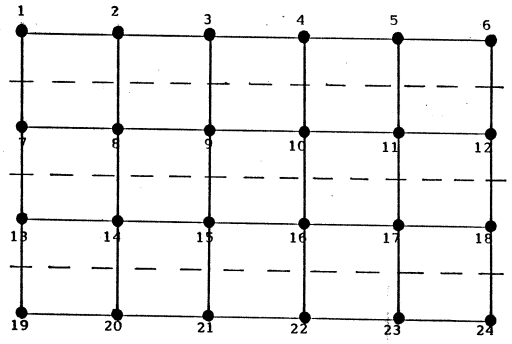
A number of programming enviroments have been developed that facilitate the creation of programs in which pools of work are dynamically shared among a number of processors [15],[2], [12]. While the self-scheduling methods described here use principles that are in many ways analogous to those described above, an explicit description of

```
/* Sparse matrix jacobi */

              Data Structures
              ---------------

struct st_matrix
    {
    int *col;
    float *value;
    }
struct st_matrix *matrix;

              Code Segment
              ------------

   for(i=0;i<num_iters;i++)
   {
    for(i=0;i<numrows;i++)
    {
      for(j=0;j<matrix[i].ncol;j++)
      {
      newsoln[i] -=matrix[i].value[j]*
              oldsoln[matrix[i].col[j]];
      }
    }
    for(i=0;i<numrows;i++)
    {
    oldsoln[i] = newsoln[i]
    }
   }
```

(a)



(b)



(c)

Figure 2. Sparse Jacobi Iterations

```
for (i=0; i < ROWS; i++)
  {
   for (j=i+1; j < ROWS; j++)
     {
       if (next(j) == i )
           modify(j, i)
     }
  }
```

Figure 3. Sparse LU Factorization

the parallel algorithm is not necessary. In self-scheduled computations as a whole, the choice of appropriately sized schedulable computational units is of great importance. Overhead can be reduced by the appropriate choice of computatational granularity, but the self-scheduled nature of the computation makes aggregating work units difficult.

The advantages of our approach include a great simplification of programming. The system design is deliberately architecture insensitive; the system scales if the architecture is augmented due to technological developments and the system has the potential to dynamically scale if portions of the system become unavailable.

The issues that must be understood in order to ensure the development of a robust system requires a substantial amount of experimental work. The problems to be studied in this context include:

- solution of sparse linear systems by Preconditioned Conjugate Gradient Methods. These methods require forward and back substitution of very sparse triangular systems, incomplete matrix factorizations, matrix vector multiplies and inner products,
- discrete event and time-stepped simulations
- realistic simulation of neural networks,
- adaptive mesh solutions of fluids problems,
- string matching problems.

The machines utilized for various portions of this work include the Intel iPSC, the Encore Multimax and the Connection machine. Work is partitioned among the processors by assigning different index values associated with a computation to different

processors. It is possible to perform these index partitions using various different techniques, as we will see later in this document.

2 Motivation for Automated Runtime System

Various research efforts have been underway to write compilers for both imperative as well as applicative languages in order to extract parallelism from user programs written in a sequential form [17], [13], [10]. While there has been considerable success in this endeavor, it has been recognized that 1) compile time data is not always adequate to exploit hidden parallelism that may manifest itself only at runtime, and 2)sophisticated aggregation and mapping techniques may be needed once this runtime parallelism is detected.

There have been several efforts in which problem specific information is utilized at runtime to detect parallelism and perform appropriate mappings of these computations onto parallel machines. Fox [8] has utilized a scatter decomposition strategy for the Caltech Hypercube. This method partitions the problem domain into a fine lattice of tasks (which are far more numerous than the processors) and then scatters each processors work assignment throughout the domain. Baden [3] has used a method that takes advantage of the locality of interactions in certain types of problems in fluid dynamics and implemented a dynamic load balancing scheme for the Intel iPSC. Nicol and Saltz [7] have implemented the triangular solve and a battlefield simulation program on the Intel iPSC and the Encore Multimax. Lusk and Overbeek [15] implement a self scheduled mechanism to dynamically allocate work to processors. While this method has the advantage of simplicity, there are many potential problems, especially with distributed memory machines.

In many of the examples we have seen, each user ends up designing their own specific runtime system. The problems with this approach are numerous. Primarily, each time there is a slight change in the application, reprogramming may be needed. Each new application will warrant a completely new runtime system specific to the problem. We believe that this approach hinders the speedy development and understanding of parallel systems and applications. Thus, our goal is to provide the runtime detection of parallelism and mapping using an automated system which is able to abstract away the unnecessary details of different computations. We use a standard directed acyclic graph, or DAG representation for this purpose. It is possible that we may be unable to generate optimal performance for any of the problems, but if the system provides good performance over a large range of application areas, then the significantly reduced programmer effort involved will be very important. This aspect becomes even more significant as the size and complexity of parallel machines as well as the applications being run on these machines, increases dramatically. There is an obvious but important analogy with the use of demand paging in virtual memory systems. One might expect

a programmer to know enough about the runtime behavior to allow him to improve performance by taking direct control over paging decisions. But he is likely to make errors, will certainly spend more time coding, the ultimate performance gains may not be substantial.

3 Methodology

There is a proliferation of languages being used to program parallel machines; from parallel versions of FORTRAN and C, to new parallel languages. Some of these languages are functional in nature and are able to exploit program parallelism without tremendous effort on the part of the programmer. They possess the useful property of being side-effect free, making the detection of parallelism much less problematic [5], [10]. One such language language being designed here at Yale is called Crystal.

