A Framework for Intelligent Parallel Compilers

Ko-Yang Wang

Report Number:
90-1044
A FRAMEWORK FOR INTELLIGENT PARALLEL COMPILERS

Ko-Yang Wang

CSD-TR-1044
November 1990
A FRAMEWORK FOR INTELLIGENT PARALLEL COMPILERS

Ko-Yang Wang
Computing About Physical Objects,
Department Of Computer Science, Purdue University, West Lafayette, IN 47907
E-mail: kyw@cs.purdue.edu
Phone number: (317) 494-4465
Fax number: (317) 494-0739

ABSTRACT

In this paper, we discuss obstacles for building intelligent parallel compilers and directions for improvements. A framework for the systematic program analysis and intelligent program restructuring control is presented. Methodologies for applying this framework to build optimizing parallel compilers, including heuristic-guided state-space search and planning, machine knowledge manipulation, system-knowledge organization and inference, opportunistic reasoning, and a problem solving model called hier-blackboard, are discussed.

Keywords: AI, architecture, compiler, heuristics, intelligence, parallelism, planning, problem-solving, program-transformation, optimization

September 29, 1990
A FRAMEWORK FOR INTELLIGENT PARALLEL COMPILERS

Ko-Yang Wang

Computing About Physical Objects, Department Of Computer Science, Purdue University

1. Introduction

Parallel programming is such a difficult task that intelligent advice or automatic program optimization by parallel compilers and parallel programming environments are highly desired. Unfortunately, progress in the parallel software systems lags far behind the rapid advances of parallel hardware technologies. Most parallel compilers and programming environments of today have the following problems:

- **Lack the needed intelligence to make critical decisions.** Most systems rely on the user to make critical optimization decisions.

- **Inefficient.** Most systems are slow and expensive to use. Lacking intelligence in the decision-making process causes these systems to waste resources on non-critical parts of the problems.

- **Hard to improve.** Most systems lack a systematic organization of the system knowledge and often have ad hoc heuristics scatter throughout the system. This makes it difficult for the systems to evolve. None of the existing parallel compilers or parallel programming environments have knowledge acquisition facilities or learning capabilities.

- **Limited scope of applications.** Most existing systems can only be applied to very limited target architectures. Transferring the knowledge in a parallel compiler for a particular machine to another requires significant effort.

Generally, in order to utilize the parallelism of an architecture, programmers have to make critical decisions about program partitioning, data allocation, memory utilization, synchronization, etc. They are forced to learn architecturally specific machine-language primitives and programming tricks, data and control dependencies of the programming languages, special options and limitations in the vendor supplied compilers, etc. This undermines the usefulness of the parallel compilers or programming environments.
1.1. Obstacles of Building Parallel Programming Software Tools

Why is the parallel programming software still in such a primitive state after extensive research efforts during the last decade? The major difficulties in generating powerful optimizing parallel programming environments can be categorized into the following list:

1. *Lack of comprehensive understanding of parallelism utilization:* Research in this area is still in its infancy.

2. *Detailed knowledge about the target machines required for program optimization:* Different parallel architectures use different techniques to speed up computations and require different tricks to utilize the features. Extensive knowledge about the underlying hardware is needed.

3. *Dynamism in program behavior:* The performance of a program may depend on the input data and the control flow of the program.

4. *Incomplete knowledge at compile time:* Parallel compilers have to base their decisions on approximate information at the compile time. These approximations are often incomplete, rough and rely on an unrealistic and simplified model of computation.

5. *Huge decision trees for parallelism optimization:* Even for a program of medium size, the decision tree can be quite large. Methodologies for pruning unneeded branches in the decision tree are needed.

6. *Use of ad hoc heuristics:* The techniques adopted by parallel compilers and programmers of parallel computers are mostly ad hoc heuristics. A huge amount of heuristics are needed in order for the compiler to perform an adequate job. Most systems lack proper knowledge organization facilities to utilize the heuristics effectively and systematically. Also, these ad hoc heuristics are mostly non-portable.

7. *Expensive dependence analysis and performance estimation:* Program dependence information and estimation of performance are usually very computation-intensive.

These difficulties have crippled most conventional parallel compilers and programming environments for multiprocessor parallel architectures. The programmer’s involvement is essential in the specification of parallelism at some levels. Although there are some parallel compilers that show a limited degree of intelligence (for example, Paraphrase and Paraphrase II
at CSRD [15, 23], PTOOL at Rice [2], PTRAN at IBM [1], and the Intelligent Parallel Compiler at Purdue [27, 29]), almost everyone agrees that current parallel programming tools need a much higher degree of intelligence. The question is: How far away are we from building intelligent parallel compilers? Have we effectively utilized the state-of-the-art technology in building the current generation of parallel compilers? Judging from the abundance of research results and the state of current parallel compilers, the answer is probably a "no." From our point of view, the root of the problem lies in the lack of systematic mechanisms for the reasoning and control of program parallelization and optimization process. Methodologies that can effectively integrate and utilize the current technology to improve parallel programming environments and compilers can have an immediate impact on the development of parallel software and should be important topics in the current research on parallel processing. In particular, combining advance program transformation techniques and the state-of-the-art AI technology in the construction of parallel compilers or programming environments is a very promising approach. Four important areas that have been largely ignored by the research community so far are:

1. **Frameworks for systematic program analysis and intelligent program restructuring control.** This is the central topic of this paper.

2. **The integration of machine properties into the program restructuring process and the representation and manipulation of machine features.** These problems were discussed in [28, 29] and will be briefly discussed here.

3. **Methodologies for analyzing, representing, organizing, and integrating heuristics for improving parallelism.** A framework for knowledge organization and control, called the **heuristic hierarchy**, is discussed in section 4.3, and the concept is extended to a new problem-solving model called the **hier-blackboard** (see section 4.4).

4. **Learning models and knowledge acquisition tools for the enhancement of the system knowledge.** A knowledge acquisition tool and several learning models, including the neural networks, the casual learning model, and case-learning model, were discussed at length in [30].
2. Models for Program Parallelism Improvement

Parallel compilers use program transformation techniques to restructure the control and data structures of the programs. Different sequences of program transformations lead to different speed up of the program. One of the major tasks of parallel compilers is to choose on an appropriate sequence of program transformations to use to fit a program on the target machine.

The key to achieving intelligent behavior in parallel compilers lies in appropriate paradigms for program optimization and sound methodologies for knowledge organization and integration.

Below we first examine existing models for selecting program transformation sequences and problems with these approaches. We then introduce a new model called the feature-directed program optimization model.

2.1. Six Models For Selecting Program Transformation Sequences

Most existing parallel program optimizers adopt one or more of the following models:

1. **Compiler option model.** The compiler provides command-line options for users to choose a sequence of transformations among a set of pre-defined sequences or to specify a user-determined sequence of transformations. The Paraphrase [15] from University of Illinois provides such an option.

2. **Annotation model.** The programmer directs the compiler to parallelize or vectorize certain program components (usually loops) or to decompose or distribute data in certain patterns by annotating the user program in forms of *user directives* or *assertions*. This model is supported by most parallel or vector compilers to make up the shortcoming of the compilers.

3. **Predefined sequence model.** The compiler builder finds one or more predetermined sequences of transformations that are supposed to be optimal for the particular target machine and applies the fixed sequence on all programs. If there is more than one pre-defined sequence to choose from, the user may use either command-line options or assertions to select alternative sequence to use. Paraphrase and many other parallel compilers
support this model.

