Knowledge-based program parallelization
A.-E. Al-Ayyoub$^a$, A. Yazici$^b$

$^a$Computer Science Department, University of Bahrain, Isa Town, Bahrain
$^b$Maths and Computing Department, Sultan Qaboos University, Oman

ABSTRACT

Most of the available restructuring compilers use program transformations techniques to improve and enhance parallelism in scientific programs. Different sequences of program transformations lead to programs with different performance characteristics. One of the major tasks of a parallelizing compiler is to choose an appropriate sequence of program transformations so as to effectively map a program onto a target machine. In this paper, essential requirements for intelligent parallelization and ways of meeting these requirements are discussed. A new knowledge-based parallelization model and a framework for realizing this model is also presented. This model is machine independent and can dynamically determine the sequence of program transformations depending on the program being parallelized and the target machine. The implementation of an experimental system (called InParS) based on this model and results of this experiment are also discussed.

INTRODUCTION

An efficient parallelizing compiler should recognize the parallel features of the underlying architecture, discover the parallelism in user programs, and efficiently match the program parallelism with the machine parallelism. Obviously, these tasks require a high degree of intelligence. Although a few existing parallelizers (e.g. SUPERB [6]) possess some degree of intelligence, none is capable of generating acceptable parallel schedules without extensive and experienced user guidance.

One key task of the parallelization process is to choose the program transformation sequence that generates a satisfactory parallel schedule. Different sequences of program transformations lead to programs with different performance characteristics. Some of the existing models for selecting program transformations sequences are examined below:

Fixed Transformation Sequence
This model can be used when the given class of programs can be parallelized for a given machine based on a particular transformation sequence. In this case the parallelizer designers may find a fixed sequence of transformations that is supposed to be optimal for the particular target machine. PARAFRASE-2 supports this model [9].
Alternative Transformation Sequences
This model is better than the previous one in the sense that it allows the user to compare alternative fixed transformation sequences. Furthermore, parallelizers supporting this model usually allow the user specify a user-defined sequence of transformations. The PARAFRASE System supports this model [8].

Tuning Model
In this model the programmer is supposed to provide the proper information about certain program components by annotating the programs in forms of compiler directives. Theses directives are then used by the parallelizer to make the parallelization decisions. VS Fortran Parallelizing/Vectorizing compiler supports this model. It can provide the user with run-time statistics on the execution time of DO loops. The user can use this information to decide on the appropriate compiler directives to be annotated.

Interactive Model
In this model the user is supposed to do all the work. The parallelizer provides the user with an interactive programming environment and a set of program transformations. The user can study the current state of the dependence graph and the program being parallelized and decide which transformation to apply next. Experienced programmers may utilize this kind of compiler to produce highly optimized programs. TINY is an example of experimental systems supporting this model [11].

These models can be compared by studying the effect of the sequence of program transformations used during program parallelization. For instance, the fixed sequence model will only examine a few pre-determined sequences of program transformations out of many possible sequences. The chance that the pre-determined sequence will be optimal is rather small. In the tuning and the interactive models the application order of program transformations depends on the experience of the user. The interactive model is superior to the tuning model or the alternative sequence model in the sense that programmers can base their decisions on the results of previous transformations. However, both models leave the difficult part of the job to the programmer. This problem is further tackled in the next section.

A FRAMEWORK FOR KNOWLEDGE-BASED PARALLELIZATION
To overcome the problems with the current parallelizing models the parallelizing compiler has to, somehow, achieve intelligent behaviour. An intelligent parallelizing compiler should utilize heuristics to select the program transformations to apply based on the current optimization stage and the target architecture. Intelligent parallelization is an iterative process of selecting and applying program transformations to match a program to a particular parallel architecture. At each step of the parallelization process the program is analyzed, the set of applicable transformations is compared based on some metrics, the most promising transformation is chosen and applied on the program. This process is repeated until the resulting program is satisfactory.