Crystal is a very high level machine independent functional language that enables the user to specify parallel algorithms. This language provides an easy to use notation that allows a direct mathematical specification of a problem. Crystal is designed to have the modularity and freedom from side effects that has been shown to be of substantial benefit in the automatic detection of parallelism. For an overview of the Crystal language constructs and programming examples, see [6], [5]. No explicit passing of messages is needed in the program specification. When the data dependencies are known at compile time, task decomposition is done automatically by the Crystal compiler. The compiler generates as many logical processes as possible and then combines clusters of logical processes to produce a problem decomposition that possesses a degree of granularity that is appropriate for the target machine [14]. A high-level pictorial description of the aspects of the system thus far discussed is given by Figure 4.

Crystal is being designed and implemented as a language that provides several useful features for massive data-level parallelism. Because many problems in scientific computing are comprised of functions that operate on large amounts of data, it is natural to link our runtime system with Crystal. We utilize a C based interface for the control of the runtime system, this provides us a very clean interface to Crystal and leaves the possibility open of linking the runtime system to other programming environments. With little additional work, it will be possible to generate performance data from applications already developed in C as well as from Crystal programs. Using these data, we hope to design performance models of parallel programs/machines. It is important to remember though that our goals are achieved with a well-defined, but functionally limited interface. The rules we impose on the interface specification are ones deemed expedient for our larger goal of building a working runtime system.

Figure 5 provides an example of the triangular solve written in the intermediate C form. The user is made to specify the scope of a computational DAG with the syntax `begin dag_block` and `end dag_block`. This structure enables the runtime system to

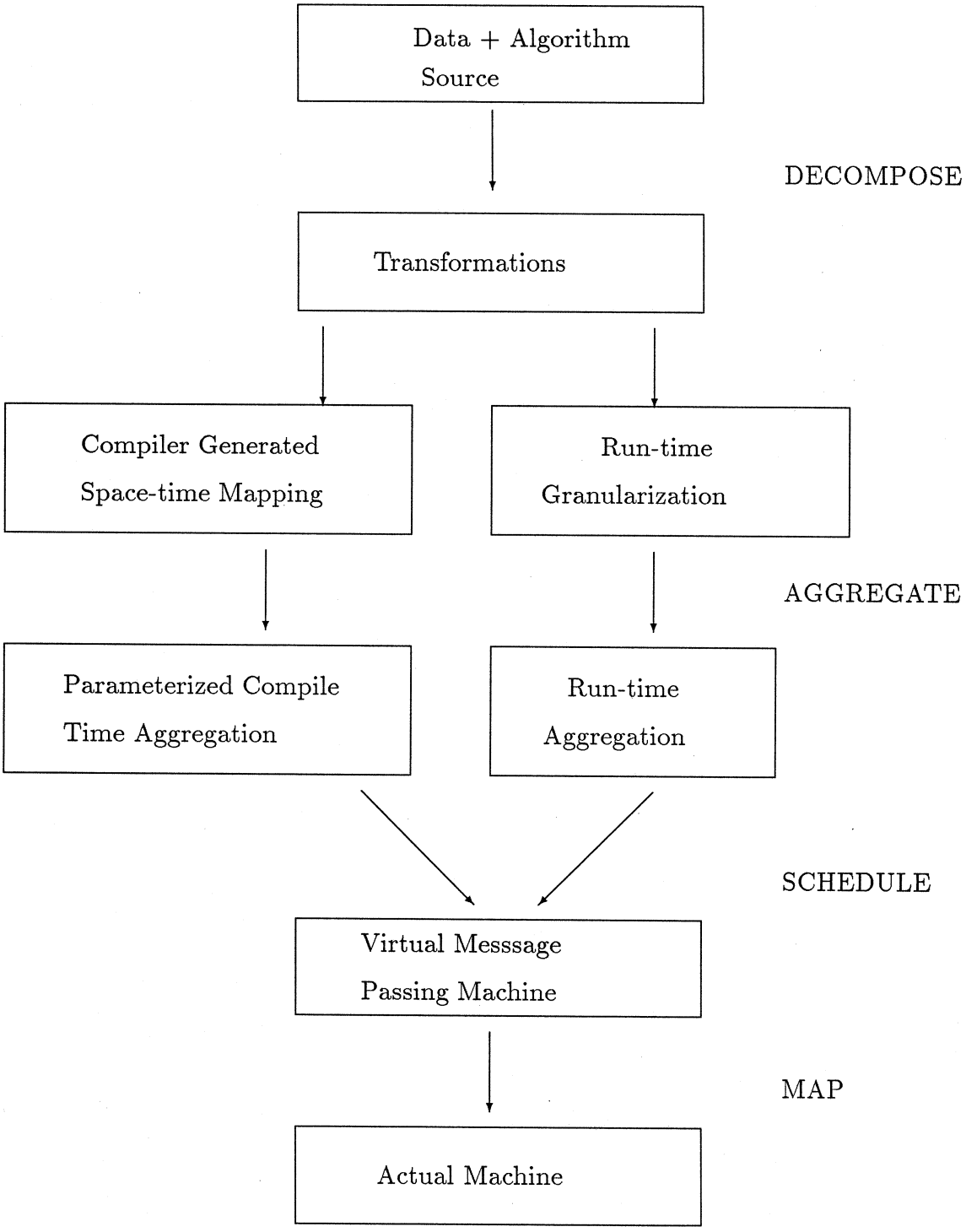


Figure 4. Data and Control Flow in Crystal Runtime System.

extract the hitherto unexploited parallelism from the code block and perform various optimizations on the resulting DAG, for the purposes of mapping and scheduling. The DAG block contains *node functions* whose individual invocations will be represented by DAG nodes, and by C code that organizes these invocations. Within this block, additional syntactic rules are in effect. Some variables may not be changed during the execution of the computation represented by the DAG. Such variables are called *fixed*; this concept gives rise to data types such as *fixed int*, *fixed float*, etc. All fixed variables are declared following the `begin dag_block` statement. We also declare *exchange* variables here. An *exchange* variable is used to exchange data between DAG nodes. A DAG node representing one function's invocation may write into an exchange variable (and may read a value that it has written itself), while one representing a different function may read from it. Another critically important type of data are integers of type *index*. Every node function invocation is to be called with a unique set of index values—the invocation is identified by the function's name and the index values. Other variables local to the dag_block are also declared here.

