Applying AI Techniques to Program Optimization For Parallel Computers

By

Ko-Yang Wang
Department of Computer Science
Purdue University
West Lafayette, Indiana

and

Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, Indiana

TECHNICAL REPORT NO. 227

Applying AI Techniques to Program Optimization For Parallel Computers

by

Ko-Yang Wang and Dennis Gannon

September, 1987

This report is based on work supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR-86-0147.

12.1 Introduction

12.1.1 The Trend Toward Parallelism

Perhaps the most important trend in supercomputer design is the reliance on parallelism to achieve performance improvements over our fastest sequential processors. During the three-year period from 1984 to 1987, the number of commercially available general purpose parallel processing systems jumped from a couple to over a dozen. The number of ways in which different architectures exploit parallelism is almost as large as the number of different companies. This is a healthy situation for computer architecture. Many good ideas are emerging. Unfortunately, each different machine presents a different architectural model to the programmer. A program that has been optimized for one system may not be well suited to another. At first glance, the differences may appear to be due to the fact that each machine supports a different set of extensions to FORTRAN, or even a different base programming language. But a deeper analysis shows that the architectural difference between machines plays a fundamental role in the organization of the computation. Surface level syntactic changes are not enough to port a program optimized for a Cray XMP to good code for a MIMD hypercube design. While this is an extreme case, it illustrates the problems faced by the small, but growing, cadre of programmers who have taken up the task of putting these machines to productive use.

Because of these problems, it has become clear that the greatest need in supercomputer development is a new generation of software tools that can help in the task of optimizing code for new architectures.

In this chapter, we describe a project under development at Purdue University and Indiana University, which is an experiment in integrating expert systems technology with the advanced compiler optimization research conducted over the last ten years by Kuck, Wolfe, and their associates in Urbana Illinois [3, 13, 14, 15, 20, 21, 23, 29], Kennedy and his students at Rice [11, 1], and Allan, Cytron, and Burke at Yorktown Heights [6, 4]. There are three key ideas that are guiding our work:

- Interactive program restructuring tools are essential in helping users move programs to new machines.
- Expert knowledge about how to choose a sequence of restructuring transformations that optimize performance can be organized as an "advice giving" system. Furthermore, performance models of the target architecture can be incorporated into a rule based system to guide the transformation process.

- New architectural models and expert programming heuristics for new target machines must be easily incorporated into such a system in a uniform manner.

Of course, interactive tools already exist. For example, FORGE from Pacific-Sierra Research provides an excellent user interface. PTOOL from Rice University [2] has an elegant way to help users identify data dependence in programs. And all automatic program restructurers, such as VAST, KAP and Paraphrase, employ powerful heuristics to retarget user code. The goal of this research is to show that an expert systems approach is a more flexible and extensible model than the conventional parallel compilers for designing a tool that can be rapidly adapted to new target machines and new heuristics for parallel program optimization.

12.1.2 Automatic Program Parallelism Optimization

The program parallelism optimization problem is the following: given a program and a target parallel machine, how can a parallel program that is both functionally equivalent to the original program and optimal for the target machine be generated?

The basic algorithm for program parallelism optimization can be outlined as the following:

```
BasicProgram
Input: a sequential or parallel program and the description of the target
machine.
Output: a parallel program that is optimal for the target machine.
Begin
    repeat
        pick the "best" transformation from the set of
                  all applicable transformations;
        apply the selected transformation to the program;
    until the resulting program is optimal for the target machine
End;
```

This algorithm is superficial in the sense that it does not specify how to determine either which transformation is the best or when the program is optimal. However, this is the algorithm that most parallel computer users use when they hand-optimize their programs. Picking the "best" transformation requires expert intelligence.

Our goal is to design an intelligent system that can perform the program parallelism optimization process for different classes of target machines automatically. Several fundamental issues must be addressed before such an intelligent system can be constructed:

Machine knowledge representation. Conventional program restructurers hide the impact of machine knowledge on the decisions made during program restructuring as a part of the process of selecting of the heuristics used in the system. - only heuristics that are effective for the target machine are included. This is possible because only one target machine is considered. However, when the program parallelization system is designed to handle different classes of architectures, the features of the parallel computers that affect program parallelism must be abstracted and represented in a uniform structure. Separating the machine features from the heuristic acquisition process allows the description of the heuristics to be based on the machine features as well the program features. In this way, a heuristic can be applied to any target machine that has the appropriate set of features.

Program representation. The program representation problem is to define internal data structures that can encode the program's semantic and parallelism constraints. A good program representation must preserve the exact semantic and parallelism constraints of the original program. The program representation scheme must also allow easy and efficient accesses and modifications.

Transformation techniques. Transformation techniques are the essential elements of program restructuring systems. Many transformation techniques have been studied during the past two decades by a number of pioneering researchers. Rather than going through the details of the mechanical techniques for modifying program structures, in this chapter we will emphasize the heuristics for applying the transformations and the effects of the transformations on program parallelism.

Restructuring heuristics. The optimal sequences of transformations needed to get good performance from a section of code is very dependent on the program and the target machine. There are no algorithms that provide the optimal sequence of transformations for all circumstances. Heuristics are usually used to perform the task and these heuristics are usually based on the particular application and make assumptions about the target machine. In order to make the heuristic general the special features of the program and the assumptions about the machine must to be made clear.

The representation and organization of transformation knowledge. The representation, organization, and integration of the transformation knowledge are the central issues for an automatic program parallelizing system. They actually

determine the effectiveness and efficiency of the system.

Parallelism metrics. Parallelism metrics are used to compare the effects of different transformations and to decide when to terminate the optimization process. Measuring the achievable parallelism of a program on a target machine must be based on the parallelism features that the machine provides and the matching between the program structure and the target machine.

The remainder of this chapter is organized into three sections. In section 12.2, we formally define the program parallelism optimization process and discuss the machine knowledge representation problem. The program representation problem and the problem of defining parallelism metrics are also briefly discussed. In section 12.3, the transformation knowledge representation problem and some program restructuring heuristics are presented. Examples that describe the work of the inference engine are also included. In section 12.4 we give a brief summary and describe the status of our project.

12.2 Abstracting Machine Features and Building the Knowledge Base

In this section, we define the program parallelism optimization process. A machine feature abstraction scheme is introduced and a function to estimate the matching between the program level parallelism and the machine level parallelism is also given.

12.2.1 Parallelism and Program Parallelization

Parallelism can be exploited at three different levels: the algorithm level, the program level, and the machine level. Each of these three levels has a conceptual concurrency model of computation and we call this model the *virtual machine* for that level.

At the algorithm level, the virtual machine is the computational model (e.g. mesh, hyper-cube, etc.) that the parallel algorithms are based upon. *Algorithm level parallelism* can be characterized as the number of virtual processors, the complexity of inter-processor communications, and the complexity class of the parallel execution time on the virtual machine model when expressed as a function of problem size.

At the program level, each parallel programming language defines a virtual machine by the semantics of its parallel control constructs. *Program level*

parallelism can be characterized by the control and data dependence constraints imposed by the language and the user's choice of data structures.

Machine level parallelism is the maximum concurrent execution capacities of the architecture and can be characterized by various machine features.

When mapping problems from the algorithm level to the program level or from the program level to the machine level, the differences in the computational models of the two levels may cause parallelism to be lost. For example, when an algorithm is translated into a program, the concurrent properties of the algorithm may be serialized by the dependence relations inherited from program constructs and data synchronization. In some cases, the concurrency is lost because the limited parallel constructs provided by the programming language simply can not express the full parallelism in the algorithm. The problems encountered in translating parallelism from the algorithm level to the program level fall into the scope of parallel programming language design will not be discussed in this chapter.

When the program is mapped from the program level to the machine level, the programs may have to be restructured, since some specific program structures or data structures may suit the target machine better than others. *Program restructuring* is the process of improving the match between the program level parallelism and the machine level parallelism by applying a sequence of program transformations to restructure the program.

12.2.2 Program Realization and Restructuring

The process of optimizing program parallelism consists of two steps: the program restructuring process and the program realization process. The program restructuring process improves the program parallelism by modifying the structure of the program representation. The program realization process maps the programs onto the computational model of the target machine by effectively utilizing the concurrency potential of the machine.

Program level parallelism can be divided into three concurrency levels: *task, micro task,* and *operation.* At the task level, a program is decomposed into large processes which may be run on different processors. At the operation level, vector operations or scalar operations are the units of computation. The size of the vector operation represents the degree of concurrency of this level. The micro task level is the level between task level and operation level and is often characterized by loop bodies. More specifically, inside a task, operations are grouped into micro tasks, which are the blocks of code that are executed between synchronization points.

Based on the dependence constraints of the program and the feature descriptions of the target machine, the program realization process partitions the program into operation blocks and composes them to form vector operations, micro tasks, tasks, and processes. Abstractly, the process can be viewed as a function:

$$\textit{Program_realization}: \textit{Computational_model} \ X \ \textit{Programs} \ \rightarrow \ \textit{Program}_A$$

where elements of $\textit{Program}_A$ are programs that are augmented with parallelism and run-time information such as processor assignments, synchronization, vectorizable or parallelizable loops, etc.

The program realization process does not actually improve the true parallelism of the program. It simply takes the current form of the computation, as represented by the program, and based on the features of the target machine, applies a mapping to realize the program into parallel form. For example, for multiprocessor systems, the outermost parallelizable loop is always used to generate tasks. For machines with vector capability, the innermost loop is the one that is vectorized (if it is legal to do so). The synchronization technique that is provided by the computational model is used to satisfy any data dependence not already satisfied by sequential execution of parts of the program.

The program restructuring process improves the match between the program level parallelism and the machine level parallelism by modifying program structure and improving the data locality in the program. In particular, it involves techniques such as changing the instruction execution order (by forward substitutions, statement reordering, etc), modifying program control (by loop interchange, loop distribution, etc), and eliminating unnecessary data accesses and modification (by data localization, block transfer, cache optimization, dead code elimination, etc). Each individual technique used to modify the structure of the program is called a *transformation.*

Abstractly, a program transformation, T, is a mapping

$$T: \textit{Program} \ \rightarrow \ \textit{Program}$$

that maps a program representation to a new program representation that has the same input-output semantics. The *precondition* of a transformation is the list of conditions that must be satisfied so that the result of the transformation will have the same meaning as the original program. If a program satisfies the precondition of a transformation, we say that the transformation is *applicable* to the program.

Program transformations are just mechanical techniques for changing the structure of the program. To have a positive effect on the performance, the transformations must be chosen based on the full knowledge of the program, the target machine, and a set of effective heuristics. The program restructuring

process is a composite function of a sequence of transformations. It uses heuristics that are based both on features of the program and the machine to guide the transformations and effectively translate the program into optimal form. Abstractly, it takes the form

Program_Restructuring: Program X Computational_model X Heuristics → Program

At the heart of the program restructuring is the set of rules in the knowledge base that represents the expertise about program constructs, transformation techniques, machine parallelism, and heuristics for improving the matching between programs and machines. These rules decide the effectiveness of the program restructuring process.