4. **Interactive model.** The compiler provides the programmer with an interactive programming environment and a set of program transformation techniques for the user to direct the program restructuring process step by step and view the intermediate results of the transformations. Experienced parallel programmers may utilize this kind of compiler (usually referred to as programming environments) to produce highly optimized programs. PTOOL from Rice university is a good example of this model [2].

5. **Heuristic-driven model.** The compiler chooses the program transformations based on heuristics. The quality of the knowledge base of the compiler determines the quality of the decisions that the compiler makes. Most parallel compilers utilize some heuristics, but very few rely exclusively on heuristics; none provides systematic processing on heuristics.

6. **Program-directed model.** In this special case of the heuristic-driven model, the compiler selects transformations based on the patterns of the program dependence graph of the program. It may also utilize a knowledge base that contains heuristics for exploiting the parallelism on the target machine. An example of such a compiler was reported in [27, 29].

2.2. Analysis of the Models

One way to compare the models is to identify the paths they explore in the decision tree of the program and how well these paths are compared to the rest in the tree. The predefined sequence model will only visit one or a few predetermined paths out of many possible paths in the decision tree. Due to the dynamism of the program behavior, the chance that the predefined sequence is the optimal path is rather small. The philosophy behind this model is that this model will achieve acceptable performance without expensive analysis for a specific type of problems that the sequence is designed for.

The user annotation model and the interactive model rely on the user to select the transformations. Thus the part of the decision tree that is visited depends on the experience of the user and how hard he or she tries to optimize the program. The only help the compilers provide is that the user does not have to write the program in assembly language (instead, the
user instructs the compiler to generate the correct assembly instructions). The interactive model is superior to the user annotation model or predefined sequence model in the sense that programmers can base their decisions on the results of previous transformations and various programming tools can be incorporated into the environment to help users understand the consequences of their decisions. However, the programmer is still the one who is supposed to make all the hard decisions. The interactive model gives the user better control of the optimization and parallelization but fails to remove the major burdens of parallel programming.

The heuristics-driven model is efficient in choosing the transformations. The number of transformation paths that the model examines depend on the size and quality of the heuristics. Some heuristics are rough and may cut off useful transformation paths accidentally. Another problem lies in the difficulty of collecting heuristics. Much effort has to be put into constructing a powerful knowledge base.

Among all the models listed above, the program-directed model has the highest potential to make the program optimization decision automatically. It may consider the whole decision tree and base its decisions on the properties of the program. The down side of this approach is that much more computing resources are needed to find the optimal path.

Why do most implementors of existing compilers avoid systematic analysis models and leave the hard decisions to users? Among other considerations, we weight the following as the two most significant factors in influencing the design decisions of compiler writers.

• Difficulty in obtaining expertise for parallel programming. Heuristics that application-programmers use are usually specific to the particular application and are not directly extendable to general problems that a compiler is facing.

• Too much computing power required for deciding on a proper program restructuring sequence at compile time. The computation and analysis of the program dependence graph also require a significant amount of computing power.

We will show below that with suitable infrastructure, the difficulties in obtaining optimizing expertise can be overcome and automatic knowledge acquisition is possible. With the rapid advances in workstation technologies and the call for utilization of parallel computing resources, using fast and cheap processing power of the workstations to optimize programs for
supercomputers diminishes the problem of needing too much computing resources and makes
this approach attractive. Also, by applying appropriate AI techniques, it is possible to cut
down the decision tree to minimize the cost of optimization. The bottom line is: the desire for
improving parallelism automatically should not be put off by pragmatic problems that can be
solved by suitable methodologies.

2.3. A New Paradigm For Improving Program Parallelism

In this section, we introduce a new paradigm for improving program parallelism that
incorporates machine features into the decision-making process and provides a good foundation
for the integration of heuristics. This new paradigm, that we call the feature-directed program
optimization paradigm, can be described as follows: A parallel architecture can be character­
ized by machine features which are properties of the machine that are related to the concurrent
execution of user programs. Also, a program can be abstracted into a list of program features
such as patterns of the computation or functionalities of the computation. Heuristics are
represented into system knowledge based on machine features and program features. The con­
trol that guides the program restructuring process utilizes these heuristics to select the program
transformation to apply. A performance evaluation unit can be defined to assess the merit of
the transformations. The diagram of the paradigm is shown in figure 1.

The feature-directed program optimization paradigm has the following advantages:

1. Forcing deep understanding of the heuristics. This paradigm requires the system imple­
menter to analyze the heuristics in a systematic way before they can be integrated into the
knowledge base. The analysis and description of the heuristics for optimizing the pro­
grams are based on features of the programs and the target machine; thus they are easier
to understand and generalize.

2. Allowing better organization of the knowledge structure. Since the heuristics are
represented as a function of machine features and program features, they can be organized
by machine features, program features, or both. In any case, this allows the heuristics to
be organized in a systematic way and be accessed and modified easily.

3. Enabling systematic analysis of the program and automatic program optimization. Ex‐
licitly relying on the features of the program and target machine allows systematic analysis
Figure 1. The feature-directed program restructuring paradigm.

of the program and makes automatic program optimization possible.

4. Allowing multiple target architectures. Since features of the target machines are explicitly spelled out, this approach also allows the construction of parallel programming environments that can work with wide varieties of different target architectures.

5. Permitting transportation of heuristics to other systems. After the machine features that a heuristic relies on are distilled and encoded, the heuristic is no longer tied to the machine that it was developed upon and can be applied to other architectures that possess the set of features that it relies on. This allows the knowledge to be generalized for other parallel machines that possess similar features. Based on the feature-directed program optimization paradigm, a general algorithm for automatic program parallelism improvement can be outlined as follows:
Algorithm 4.1: Program Restructuring

Input: A sequential or parallel program dependence graph.
Output: A parallel program that matches the target machine.

Begin

while the program does not "match with the target machine well" do

1. [the program transformation unit]:
   finds a set of applicable transformations
   [the program and machine parallelism analysis units]:
   analyzes the features of the program structure and the target machine.

2. [the intelligent program restructuring unit]:
   chooses a transformation based on features of the program and the
   target machine and a set of heuristics.

3. [the program transformation unit]:
   applies the selected transformation to the program

4. [the performance evaluation unit]:
   evaluates the effects of the transformation

end while

End

As can be seen in the above algorithm, the program-restructuring process is an iterative
process of selecting and applying the program transformation techniques to match a program to
a particular parallel architecture. At each step of the process, the program is analyzed, a
transformation is chosen and carried out, and the resulting program is evaluated. This process
is repeated until the resulting program is satisfiable. To realize this process, several questions
will have to be answered.

1 The intelligent program-restructuring unit may limit the choices into a smaller subset to
   minimize the resources used in decision making.

2 An alternative to the above algorithm is to evaluate the performance of the transformation
   before it is actually applied. In this case, step 4 is moved to the front of step 3 and forms a nest-
   ed inner loop with step 2; the inner loop terminates when a favorable transformation is found.
• "What transformations to consider?" Each transformation has different effects and purposes; it is not efficient to consider all possible transformations at each step. Therefore, a subset of applicable program transformations will have to be selected for evaluation depending on the objective of the optimization. Heuristics to decide which transformations are more promising for certain tasks are needed to limit the search tree.

• "How to select the most appropriate transformation?" What does "most appropriate" transformation mean? All transformations have tradeoff and overhead. The decision for selecting a transformation will need to be based on the particular program, target machine and the current stage and objectives of the optimization. One possibility is to define an evaluation function to estimate the possible contributions the transformation can have on the concurrency. A heuristic-oriented rule-based system can also be used to make the decision. Other possibilities include pattern matching, neural network, and fuzzy logic. We will study the framework for control decisions in the next section.