Node function definitions follow the variable declarations. The node functions follow the usual C syntax, with the additional requirement the function's index values be explicitly passed as parameters, and that pointers to any exchange variables used are also passed. Throughout the node function, any index variable is "marked" with an #. This serves to force the programmer to be additionally aware of the restrictions we place on the use of index variables. An exchange variable is identified in a function's preamble as being an *input* variable if the function reads it, or a *value* variable if the function writes it. An exchange variable may be multi-dimensional, and it is possible that only some proper subset of its dimensions are indexed by the node function's indices. For example, a two-dimensional array of floating point numbers `e_variable` that is an input variable, and is indexed only in the second dimension by index i, is declared as `input float e_variable[][#i]`. A node function may call other functions, provided that they are not node functions. All variables assigned values within a node function must either be variables local to the function, or value variables. Index variables may not be altered by a node function.

The main body of a dag_block consists of C statements which manipulate local variables, index variables, and which call node functions. The main body does not represent computational work to be done by the system. Instead, it uses C syntax to express algorithms for computing the indices of node function calls. The main body may not change the value of a fixed or exchange variable.

This ideas are made more concrete by studying figure 5. The code is written such that the solution of each row is partitioned into several sub-reductions, each one calculated by an invocation of the node function `reduce`. The main body computes the variable `num`, which which determines the number of data elements involved in a single reduction. `num` is determined by `r_size`, a fixed variable that controls granularity. All of the work performed by a given `reduce` call is serialized. `reduce` called with indices `i` and `j`

calculates the `j`th sub reduction corresponding to a row `i`, and sets the exchange variable `inter[#i][#j]` to this value. Then, for each `i`, the function `solve_row` sums up certain values of `inter[#i][]` and subtracts these from the right hand side of the equations. Note that the subset of `inter[#i][]` values that are summed are not explicitly stated in the node function preamble.

The system is being designed to symbolically transform programs specified in our expanded C syntax to: (1) a program that encodes the directed acyclic graph corresponding to the data dependencies between the variables designated and (2) a program that executes the code on a multiprocessor according to the schedule obtained through the analysis of the encoded DAG. The DAG encoding is accomplished in two stages. At compile-time, the dag_block code is analyzed to create for every node function a routine which computes the indices used to write value variables, as a function of the node function call parameters. Secondly, we create a routine that computes all node function call parameters, once the values of the `fixed` variables are given. At runtime, the `fixed` variables become established, and the index analysis code is run. The index information created by the index analysis code allows us to identify all DAG nodes, and the data dependencies (through exchange variables) between them.

The DAG analysis, as stated above, involves both parallelization and aggregation; the programs for performing the DAG analysis operate on the DAG data structures that describe the encoded DAG. In the expanded version of the runtime system, users will not need to specify the granularity, unless they so desire. Based upon the machine characteristics and computation structure, the automated system will decide the basic grain of computation and adapt this parameter during execution, if necessary.

The encoded DAG represents data dependencies that will depend on values of variables defined at runtime. In C, values of variables can be redefined during the execution of the program. Certain variables, the *fixed* variables, must remain unaltered in order for the encoded DAG to remain valid. If any of these critical data structures are modified, it will be the responsibility of the user to inform the runtime system about these changes. The status of the "flag" passed as an argument to the dag_block informs the runtime system about the current condition of the relevant data structures. Changes in structure which imply a different computation DAG necessitate the reapplication of the index analysis code, and the DAG manipulation routines. Crystal being a single-assignment language, the identification of fixed variables when the runtime system is linked to Crystal will consequently not be a problem.

4 Aggregation and Mapping of the Computations

In the previous chapters, we have discussed the two most important features of our runtime system:

```
begin dag_block(flag)
          fixed int numrows, r_size; fixed struct st_matrix *matrix;
          exchange float soln[MAXSIZE], inter[MAXSIZE]; index i,j;

/*  Main Body */

for(i=0;i<numrow;i++)
{       num = matrix[i].ncol/max_cols;
        /* Calculate and sum num elements of the row inner product */

        for(j=0;j<num;j++) {
            reduce(i,j,maxcol,matrix,inter,soln,numrow);
        }
        /* sum the partially computed row inner products */

         solve_row(i,num,rhs,soln,inter,numrow);
}

reduce(i,j,maxcol,matrix,inter,soln,numrow)
         index i,j; struct st_matrix matrix[#i];
         int maxcol,numrow; value float inter[#i][#j]; input float
soln[#i];
{
int l;
        for(l=j*(maxcol+1);((l<matrix[#i].ncol)||(l<j*(maxcol+1)));l++)
          inter[#i][#j] += matrix[#i].value[l]*soln[matrix[#i].col[l]];
}

solve_row(i,num,rhs,soln,inter,numrow)
         index i; int num,numrow; float rhs[#i];
         value float soln[#i]; input float inter[#i][];
{
int k;
       soln[#i] = rhs[#i];
       for(k=0;k<num;k++)  soln[#i] -= inter[#i][k];
}


end dag_block
```

Figure 5. Triangular Solve in Intermediate C Form

- the detection of parallelism at runtime that a compiler is not able to detect because of certain data dependencies that are manifest at runtime, and
- the aggregation and mapping of basic units of work such that the system is able to take advantage of the computation structure as well as the machine architecture.