The complexity of the problem being attacked is better illustrated through an example. Consider the simple iteration for matrix multiplication in Figure 1. Assume that the target machine architecture supports multiple processors and each processor contains vector hardware (e.g. IBM 3090/600). The vector hardware allows the exploitation of parallelism in vectorizable computations, and multiple processors allow concurrentization of loops. Also, assume that in addition to vectorization and concurrentization the parallelizer is capable of carrying out loop interchanging and loop circulation. These transformations are supported by most of the current
parallelizing compilers. A statement-by-statement data dependence graph (DDG) is enough to verify the applicability of these transformations. Perhaps, the presence of dependence direction vectors (DDV) associated with each dependence relation would simplify this validation process.

The example on Figure 1 illustrates the approach used to find a sequence of program transformations that should be applied to produce a parallel code with satisfactory performance on a given machine. The approach is that of searching a space of intermediate codes produced by applying different program transformations. These intermediate codes are the states in the search space. The initial state (node number 1) contains the serial code and a final state (e.g. node number 10) is a parallel schedule for that code. The program transformations (interchanging, circulation, vectorization and Concurrentization) are the operators that act on the programs. Starting with serial code (the initial state), the system applies all the applicable program transformations and stores the results on separate nodes and pointers are set back to the parent node (the initial node in this case). This “expanding” process is repeated until an optimal parallel code is found or no more applicable transformations are available.

**Figure 1.** Applying transformations to sequential code
Applications of Supercomputers in Engineering

Figure 1 shows only a small portion of the tree that could be generated using four transformations. It should be clear that using a given set of transformations, the number of codes that can be generated is exponential. These codes may differ greatly in their performance. The presence of such large search space prompts the investigation of knowledge-based (heuristic) search techniques to make the search more efficient.

Intelligent Parallelizing as a Heuristic Search Problem

Intelligent parallelization can be viewed as a heuristic state-space search problem. A heuristic search system develops a plan to reach a desired goal staring from initial description of the start node. This plan is then used to guide the search in a space of state descriptions, represented either implicitly or explicitly, for a path from the initial state to the final (goal) state. Before the search for this path can begin, the representation of the state space must be set up. The problem of state-space representation is common to any problem solving approach and, unfortunately, there is no general automatic method for state-space formulation. The performance (as a function of speed, quality of the generated parallel code, and space requirements) of heuristic search system often depends critically on the representation of the states. Therefore, when selecting state-space representation, one must take into account the computation time consumed by the search activities.

State-space Representation

The state-space consists of the initial, the goal and all state descriptions that can be generated form the initial state by applying different program transformations (operators) in a “legal” manner. To complete the state-space representation of the problem one must specify the format of the initial state, the set of operators, and the properties of the goal state.

In setting up the state space formulation, any kind of data structure that bears a resemblance to the physical property of the problem can be used to describe the states of the problem. The Statement-List structure is one example that might be used to describe programs at their various states [7, 13].

Operators are computations that transform a state description into another state description. Thus, operators can be considered as functions whose domain and range are set of states. An operator, denoted by \( \gamma \), is a map which can be defined as follows:

\[
\gamma : P \rightarrow P
\]

where \( P \) denotes the set of all possible states in the state-space. Let \( \Gamma \) be the set of all operators used to generate the state-space. Some examples of operators that might be used in program parallelization are: Vectorization, Concurrentization, Loop Interchanging, Loop Circulation, Loop Skewing, Loop Reversal, Loop Rotation, Loop Sectioning, Loop Distribution, Loop Fusion, Loop Coalescing and Loop Multi-versioning [4, 13, 14].

For each operator \( \gamma \) in \( \Gamma \) there are conditions under which the operator is legal. In program parallelization these are the constraints imposed by the dependence relations between the statement or variable references within the program. Therefore, the program dependence graph represents the constraints that the transformations must respect.