• "What are the effects of the target machine on the program transformation?" Different parallel architectures have different views of parallelism. This view of parallelism of the architecture directly affects the execution of the program and must be considered when choosing the transformations. To maximize the effectiveness of the compiler, the effects of the target machine on the selection of transformations need to be studied carefully. This issue is studied at length in [29].

• "When to stop the transformation process?" For the same program, there may be many different representations that have the same input-output semantics as the original program. It is impractical to try all of the sequences before choosing the best way to restructure the program. Heuristics, and some kind of metric, must be employed in order to find the most promising transformation to apply at each step.

3. A Framework for Implementing Intelligent Parallel Compilers

The introduction of the feature-directed program optimization paradigm opens up many interesting research problems that have not been addressed before. In particular, the following problems need to be solved before an intelligent optimizing parallel compiler can be implemented.
• Machine feature manipulation. How should the machine features be abstracted, represented and organized in the knowledge base and how are they integrated into the decision-making process? What effects do architectural differences have on the selection of transformations?

• Heuristic manipulation. How do we acquire, represent, organize, and utilize the program-restructuring heuristics? What is the relationship between the program restructuring heuristics and the machine features?

• Concurrency optimization and efficient decision making. How can the system optimize programs efficiently and effectively? Can the compiler itself be parallelized and therefore able to utilize more computing power?

• Learning. Can the system learn from experience to improve its own ability? Can the framework provide hooks for the learning module?

• Intelligent user interface. When the system fails to come to a conclusion about a certain situation, can the system query the user in a intelligent manner? Can the system provide a friendly user interface?

In this section we outline a framework for an intelligent parallel compiler that is designed to provide a foundation for the above problems and serves as a basis for implementing practical optimizing parallel compilers that can handle multiple target architectures. The framework is illustrated in figure 2.

Based on the framework shown in figure 2, our approach to building intelligent parallel programming environments can be outlined as follows:

1. Feature-based target machine description. The features of the parallel machines are analyzed and the target machines are described on the basis of their features. The target machine can be either represented in a heterogeneous or hierarchical structure. An object-oriented knowledge representation scheme is described in section 4.6.

2. Feature-based heuristics representation. The program transformation heuristics are encoded on the basis of the features of the target machine and the programs. The heuristics can be hooked to the hierarchy of the machine feature that is constructed by the system. This allows heuristics to be manipulated efficiently.
3. **Knowledge organization and integration.** The inference knowledge of the program transformation is organized into a structure that is called the heuristic hierarchy. This structure closely mimics the structure of the decomposed problem space. It also features competitive and opportunistic problem-solving methodologies. The heuristic hierarchy and methodologies of organizing and integrating the knowledge are discussed in section 4.

4. **Expert systems approach.** Since heuristics are used extensively to control the decision making, expert system technologies are used to build the intelligent program restructuring system for program optimization and use it as the central control unit of the intelligent parallel programming environment. Other expert systems, including the machine
knowledge manipulation expert system, explanation expert system, and program feature abstraction expert system can be added.

5. *Intelligent program-restructuring based on feature-directed model.* The control for program-restructuring is guided by features of both the target machine and the program. A knowledge base that contains a rich set of program-restructuring heuristics can be used to aid the control of the program restructuring process. Users have the option to query the system decision-making process and take over control.

6. *Utilization of AI techniques.* AI techniques are used extensively to cut down the unnecessary branches in the decision tree and improve the efficiency of the system. More details are discussed in section 4.4.

7. *Learning.* Learning models are being studied and built for program optimization.

8. *Parallelism in the decision-making process.* Parallelizing the compiler itself allows the compiler to run on a more powerful machine and use more computing resources to solve the program optimization problem and to improve the quality of the generated code. We will discuss a program restructuring model which exhibits sufficient parallelism in the program restructuring process.

Under this paradigm, the program optimization knowledge is encoded in a machine feature-dependent but machine-independent form. And the program optimization can be viewed as being programmed by the features of the machine and the program.

4. Frameworks For the Control of Intelligent, Optimizing Parallel Compilers

In this section, we shall review how the above methodologies can be integrated into a framework for implementing the intelligent parallel program optimization system.

4.1. Parallel Program Optimization as a Planning Problem

The program optimization system can be viewed as a planning system. A planning system is a program that develops a course of actions, or a *plan*, for the involved entities to reach the desired goals. This plan is then used to guide the execution of planned activities. When the activities represented in a plan are timed, the plan is called a *schedule.*
Formally, a planning system $PS$ can be defined as a quadruple:

$$PS = (S, OP, S_0, S_G)$$

Where $S$ is the set of problem states, $OP$ is the set of operators defined by a state-transition mapping from one state to another, $S_0$ is the initial state and $S_G$ is the set of goal states. The planning process is to find a plan $\Psi$ (which is a sequence of operators) that will transfer the initial state to one of the goal states. That is:

$$S_0 \xrightarrow{\Psi} S_g \quad \text{where } S_g \in S_G.$$ 

$\Psi$ is actually a sequence of operators that change the problem states:

$$S_0 \xrightarrow{\Psi_1} S_1 \xrightarrow{\Psi_2} S_2 \xrightarrow{\Psi_3} \ldots \xrightarrow{\Psi_n} S_g$$

Where $\Psi_i$'s are operators that map $S_{i-1}$ into $S_i$, that is. $\Psi_i(S_{i-1}) = S_i$. we have

$$\Psi(S_0) = S_g = \Psi_n(\Psi_{n-1}(\Psi_{n-2}(\ldots \Psi_1(S_0) \ldots ))) \text{, or:}$$

$$\Psi = \Psi_n \cdot \Psi_{n-1} \cdot \Psi_{n-2} \cdot \ldots \cdot \Psi_1.$$  

A parallel program (represented as an augmented program dependence graph) is a rough schedule for executing concurrently the computation specified in the program on a parallel architecture. The schedule generated by the compiler statically determines the flow of the control, the flow of the data, and the utilization of resources. The parallelism optimization process modifies the schedule of the operations to improve the performance of the program on the target machine.

There are many ways to formulate a parallel compiler as a planning system. The planner can use the dependence graph as constraints for generating plausible execution plans. Alternatively, programs can be viewed as problem states and transformations as operators to refine the states. The planner's objective is thus to generate a plan for refining the execution plan, or more specifically, to generate a list of transformations to improve the parallelism of the program.

For the second approach, the program optimization process can be represented by the planning problem $PS = (S, OP, S_0, S_G)$, where $S$ is the set of program dependence graphs, $OP$ is the set of program transformation techniques, $S_0$ is the original program, and $S_G$ is the
set of optimized programs. The objective of the program optimization process is to find an appropriate sequence of transformations \( \Psi \), \( \Psi_{n-1} \), \( \Psi_{n-2} \), \ldots , \( \Psi_1 = \Psi \) to translate the program dependence graph into an optimal form for the target machine.

Once the program optimization problem is defined from the state-space transformation paradigm, the problem is then to find a solution path in a search tree whose nodes are programs and whose arcs are program transformations that modify the programs. The key issue is the selection of the most appropriate operator to apply at the given state, represented as a node in the search tree. We shall now examine how a plan \( \Psi \), the sequence of transformation, can be obtained. This approach differs from ordinary planning problem in two respects: first, there is no clear definition of the goal states; second, even though optimization is desired, the user may not be able to afford the cost of finding the optimal solution, since the cost of verifying the applicability of the transformation is usually fairly high. In this case, a partial solution maybe acceptable.