The performance of a large class of problems on multiprocessor machines is largely determined by the runtime mapping of the problem onto the target machine. This mapping is specified by the assignment of indices into clusters representing schedulable units of work. In order to make reasonable decisions concerning this assignment, we must be able to take into account the data dependencies that occur within and between systems of recurrence relations.

In making assignment decisions, a variety of objectives must be taken into account:

(1) mapping and scheduling the problem in such a way that the load on the multiprocessor is balanced during each portion of the computation,

(2) partitioning the work so that the ratios of communication and or synchronization to computation are reasonably low,

(3) partitioning the work so that good performance can be obtained from any fast cache, local memory or vector processing capabilities that a target machine may have.

A variety of methods are under investigation to perform this scheduling and aggregation. All of these methods require a representation of the data dependency relations manifested by the problem. It is important to note that the data dependency relations need only be considered when they impact on scheduling and aggregation decisions. When we specify that a given computation be scheduled as an indivisible unit on a single processor, any data dependency relations that pertain only to relationships between elements within that unit can be ignored. For example, in the triangular solve function in Figure 1, we perform a reduction over the index j for each value of the index i. If we specify that all work involved in solving for a single value of the index i is to be assigned to a single processor, we need only take into account data dependencies *between* the indices i.

Before work may be scheduled onto processors, the computation written in the intermediate C form is encoded into a DAG format. A topological sort is then performed on the DAG to determine the potential parallelism. DAG manipulation techniques may then be utilized to aggregate clusters of work. Using sparse matrix computations as a model of a difficult problem, we have tested some strategies to aggregate and map these type of computations. One principal method used to aggregate involves two steps. In the first step, a sort of coordinate system is obtained for the DAG. This system is essentially a process of peeling off layers of the DAG. It is then straightforward to map the problem to a multiprocessor in a manner that restricts the fan-in and fan-out of data between processors. To the extent allowed by the data dependencies in the algorithm, mapping work so that only nearby processors have to communicate also becomes

possible. Furthermore, the coordinate system is used to allow the specification of work clusters in a parametric manner. In the applications discussed here, the clustering of work is controlled by two parameters, the *block size* which describes the number of consecutive DAG layers assigned to a processor and the *window size*, or number of wavefronts per block. The reduction in communication overhead is however, achieved at the risk of load imbalance, making this the critical tradeoff. The reader is referred to [21] for further details regarding this issue.

We seek to utilize a standard representation of the directed acyclic graph that describes the inter-index data dependencies. For dependencies involving a single index, we use the following set of data structures to represent the dependency DAG. The first data structure is a tuple of tuples *edges* that represents the edges that originate from each DAG node. The second is a data structure *incoming* that denotes the number of edges pointing to each DAG vertex.

In many cases, the best strategy for parallelizing and aggregating index sets will depend on data produced at runtime. For instance, there may be at least three different strategies for parallelizing a triangular solve. The best strategy depends on the sparsity structure of the matrix, the number of times a set of recursion relations with the same data dependencies will be solved, the number of processors available and the communication capabilities and structure of the machine. In the presence of uncertainty about the best mechanism for decomposing programs and mapping them to processors, several different strategies can be symbolically generated at compile time. The significant scheduling overhead does not occur until the dependency DAG is actually manipulated, hence the decision on the decomposition strategy to be used can be deferred until run time.

4.1 Self-Scheduled Computations

Thus far, we have dealt with preschedulable computations, i.e., those that can be completely scheduled once the input matrices are provided to the system. However, there are classes of computation which prohibit any prescheduling. An example computation of this type is the symbolic factorization whose high level code is shown in Figure 3. In general, it appears hard to classify functions as definitely requiring self-scheduling but we utilize the following heuristic: Recall that functions whose evaluations depended upon entities that were made available at runtime were candidates for runtime analysis. Once the sparse matrices were available, the computation could be completely prescheduled. However, in many computations, the input data structures are not invariant during execution and changes in these structures can influence the flow of the computation.

In general, some of these functions have the property that their outputs depend upon other functions which may only be computed sometime during the execution of the program and consequently the indices over which the function is to be computed is

unknown. By filtering out self-scheduled functions on this basis, we may be incorrectly classifying certain functions as non-preschedulable, when in fact, after some amount of analysis we may find them to be be preschedulable. In any event, after one complete execution of a function in a self-scheduled manner, the resulting DAG generated by this execution can be then used to subsequently preschedule the computation.

Given a system of recursion equations that involve a particular set of variables, we will now describe how the self-scheduling process will proceed. First, *evaluation requests* are placed for all function evaluations that we know must be calculated. These evaluation requests will be represented by DAGs which may be constructed and evaluated as described in [1]. In cases where the meaning may be ambiguous, these DAGs will be called function evaluation DAGs, to distinguish then from the DAGs used in calculating prescheduled partitions. Function evaluation DAGs represent the data dependencies involved in computing the value for a particular function evaluation. These DAGs do not represent any flow of control: an evaluation request (characterized by a function evaluation DAG) is only placed when conditionals are resolved and a definite expression is obtained describing how a function evaluation is to be computed.

Each function evaluation DAG D is assigned to a single processor P, and is evaluated until a reference is made to a function evaluation `e(...)` whose value has not been computed. At this computation of D is suspended, and we arrange to make D ready for reactivation on processor P when the function evaluation `e(...)` needed by D becomes available. Data pertaining to each function evalution D will be stored and retrieved using hashing functions known to the system as a whole. This data will include the function evaluation DAG, the value of the function evaluation when this is known, and pointers to other function evaluations awaiting the calculation of the value of D. Work with the Linda system [2] has demonstrated that in many cases, such database like operations can be performed in a multiprocessor setting without incurring unacceptable overheads.