Goal States

Applying operators to the old state descriptions involves generation of new states descriptions, then checking the new ones to see if they describe the goal state. Intelligent parallelizing differs from an ordinary heuristic search problem in two respects: first, no clear definition of the goal states; second, finding the optimal solution may be too expensive (due to the fairly high cost of verifying and applying program transformations) and approximate solutions may be acceptable. Thus, it is the user's responsibility to describe the characteristics of the desired results (goal states). One approach is to make guesses on the percentage of operations overlap. Suppose, for example, that the loop below were compiled for a target machine with multiple processors:
The dependence graph on the right indicates that the execution of statements \{S_1, S_3\} and \{S_2, S_4\} in successive iterations can be overlapped. Thus, a guess of 50% overlap could be a good description of a goal which could be:

\[
\begin{align*}
\text{DO } &1=2, N \\
S_1: &\quad A(I) = B(I) + C(I-2) \\
S_2: &\quad D(I) = A(I) \cdot 2 \\
S_3: &\quad C(I) = A(I-1) + C(I) \\
S_4: &\quad E(I) = D(I) + C(I-2)
\end{align*}
\]

ENDDO

Another more attractive approach for describing goal states is by using execution time estimates. This approach involves heuristics to compute the execution time estimates of the programs. Again, the user has to make a guess on the execution time of the goal node. This guess is used as a threshold for search control. Any state having execution time estimate less than or equal to the threshold can be characterized as a goal state. Note that low guesses of the execution time of the goal state will force the parallelizer to hopelessly dig deep searching for, perhaps, non-existing goal description in the space. On the other hand, with high guesses the parallelizer may stop the search soon since the probability of finding a state description that satisfy the goal is high, rather, few search steps may lead to a better results. Therefore, accurate guesses (may be obtained by using empirical knowledge) can lead the searcher to find its way to the optimal parallel schedule for a given architecture.

Graph Notation The state-space can be conceived as a directed graph. A graph consists of a set of nodes and edges or arcs (the number of nodes are not necessarily finite). Every graph has a unique node \(p\) called the start node. Certain pairs of nodes are connected by direct arcs, which represent the operators available to the parallelizer.

A sequence of nodes \(n_1, n_2, ..., n_k\), where each \(n_i\) is a successor of \(n_{i-1}\), is called a path of length \(k\) from \(n_1\) to \(n_k\). One can see that the problem of finding a sequence of operators transforming one state into another is equivalent to the problem of finding a path in a graph.

A graph may be specified either explicitly or implicitly. In an explicit specification, the nodes and the arcs are explicitly given (say) in a table. In an implicit specification a finite set of start nodes \(\{p_i\}\) and the set of successor operators \(\Gamma\) are given. The application of \(\Gamma\) to the members of \(\{p_i\}\), to their successors and so on, makes explicit graph implicitly defined by \(\Gamma\) and \(\{p_i\}\). The process of applying \(\Gamma\) to a state description is called node expansion.

The process of searching through a state-space for a solution sequence of operators corresponds to making explicit portion of an implicit graph sufficient to include a goal node. Searching graphs in this manner is therefore a central element of state-space problem solving. In the next section an informed state-space search method that might be employed in an intelligent parallelizing compiler will be discussed.
Informed Search Strategy for Intelligent Parallelization

Using the above state-space formulation, a solution (sequence of program transformations) can be obtained by applying operators to state descriptions until an expression describing a goal state is encountered. The following steps merely describe the major elements of the search process [2, 3]:

- Given an implicit graph $G=(\rho, \Gamma, \tau)$, where $\rho$ is the start node associated with the initial state description, $\Gamma$ is the set of operators, and $\tau$ is the goal node associated with the final state description.

- The successors of a node are calculated using the operators that are applicable to the state description associated with the node.

- Pointers are set up from each successor back to its parent node. These pointers indicate a path back to the start node when a node satisfying the goal description is finally found.

- The successor nodes are checked against the goal description. If none of the successors match the goal description, the process of expanding nodes continues. When a goal node is found the pointers are traced back to the start node to produce a solution path. The operators associated with the arcs of this path are then assembled into a solution path (i.e. sequence of program transformations).

A complete specification of a search process must describe the order in which the nodes are expanded. Expanding the nodes according to their generation order (FIFO) will simulate breadth-first search. If the most recently generated node is expanded first (LIFO), then the result is depth-first search. Breadth-first and depth-first search methods are called blind-search methods since the order in which nodes are expanded is unaffected by the location of the goal.