We tried two different approaches for realizing the control of the parallel program optimization problem. The control framework is the basis for the implementation of intelligent optimizing parallel compilers. The first approach uses the heuristic-guided graphic-search algorithms and the second the approach of rule-based systems. For the latter approach, a new problem-solving methodology based on the structural decomposition of the problem solution space will be presented. Its application to the program transformation problem will be discussed.

To guide the selection of the operators and identify the goal states, a performance objective function \( E \) is defined. The function \( E \) is a mapping from the state space to a real number which represents the performance measure of the states. This performance objective function is usually based on certain heuristics. Search algorithms based on the performance evaluation function are called heuristic-guided graph-search algorithms. The objective of the problem optimization process is to find the transformation sequence \( \Psi \), \( \Psi = \Psi_n \cdot \Psi_{n-1} \cdot \Psi_{n-2} \cdot \ldots \cdot \Psi_1 \), which transfers \( S_0 \) into \( S_g \) such that \( E(S_i) \geq E(S_g) \) for all \( 1 \leq i < n \). The definition of this performance objective function is subjected to the degree of optimization, the affordable resources, the knowledge of the architecture, etc. Modifying this function will change the characteristic of the program optimization system.
4.1.1. Heuristic Guided State-Space Search

The control of selecting transformations may be guided by heuristic functions or other systematic approaches such as rules in a rule-based system. For instance, algorithm $A^*$ [21] can be incorporated. The algorithm $A^*$ is a variant of the best-first search of a problem graph. It transforms the planning process into a graph search problem guided by a heuristic function $f$. The evaluation function $f(S_i)$ at any node $S_i$ estimates the cost of the minimal cost path from the start node $S_0$ to the node $S_i$ (denoted $g(S_i)$) plus the cost of a minimal cost path from node $S_i$ to a goal node $S_G$ (denoted $h(S_i)$). That is, $f(S_i)$ is an estimation of the cost of a minimal cost path constrained to go through node $S_i$.

$$f(S_i) = g(S_i) + h(S_i).$$

At each stage of the node expansion, the algorithm chooses the node that achieves the minimal evaluation function to expand. This algorithm is called Algorithm $A$.

Let $cost_{\text{min}}(S_i, S_j)$ be the actual cost of a minimal path between the two nodes $S_i$ and $S_j$. We define function $h^*(S_i)$ to be the cost of minimum cost path from node $S_i$ to any of the goal state.

$$h^*(S_i) = \min_{S_g \in S_G} \left\{ cost_{\text{min}}(S_i, S_g) \right\}$$

We also define $g^*(S_i) = cost_{\text{min}}(S_0, S_i)$, which is the cost from a start node to the node $S_i$. And $f^*(S_i) = g^*(S_i) + h^*(S_i)$ is the cost of an optimal path from $S_0$ constrained to go through node $S_i$. When the estimation function $h$ is a lower bound of $h^*$, this algorithm is called $A^*$.

Lemma 4.1. Algorithm $A^*$ will terminate if there is a path from $S_0$ to a goal state $S_g$ [21].

Lemma 4.2. Algorithm $A^*$ is admissible (that is, $A^*$ will terminate by finding an optimal solution if there is a path from $S_0$ to a goal state $S_g$) [21].

The efficiency of the $A^*$ algorithm depends on the choice of the evaluation functions. The precision of $h$ depends on the amount of heuristics it possesses. When $h = 0$, it reflects complete absence of any heuristic information about the problem and results in a breadth-first search. However, since such an estimate is a lower bound on $h^*$ the algorithm is still an admissible algorithm. Another interesting property about $A^*$ is that the more "informed" the
algorithm is, the fewer nodes it will expand. This property is described in the following Lemma.

Lemma 4.3. Let $A_1$ and $A_2$ be two versions of algorithm $A^*$ that use different evaluation functions. If $A_2$ is more informed than $A_1$ (i.e., $h_1(S_i) \leq h_2(S_i)$ for all $i$), then at the termination of their searches on any graph having a path from $S_0$ to a goal state $S_g$, every node expanded by $A_2$ is also expanded by $A_1$. It follows that $A_1$ expands at least as many nodes as does $A_2$ [21].

The major question in applying the $A^*$ algorithm involves defining the evaluation function and goal states. There are many ways to define the evaluation functions; the more accurate the estimation is, the fewer nodes the algorithm has to visit. On the other hand, accurate estimation of the program performance is usually very expensive to obtain. So the proper choice of the estimation lies in the compromise between the cost of computing the evaluation function and the cost of traversing the node (applying the transformations). More details about evaluation functions and their application in controlling the search and learning processes are discussed in [30]. Here we give a simple example. One heuristic for estimating the execution time involves using the statement counts. For a sequential statement block, this number is the sum of the number of the statements in each of the components. For loops, this is the number of iterations times the number of statements inside the loops. For conditional statements, a probability value can be assigned to each branch and the statement count for each branch is the number of statements in the branch multiply the probability that the branch would be taken. The statement count for the condition statement is then the maximum number of statements of the two branches. The statement count for a parallel loop is defined to be the maximum number of statements in each of the parallel tasks. This estimation function described above is rough but cheap to compute. The synchronization cost, memory utilization, and cache miss-ratio are all ignored in the estimation. However, it can serve as a framework for more sophisticated performance estimation. For example, for memory optimization, the unit cost for a statement can be replaced by the cost of memory accesses in the statement. The sequential statement count $T^1_{sc}$ is the total number of statements in the program and the parallel statement count $T^P_{sc}$ is the number of statements for concurrent execution on $P$ processors. The speedup based on this performance estimation function is thus defined to be $SP_{sc} = \frac{T^1_{sc}}{T^P_{sc}}$. For
algorithm \( A^* \), the cost of the arc between a pair of nodes is defined to be the changes in statement count that the transformation would have on the program. And the heuristic functions \( g \) and \( h \) are defined as \( g(S_i) = 0 \) and \( h(S_i) = SP_{sc} \).

Lemma 4.4. Algorithm \( A^* \) defined above using the estimation functions \( g \), \( h \) and \( f \) is admissible.

Proof: Since the arcs in the state space graph represent changes in statement counts, the cost of the path from \( S_0 \) to node \( S_i \) is the statement count of the program \( S_i \). Subsequently, the function \( h^*(S_i) \) is identical to the function \( h(S_i) \). And the function \( h \) is always a lower bound on \( h^* \), so by lemma 4.3 we know that the \( A^* \) algorithm will always find a program that would generate most parallel statements.

Since this heuristic ignores the impact of synchronization cost and differences in cost of different operations, the "optimal" program generated by the algorithm may not be optimal for general programs. However, this heuristic is useful for "embarrassingly parallel programs" since the lack of communication between the processes makes factors ignored by this heuristic unimportant, and the parallel statement count is thus a good indication of the concurrent performance.