12.2.3 Problems in Program Parallelism Optimization

Corresponding to the concurrency levels of the program parallelism, the task of improving program parallelism can be subdivided into the following problems:

Partitioning problem. How does one partition a problem into tasks and micro tasks and form good vector operations? If the current structure of the program does not suit the hardware, various transformation techniques should be used to improve the program structures and to achieve a better partition.

Synchronization problem. When mapping a sequential program to a multiprocessors 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, it 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.

Scheduling problem. The scheduling of the processes is another important factor in obtaining optimal performance. Traditionally, this problem is viewed as the task of the operating system. However, studies have shown that static estimates done at compile time can simplify the task of the operating system at run time [6]. There are techniques (e.g. do-across) that can estimate the required minimum process delay time and significantly reduce the amount of time the processor in "busy-wait" loops. Run-time tests can also be generated at compile time to guide the execution of the process.

Memory utilization problem. 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.

Due to the complexity of the task, most algorithms used in solving the above problems are heuristic-driven. Some useful heuristics for program restructuring are discussed in section 12.3 and others can be found in the literature on program transformations.

12.2.4 Program Representation

The state of the program can be represented by *program dependence graphs* which consist of the *control flow subgraph* [7] and the *data dependence subgraph* [14, 29] of the program. The data dependence subgraph represents the set of essential constraints on the execution order of the operations. The control flow subgraph specifies the preconditions on the operations which are required for them to be actually executed. Together, these two subgraphs form a complete summary of the semantics of the program. The dependence relations in the program dependence graph specify the sequential order that the program parallelization process must respect. Violating the dependence relations may cause data access and modifications to happen in the wrong order which will change the meaning of the program. Program dependence graphs have been studied extensively, details of the representation and computation of the graph can be found in [7, 14, 4, 26, 29].

12.2.5 The Representation of Machine Structures.

One of the major advantages of multi-target optimization systems over dedicated single-target optimization systems is that the heuristics can be shared among all target machines that have the same properties. When a heuristic is synthesized, the influences of the target machine must be distilled to identify the properties of the target machine that actually affect the heuristic. These properties of the machine must be represented in a uniform structure so that different parallel computers can be easily characterized. The properties of the target machines that affect program parallelism optimizations are called *machine features.*

The space of all possible values of a feature is called the *feature space.* A feature space may be either a subspace of the reals or a discrete space. The cross product of all the feature spaces forms the space of all possible computational

models, which we call the *Computational_Model.* An element in the Computational_Model represents the computational model of a particular target machine. The computational model is the abstraction of the properties of the target machine that influence program parallelism optimization. It represents the program restructuring system's understanding of the target machine.

Since the intelligent program restructuring system can reason, not all of the hardware properties need to be included in the computational model. Instead, properties that can be derived from other features can be omitted from the computational model, since they can be derived by the system when they are needed. This helps keep the size of the feature list manageable.

We represent the Computational_Model as a frame "slot filler" model. This frame model of representing the computational model is called the *raw model.* Each individual feature is an slot of the raw model to be filled. The computational model of a target machine can be defined by filling the feature space attributes in the raw model with the correct values. Not all the slots have to be filled when abstracting a machine feature. A set of rules can be used to derive default values for the unfilled slots.

The computational model of the target machine can be divided into the following four categories:

1. Processor hierarchy
2. Processing units
3. Memory hierarchy
4. Networks/Busses

Each of these 4 subspaces consist of a list of features. In the following three subsections, we examine the elements of these features and discuss their attributes in the program restructuring process.

12.2.5.1 Processor Hierarchy and Processing Elements

The set of computational elements (PEs) in a parallel computer can be characterized by the following components of the feature space:

1. Number of processors.
2. Modes of computation: (SIMD, MISD, MIMD, etc.)
3. Methods of scheduling: (data driven, data-flow, demand driven, control flow)
4. CPU scalar speed.
5. CPU scalar instruction type: (stack, two address, three address, etc.)

Vector attributes --

6. Vector instructions: (diadic, triadic-vec-vec-vec, triadic-vec-vec-scalar, etc.)

7. Vector instruction speed.
8. Vector startup time.
9. Vector operands: (register, memory)
10. Vector results: (register, memory)
11. Number of vector registers.
12. Size of vector registers.
13. Chaining.
14. Cost of non-uniform stride.
15. Cost of scatter-gather.
16. Vector reductions: (max, add, inner-product, etc.)
17. Horizontally coded multiple function units.
18. Special restrictions/features: (list)

The number of processors affects the way in which a program is partitioned into tasks. For example, when partitioning a nested loop, the best way to create tasks is to first match the number of iterations of the outermost loop with the number of processors, then block the loop to form tasks. Loop interchange can be used to cause the best matching loop to be the outermost loop.

For processors with vector processing capabilities, issues such as where the operands are stored (in memory or in register), whether it has vector registers, and the size of vector registers affect the way that data is decomposed and how vector operations are formed. Vector operation start up time and relative speed of vector/scalar operations are critical in justifying whether a loop should be translated into vector operations. In addition, the use of special vector instructions (e.g., triadic vector operations, inner product reductions, vector operand gathering) can be more important than the absolute speed of the vector processors.

The processors may have a special hierarchy that the programmer must keep in mind. This processor hierarchy, usually based on processor clustering, affects task decomposition. Features in this category include:

1. Cluster size.
2. Shared resources within clusters: (memory, synchronization hardware, etc.)
3. Task switching time within a cluster.
4. Processor scheduling within a cluster: (loop oriented, data-driven, etc.)
5. Special topological constraints: (mesh, cube, etc.)
6. Cluster task granularity.
7. Cluster scheduling policy: (users or special operation system policy)

A cluster can be viewed as a collection of processors that is capable of executing a collection of very finely grained tasks in a tightly-coupled manner which is

not possible by the set of all processors. For example, the computational complex (CEs) of the Alliant FX/8 forms a cluster that is distinct from the interactive processors (IPs) system. A system may support multiple clusters with multiple processors per cluster (as in the Cedar system), or it may be viewed as one tightly-coupled cluster of processors (as in the Connection Machine) or a loosely-coupled system of one-processor clusters (as in the Cray XMP). In a machine with multiple clusters, there will often be two levels of scheduling: a "micro-task" level that manages jobs within each processor and a "process" level that assigns processes to each cluster.

12.2.5.2 Memory Hierarchy.

The memory hierarchy of a parallel computer consists of global memory, local memory, and cache memory, as well as the networks or busses that connect these components. Global memory is shared by all processors, and can be either physically centralized in one memory module (as in the Alliant FX/8) or distributed among processor units (as in the BBN Butterfly and the IBM RP3). Local memory is owned exclusively by individual processors. Processors are not allowed to access other processors' local memories directly. However, some computers have a centralized controller which can access all local memories (as in the Pringle [9, 10], or the Connection Machine). The feature space for the memory hierarchy consists of the following items:

1. Size of cache.
2. Cache sharing: (shared cache, private cache, etc)
3. Cache coherence strategy: (compiler managed, snoopy cache, etc.)
4. Cost of cache data fetch relative to register fetch.
5. Size of local memory.
6. Cache shared by cluster.
7. Cost of local memory data fetch relative to register fetch.
8. Size of global memory.
9. Interleaved or non-interleaved global memory.
10. Centralized or distributed global memory.
11. Cost of "near" global fetch relative to register fetch.
12. Cost of "far" global fetch relative to register fetch.
13. Vector prefetch mechanism: (from global to local, from global to cache, none)
14. Special synchronization memory commands: (fetch-add, locks, memory tags, etc.)

Normally, accessing data from the global memory is slower than accessing data from a local memory, which is in turn slower than accessing data from a cache. In multiprocessor systems, an excessive amount of shared data access and synchronization might cause network contention and, as a result, saturate the entire system. For example, on the BBN Butterfly, if all processors make frequent

references to the same critical section lock or data structure, a memory "hot spot" is created. If the data is not a critical section lock, then a local copy can be made. This can double performance on many algorithms.

Management of cache and local memory is also critical. If the cache miss-ratio or the locality of an algorithm is bad, then the system utilization will be low since most of the processing power will be wasted waiting for data. On the Alliant FX/8, cache is shared by all computational elements. Because the cache is twice the speed of main memory, bad cache management can cut performance in half.

Although better locality always means better memory utilization, the cost ratios of data accesses from different components of the memory hierarchy plays an important role in resolving conflicts between improving data locality and decreasing the number of instructions. We will discuss this issue in more detail in the next section.

Different machines may have different memory hierarchies. On some machines, one or more components in the memory hierarchy may be missing. For example, the connection machine has no cache, most MIMD hypercubes have only local memory; message passing strategies are the basis of all synchronization and access to shared information. Data flow machines have a completely different memory model. The Pringle has no shared memory; processor communication is done by message passing through reconfigurable processor-to-processor routing switches. Each processor in the Pringle has only eight ports, so a message routed to another processor might need to go through a couple of hops, and setting up an optimal message routing network for a given algorithm is a non-trivial task. Although some heuristics for data allocation and routing on non-shared memory machines like the Pringle do exist, the data decomposition problem for non-shared memory remains largely unsolved. More effort is needed before an optimal result can be achieved.

On the other extreme are the IBM RP3 and Cedar, which both have a complete memory hierarchy that includes cache, local memories, and global memories. On the RP3, global memories and local memories reside in the same memory modules that belong to individual processing elements. The same mechanism is used in the BBN Butterfly Uniform system. On the RP3, a sophisticated memory addressing scheme allows the boundaries between global and local memories to be adjustable. On both machines, it is more expensive for a PE to access another PE's global memory than it is for the PE to access its own. Therefore, it is very important that the locality is explored on these machines. The Butterfly provides a block transfer operations which makes localizing frequently used data attractive.

The Alliant FX/8 has no local memory, and its two 32KB caches are shared by eight processors. The shared cache is connected to the processors by an 8 x 8 crossbar switch, and is connected to memory through a high speed bus (188 MB read-access per second). Therefore, cache utilization for the Alliant is important. Examples of data utilization for the Alliant will be discussed in the next section.

12.2.5.3 Interconnection Networks and Busses.

The connections between processors, or between processors and the memory hierarchy, or between the components of the memory hierarchy may utilize either busses or complicated networks. There are a number of factors that are very important to understand:

1. Network topology: (bus, ring, cube, mesh, tree, banyan, etc.)
2. Network bandwidth.
3. Delay per network stage.
4. Packet or circuit switched.
5. Packet size.
6. Maximum pending memory references a processor can have in the network.
7. Routing type: (self-routing, compiler routing, both)
8. Performance penalty of self-routing.

Network topology plays an important role in the way data structures are distributed around the system. On networks with a low bisection width, such as a tree, certain data movements are notoriously slow. For example, a matrix transpose is extremely slow on trees and rings. A complete study of the role of topology in parallel algorithm design is found in the paper [8].