The assignment of work to processors depends on the partitioning of responsibility for function evaluations. Due to the lack of global knowledge in self-scheduled computations, the assignment of function evaluations will generally be performed using simple heuristics for job placement. Much of the literature on this subject can be found in [16]. Migration of function evaluations between processors could also be carried out, but is likely to be infeasable in many circumstances due to the overhead incurred by this migration. Unlike prescheduled computations, work performed by the self-scheduling system takes into account only local information about the workload of the problem. Consequently, the self-scheduled system cannot be expected to provide the optimizations produced by the prescheduling system.

5 Experimental Data

We will present the results of a number of experiments that underscore the need for and the feasibility of a runtime system such as the one outlined here. The runtime system is designed for two basic functions; (1) detect parallelism in cases where substantial parallelism is present but where this parallelism cannot be identified by the compiler and (2) aggregate computational work so that the costs of inter-processor interaction are minimized. We will use measurements that are obtained from measuring the performance of preconditioned Krylov space iterative solvers implemented on the Encore Multimax and the Intel iPSC multiprocessors. In the following, *speedup* is defined as the time to run an optimized sequential algorithm divided by the time required to run the parallel algorithm. The term *efficiency* is defined as the speedup divided by the number of processors.

5.1 Results from a Shared Memory Machine

A sequential version of PCGPAK [18] [22] was ported to the Encore Multimax/120 machine and code was benchmarked on eight test problems to identify the tasks taking most of the time. Parallel implementations of these tasks were developed and benchmarked. The following four operations are performed repetitively each iteration: (1) the SAXPY operation, adding a scalar times a vector to a vector, (2) the calculation of the inner product of a vector, (3) sparse matrix vector multiples and (4) sparse triagular solves. The relative contribution of each task to the sequential running time of the program was measured; these relative contributions were used to to estimate the overall parallel efficiency of the iterative portion of the PCG algorithm.

As expected, the first three operations proved to be quite parallelizable, with efficiencies measured in all cases of well above 0.90. Since the matrices used in the preconditioning are obtained from incomplete factorizations, the number of non-zeros *within* each row tends to be very limited when sparse systems of equations are solved using these methods. Exploitation of inter-row parallelism is consequently essential for obtaining significant speedups in sparse triangular solves.

The PCGPAK code stores all matrices using sparse matrix storage methods, inter-row data dependencies arising from the triangular solve consequently depends on the values of the elements of the sparse matrix. Since the matrix elements are not defined until runtime, methods such as those described above must be used in order to schedule the concurrent execution of rows. We depict data from the solution of two problems described in [18], the matrix describing the system of equations to be solved for the first model problem has 3969 unknowns with 34474 non-zero elements, the matrix describing the second system of equations has 27000 unknowns and 105301 non-zero elements. These matrices are quite sparse with an average of under 9 and 4 nonzeros per row respectively.

In Figures 6 and 7, we depict for varying numbers of processors the following efficiencies: (1) that obtained from the sparse triangular solve parallelized by scheduling rows concurrently, (2) that obtained from parallelizing only the inner products of the triangular solves, (3) the estimated overall efficiency obtained from the entire iterative portion of the calculation, when the triangular solve parallelized by rows is employed, and (4) the estimated overall efficiency for the iterative portion of the calculations that would result when only the inner products of a triangular solve are parallelized. It is clear in both of these cases that runtime parallelization is crucial if we are to obtain good efficiencies in this situation.

5.2 Results from a Message Passing Machine

Computational granularity is a crucial determinant of performance in message passing multiprocessors which possess relatively high communication latencies. It become crucial to aggregate or clump work to varying degrees using the methods described in [21], This aggregation leads to a controlled tradeoff between load imbalance and communication requirements. In particular, these methods of aggregation reduce the number of communication startups required for the problem. The results presented in this subsection deal with the problem of triangular solve, and tested on the Intel iPSC.

The granularity of parallelism is parameterized using the concepts of windows and blocks. A window and block size of 1 corresponds to a partition in which all parallelism that can occur between individual row substitutions in the solve is extracted, and processor work assignments are carried out in a way that attempts to maximize concurrency. As window and block sizes increase, the size of the scheduled computational grains increases.

An illustration of the considerations involved in making the tradeoff between the costs of communication and of load imbalance will be presented using data from a triangular system generated from a zero fill incomplete factorization of a sparse matrix generated by a 120x120 five point template. The tradeoffs between load imbalance and communication costs in this model problem have been formally analyzed in some detail [20], [21].

In Table 1 we depict parallel efficiency, estimated optimal time, the estimated communication time, and the total time required to solve the problem on a 32 node Intel iPSC. The estimated optimal time indicates the computation time that would be obtained in the absence of any multiprocessing overheads including communication delays. This figure is obtained by dividing the sequential time by an operation count based estimate of the speedup. The communication time estimate is obtained by running problems in which computation is deleted but communication patterns are maintained. We note that when we employ a very fine grained parallelism (window and block size equal to one), we pay a very heavy communication penalty relative to the computation time. The completion of this non-computationally intensive problem requires 240 phases, each one

Table 1: Matrix from 120x120 mesh, 5pt. template

window block size	efficiency	total time	estimated optimal time	communication time
1	0.06	1.25	0.09	1.09
2	0.12	0.60	0.11	0.49
4	0.18	0.40	0.15	0.25
8	0.15	0.48	0.31	0.10
10	0.13	0.56	0.36	0.08

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. 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 small to intermediate sized problems. The example described here illustrates this well, we obtain a three fold improvement in efficiency through a moderate increase in granularity, i.e. speedup increases to 5.8 from 1.9. The absolute efficiency obtained for this rather small problem is still quite limited, even with the appropriate choice of granularity.