4.1.2. Non-linear Planning and the Coordination of Multiple Thread State-space Search

For a finer formulation of the planning problem where each node represents the scheduling of an operation, the goal of the system is to generate an execution plan. This problem can be translated into a multiple-task scheduling process where nodes in the dependence graph are the operations to be performed and dependence arcs define the precedence relations between the operations. The output of the planning system is a \( P \)-thread parallel program with annotations for data decomposition and allocation. The control threads of the program compete for shared resources such as the memory, communication network and processors but cooperate to carry out the objectives of the program through synchronized communication. The resulting plan is partial ordered since the dependence relations need to be respected. This type of planning is commonly referred to as non-linear planning [24, 25, 31]. The complexity of the search algorithm depends on the number of nodes generated. The size of the search tree is bounded by \( b^d \), where \( b \) is the branching factor and \( d \) is the depth of the tree. Obviously, the
search tree for parallel program optimization grows very rapidly. To overcome this complexity, the problem can be decomposed into $P$ subproblems where each subproblem plans the execution for a particular processor. The algorithm $A^*$ can be applied to generate the plans for the subproblems. The coordination between these $P$ planning processes is the global strategy to avoid violating precedence constraints and resource-conflicts. The interaction between the machines is the data dependence between the blocks of statements assigned to each of the statements. The cost of an arc is defined to be the time to perform the operation that the arc points to. Communication and synchronization time need to be added to the processing time of the operation if data need to be obtained from remote processors. For this problem, the performance evaluation function $f$ is defined to be $g + h$, where $g(S_j)$ is defined to be the time it takes to perform operations from the beginning up to operation $S_j$, and $h(S_j)$ is the estimated execution time for the remaining operations. The parallel execution time is the maximum execution time of each of the processors. And the goal of the compiler is to find a schedule (a parallel program) so that the parallel execution time of the program on a $P$ processor machine is minimum.

Depending on the granularity of the schedule, the program dependence graph can be abstracted at different levels. It is generally impractical to schedule programs at a fine grain level, since this multiple task scheduling problem is an NP-complete problem and too much computing power will be required for scheduling large programs. Even at the task level, the problem is still relatively complicated. To summarize, the state-space search approach for parallel program optimization applies a heuristic function as the guideline for searching for the solutions. Heuristics are quantified into heuristic functions to help to choose the "best" node to expand. As indicated by Lemma 4.3, the quality of the algorithm depends on the quality of the heuristics used and the quality of the quantification of the knowledge. The advantage of this approach is clear: systematic processing is possible. Also, the behavior of the algorithm can be controlled by the heuristic function so the characteristic of the algorithm can be modified by changing the heuristic functions based on the objective of the system. Based on this model, new heuristics can be tested or compared to existing heuristics. On the other hand, the disadvantage of the approach is that it is not always possible to characterize the heuristic by numerical values. Distortions to the heuristics are likely in the process. Also, for complex
problems that involve many different heuristics at different stages of the problem (such as the problem for program optimization), trying to merge all heuristics into one heuristic function and applying it to every step on search steps is difficult and inefficient.

One important characteristic of the above approach lies in the specification of the problem. The separation of control knowledge (planning), data (state descriptions and goals), and problem-solving knowledge (operators) that this approach and most other AI problems have allows the system to be adaptive to different objectives of the problem at different stages of problem solving. This suggests an alternative approach which hierarchically decomposes the problem into subproblems and uses different sets of rules that are specialized for the subproblems in order to select the best node for the particular stage of the problem-solving process to expand. Below we will discuss the control of the parallel program optimization problem and discuss possible methods to decompose the problem.

4.2. Hierarchical Decomposition of the Parallelism Improving Problem

To solve the problem efficiently we can decompose the program parallelism optimization process into modules of subproblems. The process of program parallelism optimization can be classified hierarchically into three problem solving modules that we call the parallelism-defining layer, the parallelism-matching layer, and the parallelism-matching control layer. Depending on the complexity of the subproblem, each of these three layers may be further decomposed into finer subproblems.

The parallelism-defining layer abstracts the program parallelism and the machine parallelism into lists of machine and program features. The parallelism-matching layer matches the program onto the target machine by performing a sequence of program transformations. Since a single transformation may be applied to serve for different purposes, a transformation may belong to different categories. Therefore, we separate the heuristics in the program restructuring control layer into two sub-layers: the program restructuring subgoal selection layer and the transformation layer. The transformation layer contains the transformation techniques which we term transformation modules.

Each transformation module consists not only of the description of the transformation technique, the conditions for the transformation to be applicable and the procedures to carry
Figure 3 A hierarchical decomposition of the process of optimizing program parallelism. Out the transformation, but also the heuristics about the feasibility of the transformation under various circumstances, short-cut rules in applying the transformation, methods of estimating the effects of the transformation, etc.

The program parallelism improving process can be decomposed into the following five subproblems:

- **Improving general program parallelism.** The major purpose of this process is to improve the structure of the problem to prepare for other processes below. This goal can be achieved by cutting down on the amount of data or control dependence presented in the program dependence graph. Machine independent transformations for removing redundant code, breaking dependence cycles, and improving localities can be applied.
• Creating tasks. Decomposition of the control structure of the program to create tasks and vector operations. One major consideration involves balancing the loads.

• Scheduling tasks. The scheduling of the tasks/processes is another important factor in obtaining optimal performance. Traditionally, this problem is viewed as the task of the operating system. However, studies [6] have shown that static estimates done at compile time can simplify the task of the operating system at run time.

• Minimizing synchronization. When a sequential program is mapped to a multiprocessor machine, the proper synchronization operations must be inserted in the code to preserve the meaning of the original program. Synchronization costs penalize the program performance, and, in the worst case, may serialize the whole computation. Fewer synchronization points mean less processor idling time and better system performance. Grouping closely related micro-tasks into one task, copying repeatedly used data into local memories, and changing data access patterns may have a positive effect on minimizing the synchronization cost.

• utilizing memory. Since the data access time for different components of the memory hierarchy may be different, the utilization of fast memory components (like cache) and the removal of unnecessary data accesses will shorten the access time and speed up the computation. Array decomposition, data copying, scalar gathering, stride mining, loop interchanging, loop blocking, and other transformations can be used to achieve a lower cache miss ratio and improve locality.

Each of these five subproblems may select any of the transformations in the underlying transformation layer. The selection of the transformations is based on the heuristics in the transformation layer and the features defined in the parallelism-defining layer. Since these five problems are interrelated, the restructuring control process coordinates the interaction between them.

The parallelism-matching control layer is the topmost layer of the hierarchy and it represents the process that controls the overall optimization of the program. It uses the focus selection process to decompose the program into tasks which we call program focuses. It then matches them with the machine model individually, and finally adjusts the results based on global considerations.
Global coordination between different focuses is often needed. For example, the memory access optimization subgoal will try to optimize the memory accesses and decompose the array storages based on the program focus and the machine model to which it is assigned. The array decompositions chosen in the subgoal may be changed when global consideration and adjustments are made.

The performance evaluation process evaluates the performance of transformations on the program focus and provides quantified evaluation for the parallelism matching control layer to make decisions.

This hierarchical decomposition of the program parallelism optimization problem divides the problem into interacting processes based on the hierarchical structure we described above. It models the conceptual interactions between different functions of the program restructuring process into a concrete structure so that controls in these functional units can corporate and interact with each other. This hierarchy decomposition allows specialized heuristics to solve the problem so it can significantly improve the flexibility and efficiency of the transformation process. It also provides a model for the decomposition and organization of the heuristics.

4.3. Heuristic-Guided Reasoning and the Expert Systems Approach

For the rule-based approach, at each stage of the program optimization process, the control of the process utilizes a set of rules to decide how to restructure the program. A prototype implementation that used the rule-based expert system approach was reported in [27]. In this experiment, control heuristics were encoded into Prolog predicates which chose the "most appropriate" program transformation to apply. The experiment with the expert system obtained a mixed result, while the system was able to generate efficient code for some particular programs, it failed in many other cases. The major drawback of the system was its lack of heuristics. It employed only about three dozen rules for choosing the transformations. On the other hand, this experiment exposed a common problem about the flat-structured first generation rule-based systems - the fragmentation of the knowledge and lack of systematic knowledge acquisition tools. This makes enhancing the ability of the system a very involved process. We concluded that structured organization of the knowledge is desired and systematic integration of the heuristic is a key issue to automatic learning of the system. The heuristic
hierarchy reported in [29] was our first attempt in moving to more intelligent expert systems.