From the point of view of a program restructurer, there are two issues which are more critical. First, if the network is not self-routing, then the compiler needs to plan a path and generate switch settings for the network. Many non-shared memory machines require that each intermediate processor be programmed to intercept and forward cross network traffic as part of the target code. Second, if the network is such that some processors are "nearer" than others, and if the message delay from a far processor is significantly more than from a near processor, optimal data structure decomposition becomes critical. Not only is this problem NP-complete, there are also very few good heuristics for it. In addition, for dynamic allocation of new processes, it may cost more for a processor to start-up a new process on a remote processor than it does for it to do the computation itself. The program restructuring system has to consider all these differences in network implementation before it can actually perform task and data decompositions.

Some interconnection networks have special properties to enhance the capabilities of the system. For example, the IBM RP3 has a combining network which supports fetch-and-op kinds of operations, making the implementation of system primitives much easier; in particular, it supports the implementation of task queues and makes self-scheduling loops possible. (On the Cedar and BBN Butterfly these same operations exist, but they are done by the memory controllers rather than the network.) For machines that support self-scheduling loops, the program restructuring system can leave the task scheduling problem to the operating system of the machine by transforming the outermost loop into a self-scheduling loop. However, the self-scheduling loop makes the global array decomposition almost impossible, since it can only be known at run time which loop will be run by which processor. Our experience shows that the data decomposition is usually more important than the loop scheduling, so in programs that have decomposable arrays (i.e. arrays that can be allocated into the local memories of the processing units) data decomposition should be favored.

In multistage networks, non-uniform network traffic, known as "network hot spots", is typically (but not uniquely) produced by shared locks or data synchronization. This can generate effects that severely degrade the network traffic. Studies have shown that combining data access requests within the switches is an effective technique for dealing with a hot spot contention problem that is caused by global shared locks [22]. For machines that have no combining network, balancing the operation load is one of the major challenges to the program restructuring system.

12.2.6 Program and Machine Feature Abstraction

As we discussed above, the program parallelism abstraction process bases its decisions on the features of the program at hand and the target machine. The features of the program and target machine are abstracted into concepts that can be used by various heuristics. In the case of program representation, this feature abstraction can be done by either matching patterns or checking program dependence relations to find out whether the program region under consideration matches some predefined "concepts." For example, an inner-product operation can be recognized by matching the pattern that a statement inside a loop accumulates the product of corresponding elements of two arrays into a variable. A more complicated example is the concept of "vectorizable", a loop is vectorizable if each statement, S, in the loop can be executed for all values of the index set of the loop before executing any of the statements in the loop following S, and this alternate execution order will compute the same result. The vectorizable concept can be captured by examining the dependence relations of the loop. A procedure (or rule) that does the test inserts the fact *"the current loop is vectorizable"* into the solution space if the test is true.

As for the machine, we should note that there are usually some heuristics which accompany the features of the machine. These heuristics are the methodologies of utilizing the properties of the machine. Examples of this are: "improve locality if the machine has cache or local memories," and "generate P(= number_of_processors) tasks if task creation cost is high." It is the collection of these methodologies that really defines the computational model of the machine.

There is a fundamental difference between the abstraction of features of the program and the abstraction of features of the machine. That is, the features of the machine are static, but the features of the program are dynamic. The facts that are derived by the feature abstraction process will stay true throughout the optimization process for the machine, but the facts about the program may be changed as the structure or data distribution of the program is changed. Therefore, the feature abstraction process for the machine is done at the time the target machine is chosen but the feature abstraction process is done during the program restructuring process. Another dynamic aspect of the feature abstraction process is that only the features of the program that are currently important are abstracted. For example, it would make no sense for the restructuring system to check whether a loop is "vectorizable" when it is trying to figure out how to create tasks from a simple loop. However, if the loop is a nested loop and the machine supports both multiprocessing and vector processing, then the loops will be checked for "vectorizability" since the best way to schedule the loops is to create vector operations from the innermost loop and create most tasks out of the outermost loop.

12.2.7 The Parallelism Metric

In order to justify the merit of a particular transformation, a valuation function which evaluates both the degree of program parallelism and the matching between the program and the machine is needed. The valuation function:

$$Matching: Computational_Model \ X \ Program \rightarrow R$$

returns a simple real valued index that estimates the matching between the computational model and the current structure of the program. The matching function is a weighted linear combination of several factors. Among these are: how well the size of the program structure fits the size of the target machine (size matching), how well the data access pattern matches the data distribution on the memory hierarchy (data access matching) and how much synchronization delay is needed (scheduling matching). Each of these factors can be defined as a *match function* that maps the cross product of the spaces of the computational model and the program into a subset of the real space.

The size matching function quantifies the structure matching between the program and the target machine. For example, an outer loop that generates only two tasks on a machine with 100 processors would get a rather low score. For machines with vector instructions, the size matching function estimates the efficiency of the vector instructions.

Data access patterns are also measured. If possible, data that is repeatedly referenced should be kept in local memory or cache to reduce the network traffic. The most common example of repeatedly referenced data is the array references inside loops. The subscripts of the references, plus the loop bounds, give a good estimate of the number of array references in the loop. Non-unit stride array references are discriminated against when cache size is relatively small since these references will generate a much higher cache miss ratio than unit-stride array references.

Shared data accesses might cause memory contentions and serialize the data accesses and thus degrade the system performance. The more shared data references that a program has, the higher its synchronization cost will be. So the shared data synchronization factor can be defined to be the reciprocal of the number of shared data accesses in the program region under consideration.

Task scheduling and synchronization are also modeled by the match function. Based on a do-across schedule [6], an estimate is made of processor utilization. This estimate contributes to the final value.

Once processor assignment is completed, only the cross-task dependence may produce inter-processor synchronization. Another source of synchronization cost is the serialized access of shared variables. This kind of data synchronization can also be characterized by inter-task data dependence.

The number of inter-task dependence, IDEP, can then be used to quantify the effectiveness of the synchronization factor. The fewer of these dependence there are, the better the matching is. Let NDEP be the total number of dependence in the focused program region. The synchronization matching factor, SYNC, is defined as:

$$SYNC = (NDEP - IDEP) / NDEP.$$

A large number of other factors go into the evaluation of the Match function. A much more detailed discussion is given in [26].

The weighted-combination approach of computing the match function has the following advantages:

Dynamism. Weights of the components can be adjusted dynamically and this makes the matching function very flexible and powerful. Different architectures can have different weights to suit their particular configurations. For example, on a vector machine that has vector registers, the weight of the size matching can be increased so that longer vector operations will be generated, and bad stride vector operations will be avoided. During the program transformation process, some factors can be intentionally ignored to resolve conflicts, or to allow alternative paths to be explored.

Simplicity. Each individual matching function focuses on the matching between the program and a set of particular features of the machine, making it easier to compute.

Modularity. When new factors that affect matching are introduced, they are vary easy to be added into the matching function. One only needs to define the sub-function and give it a weight that represents its importance in matching parallelism.

Topics discussed in this section form the foundation of the program parallelism optimization process. However, what really decide the effectiveness of the program parallelism optimization systems are the heuristics which are based on this foundation and the program transformation techniques which are used to restructure the program to match the machine. In the next section, we will discuss the mechanism used to organize the heuristics that deal with program transformation theory and we will describe the operation of the inference engine.

12.3 Intelligent Program Transformations

In this section, the organization, integration, and interpretation of program transformation knowledge are discussed. An example of optimizing a matrix-vector multiply program for three different parallel machines (BBN Butterfly, Alliant FX/8, and Purdue Pringle) is given to describe the operation of the inference engine.

12.3.1 System Organization

There are three major components in the expert systems organization: the knowledge base, the inference engine, and the user interface mechanism. The knowledge base contains the domain dependent rules, facts, heuristics, and procedural knowledge. The inference engine is the mechanism used to select and apply the rules in the knowledge base to solve the problem. The user interface mechanism contains the utilities to build user friendly interfaces. These include a menu selection mechanism, graphics interface utilities, an explanation mechanism,

and help utilities. The inference engine and the user interface are domain independent, and they can be used to construct other expert systems by adding a domain dependent knowledge base.

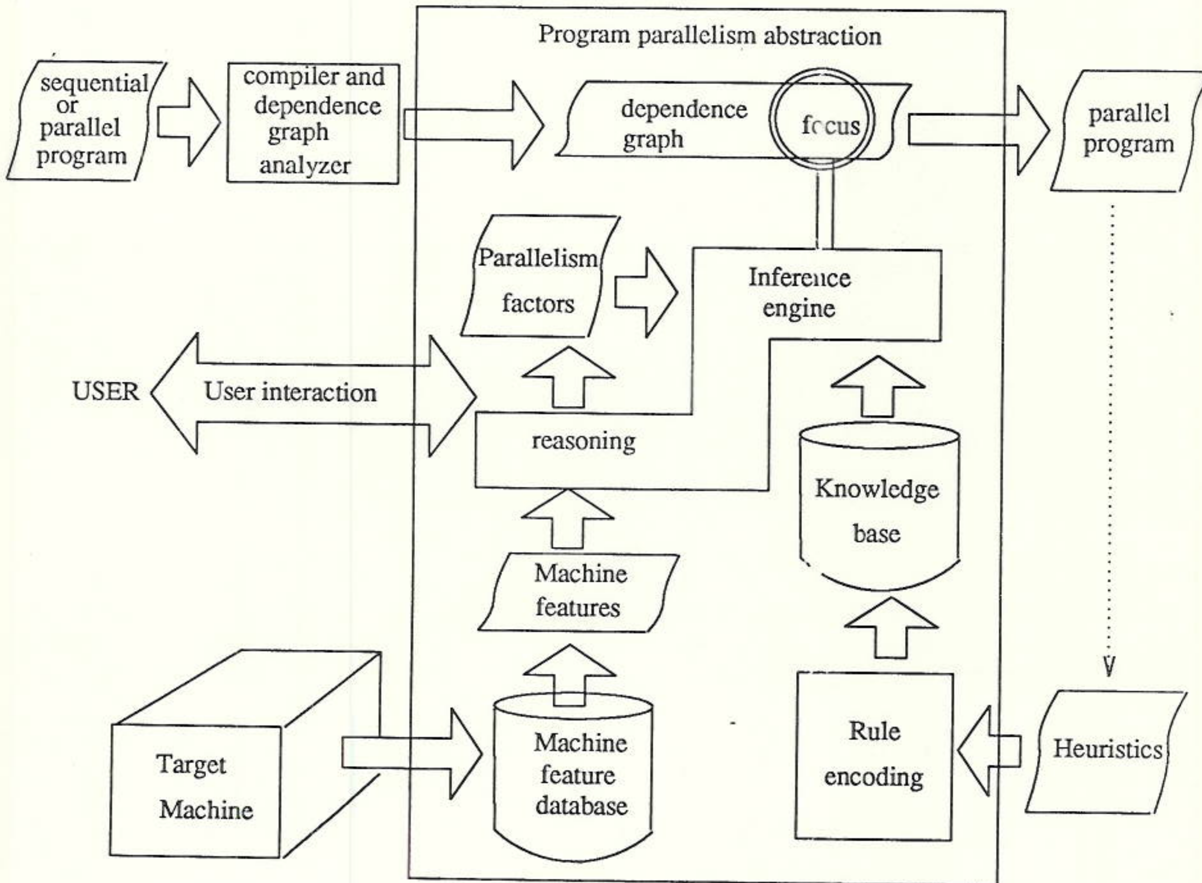

Figure 12.1 System Organization.