In Tables 2 and 3 we present results from somewhat larger problems solved on a 32 node Intel iPSC hypercube. A matrix A is formed using a mesh with a 5 point template. A different matrix B, with half the number of rows is generated by forming the reduced system; this matrix B is factored without fill to form the triangular system C. We depict the total parallel efficiency, the total execution time, the number of computational phases and the estimated communication time. Note that the best efficiencies we are able to obtain increase with the size of the problem. In the smaller of the two problems (Table 2, 200 by 200 mesh) the best efficiency is 31% while in the larger problem (Table 3, 300 by 300 mesh) the best efficency increases to 53%. The ability to aggregate work plays a central role in the increased efficiency from larger problems. When matrix rows are assigned to processors in an unaggregated manner, each row corresponding to an interior mesh point must communicate its value to another processor. Consequently, the communication volume does not tend to decrease as problems become larger.

The aggregation here is performed using graph techniques on sparse matrices. In the absence of a method that is able to capture the geometrical relationship between matrix rows, it is not possible to optimize the mapping of the unaggregated problem. Aggregation allows a tradeoff between communication costs and time wasted due to load imbalance. For a given size machine, as problem size grows, one can obtain the same load balance by aggregating work into increasingly large chunks. This leads to increasingly favorable ratios of computation to communication costs.

In Tables 2 and 3, we also depict the performance figures when the problems were executed without the use of gray coding. This was a conservative attempt to estimate the

Table 2: Matrix from 200x200 mesh, 5pt. template reduced system

window block size	efficiency	total time	phases	communication time
1- no grey code	0.06	4.58	398	-
1	0.12	2.34	398	1.92
2	0.21	1.35	200	1.06
4	0.31	0.90	100	0.45
6	0.24	1.21	67	0.65
8	0.18	1.54	50	0.57

Table 3: Matrix from 300x300 mesh, 5pt. template reduced system

window block size	efficiency	total time	phases	communication time
1- no grey code	0.08	7.61	598	-
1	0.16	3.99	598	3.45
2	0.31	2.07	299	1.57
4	0.53	1.21	149	0.88
6	0.44	1.44	99	0.65
8	0.33	1.95	75	0.45

effect of not taking the problem geometry into account. In this case we assume that the assignment of work to processors was such that each processor needed to communicate with only one other processor in each phase. Clearly, when one takes into account the geometry, the performance difference is substantial.

For reasons that related to the consequences of the 16 bit architecture of the current iPSC machine, the solution of large sparse irregular problems proved to be extremely difficult from a programming and debugging point of view. For this reason we were not able to run extensive tests on problems of the size that we feel justify the use of a multiprocessor. The absolute sequential time of the most computationally intensive of these solves is 20.4 seconds. Our results on smaller problems and the detailed breakdown of timings indicate clearly that the trends are in our favor; through the use of aggregation we can take advantage of the widely documented improvements in efficiency that come from increasing problem size, given a fixed number of processors.

6 Conclusions

In this paper, we have presented a framework for the automated runtime system (Crystal/ACRE) being designed at Yale. The goals of the runtime system effort are manifold:

- to exploit the parallelism in programs which only becomes apparent at runtime and consequently is not detected by compile time techniques,
- to manipulate the resulting DAGs into more efficient schedulable units,
- to automate the entire process and be able to generate performance data from a large variety of problems much more quickly than if the mapping and scheduling were specified by the programmer,
- to generate realistic performance models of parallel systems.

Figure 1 shows a detailed description of the various possible mechanisms by which a user program may be transformed into an efficient representation, given knowledge of the underlying machine and its characteristics. When the data dependencies of the problem are known at compile time, task decomposition is automatically performed by the compiler which generates a virtual systolic array representing the computation. The involvement of the runtime system in such cases is not significant. However, there are problems where workloads cannot be fully characterized during compilation due to data dependencies that become manifest only at runtime. Typically, a large body of scientific problems possess this characteristic.

One of the very first steps in the process consists of selecting the appropriate level of granularity at which the computation is to be scheduled. The appropriate granularity will be a function of the machine architecture as well as the structure of the sparse matrix. In cases where the computation can be symbolically prescheduled at compile time, a transformed program is generated by symbolic manipulation. Once the input data is available, the computation is represented by DAGs at various levels of granularity. Runtime aggregation of these DAGs is performed to block units of work together to improve performance. The resulting virtual message passing machine is then mapped onto the physical machine. From our initial results described in Section 5, it is quite clear that not only is the runtime detection of parallelism possible, it is absolutely essential if maximum performance improvement is desired for parallel programs.

In certain computations, it is not possible to generate DAG's representing the computation even after the input data is available to the system. We have seen an example of such a phenomenon in the symbolic factorization code described in Figure 3. A mechanism to handle such problems in a self-scheduled manner was also described earlier in the paper.

We have also provided some initial results that indicate the usefulness of a runtime system. We have seen that for the PCG computation on the Encore Multimax, the

efficiencies generated by runtime parallelization were significantly higher than those without runtime support. We can thus claim that an efficient, automated runtime system is desirable and that it can be built using the principles outlined in this paper.