4.3.1. The Heuristic Hierarchy

While the modularity and integratability of the rule-based expert systems make modifying the knowledge base easy, its opacity of knowledge and inefficiency in execution are the major drawbacks. For example, translating a heuristic into a set of rules causes the knowledge to be fragmented. Even though there are still strong relations between many of the rules, the fragmentation causes an unfortunate loss of coherence. Furthermore, this makes maintenance and modification of the knowledge base difficult.

To improve the integration and modularity of the knowledge, we organize the heuristics by the decomposition of the problem-solving methods. This organizes the rules into a hierarchical knowledge structure called the heuristic hierarchy [29]. A heuristic hierarchy consists of one or more hierarchical layers. Based on the functionalities of the rules, rules in the same layer are divided into groups of rules that are called actions. Each action has a goal associated with it; invoking the action is an attempt to accomplish the goal of the action. The top layer of a hierarchy contains only one action, which is the entrance point of the control flow, and the goal of this action is the goal of the hierarchy. The heuristic hierarchy is a way to simplify the modeling of the problem into structured units. Layers in the hierarchy represent the conceptual hierarchical levels of the problem-solving process where in each layer the different actions represent possible solution steps that can be utilized to achieve the goals of the subproblem that the layer faces. The heuristic hierarchy integrates rules into conceptually and logically related units whose relationship reflects the control flow of the problem solution. Horizontal relations among the actions represent the parallelism or independence that can be exploited in a layer by employing multiple actions at the same time and vertical relations represent the inherited sequential control flows among adjacent layers. The hierarchical structure-organization of the heuristics is simple, modular, efficient, and flexible.

Note that the purpose of introducing the hierarchical structure is not to impose a tightly coupled structure into the knowledge base, because not all knowledge can be represented in structured or procedural form. Also, if the structure of the rules is too tight, then the flexibility of the rule-based system may be lost. The purpose of the hierarchical structure is to provide a
knowledge organization structure that matches the hierarchical structures in top down problem-solving processes. The hierarchical structure preserves all the advantages of a rule-based system but has better efficiency, modularity, and flexibility in the way it represents knowledge.

An example which applies this technique to the hierarchical decomposition of the parallelism optimization process was presented in [29]. The implementation of the control in each layer determines the efficiency and effectiveness of the subsystem. One can apply forward reasoning, backward reasoning, or opportunistic reasoning to achieve the best result. By merging the flexibility in opportunistic reasoning of a blackboard architecture and the well-structured control in the heuristic hierarchy, we derived a new problem-solving model called hier-blackboards. A hier-blackboard is a hierarchical problem-solving model that utilizes the inference power of opportunistic reasoning but follows the control flows inherited from the sub-problem decomposition. This achieves a very flexible model that is well-suited to solving complex problems such as optimizing program parallelism.

4.4. Opportunistic Reasoning and Blackboard Architectures

In this section, we will discuss a new problem-solving model called hier-blackboard. The hier-blackboard extends the power of the heuristic hierarchy by employing a heuristic hierarchy for structured organization and dynamic control, opportunistic reasoning for inference, and blackboards for information sharing and communication. This results in a flexible and powerful problem-solving model that is well-suited to complex and ill-conditioned problems such as program parallelization and optimization.

4.4.1. The Blackboard Architecture

The blackboard architecture combines the blackboard and the opportunistic reasoning model. The opportunistic reasoning model is a problem-solving model in which pieces of knowledge are applied either forward or backward at the most opportune time [11, 12, 20]; whereas the blackboard is a centralized knowledge representation method in which solution states and information are kept in a shared blackboard.
In a blackboard system, the solution space is divided into one or more application-dependent hierarchical levels and is stored in the blackboard. Information at each level in the hierarchy represents partial solutions currently known to the level. The problem task domain is divided into loosely coupled subtasks which correspond to areas of specialization within the task. Accordingly, the knowledge for computing intermediate results and performing subtasks is organized into modules called knowledge sources. The knowledge sources are logically independent and specialized entities, and they communicate with each other only through the uses of the blackboard. During the problem-solving process, the knowledge sources post information or intermediate results onto the blackboard to update the state of the solution incrementally and they act according to the information in the blackboard. If more than one knowledge source is willing to make contributions, the conflict is resolved by a unit called control. The control uses control strategies to choose the most appropriate knowledge source(s) to update the solution state in the blackboard. Opportunistic reasoning is applied within the overall organization of the solution space and task-specific knowledge; that is, which module of knowledge to apply is determined dynamically, one step at a time, resulting in the incremental generation of partial solutions. This problem-solving process is repeated until an acceptable solution is found or the process cannot continue for lack of knowledge or information.

4.4.2. Blackboard Systems and Production systems

Several differences distinguish blackboard systems from the production systems. First, the knowledge is organized into independent or semi-independent models in the blackboard system, while in production systems all knowledge is represented as production rules. Second, the control decision is distributed into the knowledge sources, whereas in production systems the control is sequential.

4.4.3. Advantages of the Blackboard Model

The blackboard model has been a favorite choice for solving ill-conditioned or complex problems because of its following properties: modularity in knowledge organization, flexibility in opportunistic reasoning and parallel potential in implementation.
Studies have shown the effectiveness of the opportunistic reasoning in complex and ill-structured problem domains [7, 20]. An ill-structured problem is characterized by poorly defined goals and an absence of a predetermined decision path from the initial state to the goal state. In our case, the problem of optimizing program parallelism falls into this category. The blackboard approach requires no \textit{a priori} determined path; the decision of what to apply next is made during the problem-solving process at run time.

4.4.4. Weakness of the Blackboard Model

The blackboard model has the following weaknesses:

1. \textit{Centralized and global data is a bottleneck for parallel implementation.} All modifications to the blackboard are visible and monitored by all knowledge sources; this can be a nightmare for parallel implementation. This problem can be solved by designating private blackboard sections to knowledge sources. But this solution is vague in structural organization.

2. \textit{Lacking general guidelines for implementation.} The division and organization of the solution states, solution knowledge, and solution tasks make a great deal of difference in the efficiency, clarity and effectiveness of the implementation. General guidelines in this regard are needed, but such criteria are difficult to come up with because of the diversity in different problem domains. Also, the hierarchical structure of the domain knowledge is blurred by the flat structure of the knowledge sources and control.

3. \textit{Expensive to build.} The blackboard model should be used only when its advantages justify the cost of building it.

4.4.5. The Hier-Blackboard Model

In this section we introduce a hierarchical multi-blackboard model that we call \textit{hier-blackboard}. The key idea here is to generalize the concept of knowledge sources to map the structure of knowledge sources to the structure of the solution method and localize the interaction between knowledge sources. A commonly used problem-solving method for complicated problems is the \textit{divide-and-conquer} approach by which the problem is divided into subproblems that can be solved directly or be further divided into subproblems until the subproblems
can be solved. This achieves a hierarchical level of problem partitioning which is a natural structure for the organization of the problem-solving knowledge. When the task of the system is divided into analytic levels of knowledge sources that implement the subtasks, the knowledge sources can be implemented as a blackboard subsystem if the subtasks they are responsible for are complicated. By applying this methodology recursively to the derived blackboard sub-systems, we derived a model that is logically and structurally tailored to the particular problem at hand.

4.4.5.1. The Framework of the Hier-Blackboard Model

A hier-blackboard system consists of the following four types of components: knowledge sources, blackboards, controls and communications.