The blind-search methods are exhaustive methods for finding a path to a goal. In principle, these methods provide a solution to a path finding problem, but it is often infeasible to use these methods because the search will expand many nodes before a goal is found. One way to reduce the search is to provide a more informed $\Gamma$ that does not generate so many extraneous successors. Another method is to use heuristic information, obtained from the problem domain, to pull the search toward the goal by selecting the most promising nodes to be expanded first. Heuristics can be specified to reduce the search effort without sacrificing the guarantee of finding a minimal cost path [2, 3]. Search algorithms that use such information are called heuristic search algorithms. Below, we will discuss a heuristic search algorithm called $A^*$ which is a good representative of heuristic search algorithms that has been revised and improved many times in the literature [2, 12].

Algorithm $A^*$ The selection of transformation sequences that lead to an optimal parallel schedule may be guided by heuristic information obtained from the program, the current parallelizing stage, and the target architecture. For this purpose Algorithm $A^*$ can be used. The search process in algorithm $A^*$ is guided by an evaluation function $f$. The evaluation function $f(n_i)$ at any node $n_i$ estimates the cost of the minimal path from the start node $\rho$ to the node $n_i$ (denoted $g(n_i)$) plus the cost of a minimal path from $n_i$ to a goal node $\tau$ (denoted $h(n_i)$). Thus, $f(n_i)$ is an estimate of the cost of a minimal cost path constrained to go through $n_i$ (i.e. $f(n_i) = g(n_i) + h(n_i)$).
In program parallelization, the function $f$ does not depend only on the current program description but also on the target machine. Therefore, $g(n_i)$ can be defined as the estimated performance of the program described by $n_i$ on a given target machine, while the heuristic function $h(n_i)$ can be defined as the estimated performance improvement the program parallelization process can have on the program described by $n_i$ on a given target machine. This definition realizes intelligent parallelization at least because of the following three reasons:

1) The features of the program being parallelized and the target machine can be quantified in terms of numbers so that the parallelizer can easily compare alternative codes.

2) It is already known that algorithm $A^*$ will terminate if there is a path from $p$ to $\tau$ [2]. That is, the algorithm will terminate if there is a parallel schedule with the desired performance.

3) Algorithm $A^*$ is admissible and it will always find an optimal solution if there is any [2]. As a matter of fact, this depends of the user requirements that might be annotated with a threshold. Sometimes, the user may not be able to afford the high cost of the extensive search for an optimal solution. In this case, partial solutions may be acceptable.

The major question in applying the $A^*$ algorithm involves defining the evaluation function and goal states. There are many ways to define evaluation functions; the more accurate the evaluation function, the fewer node the algorithm has to expand. On the other hand, accurate estimation of the program performance is usually very expensive. So the proper choice of the estimation lies in the compromise between the cost of computing the evaluation functions and the cost of expanding a node (verifying and applying transformations). This point is further discussed in the next subsection.

Heuristics: The Basis for Intelligent Parallelizing

It has been mentioned that intelligent parallel compilers use heuristics and program transformation techniques to improve parallelism in user programs on a given architecture. Accurate performance prediction of a program is vital in the search process of intelligent parallelizers. Program performance prediction can usually be done through heuristics obtained from the program dependence graph and the target machine.

One heuristic for estimating the program execution time involves using the primitive operation count [10]. This heuristic function is rough but cheap to compute. The most important criterion for a heuristic to meet is that it has to be inexpensive and accurate. Because the heuristic is computed once each for each newly generated node, it has to be inexpensive. A heuristic has to be accurate because inaccurate estimates would mislead the search process. A compromise between cost and accuracy of a heuristic may be the proper choice. An accurate heuristic should consider machine dependent features such as memory utilization, synchronization cost, cache hit ratio, data transmission cost, DO-across delay, DO-across parallelism degree, number of data synchronization points, etc.

Roughly speaking, an evaluation function $f$ can be defined as a function whose domain is a set of possible states and its range is the set of positive real numbers. One possible way of defining the evaluation function $f$ is to define the function $g$ to be the estimated performance of the program based on the current state description (e.g. operation count), and the function $h$ to be the estimated potential improvement for this state (e.g. number of vectorizable or concurrentizable loops in a program, or number of breakable vectorization or concurrentization preventing dependences in a DDG of a program).
Applications of Supercomputers in Engineering

Comparison with other Models

One way to compare a heuristic-based parallelizing model and the existing models is to compare the effects of the paths that these models visit during the parallelizing process. For instance, the fixed sequence model will only visit a few fixed paths out of many possible paths in the graph. Due to the dynamic nature of the program’s behaviour, the chance that this sequence is the optimal one is rather small.