The organization of the system components is shown in figure 12.1. As the figure shows, the inference engine analyzes the machine feature list to form the *parallelism factors,* which are the key components of the computational model discussed in the last section. It selects part of the dependence graph as the *program focus,* and it analyzes and restructures the focus region based on the parallelism factors and the heuristics in the knowledge base. The structure of the knowledge in the knowledge base is discussed in the next two sections. Figure 12.2 illustrates the process of building the domain dependent knowledge base.

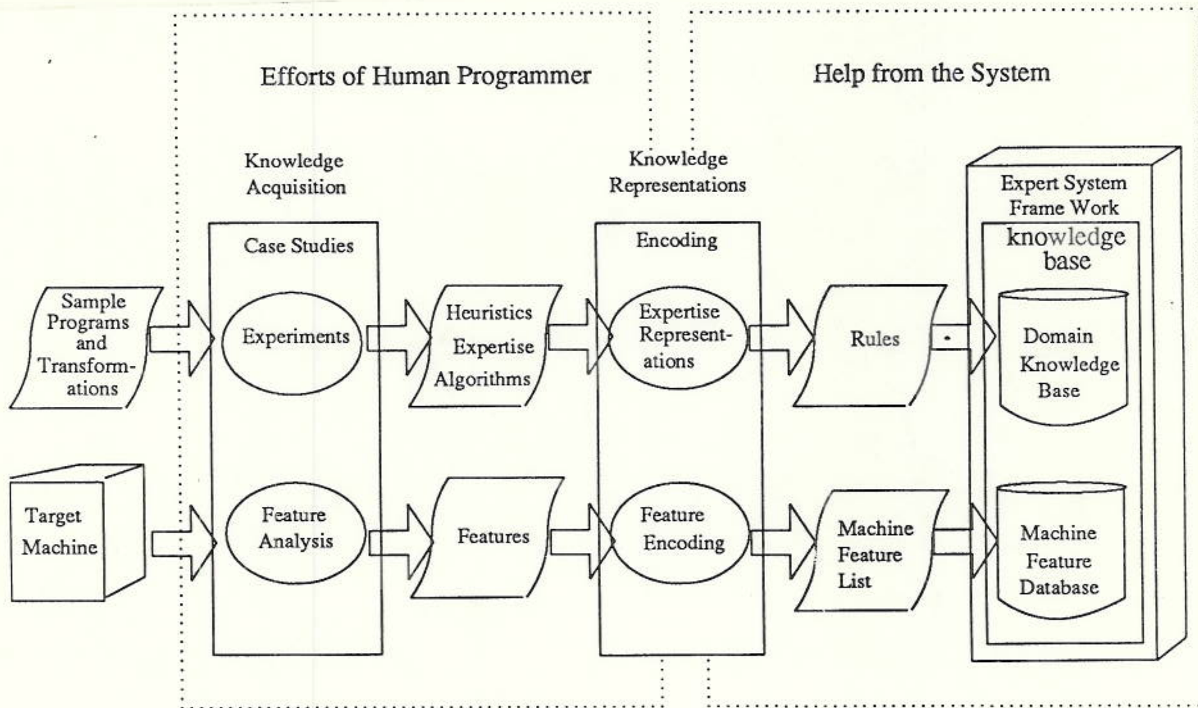

Figure 12.2 Process of constructing domain dependent knowledge base.

12.3.2 Heuristic Hierarchy

While the modularity and integratability of the rule-based expert system makes modifying the knowledge base easy, its inefficient execution and the opacity of the knowledge are the major drawbacks.

For example, translating a heuristic into a set of rules causes the knowledge to be fragmented, this makes the maintenance and modification of the knowledge difficult. Even though there are still strong relations between many of the rules, the fragmentation causes an unfortunate loss of coherence.

In order to improve the integration and modularity of the knowledge, and the efficiency of the system, we have devised a new hierarchical structure to organize the heuristics. This *heuristic hierarchy* is used to integrate the rules into conceptually and logically related units. Since this is a new concept, we devote the remainder of this section to a general description of heuristic hierarchies. In section 12.3.3 we detail the organization of the hierarchy for the program restructuring system.

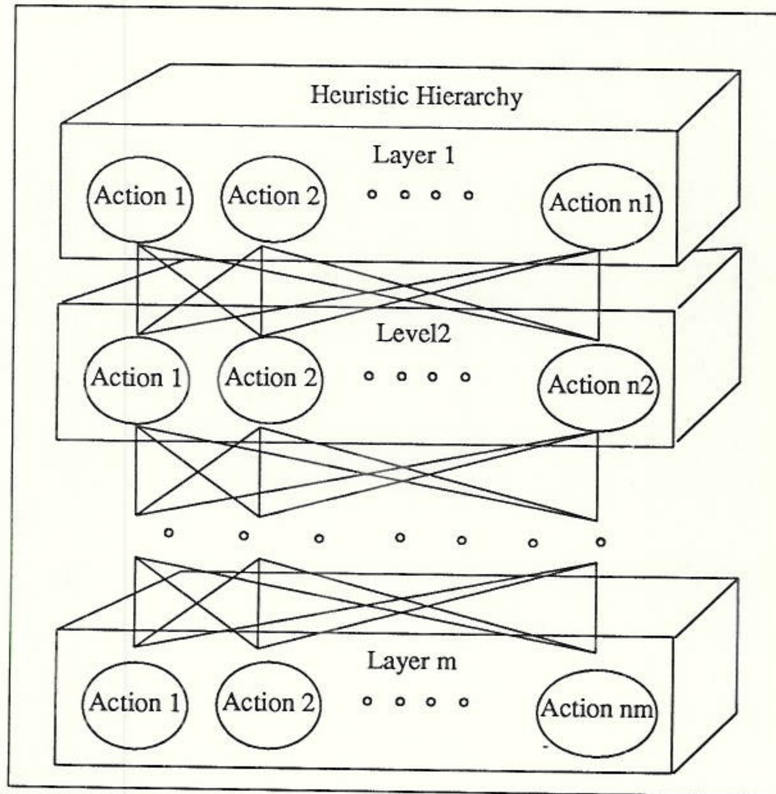

Figure 12.3 Heuristic Hierarchy.

As shown in figure 12.3, a heuristic hierarchy consists of one or more *layers;* rules in the same layers are divided into groups that we call *actions.* Each heuristic hierarchy has a goal and some rules associated with it to accomplish the goal. The actions in the top-most layer represent possible solution steps that the

hierarchy can use in trying to accomplish its goal. In other words, the rules of a heuristic hierarchy can use any actions of the top layer in attempts to satisfy the goal of the hierarchy. For each action, there is a goal for the rules in the action to accomplish. The rules in the action can select among the actions in the lower layer to satisfy its goal. Similarly, the actions in the lower layer may in turn select the actions in the next layer when trying to satisfy their goals. There are no goals associated with the layers because a layer represents a conceptual level of the problem solving process in which different actions can be applied to achieve the goal of the control flow that calls the action.

A complicated action can be organized into a heuristic hierarchy. This recursive definition makes the heuristic hierarchy very flexible and it can be constructed corresponding to the step-wise refinements in a top-down problem-solving approach. In a top-down problem-solving process, the problem is divided into multiple stages that represent the problem solving steps of the process. Each stage can be refined stepwise as the system is implemented.

The inference engine of the heuristic hierarchy works as follows: the process tries to satisfy its goal by executing the rules of the hierarchy. The rules may select any of the actions in the top layer. An action works just like a hierarchy, except that the actions in the next layer may be called by any rules in the action. When a rule fails to satisfy the goal, other rules in the group are tried until either the goal is accomplished or all the rules have been tried. In either case, the control goes back up one level to the previous layer. If the selected action fails to satisfy the goal, an alternative action in the lower layer is selected. This process is repeated until the goal of the hierarchy is either satisfied or failed, and the control flow goes back to the caller of the hierarchy.

This hierarchical structure organization of the heuristics is actually a simplified hierarchical production system. It has the following advantages:

Modularity. Conceptually related rules can be grouped together. Grouping related rules together makes implementing, understanding, maintaining and updating the knowledge base easier. The knowledge representation process that translates heuristics into rules can be done in either a top down or a bottom up fashion.

Efficiency. Only a small subset of the knowledge base needs to be considered at any given instance. The size of the knowledge base for real problems is usually very large. It is very inefficient to perform rule selection and backtracking when a flat structure knowledge base is used.

Flexibility. The order of the actions to be taken can be decided dynamically.

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 a 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.

The hierarchical structure of the rules can be specified by the following hierarchy declaration:

*hierarchy(name, [layer(name, [action] *)] *);*

where the notation *[expression]** represents a list of one or more expressions of the same type. Examples of this will be shown in the next section.

The lexical order of the layers represents the level of the layers from top down. The lexical order of the rules decides the default ordering of the rules to be applied. This default ordering can be overwritten by explicit rules. The order of the actions is irrelevant, since they are selected by the rules in the upper layer.

In the system, knowledge and heuristics are represented as rules of the following form:

```
[Rule, [action_name*]]:
    If
        {condition list}
    then
        {action list}.
```

The action_name* is used to label the action(s) in the hierarchy to which the rule belongs. These hierarchy declarations provide an easy way for the system engineer to specify the structure of the heuristics and keep closely related rules together.

12.3.3 Program Transformation With Heuristics

The program restructuring process is an iterative process. At each step, the dependence graph of the program focus region is analyzed, and a transformation that can improve the parallelism matching between the program and the machine is chosen and carried out. There are two difficulties with this process. The first problem is *"when and how to apply which transformation?"* Different sequences of

transformations may lead to different results. Also, a transformation may have different effects when it is applied to different program states.

The second problem is *"how does the system detect that the program is in its optimal form and stop the transformation process?"* Unlike some other AI problems, there is no good description of the goal states. The goal of performing the transformations is to optimize the matching between the program and the computational machine model. For the same program, there may be many different representations of the program that have the same input-output semantics. The problem is to find a sequence of transformations that transforms the current representation of the program into a representation that allows maximum parallelism on the target machine.

Since it is expensive to test the applicability of the transformations and apply the transformations, and since there may be many different applicable transformation sequences for a given program, it is impractical to try all of the sequences and then to choose 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. The matching functions described in section 2 can be used to measure the effectiveness of the transformations. But we should also note that the matching function can only be used to compare the relative merit of the transformations since an optimal form can only be found after we try all the possible transformation paths.