Knowledge Sources

Knowledge in a hier-blackboard system is organized into hierarchically structured knowledge sources according to the problem domain. In our model, a knowledge source is a conceptual unit that interacts with other knowledge sources which share the same blackboard through explicit information updates on the blackboard. In other words, a knowledge source can be a set of rules, a procedure, or a blackboard subsystem if the subtask warrants the creation of such a subsystem.

Blackboards

There is a master blackboard which is the blackboard at the top level of the hierarchy that holds the global solution state of the problem. Optional blackboards can be added in the hierarchical tree structure. The blackboards in the sub-blackboard systems are used to hold private entities needed and produced by the local knowledge sources. Each blackboard subsystem works on its private blackboard until global information updates or accesses are needed; in that case, they simply act like other knowledge sources and compete to update the blackboard at the higher level.
Control

For each blackboard in the model, there is a blackboard controller which can be a set of knowledge sources or a separate module that monitors changes on the blackboard and decides what knowledge sources in the system should be executed in case of conflicts. The idea of having a control blackboard [12] can also be incorporated into this model.

Under the supervision of the control, the problem-solving process proceeds through a series of solution cycles. During each cycle, specialized knowledge sources check the blackboard; they self-nominate (if possible) by reporting their possible contributions to a special section of the blackboard that is called the registration-board. The control checks the registration board and picks the most appropriate knowledge sources to perform their actions. User-supplied control strategies can be consulted by control to select the "most appropriate" knowledge sources among those who self-nominated. This process continues until the problem is solved or terminates in failure.

As an analogy, the control structure of a hierarchical blackboard system is very similar to the corporate hierarchy of a company. The divisions are connected by tree-structured command channels. Each subdivision can make decisions on its own, but inter-division matters need to be solved by going through their supervisors.

Communication

As the structure of the system is defined, the communication channels between the knowledge sources and the blackboards are defined implicitly. Interaction between the knowledge sources in the same blackboard subsystem is done through the updates of the local blackboard. Inter-blackboard communication is carried out through communication channels that are specified by the system organization. If a knowledge source is implemented as a blackboard sub-system, the communication control in the sub-system will coordinate its knowledge sources and update the shared blackboard at the upper level under the request of its knowledge sources. The communication in the blackboard hierarchy is done by messengers who run up or down one level in the hierarchy to deliver messages between the levels. A messenger is a special knowledge source who monitors a designated area called a message box in the blackboard for communication. When certain data in the message box is updated,
certain actions are triggered and the messenger delivers the message to the target blackboard. For example, if one blackboard subsystem decides to modify a piece of the global problem state, it gives the message to its messenger for its parent, and the messenger delivers the message to the upper lever. The messenger in the upper level updates the information in its blackboard. The update of the information triggers an action which is to update the blackboard one more level up. In this way, the message will be propagated to the master blackboard. Note that the original knowledge source which initializes the chain of update actions may not even know that the information has been sent up the hierarchical ladder since it is only responsible for contributing to its own blackboard. The messenger mechanism helps to make the blackboard sub-systems modular and clean. The blackboard update operations can be implemented efficiently based on the target architecture to minimize the communication overhead.

Primitives for communication between blackboards include access and update of entities of the parent or children blackboards. For update operations, a condition may be sent along with the request. The condition will be checked on the target blackboard with information in the target blackboard. This last operation is very powerful since the knowledge source does not need to know the state of the parent blackboard system to update information. This conceptual similarity is accomplished by the messengers who act as representatives for the subsystems to their parent blackboards. This object-oriented design shields internal operations inside the subsystems and allows them to appear to their parent blackboards as regular knowledge sources. The messenger model we provide here is simple, but powerful, and efficient operations can be defined through this framework. For example, an entity in the message board can be set up so that whenever data is written to it, the data will be immediately sent up to the message board of the parent. In this way, data can be pipelined up the blackboard hierarchy.

Issues for Parallel Implementation

Most early research in exploring parallelism of the blackboard was based on models of hardware architectures. Parallel implementation of blackboards consists either of distributed systems such as TRICERO [32] and the distributed vehicle monitoring test-bed [16], or concurrent blackboards such as CAGE and POLIGON [19, 20]. The distinction between the structures of the underlying computational model and the solution model allows the implementation
of the solution to be constructed in a more clean and structured fashion. The multiple level structure of the hier-blackboard model can be mapped onto an actual hardware by a dynamic or static scheduling procedure. For an unlimited processor model, the mapping is simple, since a processor can be assigned to each basic knowledge source. When this unlimited processor model is mapped to the actual machine that has a finite number of processors, the static mapping will have to be based on the estimated costs, the structure of the hier-blackboard, and the locality of the data. Due to the nature of the uncertainty, the dynamic task-allocation scheme may have an edge on static task-allocation.

The actual instructions to update the blackboards can be tuned to the underlying hardware, but this is shielded from users with the blackboard update operations.

Simulation of Parallel hier-blackboard on Sequential Machines

When several knowledge sources need to share a processor, a special knowledge source called a scheduler to control the execution of a set of knowledge sources on a processor is provided. The scheduler monitors the regions of the blackboard (not necessarily in the same processor) that its knowledge sources are interested in and activates them when appropriate. In the extreme case, all knowledge sources of a blackboard subsystem share a processor, and the scheduler enables a sequential simulation of the parallel model.

4.4.5.2. Comparison With Other Blackboard Models

The hier-blackboard is a generalized blackboard model. It inherits all the benefits of the blackboard problem-solving model but provides the following unique advantages over the traditional flat-structured blackboard model:

1. Better framework for organization of problem-solving knowledge. The hier-blackboard model provides a framework for better structuring of problem-solving knowledge by matching the structure of the knowledge sources with the decomposition of the problem space and the solution methods.

2. Flexibility in opportunistic reasoning. Opportunistic reasoning can be applied only at the spots that need the power and flexibility of opportunistic reasoning. Simpler subproblems
can be solved by using more straightforward approaches such as rule-based systems.

3. **Locality and efficiency.** Localized communication improves both the locality and the efficiency of the system.

4. **Higher potential to be parallelized.** The hier-blackboard model is designed with concurrency in mind. Higher locality means higher potential for parallel implementation. Built-in knowledge sources such as the scheduler, control, and the messenger simplify the implementation of the parallel problem-solving model.

5. **Flexibility in parallelism, message-passing, and structure.** The structure of the hier-blackboard model is very flexible. The hier-blackboard model can be applied at different degree of parallelism ranging from sequential to highly parallel. Its message-passing method can range from the centralized blackboard model to the distributed message-passing provided by the messenger mechanism. Depending on the problem domain, the structure of the knowledge sources can be flat or a complicated multiple level hierarchy. The flexibility in the structure and the built-in specialized knowledge sources make the implementation much easier than on traditional blackboard systems.

4.5. **Applying the Hier-Blackboard to Parallel Compilers**

To apply the hier-blackboard model to parallel compilers, we decompose the process of optimizing program parallelism hierarchically based on the method we described in section 4.2. The most complicated subproblems in the diagram as shown in figure 3 are the parallelism matching process and the five subproblems for improving general parallelism, creating tasks, allocating processors, minimizing synchronization, and optimizing memory access. So these modules are implemented as blackboard subsystems with the parallelism matching module having five knowledge sources that are implemented as blackboard subsystems. The topmost layer, the parallelism matching control layer that controls the performance evaluation and the selection of the focuses, can be implemented as the control module for the blackboard system for parallelism matching. We encapsulate each program transformation technique into an object that contains the procedure for performing the transformation, the applicability test, heuristics of utilizing the transformation for different purposes for different kinds of programs and target architectures, and heuristic for selecting appropriate arguments to apply (method of
application). These modules form the knowledge sources for the subsystems for improving general parallelism, creating tasks, allocating processors, minimizing synchronization, and optimizing memory access. We also chose to implement the program parallelism analysis and machine parallelism analysis as rule-based systems that can be activated by program parallelism matching subsystem and its knowledge sources.