The tuning model and the interactive model relay on the user to select the transformations. Thus, the part of the implicit graph that is generated which will include the solution path depends on the user experience. Thus, the programmer is still the one who is supposed to make all the hard decisions.

Heuristic-based parallelizing differs from traditional parallelizing approaches in the following respects: (a) Parallelism enhancement: In heuristic-based model, the parallelization challenges are handled by the system using different heuristics. (b) Use of heuristics: Target machine features and program characteristics are encoded and employed in terms of heuristics. (c) Expendability: The major ability of heuristic-based parallelizing model is that new program transformations can be incorporated and their interaction with the previously existing ones can be studied. This is possible because knowledge-based parallelization model does not assume any fixed transformation sequences; rather, the optimal transformation sequence is dynamically determined.

AN EXPERIMENTAL INTELLIGENT PARALLELIZING SYSTEM

In this section we describe the implementation of an experimental Intelligent parallelization system, called InParS, designed to demonstrate the ideas proposed in the previous section. The system transforms Fortran-like DO loops into vector code well-suited for IBM 3090/180S Vector Facility. InParS assumes no pre-selected program transformation sequences; rather, the program parallelization unit analyzes features of the program and the target machine, and utilizes heuristics to choose the program transformation sequences dynamically. The goal of this experiment is not to build a production quality parallelizer for the IBM 3090/180S; rather, a tiny parallelizer that demonstrates the ideas of knowledge-based parallelization.

The basic components of InParS are illustrated in Figure 2. The system can be characterized by its sophisticated Dependence Analysis Component [7, 13, 15]. This component creates a dependency graph for a given program and updates it whenever required. The Transformation Catalog contains rules which specify the conditions under which various transformations can be applied. These rules check for the presence or the absence of appropriate dependence relations among statements to determine whether a transformation can be applied. According to the conclusions of these rules the system calls the proper routines to apply the associated transformation and to record changes on the dependency graph.

The Architecture Database enables the Performance Prediction Component to provide the heuristic functions with the required information to evaluate alternative codes. These heuristic functions guide the selection among codes generated by applying alternative sequences of program transformations. The entire process of applying appropriate program transformations and generating subsequent program descriptions is controlled by the Heuristic Search Subsystem. The following subsections describe each component in greater detail.
Dependence Analysis Component
The objective of dependence analysis is to test every relevant pair of statements in a program for every valid execution order direction vector. Thus, dependence analysis is inherently expensive. Moreover, most of the recently available tests are relatively expensive to perform. For this reason a large amount of the effort was spent for the design and implementation of the Dependence Analysis Component (DAC).

The dependence tests which are: gcd test, Exact test, Banerjee-Wolf test, I test, Lambda test, and a modified version of Lambda test [5] are all implemented and incorporated in a hierarchical manner described in [7]. The input to DAC is a Fortran-Like DO loop and the output is the data dependence graph for that loop. The produced data dependence graphs possess a high degree of knowledge about the flow of data in the program and is accurate enough to verify the application of most of the available program transformations.

The Transformation Catalog
The transformation catalog consists of rules to verify the applicability of program transformations and a set of associated procedures to carry out these transformations. The accuracy and flexibility of the dependence information provided by DAC makes it possible to include a wide range of program transformations. Furthermore, because there is no imposed pre-defined sequence of program transformations, addition of new transformations can be done without critical side effects, and the interaction of the newly added transformations and the previously existing ones can be studied.
Applications of Supercomputers in Engineering

The set of program transformations currently implemented in the experimental parallelizing system is rich enough to generate efficient vector code for IBM 3090/180S VF. More sophisticated program transformations can be implemented as well. However, the implemented set of program transformations is representative and enough to demonstrate ideas of knowledge-based parallelization.