As regards the status of the various parts of the runtime system is concerned, we have preliminary designs of the intermediate C syntax as well as the design of the DAG encoding mechanism for the prescheduled system. We are in the process of developing the system that encodes a DAG from the intermediate C program. The designs for the self scheduled systems are still being developed. As regards running software is concerned, we have a schedule executer program, a version of which runs on the Intel iPSC and another the Encore Multimax. The experimental results presented in this paper were obtained using this schedule executer.

Acknowledgements: We would like to thank Martin Schultz, Doug Baxter, Stan Eisenstat and Scientific Computing Associates for work in the parallelization of PCGPAK.

References

[1] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelerter. Linda and friends. *IEEE Computer*, August 1986.

[3] S. B. Baden. *Run-Time Partitioning of Scientific Continuum Calculations Running on Mutiprocessors.* PhD thesis, Mathematics Dept., University of California, Berkeley, June 1987.

[4] M. J. Berger and A. Jameson. Automatic adaptive grid refinement for the euler equations. *AAIA Journal*, 23:561–568, August 1985.

[5] Chen. *Can Data Parallel Machines be Made Easy to Program.* Technical Report YALEU/DCS/RR-556, Department of Computer Science, Yale University, August 1987.

[6] M. C. Chen. Very-high-level parallel programming in crystal. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

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

[8] G. C. Fox and S. W. Otto. *Concurrent Computation and the Theory of Complex Systems.* Report CALT-68-1343, Caltech, 1986.

[9] G. A. Geist and M. T. Heath. Matrix factorization on a hypercube multiprocessor. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, pages 161–180, September 1986.

[10] P. Hudak. Para-functional programming. *Computer*, Aug 1986.

[11] I. Ipsen, Y. Saad, and M.H. Schultz. Complexity of dense linear system solution on a multiprocessor ring. *Lin. Algebra Appl.*, 77:205–239, 1986.

[12] J. F. Jordan, M. S. Benten, and N. S. Arenstorf. *Force User's Manual.* Department of Electrical and Computer Engineering 80309-0425, University of Colorado, October 1986.

[13] K. Kennedy. Compilation for n-processor architectures. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, page 15, October 1985.

[14] J. Li, M.C. Chen, and M.F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report YALEU/DCS/RR-513, Department of Computer Science, Yale University, February 1987.

[15] E. Lusk, R. Overbeek, and et. al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc., 1987.

[16] R. Mirchandaney. *Adaptive Load Sharing in the Presence of Delays.* PhD thesis, ECE Dept., University of Massachusetts, August 1987.

[17] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, Dec 1986.

[18] *PCGPAK User's Guide.* 1984.

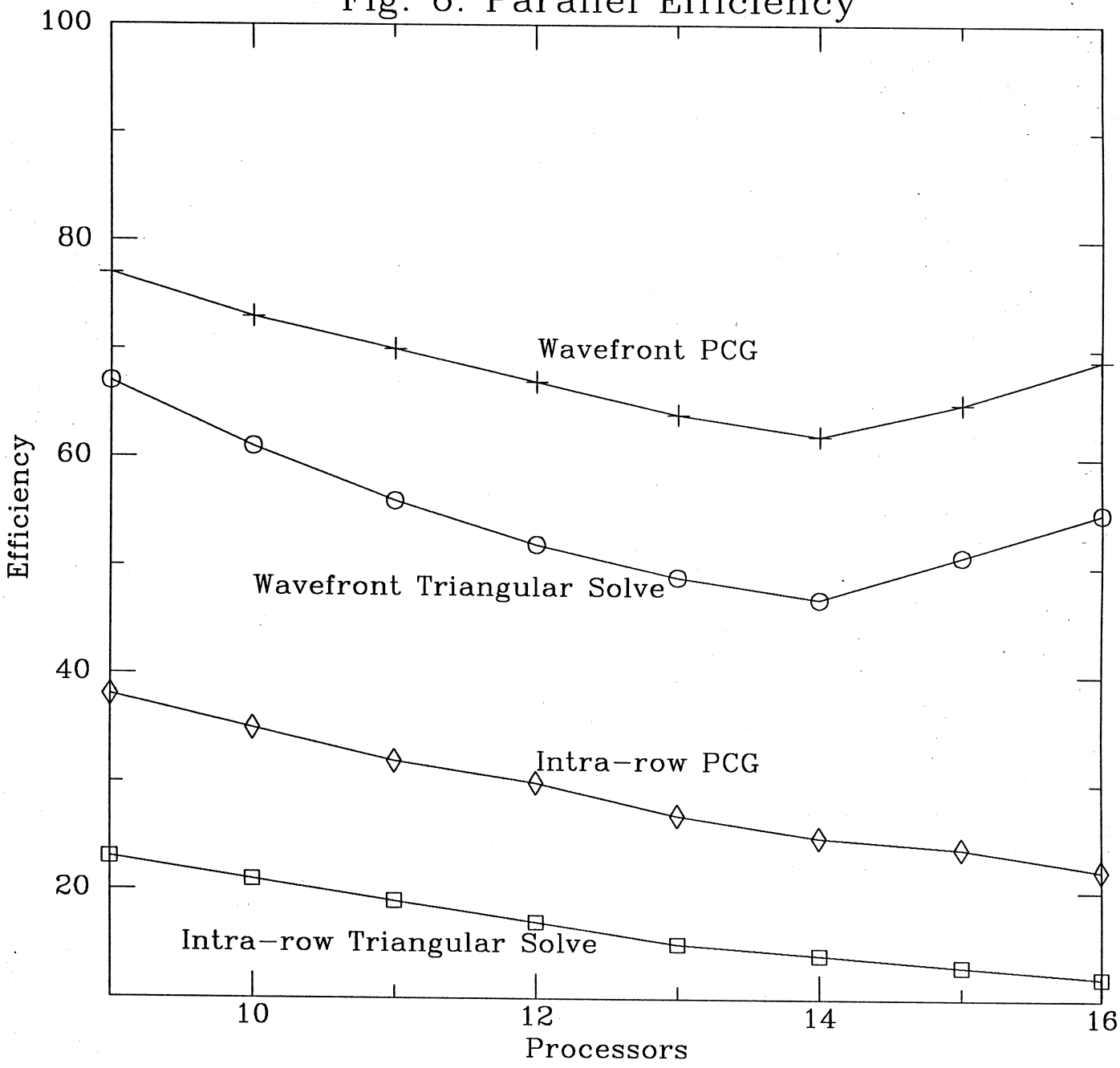
[19] Y. Saad. Communication complexity of the gaussian elimination algorithm on multiprocessors. *Lin. Algebra Appl.*, 77:315 –340, 1986.