On the other hand, the user selectable optimization degree indicates how deep the user wants the system to explore. The user can control the optimization depth by choosing the optimization degree or by stopping the process during an interactive session. The optimization degree is a real number between 0 and 1. If the user specifies an optimization degree of 1, the system tries all possible transformation sequences and selects the best sequence to apply. If the optimization degree is set to zero, no program restructuring effort will be tried, the system takes the program as it is and applies the program realization process to parallelize the program. When the optimization degree is set to some number between 0 and 1, the heuristics will be applied in selecting transformations. The higher the optimization degree is, the more aggressive the system is in trying different transformations. The optimization degree also sets a limit for the parallelism matching index to compare against. The attempt at restructuring the program is stopped when the parallelism matching index passes a certain limit, or when the heuristics are exhausted. Another advantage of using a user selectable optimization degree is that different optimization degrees can be set for different regions of the program. During an interactive session, the user can concentrate the attention of the system (as well as his own) on parts of the program that he considers more critical.

Empirical studies of the sequences of transformations have been reported by Kuck and his associates. A number of fixed sequences of transformations, tailored for different architectures, have been investigated and built into the Parafrase project [3, 13, 14, 21]. Although these sequences work well for certain programs on the architectures and problems for which they are designed, the inflexibility of the fixed sequence of transformations may limit potential optimization. In fact, recognizing the shortcomings of fixed sequences of transformations, the Parafrase system relies on the user to provide the sequences of transformations as options for particular applications that the user knows well. Also, the user can provide assertions or directives to help the compiler recognize the parallelism that it overlooked.

In our system, heuristics are organized into heuristic hierarchy structures. The heuristic hierarchy and other user interface mechanisms are built on top of the UNIX C-Prolog. In the following subsections we explain the organization of the heuristics and illustrate the operation of the inference engine with an example.

12.3.4 Organization of Transformation Heuristics

There are three kinds of transformation heuristics: the heuristics to define program parallelism and machine parallelism, the heuristics to restructure the program to match parallelism between the program and the machine, and the heuristics to control the parallelism matching process. These three kinds of heuristics correspond to the three layers in the heuristic hierarchy which we call the *parallelism-defining* layer, the *parallelism-matching* layer, and the *parallelism-matching control* layer. Each of these three layers are further divided according to the purpose and effects of the heuristics. The hierarchy structure of the transformation heuristics is shown in figure 12.4.

The parallelism-defining layer is the basis of the program restructuring process. It defines the program parallelism and the machine parallelism by asserting facts about parallelism into the solution state. The computational model represents the machine parallelism and its construction is based on the machine features and the heuristics of utilizing them. The program parallelism is represented by program dependence graphs. The parallelism matching functions and the heuristics (for analyzing the matching between the program and the computational model) are included in this layer. Customized conflict resolution strategies and inference rules can be added to this layer as well.

The program parallelism optimization process improves the matching between the program and the machine by repeatedly selecting program regions and restructuring them. Corresponding to this process, the parallelism-matching

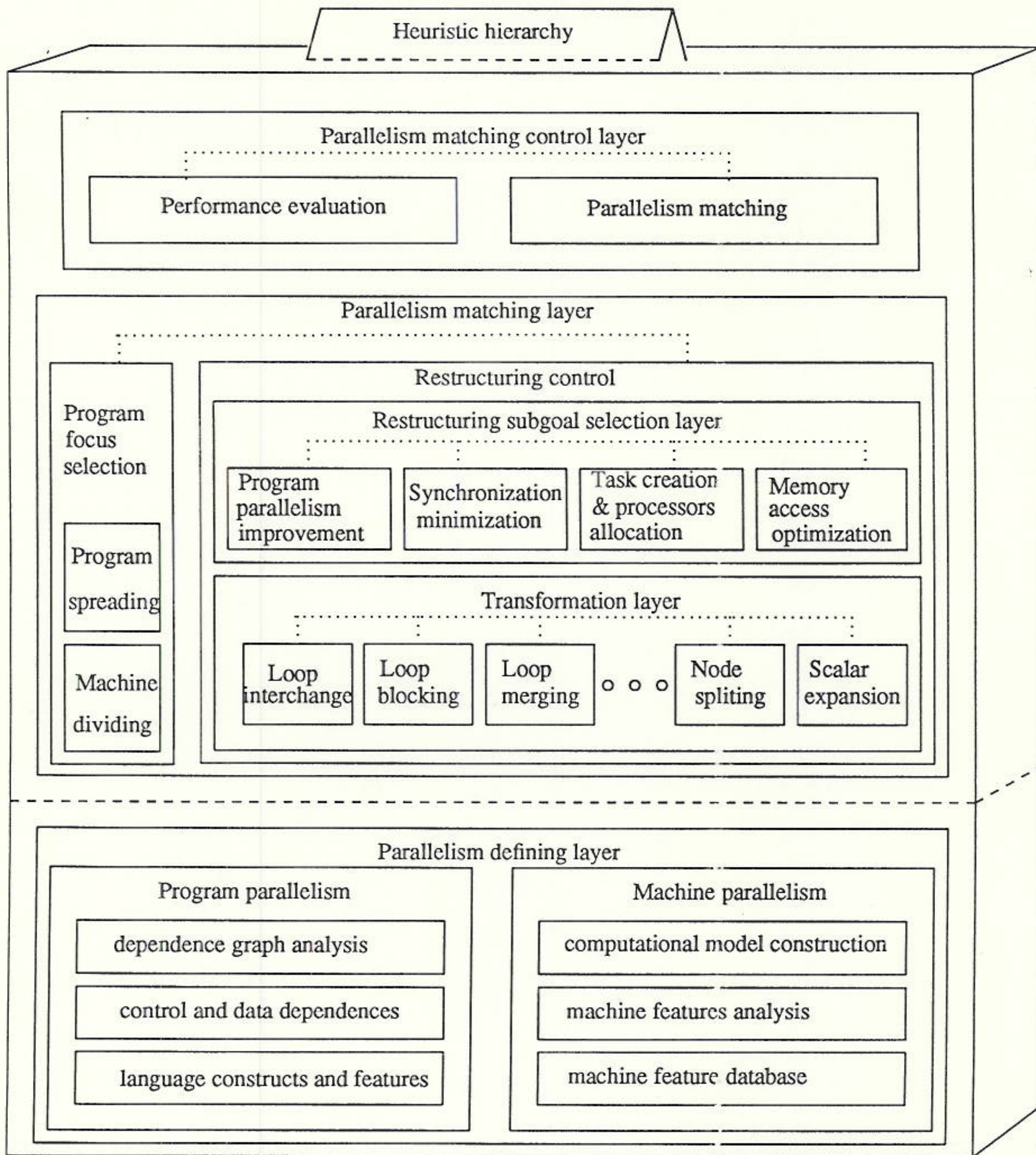

Figure 12.4 Heuristic hierarchy of transformation knowledge

layer consists of two actions that are implemented as hierarchies: the *program focus selection* and the *program restructuring control.* The program focus selection process is responsible for selecting the program fragment to optimize, and the program restructuring control process utilizes heuristics to optimize the program focus.

The program restructuring control process is the part of the heuristic hierarchy that actually selects and performs the transformations. Corresponding to the problems of parallelism optimization discussed in section 12.2, the purposes of the transformations can be classified into the following four categories: improving program parallelism, minimizing synchronization, creating tasks and allocating processors, and utilizing memory usages. Since each transformation may fit into several categories, we separate the heuristics in the program restructuring control layer into two layers: the *program restructuring subgoal selection* layer and the *transformation* layer. The restructuring subgoal selection layer contains the heuristics for solving the four problems mentioned above, and the transformation layer contains the transformation techniques which we termed *transformation modules.*

Each transformation module consists of the description of the transformation technique, the conditions for the transformation to be applicable and the procedures to carry out the transformation. Also included in the module are the heuristics about feasibility of the transformation under various circumstances, short-cut rules in applying the transformation, methods of estimating the effects of the transformation, etc. As an example, the module for "loop interchange" is outlined below. The direction vector notation is taken from [29].

Module Name : Loop interchange

Purpose : Change the order of headers of
nested loops into 'optimal' ordering.

Description : Based on heuristics, compute the loop order
that matches the computational model best.

Restrictions : Loop orders that cause a dependence to
have direction vectors in the form of
(..., <, ..., >, ...) is prohibited.

Test Algorithm : Procedure legal_order(L, ORD)

```
Given a loop order ORD,
   for each dependence DEP in the loops do
      if the direction vector of DEP has the form
            (..., <, ..., >, ...) according to ORD
      then    return(fail);
   end for
return(true);   /* The order is legal */
```

Applying Algorithm: /* find the best ordering of the loops. */

```
procedure best_loop_order(L)
old-ord = generate-loop-order(L);
while ((new-ord = generate-loop-order(L)) != NULL) do
      old-order = better-order(old-ord, new-ord);
return(old-ord);
```

Transformation Algorithm: Loop_interchange(Outmostlp, Norder)

```
change all distance vector according to Norder;
update control dependence of the loop headers;
```

Heuristics :

```
if has_IO(FOCUS)
then fail.

if (is_loop(FOCUS)) and (not nested(FOCUS))
then apply loop_distribution(FOCUS).

if (nested_loops(L1, L2)) and
   (in(S1, L1) and in(S2, L2)) and
   (dep(S1, S2, [<, >]))
then not interchangeable(L1, L2).

if ('memory optimization dominates instruction minimization')
then
   (set(weight, size-matching, light)) and
   (set(weight, memory-access-matching, heavy))
   ......
```

The program restructuring process can be divided into the following stages that we termed the *program restructuring subgoals.* These include the *program parallelism improvement subgoal,* the *synchronization minimization subgoal,* the *task creation and processor allocation subgoal,* and the *memory-access optimization subgoal.* A transformation might be applied in different situations for different reasons. Therefore, each subgoal category may select any of the transformations in the underlying transformation layer. Rules in each of the program restructuring subgoals select the appropriate transformations to apply. The selection of the transformations is based on the heuristics in the transformation layer and the parallelism-defining layer.

The program parallelism improving subgoal consists of rules about the methods of improving program structures. This goal is achieved by restructuring the program to cut down on the amount of data or control dependence presented in the program dependence graph. The synchronization minimization subgoal

contains the heuristics for trying to decrease the cost of inter-processor synchronization. The task creation and processor allocation subgoal is formed by the heuristics for both decomposing the program into tasks and matching the tasks against the target machine. The memory-accesses optimization subgoal is aimed at utilizing the memory hierarchy. Issues considered here include array decomposition and allocation, cache utilization, inter-task communications minimization, and improving locality.

The program focus selection layer cooperates with the parallelism matching control layer in selecting the appropriate program focus. It consists of rules to select a portion of the program to serve as the current focus of program restructuring. Depending on the size and the structure of the program, as well as the optimization degree that the user sets, the size of the program focus ranges from a loop to the whole program. If the program is complicated, a divide-and-conquer strategy is used to subdivide the program. The program is divided into several "super-tasks" and each super-task is restructured separately. Then the restructured portions are combined based on global considerations. Depending on the dependence relations, the super-tasks of programs can be executed either sequentially or simultaneously. If these super-tasks are to be executed sequentially on the target machine, then each part is restructured based on the computational model of the original machine. On the other hand, if some super-tasks of the program are to be executed simultaneously, then the machine is subdivided into several independent virtual machines (or clusters) and the super-tasks are assigned to the virtual machines.