Architecture Database: Organization of 3090's Vector Facility
The target machine may support hardware that allows parallelism on various levels. InParS supports most of the architectural features of IBM 3090/180S Vector Facility. The IBM 3090/180S VF arithmetic and logic unit uses pipeline technology. The vector facility has two pipelines, each of which consists of circuits to carry out arithmetic and logical operations. The first pipeline carries out addition, subtraction, comparisons, and logical operations. The second pipeline takes care of multiplications and divisions. Once the pipe has been initialized it can deliver a result on each machine cycle (15 ns). This will happen only when a sufficient supply of data is available at pipeline's input storage. Usually data are coming from a vector-register and general registers to supply data to the pipelines at adequate speed [1].

Performance Prediction Component
Accurate performance estimation of a program is vital in the search process of intelligent parallelization. Performance Prediction Component maps state descriptions of programs onto positive real numbers. It can be viewed as a function whose domain is a set of states and its range is the set of positive real numbers.

The performance of a program is estimated by counting the number of scalar and vector operations in a program. The philosophy behind this classification is that a scalar operation usually takes more time than a vector operation. Furthermore, the parallelizing system directs its goal toward increasing the number of vector operations in a program. Therefore, classifying operations into scalar and vector types can be a good criterion for node selection and expansion in the vectorization process.

The Performance Prediction Component uses two counters SOC and VOC to count scalar and vector floating-point operations, respectively. The actual operation count is then equal to SOC (or VOC for vector operation count) times the number of iterations for the surrounding loops. For example, the following loop

```plaintext
DO  I = 1, 10
    DO  J = 1, 19
        S1:  A(I,J) = B(I,J) - C(I,J)*PI
        S2:  D(I,J) = D(I-1,J+1)+A(I,J)
    ENDDO
ENDDO
```

has 190 scalar operations (additions) and 380 vector operations (subtractions and multiplications). Note that statement $S_2$ is non-vectorizable due to the dependence cycle on $S_2$. Thus, the execution time $T$ of a program $P$ can be estimated using the formula

$$T(P) = \alpha \cdot \text{VOC}(P) + \beta \cdot \text{SOC}(P)$$

where $\alpha$ is the average execution time of vector floating-point operations and $\beta$ is the average execution time of scalar floating-point operations.
Applications of Supercomputers in Engineering 73

Heuristic: The above defined $T(p)$ can be used by Algorithm $A^*$ as a heuristic function to evaluate alternative program descriptions. The evaluation function $f$ can be defined as follows: 

$$f(P) = g(P) + h(P),$$

where $g(P) = 0$ and $h(P) = T(P) \forall P \in G$ (G is the search graph). Algorithm $A^*$ with $f(P)$ as an evaluation function is admissible.

Proof: The operators ($\gamma_1, \gamma_2, \gamma_3$ and $\gamma_4$ defined below) do not change the number of operations in a program, rather, they may change the order in which these operations are performed. Therefore, the sum $\text{SOC}(P)+\text{VOC}(P)$ is the same for all $P$ in $G$.

$$\text{VOC}(P)+\text{SOC}(P) = \text{VOC}(P)+\text{SOC}(P) \quad \forall P, P' \in G$$

and

$$\text{VOC}(P)+\text{SOC}(P) = \text{VOC}(\tau)+\text{SOC}(\tau) \quad \forall \tau \in G, \tau \text{ is a goal}$$

One should notice that the relations $\text{VOC}(\tau) \geq \text{VOC}(P)$ and $\text{SOC}(\tau) \leq \text{SOC}(P)$ are always true because in the goal node may be some loops are vectorized. Now, knowing that $\alpha \leq \beta$, the above equality yields to

$$\text{VOC}(P) - \text{VOC}(\tau) = \text{SOC}(\tau) - \text{SOC}(P) \leq 0$$

$$\alpha (\text{VOC}(P) - \text{VOC}(\tau)) \geq \beta (\text{SOC}(\tau) - \text{SOC}(P))$$

$$\alpha \text{VOC}(P) + \beta \text{SOC}(P) \geq \alpha \text{VOC}(\tau) + \beta \text{SOC}(\tau)$$

$$h(P) \geq h(\tau) \quad \forall \tau \in G \text{ and a goal } \tau$$

Thus, $h(P)$ is always bounded by $h(\tau)$, therefore, Algorithm $A^*$ when using the above $f(P)$ as an evaluation function is admissible. \quad \blacksquare

One more issue that worths mentioning here is the non-contiguous (non-stride-1) memory references. Referencing non-contiguous words of memory could dramatically degrade the program performance. This is valid for most vector machines. Therefore, when designing a model for program performance prediction, vector processing based machines we should count for non-stride-1 memory references.