[20] Y. Saad and M. Schultz. *Parallel Implementations of Preconditioned Conjugate Gradient Methods.* Department of Computer Science YALEU/DCS/TR-425, Yale University, October 1985.

[21] J. Saltz. *Automated Problem Scheduling and Reduction of Communication Delay Effects; submitted for publication.* Report 87-22, ICASE, May 1987.

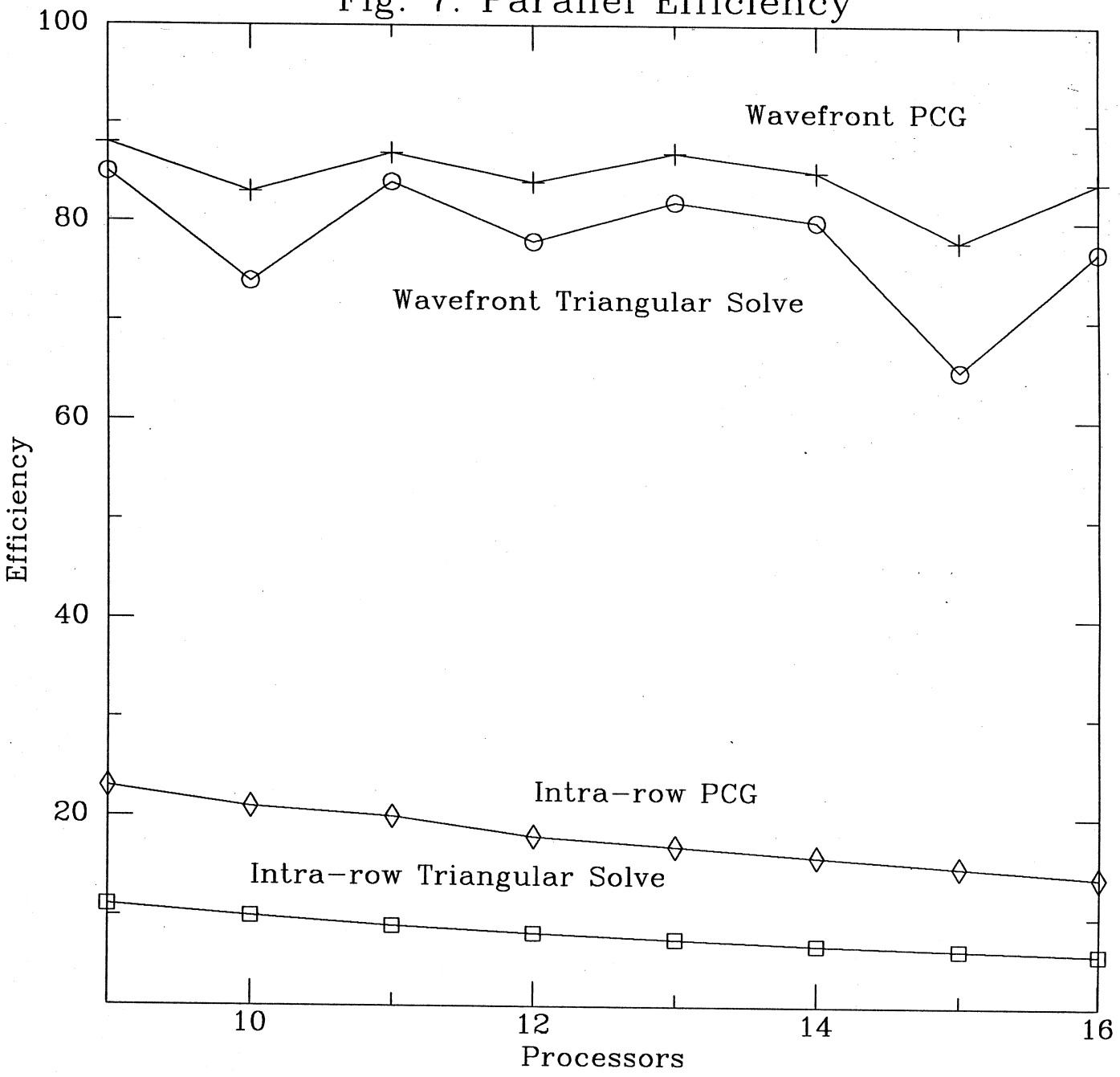
[22] M. Schultz, D. Baxter, S. Eisenstat, and J. Saltz. *Building Software Packages for Large Sparse Linear Systems of Equations on Shared Memory Multiprocessors.* Technical Report SCA-115, Scientific Computing Associates, 1987.

Fig. 6: Parallel Efficiency



Triangular System from ILU(2) Factorization
of Matrix from 127x127 grid, 9pt. Template

Fig. 7: Parallel Efficiency



Triangular System from ILU(0) Factorization
of Matrix from 30x30x30 grid, 7pt. Template