Note that when a program is divided into sub-programs, and the sub-programs are restructured separately, the memory accesses 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 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 subgoals in the parallelism-matching layer 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.

The hierarchy structure significantly improves the flexibility and efficiency of the transformation process. The rules in a layer may select any of the actions (subgoals) in the lower layers. Thus no fixed ordering for applying the actions needs to be specified. This allows the system to be very flexible in deciding the

sequences of the transformations. Unrelated rules do not need to be checked, since only the set of rules in the subgoal selected by the upper layer needs to be evaluated. Furthermore, back tracking only occurs within the set of rules in the same layer.

12.3.5 Applying a Transformation Hierarchy to Program Transformation

The program restructuring process starts by examining the rules on the top layer of the hierarchy. After the focus of the program is chosen, the transformation subgoals on the next layer are selected and the rules associated with the subgoal are involved in selecting the applicable transformations. Similarly, when a transformation is chosen, the rules associated with it are applied to decide the merits and methods of performing the transformation on the program focus.

The flow of control is decided by the rules in the heuristic hierarchy. We will illustrate the decision making process of the system with a simple example. A matrix-vector multiply is a nice illustration of the ideas behind the system, since very few data dependence are involved and many transformations are possible. The program is a simple nested iteration.

```
for i in [1 .. n] do
      for j in [1 .. m] do
            y[i] = y[i] + a[i,j]*x[j];
      end for;
end for;
```

To simplify the discussion we assume that the result vector y has been previously initialized to zero. We seek to transform this program to programs suitable for three different machines: the BBN Butterfly, the Purdue Pringle, and the Alliant FX/8. The rules used in this example are listed in the appendix.

12.3.5.1 Mapping onto the BBN Butterfly

First, we consider the Butterfly. As we discussed in section 2, the machine feature database is first consulted in the construction of the virtual computational model. For example, the fact "parallelize outermost loop without blocking" is added by rule 12.a.1 (listed in the appendix) because the Butterfly provides a mechanism, "GenOnIndex," which can schedule the loops automatically. The system discovers, among other facts, that memory optimization dominates instruction minimization (rule 12.a.5), locality is important, and local memory should be used whenever possible (rule 12.a.6). These facts are added to the system's state space in the working memory.

Next, the transformation heuristic hierarchy is used to optimize the program. First, the parallelism-matching control layer is involved to control the restructuring of the program. In this example, it is trivial to select the program focus. By rule 12.b.1, the whole subroutine is chosen as the program focus, since the original program consists only of a single statement inside the doubly nested loop.

The next step is for the program restructuring control layer to decide which sequence of program restructuring subgoals to achieve. Due to the simplicity of the dependence graph of this program, none of the transformations which are used to break the data dependence cycles are needed. Thus, the parallelism improvement subgoal is skipped (rule 12.c.1). For the sake of flexibility, it is best to do processor assignment toward the end of the transformation process. However, array decomposition can be done only after tasks are created. So there is a conflict in deciding which of the two subgoals, *task creation and processor allocation subgoal* or *memory access optimization subgoal,* should be done first. Our solution to this problem is as follows. First, we find the tentative process allocation scheme and block the outermost loop to create "processes." The newly created outermost loop is marked, but is not actually parallelized. The loop instances of this marked loop form the tentative processes, and this information will be used to guide the array decompositions in memory access optimization subgoal. The actual processor allocation is carried out at the end of the transformation process if the marked loop remains marked by then. This heuristic is encapsulated in the default ordering of rules 12.c.4, 12.c.5, and 12.c.7.

After the task creation and processor allocation subgoal is picked, the system concentrates its restructuring efforts on the loop structures. At this stage, applicable transformations include loop interchanging and loop blocking (to create processes). According to the heuristic (rule 12.e.1), if the program focus is a nested loop, then loop interchanging is checked to find the best order of the loops before the processes are created.

Therefore, the control goes down to the lower level transformation layer, and rules associated with loop interchanging are applied. We assume that the arrays in Butterfly are stored in row order. There are no dependence relations that prevent us from interchanging the loop, so the loop is interchangeable. However, if loop j is changed to be the outermost loop, the array a will be accessed in columns no matter how we block the outer loop to form processes. This is not attractive because it increases the inter-task communications significantly. Therefore, based on the rules associated with loop interchange, the system decides that the original loop order is the best and that no loop interchange is needed.

The next step is to find a tentative way of allocating the processes to the processors. Since the Butterfly has an instruction, *GenOnIndex,* that can schedule

the loops automatically, we can parallelize the outermost loop without blocking (rule 12.a.1). As a result, the outer loop i is marked to form tasks (rule 12.e.4). There are n instances of the loop i, so n tasks are formed if each loop instance is viewed as a task. This information will be used to guide the array decompositions when the memory access optimization subgoal is involved.

After the processor allocation phase, rule 12.c.3 chooses the memory access optimization subgoal. Since local memory access is faster than global memory access on the Butterfly, locality is important (rule 12.a.6). Also, the Butterfly supports a "block-transfer" instruction, which allows a block of memory to be transferred to, or from, the local memory to speed up the data transfer. This makes copying array references inside loops into local memory beneficial. In the matrix-vector multiply program, there are two array references in the nested loops. Each element of array x is accessed once by every instance of the loop j. Also, elements of the i-th row of the array a are accessed exclusively by loop instance i. Since loop i is marked to be parallelized in the "processor allocation" subgoal, every processor that runs loop instance i will have to access every element of the array x and the i-th row of array a once. Rule 12.f.1 suggests we copy array x and array a into local memory with block transfer operations. Since the i-th iteration accesses only the *i-th* row of the array a, there is no need to copy the whole array. The block transfer operation on array a is later changed by rule 12.f.2 into a block transfer operation on row i of the array a in loop i. This gives us (by applying rule 12.f.3):

```
for i in [1 .. N] do
      block_transfer(x, x_local, sizeof(x));
      block_transfer(a[i, *], a_local, sizeof(a[i, *]));

      for j in [1 .. M] do
            y[i] := a_local[j] * x_local[j];
      end for
end for
```

Since the block transfer statement of copying array x does not depend on loop i, it can be moved outside loop i to form another parallelized loop of P instances, where P is the number of the processors (rule 12.f.4). In this way, the array is copied P times instead of N times, as it was in the original form.

After the memory allocations are complete, the parallelism improving subgoal is tried. This is to see if there is any chance to improve the program further. It is relatively easy for the system to recognize that the inner loop j is an inner-product operation (rule 12.d.1), so the loop is replaced by an inner-product operation (rule 12.d.2). The final step involves the processor allocation subgoal again. Since no transformation that might prevent the parallelizing of the outermost loop i (which

is marked for parallelizing) has been performed, the loop is directly parallelized as shown below.

```
coprocess k in [1 .. P] do
     block_transfer(x, x_local, sizeof(x));
end coprocess

coprocess i in [1 .. N] do
     block_transfer(a[i, *], a_local, sizeof(a[i, *]));
     y[i] := inner-product(a_local[*], x_local[*]);
end coprocess
```

12.3.5.2 Mapping onto the Pringle/CHiP

The Pringle/CHiP architecture consists of an array of 64 processors which communicate with each other via a packet-switched message network. There is no shared memory, and each processor runs one process. The communication pattern of messages between processors, defined at compile time as a communication graph, is used to configure the switch network at load time. Each of the memory modules is dual ported. One port goes to the processor while the other goes to a global bus, this allows the local memory of each processor to be a page of the global address of the front-end host. Downloading programs and data to each processor and loading the results of a computation to the host is done over this bus.

For the same reason as in the case of Butterfly, the system decides not to change the original order of loops after the rules in the transformation module, *loop interchange,* are used to decide the order of loop headers. Making the program restructuring task different here are the facts that process creation time on the Pringle is expensive, and no self-scheduling primitive is available. The best strategy for processor allocation on the Pringle is to create P processes to run on the P processors that the Pringle has (rule 12.a.2). So, the n instances of the outermost loop i are blocked to form P tasks (rule 12.e.3). The result is shown below:

```
coprocess k in [0 .. P-1] do
     for i in [k*n/P .. (k+1)*n/P] do
          for j in [1 .. m] do
               y[i] := y[i] + a[i, j] * x[j];
          end for;
     end for;
end coprocess;
```

Next, the memory access optimization subgoal is invoked to allocate the data. Since the Pringle is a non-shared memory machine, all the data must be

distributed among the processors. Array decompositions are done by means of inter-process dependence analysis. By checking the bounds of the loops, the system discovers that the processor which runs process k (k-th iteration of the coprocess loop) accesses only rows $k*n/P$ to $(k+1)*n/P$ of the array a. In terms of the dependence relations, this means that no out-of-bounds dependence (dependence edge that has only one end in the loops) or cross-iteration dependence (dependence whose source and sink are in different loop iterations) of the array a exist. So, it is best to store these rows of the array in the local memory of the processor that runs the task. By rule 12.f.11, the array a is divided into P blocks according to the memory access pattern, and the P blocks are allocated to local memories in the corresponding processors. Similarly, array y can be blocked into P "chunks" and stored in the local memories of the processors. Therefore, each of the processors computes n/P components of the y vector.

Since each process uses all the elements of array x, the processor that runs the process needs to access the whole array x no matter where the array is allocated. If we are free to allocate the array x anywhere, the most direct method is to put it in one processor, say PE0, and then "broadcast" it to other processors by means of a pipeline process (rule 12.f.12). To accomplish this, each element of x is passed from one processor to the next by using a "channel" variable. This transformation is termed "pipelining," which is a modified version of the transformation "scalar expansion" to pass the data through "channel_variables" instead of temporary variables. The channel variable $Ch_x[k]$ implements a communication channel between processor k and processor $k+1$. Processor $k=0$ reads the value of $x[j]$ and puts it in $Ch_x[0]$. Processor $k=1$ reads the value in $Ch_x[0]$ and puts it into $Ch_x[1]$, etc. The result of the transformation is shown below:

```
coprocess k in [0 .. p-1] do
      local tmp;
     for j in [1 .. m] do
           tmp =  if (k===0) then x[j] else Ch_x[k-1];
           Ch_x[k] = tmp;
           for i in [k*n/p .. (k+1)*n/p] do
                 y[i] = y[i] + a[i,j] * tmp;
           end for;
     end for;
end coprocess;
```

On some non-shared memory machines it is too costly to send a message consisting of only one word (for example, the Intel IPSC and the N-cube). In this case, it is best to send large segments of the x vector through the pipeline at a time.

Perhaps the most important problem to be solved for both non-shared memory machines and shared-memory machines that require programs exploit locality is how to analyze a program and derive an optimal partition of the data structures.