System Controller

The System Controller consists of a heuristic search subsystem (i.e. Algorithm $A^*$) and other utilities used in node evaluation and expansion processes. The heuristic search subsystem develops a plan to apply program transformations to the scalar program description until a program description with satisfactory performance is found.

The heuristic search subsystem uses a recursive data type called $STATE$ [13] to describe programs at their various stages in the search process. The system uses the following set of informed operators ($\Gamma$):

- $\gamma_1$: Vectorize inner-most loop: This operator vectorizes all loops at the lowest level. Loops that contain dependence cycles are left unchanged.

- $\gamma_2$: Distribute: This operator distributes a given loop around the $\pi$-blocks [4] in that loop.

- $\gamma_3$: Circulate to remove non-stride-1 array references: This operator orders loops so that loop index which most frequently appears in the first dimension of array references is the index of the inner-most loop. The second inner-most loop is the one at which its index most frequently appears in the second dimension, and so on.
74 Applications of Supercomputers in Engineering

\( \gamma_4 : \text{Circulate to increase vector length: This operator brings the longest vectorizable loop to the inner-most position.} \)

Given the initial state \( p \) (the scalar code), the heuristic search subsystem applies all applicable operators to \( p \). The resulting codes (after applying \( \Gamma \) to \( p \)) are stored in new nodes and then added to a list called OPEN (initially empty). The node \( p \) is then added to a list called CLOSED. The heuristic search subsystem selects a node, say \( n \), from OPEN whose execution time is minimum (as found by Performance Prediction Component and other heuristic merits), removes \( n \) from OPEN and adds it to CLOSED, and expands that node. Nodes generated by \( \gamma_1 \) are checked against the goal description and added to CLOSED if they do not satisfy goal description. Nodes generated from \( \gamma_2, \gamma_3 \) or \( \gamma_4 \) are added to OPEN. This process is repeated until a vector code with satisfactory performance is generated.

The heuristic search subsystem uses the fact that the CPU of IBM 3090/180S VF is organized in such a way that stride has a greater impact on store operations than on load operations. Therefore, the rule: "vectorize on the first index of the output variable of an assignment statement" can be a very useful heuristic that may be used to prune the search. The search process is discontinued if the performance of a final state is close (determined by a threshold) to the almost optimum performance expected. The almost optimum performance is determined by an expert and this is in general depends on the program loops.

**Vectorizing Examples and Experimental Results**

Program segments faced by parallelizing compilers can be classified into two categories: programs with few dependence relations and programs with many dependence relations. The former category of programs is considered to be hard for knowledge-based parallelizers. This is due to the high branching degree in the search graph (i.e. there are few constraints on the transformations). In InParS the search trees are relatively small even though few dependence relations are present in the program segments (usually loops). The main reason for that is the use of well-defined informed set of operators \( \{ \gamma_1, \gamma_2, \gamma_3 \text{ and } \gamma_4 \} \) which try to vectorize loops with minimum stride and maximum length. The second reason is that the IBM 3090/180S does not support multiprocessing, which in turn reduces the number of final operators and the size of the search tree. Examples to illustrate how the vectorization process in InParS is conducted are shown in Figures 3 and 4.