Figure 4. The structure of a program parallelism optimizing system based on the hierarchical blackboard model.

4.6. Machine Feature Manipulation

Properties of the target machine that affect the concurrent execution of machine are called machine features. Intelligent parallel compilers that utilize the feature-directed program optimization paradigm are actually "programmed" by machine features of target architectures.
The representation scheme for machine knowledge not only needs to represent the value of the machine features but also the relationship among the features. An object-oriented machine feature representation scheme was described in [28]. The representation scheme supports inheritance, specification, qualification, and modification. Inheritance simplifies the modeling of the target architecture. Specification overwrites defaults inherited from the class and allows distinctions between different instances of the same class. Qualification associates conditions with the specification and inheritance. A fragment of the machine specification for memory hierarchy is shown in figure 5.

```plaintext
class memory_hierarchy with
    type: one_of [shared,distributed,hybrid],
    structure: list_of [global,cluster,local,cache].

class memory with
    type: one_of [shared,distributed,hybrid],
    size: integer, % 1 unit = 1000
    ratio_of_fetch_multiply_op: real,
    ratio_of_fetch_register_fetch: real,
    interleave: integer,
    prefetch: boolean.

class local_memory instance of memory subclass_of memory_hierarchy with
    type: local,
    size: (integer,4000), % specify default value
    attached_to: processor,
    connection: bus.
```

Figure 5. A sample machine specification for the feature class local_memory.

The machine knowledge representation scheme provides inference support and knowledge encapsulation. For different phases of the parallel compilers, the machine features are manipulated at different levels and the perception of the target machine may be different also. A machine knowledge manipulation system based on this machine knowledge representation scheme and an interface to a SQL relational database interpreter is described in [28].

5. Employment of AI Technologies

From our experience, there are many areas where AI techniques may help in the construction of parallel compilers. We will briefly discuss some of them here.
• Search algorithms. In section 4, we discuss the use of AI search algorithms, such as $A^*$, for searching through the decision tree in the program optimization process. When equipped with suitable performance functions, search algorithms can find optimum solutions.

• Goal reduction. Goal reduction techniques (such as forward-chaining, backward chaining, hybrid methods, etc), can be used as goal searching and processing and to cut down the search trees to improve efficiency in decision making. Furthermore, resolution and unification can be used to deduce the search goals. The theory proving system can be used to deduce and analyze heuristics and find the inconsistency in the knowledge. The hier-blackboard model discussed in section 4.5 is a framework for goal reduction.

• Constraint propagation and satisfaction. Static analysis of the program has its limits. For instance, the program dependence test may be obscured by a variable in the loop bounds, or the task decomposition may be crippled by the unknown loop bound in the outermost loop. Some of these decisions can be postponed until run time. To insure that only the minimal run time test is generated, constraint propagation can be used to propagate the critical conditions for such run time tests at the needed points.

• Planning. Using planning to select program transformation sequence or generate the schedule of parallel execution is discussed in section 4.

• Generate and test. The model consists of a generator and a tester, where generator generates a number of possible cases, the tester eliminates the inapplicable or non-promising ones. A example is to apply this technique to data decomposition, where a data decomposition generator can generate possible decompositions of arrays and an examining expert can be used to eliminate less promising compositions for limiting the selections.

• Pattern recognition. Pattern recognition can be used to recognize the program parallelism, machine features, and opportunities for improving parallelism. An example of using pattern recognition to abstract program features involves recognizing opportunities for pre-optimized algorithm substitution.

• Man-machine interface. Advances in the man-machine interface such as natural language processing and visual programming can be used to achieve intelligent user-interaction.
• Knowledge engineering. Knowledge representation and manipulation techniques can be employed to represent, organize, and integrate the program transformation heuristics and manipulate machine features.

• Learning. Learning models and knowledge acquisition techniques can be applied to enhance the power of the system greatly.

• Problem-solving models. Problem-solving models such as rule based systems, blackboard systems, and object-oriented models can serve as frameworks for reasoning in intelligent compilers.

6. Conclusions

Our paradigm for building intelligent parallel compilers and programming environments differs from traditional compiler approaches in the following respects:

• Model of parallelism optimization. Most compilers and parallel programming environments either use predefined program transformation sequences or rely on users to select the transformation sequences. In our system, we utilize the feature-directed program optimization model which opens up a completely new avenue for research into intelligent parallel compilers.

• Reliance on the amount of the knowledge incorporated. One major difference between our approach and conventional parallel compilers lies in the degree of reliance on the knowledge stored in the knowledge base to guide the optimization decision control. The size of the knowledge base not only reflects the quality of the optimization capability of the system but also the reduction in the amount of the knowledge that users have to possess. This factor may not be as crucial on other systems since they push hard decisions onto the users or use scattered heuristics with little or no knowledge manipulation facility.

• Organization and integration of the knowledge. Another major difference is that the knowledge employed is explicitly represented in our system but implicitly hidden in most conventional parallel compilers. In our system, knowledge about the target machine is encoded and employed in terms of machine features. Specification, organization, integration and utilization of the heuristics are all based explicitly on the machine features and
program features.

- **Degree of the user involvement.** The experiences and capabilities of programmers vary widely; the programming environment should provide different degrees of help based on the user's preference and requirement. Our reasoning model provides a mechanism for the user to intervene in the decision-making process by selecting the degrees of interaction and optimization; and the system can be adjusted to suit different needs of different users.

- **Extensive utilization of AI techniques.** It is surprising that AI technologies and knowledge manipulation issues have been ignored by the parallel compiler community for so long even though compiling for parallel machines has been largely based on heuristics.

- **Multiple target machines and knowledge generalization.** One key issue in deciding whether a parallel compiler can be successful lies in the ability of transporting and integrating experiences learned from a particular machine to other machines. Integrating knowledge for optimizing different kinds of parallel architectures into one system eases the problem of knowledge transferral and knowledge generalization. Accumulating the abilities of the system can greatly enhance the capability of the system as the development of the system progresses. This feature is particularly valuable for complicated software systems such as compilers since the cost of building such a system from scratch for each individual target machine is so high. With this approach, only the back-end (code generation) needs to be specialized for the target architecture. More important, users' parallel programs can be immunized from machine-dependent constructs to preserve portability without sacrificing efficiency. As a result, user programs can be closer to the algorithm specification and thus easier to debug. Another advantage of the multiple target parallel programming environment is that it provides a uniform environment for the user to work with and save a great deal of learning time for different architectures and environments.

The combination of expert systems, knowledge acquisition, and AI techniques for analysis, collection and accumulation of the system knowledge provides a practical alternative to traditional parallel compiler approaches. This paradigm can be used to build compilers that are far more powerful than even the best parallel programming environments available today.
On the other hand, with this approach the need for efficient decision-making processes and new methodologies for representing, organizing, integrating and utilizing the knowledge becomes even more important. Methodologies for applying state-of-the-art AI techniques to these problems to realize this new framework in the construction of parallel compilers is studied in [29] and briefly in this paper; further research effort is needed for constructing a practical, powerful, intelligent parallel compiler.

ACKNOWLEDGMENTS

I would like to thank Dr. E. Houstis and Dr. Rice for their continuous encouragement and support. I also would like to thank Dr. D. Gannon and Dr. P. Mehrotra for their valuable discussions and suggestions at various stages of this research.

BIBLIOGRAPHY