12.3.5.3 Mapping onto the Alliant FX/8

In the case of the Alliant FX/8 there are three important programming issues. First, because of the powerful vector instruction set in each processor, one should exploit as many vector operations as possible. Second, since cache access is twice as fast as a memory access, the programmer must force as many memory accesses to be from the shared data cache as possible. Third, because only one operand in a vector instruction may come from memory or cache, it is important to keep vector operands that are used repeatedly in vector registers.

Most parallel compilers can recognize the inner-product operation in the original matrix vector multiply program and translate the program into the following form:

```
for i in 1 .. n do
    y[i] = inner_product(A[i, *], x);
```

Although the Alliant supports fast inner-product operations, this transformation does not really utilize the parallelism capabilities of the Alliant FX/8. Each processor that runs the program accesses the array x n times, so the array x needs to be brought into the cache repeatedly. Since each vector register in the Alliant FX/8 can hold only thirty-two words of data, the vector x and the matrix a in the sample program need to be loaded into the vector registers repeatedly. This data traffic floods the bus and slows down the computations significantly.

In general, without intelligent program analysis, this communication bottleneck problem is hard to solve. Our system tries to improve the matching between the program and the computational model of the Alliant by examining and managing the memory accesses intelligently.

As in the case of the Butterfly, task creation and processor allocation is the first subgoal selected. Since the Alliant has a vector capability, both the vector processing parallelism in the innermost loop and the multi-processing parallelism in the outermost loop need to be explored. Before the outer loop is blocked to form tasks and the inner loop is blocked to form vector operations, loop interchange is considered to find the best ordering of the loop headers (rule 12.e.1). So control goes down to the transformation layer, and the rules associated with the transformation "loop interchange" are applied. First, the nested loops i and j in the original source are checked, and the conclusion that they are interchangeable

is reached. Next, rules about loop orders are applied to decide the best order of the loop headers. Program size matching and memory utilization matching indices can be used to select the loop order. Rule 12.a.5 suggests that memory optimization dominates the instruction minimization, so memory optimization matching is considered.

The matrix-vector multiply program accesses vector x n passes in total, one pass for each loop instance of loop i. Loop j is the loop that scans through vector x. If loop j is the inner loop, and loop i is the outer loop, then each value of the vector x will be accessed once by every loop instance of loop i. Therefore, the vector needs to be brought into cache repeatedly. On the other hand, if loop i is the inner loop and loop j is the outer loop, the value *x[j]* is brought into the cache and used by all loop instances of the inner loop i for each loop instance of the outer loop j. In this loop order, the network traffic for references of vector x is decreased significantly. Therefore, the loop order where loop j is outside is preferred according to the memory allocation matching function. In other words, the loops need to be interchanged.

After the loops are interchanged, the innermost loop is blocked to form vector operations, and the outermost loop is translated into tasks and may be blocked to form processes. For the vector loop blocking, the inner loop i is blocked according to the vector register size of the Alliant (rule 12.e.2). The vector operation is created by vectorizing the innermost loop after the blocking. The resulting program is shown below. Each loop instance of the outermost loop j forms a task. Since the Alliant instruction set provides a means to automatically allocate the processes to the 8 processors, no loop blocking is needed to match the number of processes with the number of processors (rule 12.a.1). Subsequently, loop j is marked to be parallelized.

```
for j in [1 .. m] do
    for k in [0 .. n/32-1] do
        k1 = k * 32 + 1;
        k2 = (k+1) * 32;
        y[k1 .. k2] sum= a[k1 .. k2, j] * x[j];
    end for;
end for;
```

The next step is to perform memory access optimization. Rule 12.a.7 suggests that keeping one vector operand in a vector register is beneficial. Since vector segment y[k*32+1 .. (k+1)*32] is used repeatedly by each instance of the outer loop j, it is best to keep this segment in the vector register. This can be accomplished by interchanging loops j and k (rule 12.f.13). Note that in the previous task creation and processor allocation subgoal, the loop j is marked as "to be parallelized". However, according to rule 12.f.14, the utilization of vector registers and vector operations is weighted to be more important. So the previous decision

is revoked, and the loops are interchanged. Loop k becomes the outermost loop to be parallelized. The resulting program is:

```
coprocess k in [0 .. n/32-1] do
    local k1, k2 : int;
    k1 = k * 31 + 1;
    k2 = (k+1) * 32;
    for j in [1 .. m] do
        y[k1 .. k2] sum= a[k1 .. k2, j] * x[j];
    end for;
end coprocess;
```

In the final version, each 32 word y vector segment can be saved in a register for the lifetime of the process and can be written to memory only at the end of the computation. Experiments performed in collaboration with Dan Sorensen at the Illinois Center for Supercomputer Research and Development [CSRD] have shown that this implementation of the program is the fastest version of a matrix-vector multiply available for the machine.

The matrix-vector multiply example described above served three purposes:

1. It demonstrated how the inference engine works.
2. It illustrated the fact that a different sequence of transformations was required to produce an optimal program for each machine.
3. It showed the complexity of the program parallelism optimization process.

Many heuristics were needed even for this simple program. This reinforces our view that an expert systems approach is a more flexible and extensible approach than the conventional hard-wired heuristics approach.

On the other hand, the example described above is far too simple to illustrate many of the most interesting and important issues in program restructuring. In particular, it fails to illustrate the issues relating to the introduction of synchronization needed in many problems to satisfy data dependence constraints between parallel tasks. This topic and many other are considered in greater depth in [26].

12.4 Conclusion

Different parallel architectures use different properties of parallel algorithms to speed up computation. These properties require different programming methodologies and heuristics in order to be well utilized. Most users of scientific parallel computers use the following approach: they study the target parallel architecture

extensively, then develop tricks and expertise to utilizing the architecture. From these experiences, they carefully code their applications to exploit the parallelism provided by the hardware. This "study and experience cycle" may need to be repeated many times before the resulting program achieves a satisfiable speed-up. As a result, users need to pay a great deal of attention to the problem of matching program parallelism to machine parallelism for each application. Furthermore, algorithms tailored to suit the particular underlying hardware may not be easily ported to other machines without major modifications. It is clear that this approach is expensive in human terms, i.e. software development and maintenance grow as the diversity of parallel machines increases.

Although most program transformation techniques are machine independent, the heuristics of applying these techniques to the target machine are not. These heuristics are based on extensive study of the particular target machine and are usually hard-wired into a compiler. As a result, existing parallel compilers/restructurers can only generate parallel code for one particular target machine. Much effort must be spent in order to build compilers for different machines even though much of the knowledge can be transferred with minor modifications. Furthermore, the transformation sequence is often predefined by the compiler or specified by the user as an option to the compiler. Given the dynamic nature of programs, this approach is not flexible and may not be able to generate optimal code across a wide spectrum of algorithms.

Building an interactive program restructurer is an attempt to improve the programming environment to allow users to experiment with different program restructuring sequences interactively. But the user still has the burden of matching program parallelism with the underlying machine. From our point of view, what the user really needs is a user friendly environment that is capable of exploring program parallelism and providing expert advice for different architectures when it is requested to do so.

The expert systems approach of program parallelism optimization has the following advantages over the conventional hard-wired approach:

Modularity. The heuristic hierarchy structure provides a means to organize the program transformation heuristics into a modular form for easy understanding and maintenance of the system. Basing heuristics on both the program features and machine features can clean up the heuristics and allow the heuristics to be used for different parallel machines. It also makes modifying and expanding the system easy. New heuristics can be easily installed. Porting the system to new target machines is just a matter of specifying the machine features and providing a mechanism go generate target code for that machine.

Flexibility. The decision of which transformation to apply is made dynamically during the program optimization process. Both current program structures and the target machine features are considered as the program is optimized. This allows the system to select transformations that suit the particular program and target machine well.

Retargetablity. The system can handle different kinds of target machines. It would be very difficult, if not impossible, to implement a program parallelism optimization system using the conventional hardwired approach.

In its current form, our system consists of three major components: an interactive incremental parser/structured editor for a simple functional language BLAZE [MeVR85] or FORTRAN, an interactive graphics based program restructurer that allows the user complete control over the program restructuring process, and the knowledge base and inference engine described in this paper. All three components now work in prototype form only, and much work remains to be done before we will know if this experiment has been a success. Experimental results and many more details about the system will be published in a later volume [26].

References

[1] J.R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. Thesis, Rice University, Houston, Texas, April 1983.

[2] J. Allen, and K. Kennedy, "A Parallel Programming Environment," technical report, Rice COMP TR84-3, July 1984.

[3] W. Abu-Sufah, D. Kuck and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," *Proc. of the 1979 National Computer Conf.,* June, 1979, 969-974.

[4] M. Burke, R. Cytron, "Interprocedure Dependence Analysis and Parallelization," *Proc. of the 1986 Compiler Construction Conference.*

[5] P. Cohen, E. Feigenbaum, "The Handbook of Artificial Intelligence," Vol. 3, William Kaufmann, 1981.

[6] R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois, Urbana-Champaign Aug., 1984 Report No. UIUCDCS-R-84-1177).

[7] J. Ferante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10208, Aug. 1983.

[8] D. Gannon, J. Van Rosendale, "On the Communication Complexity of Parallel Numerical Algorithms," *IEEE Trans. on Computers,* Dec. 1984, C-33 #12, 1180-1194.

[9] A. Kapauan, D. Gannon, L. Snyder and T. Field, "The Pringle Parallel Computer," *Proc. of the 11th International Symposium on Computer Architecture, IEEE,* 1984, 12-20.

[10] A. Kapauan, K. Wang, D. Gannon, J. Cuny and L. Snyder, "The Pringle: An Experimental System for Parallel Algorithm Design and Testing," *Proc. of the 1984 International Conference on Parallel Processing,* Keller, editor.

[11] K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University, October 1980

[12] J. Kowalik, "Parallel MIMD Computation: Hep Supercomputer and Its Applications," The MIT Press, 1985.

[13] D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proc. of the 4th Inter'l Computer Software and App. Conf.,* October, 1980, 709-715.

[14] D. J. Kuck, R. H. Kuhn, B. Leasure, D. H. Padua and M. Wolfe, "Dependence graphs and compiler optimizations," *Proc. of the 8th Annual ACM Symposium on Principles Of Programming Languages,* Williamsburg, VA., January 1981.

[15] D. Kuck, M. Wolfe, and J. McGraw, "A Debate: Retire FORTRAN?," *Physics Today,* May, 1984, 67-75.

[16] M. Minsky, "A Framework for Representing Knowledge," in P. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975, 211- 277.

[17] P. Mehrotra, J. R. Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," Report No. 85-29, ICASE, NASA Langley Research Center, Hampton, Va., May 1985. (To appear in *Journal of Parallel Computing).*

[18] D. Nau, "Expert Computer Systems," *IEEE Computer,* Feb, 1983, 63-85.

[19] N. J. Nilson, "Problem-solving Methods in Artificial Intelligence," McGraw-Hill, 1971.

[20] D. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois, Urbana-Champaign, Nov. 1979.

[21] D. Padua and D. Kuck, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers,* Vol. C-29, No. 9, September, 1980, 763-776.

[22] G. Phister, A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proc. of the 1985 International Conference on Parallel Processing,* 1985, 790-797.

[23] C. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Thesis, University of Illinois Center for Supercomputer Research and Development. CSRD TR.595, Aug. 1986.

[24] J. Schwartz, "Ultracomputer," *ACM Transactions on Programming Languages and Systems,* Vol. 2, No. 4, October, 1980, 484-521.

[25] K. Wang, "An Experiment in Parallel Programming Environment: The Expert Systems Approach," in K. S. Fu, editor, *Some Prototype Examples for Expert Systems,* TR-EE 85-1, Purdue University, Mar. 1985, 591-624.

[26] K. Wang, "A Program Transformation Expert System: Methodologies for Mapping Programs to Different Parallel Computers," Ph.D. Thesis, in preparation, Dept. of Computer Sciences, Purdue University, West Lafayette, IN 47907.

[27] S. Weiss and C. Kulilowski, "A Practical Guide to Designing Expert Systems," Rowman and Allanheld publishers, 1984.

[28] T. Winograd, "Frame Representations and the Declarative/Procedural Controversy," in Bobrow and Collins, editors, *Representation and Understanding: Studies in Cognitive Science,* Academic Press, 1975, 185-210.

[29] M. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Urbana-Chanpaign, 1982, Report no. UIUCDCS-R-82-1105.

Appendix

Rules used in the examples.

Cnstruction of the Computational Model.

Process Creation.

```
[Rule 12.a.1, ['computational model construction']]
if 'has self-scheduling-loop primitives'
then
   assert('parallelize outermost loop without blocking').

[Rule 12.a.2, ['computational model construction']]
if ('process creation cost'(high)) and
   (number-of-processors(P))
then
   assert('number of processes to create'(P)).

[Rule 12.a.3, ['computational model construction']]
```

```
if 'process creation cost'(low)
then
   assert('parallelize outermost loop without blocking').
```

Locality

```
[Rule 12.a.4, ['computational model construction']]
if has-cache
then
   assert('locality is important').

[Rule 12.a.5, ['computational model construction']]
if 'data access/process cost ratio'(large)
then
   assert('memory optimization dominates instruction minimization').

[Rule 12.a.6, ['computational model construction']]
if 'shared/local memory access ratio'(large)
then
   (assert('locality is important')) and
   (assert('use local variable whenever possible')).

[Rule 12.a.7 ['computational model construction']]
if ('has vector register')
then
   ('try to keep vector operand in register')
```

The Program Focus Selection Subgoal

```
[Rule 12.b.1, ['program focus selection']]
if ('nested loop'(PDG)) and
   (nested-in(BB, PDG)) and
   ('single statement block'(BB))
then
   FOCUS = PDG.
   ......
```

The Transformation Selection Subgoal.

```
[Rule 12.c.1, ['program restructuring subgoal selection']]
if ('nested loop'(FOCUS)) and
   (nested-in(BB, FOCUS)) and
   ('single statement block'(BB))
then
   select('task creation and processor allocation').

[Rule 12.c.2, ['program restructuring subgoal selection']]
```

```
if ('compound statement'(Focus))
then
   select('parallelism improvement').

[Rule 12.c.3, ['program restructuring subgoal selection']]
if ('tasks created')
then
  select('memory access optimization').

[Rule 12.c.4, ['program restructuring subgoal selection']]
:- (select('task creation and processor allocation')).

[Rule 12.c.5, ['program restructuring subgoal selection']]
if (('has cache') or ('has arrays in'(Focus)) or ('locality is important'))
then
   select('memory access optimization').

[Rule 12.c.6, ['program restructuring subgoal selection']]
if 'multiple tasks are created'
then
   select('parallelism improvement').

[Rule 12.c.7, ['program restructuring subgoal selection']]
if (('task created'(FOCUS)) and (not 'parallelized'(FOCUS)))
then
   select('task-creation and processor allocation')
```

Parallelism Improvement Subgoal

```
[Rule 12.d.1, ['parallelism improving']]
if (is-a-loop(L)) and
   (L = (for i in [RANGE] do A += B[i] * C[i]; end for))
then
   is-inner-product(L)

[Rule 12.d.2, ['parallelism improving']]
if ('has built-in fast inner product') and
   (is-in(L, FOCUS)) and
   (is-inner-product(L))
then
   apply(transformation(inner-product, L)).

[Rule 12.d.3, ['parallelism improving']]
if ('has fetch and op operations') and
   ('recurrence relation'(STMT))
then
   ('change into accumulation'(STMT)).
```

```
[Rule 12.d.4, ['parallelism improving']]
If ('nested-loops'(Focus)) and
   (not 'perfectly-nested-loops'(Focus)) and
   (('is-multi-processors' and high('task-creation-time')) or
     ('has vector operations'))
then
   apply('loop distribution').
   .......
```

Rules about task creation and processor allocation

```
[Rule 12.e.1, ['task creation and processor allocation']]
if (is-nested-loop(FOCUS))
then
   select(loop-interchange(FOCUS)).
```

```
[Rule 12.e.2, ['task creation and processor allocation']]
if (is-nested-loop(FOCUS)) and
   ('has vector operations') and
   ('size of vector registers'(V)) and
   (V <> 0) and
   (innermost-loop(FOCUS, INNER)) and
   (num-of-iterations(INNER, N)) and
   (V < N)
then
   'loop blocking'(INNER, N).
```

```
[Rule 12.e.3, ['task creation and processor allocation']]
if (is-a-loop(FOCUS)) and
   (outermost-loop(FOCUS, OUTER)) and
   (num-of-iterations(OUTER, N)) and
   (number-of-processor(P)) and
   (N > P)
then
   'loop blocking'(OUTER, P).
```

```
[Rule 12.e.4, ['task creation and processor allocation']]
if ('parallelize outermost loop without blocking') and
   (is-nested-loop(FOCUS)) and
   (outermost-loop(FOCUS, OUTER))
then
   (parallelize(OUTER))
      .......
```

Memory Access Optimization.

```
[Rule 12.f.1, ['memory access optimization']]
```

Assume L2 is the innermost loop that is nested in L1 such that array references of X depends on the loop index of L2. Also let X-sub be the part of the array X whose references depend on the loops inside L2.

```
if (has-instruction(block-transfer)) and
   (shared-array(X)) and
   (parallelize(L1)) and
   (referenced-in(X, L1)) and
   (innermost-depends-on-loop(L1, X, L2)) and
   (sub-depends-on(X, X-sub, L2)),
   (N = sizeof(X)) and
   ('minimal number of references to justify cost of block-transfer' = B) and
   (N > B)
then
   (apply('block transfer'(X-sub, L2))).
```

```
[Rule 12.f.2, ['memory access optimization']]
if (apply('block transfer'(X, L)) and
   (parallelize(L)) and
   ('nested in'(L1, L)) and
   (sub-depends-on(X, X-sub, L1))
then
   (apply('block transfer'(X-sub, L1))).
```

```
[Rule 12.f.3, ['memory access optimization']]
if (apply('block transfer'(X, L))) and
   ('nested in'(L, LO))
then
   ('create temp array'(amp, LO) and
   ('create statement'(S, block-transfer(X, tmp, sizeof(X))) and
   ('insert in front of'(S, L2)) and
   (substitute(X, tmp, L)).
```

```
[Rule 12.f.4, ['memory access optimization']]
if (S = ('block transfer'(A, L, N))) and
   (shared(A)) and
   (local(L)) and
   (nested-in(S, L0)) and
   (parallelized(L0)) and
   ('not depends on'(A, L0)) and
   ('number of processors'(P))
then
   (create-loop(LL, 1..P)) and
   (add-stmt(LL, S)) and
   (parallelized(LL)) and
   ('insert in front of'(LL, L0)).
```

```
[Rule 12.f.5, ['memory access optimization']]
if (S = ('block transfer'(L, A, N))) and
   (shared(A)) and
   (local(L)) and
   (nested-in(S, L0)) and
   (parallelized(L0)) and
   ('not depends on'(A, L0)) and
   ('number of processors'(P))
then
   (create-loop(LL, 1..P)) and
   (add-stmt(LL, S)) and
   (parallelized(LL)) and
   ('append to'(LL, L0)).

[Rule 12.f.6, ['memory access optimization']]
if ('has cache') and
   ('mostly used array'(A, FOCUS))
then
   ('keep in cache'(A)).

[Rule 12.f.7, ['memory access optimization']]
if ('locality is important') and
   ('has local memory') and
   ('data accessing ratio of shared memory-local memory' > 2) and
   (shared-array(A))
then
   (allocate array A to the local memory of each processor).

[Rule 12.f.8, ['memory access optimization']]
if (has-local-memory)
   ('mostly used array'(A, FOCUS))
   (shared-array(A))
   (appears-in(A, S)) and
   ('in nested loops'(S, [L1.. Ln])) and
   ('not depends on loops'(A, L1))
then
   ('create tmp'(tmp, L1)) and
   ('create statement'(S1, (A := tmp))) and
   ('insert in front of'(S1, S)),
   (substitute(A, tmp, L1)).

[Rule 12.f.9, ['memory access optimization']]
if ('mostly used array'(A, FOCUS)) and
   (shared(A)) and
   (appears-in(A, S)) and
   ('in nested loops'(S, [L1.. Ln])) and
   ('depends on loops'(A, L1))
```

then
    (find the plausible loop order ORD with most inner loops that A depends on) and
    'loop interchange'(L1, ORD) and
    (innermost-depends-on-loop(L1, X, LL)) and
    ('create tmp'(tmp, LL)) and
    ('create statement'(S1, (A := tmp))) and
    ('insert in front of'(S1, S)),
    (substitute(A, tmp, LL)).

[Rule 12.f.10, ['memory access optimization']]
if ('has local memory') and
    ('mostly used array'(A, FOCUS)) and
    (shared(A)) and
    ('not modified'(A)) and
    (cache-size(C)) and
    (sizeof(A) > C)
then
    ('create tmp'(tmp, FOCUS)) and
    (scalarize(A, tmp)).

[Rule 12.f.11, ['memory access optimization']]
if ('non-shared memory') and
    (parallelized(L)) and
    (array(A)) and
    (appears-in(A, L)) and
    ('no inter task dependence exist'(A, L)) and
    (sub-depends-on(A, A-sub, L))
then
    allocate-local(A-sub, L).

[Rule 12.f.12, ['memory access optimization']]
if ('non-shared memory') and
    (parallelized(L)) and
    (array(A)) and
    (appears-in(A, L)) and
    ('has inter task dependence in'(A, L))
then
    'pipelining references'(A, L).

[Rule 12.f.13, ['memory access optimization']]
if ('has vector register') and
    ('is a vector'(V)) and
    (appears-in(V, S)) and
    ('in nested loops'(S, LList)) and
    (member(LL, LList)) and
    ('not depends on'(A, LL))
then

'interchange loops to move LL into innermost'

[Rule 12.f.14, ['memory access optimization']]
if ('has vector register')
then
('vector register optimization dominates memory access optimization')
.....