Figure 3 demonstrates how InParS treats statements with different strides. If either of the loops \( I \) or \( J \) were vectorized, there will be always a non-stride-1 store. A better solution can be obtained if the loops are distributed around the statements and then circulated to remove non-stride-1 stores. This is achieved in InParS by applying \( \gamma_2 \), then \( \gamma_3 \) and finally \( \gamma_1 \) (node number 5). One should notice that another alternative vector code can be generated by applying \( \gamma_1 \) to the code in node 1. However, the resulting code will contain a non-stride-1 store operation which viewed with disfavor when vector processing is concerned. An obvious heuristic function that characterizes node 5 as a goal would assume sort of penalty on non-stride-1 array references. This penalty can be expressed in terms of the average time needed to calculate the effective addresses of the non-stride-1 memory references. This heuristic is included in the performance prediction component of InParS.
Applications of Supercomputers in Engineering

\[ \begin{align*}
&\text{DO } i = 1, 128 \\
&\quad \text{DO } j = 1, 128 \\
&\quad \quad C(i,j) = A(i,j) \times B(j,i) \\
&\quad \quad D(i,j) = C(i,j) + E(i,j)
\end{align*} \]

\[ \begin{align*}
&\text{DO } i = 1, 128 \\
&\quad \text{DO } j = 1, 128 \\
&\quad \quad C(i,j) = A(j,i) \times B(i,j) \\
&\quad \quad D(j,i) = C(i,j) + E(j,i)
\end{align*} \]

Figure 3. An example on the vectorization process in InParS.

\[ \begin{align*}
&\text{DO } K = 1, N \\
&\quad \text{DO } i = K + 1, N \\
&\quad \quad A(i,K) = A(i,K)/A(K,K) \\
&\quad \quad A(i,j) = A(i,j) - A(K,j)\times A(i,K)
\end{align*} \]

\[ \begin{align*}
&\text{DO } K = 1, N \\
&\quad \text{DO } i = K + 1, N \\
&\quad \quad A(i,K) = A(i,K)/A(K,K) \\
&\quad \quad A(i,j) = A(i,j) - A(K,j)\times A(i,K)
\end{align*} \]

Figure 4. An example on the vectorization process in InParS.
Figure 5. The actual CPU time (a) and the performance (b) for codes generated by InParS and VS Fortran for the loop in Figure 4.
The loop nest in Figure 4 shows an example on loops that cannot be handled by user-guided vectorizing compiler such as VS Fortran Vectorizing Compiler without wise user generated compiler directives. However, the loop possesses some vector operations, and can be explored as shown in node 6 of Figure 4. In this example, accurate dependence computation on DAC allows InParS to recognize such parallel operations. Particularly, only the extended version of Lambda test [5] can prove the absence of dependence relations inhibiting vectorization in VS Fortran Vectorizing Compiler. InParS would produce the optimal vector code by applying $\gamma_2$, then $\gamma_3$ and finally $\gamma_1$ to the code shown in node number 1 (the initial code). Figure 5 shows the actual CPU time and the performance for codes generated by InParS and VS Fortran for the loop in Figure 4 for different N values. One can notice that for N=1600 InParS was able to obtain 25 MFLOPS at the time VS Fortran provided only 2 MFLOPS.

CONCLUSIONS

In this paper some automatic parallelizing models are compared. Difficulties with these models are considered. A new model for knowledge-based parallelization and a framework for realizing this model are also presented. The implementation of an experimental knowledge-based parallelizer, called InParS, is discussed. This system integrates a powerful dependence analysis component, a transformation catalog, AI techniques, and other utilities. Examples and experiments have shown that InParS can produce results as good as VS Fortran Vectorizing Compiler. For some cases (e.g. LU-factorization) InParS is able to produce results even much better than those of VS Fortran Vectorizing Compiler.

One key ability of InParS is that new program transformations can be incorporated and their interaction with the previously existing ones can be studied. This is possible because knowledge-based parallelization model does not assume any fixed transformation sequences; rather, the optimal transformation sequence is dynamically determined.

Knowledge-based parallelization model relays on heuristic information to decide which program transformation to apply next. Although a rough and cheap heuristic function where employed in InParS, it was accurate enough to direct the system towards the desired goal in a reasonable amount of search. Still, it is always possible to develop new heuristics or enhance existing ones without affecting the other components of the parallelizing system. Therefore, powerful heuristics can be gradually developed and enhanced.
REFERENCES

1. Book:

2. Paper in a journal

3. Paper in a Conference Proceeding:

4. Technical Report:

5. Thesis: