Parallelization

Synonyms

Parallelization

Definition

Autoparallelization is the translation of a sequential program by a compiler into a parallel form that outputs the same values as the original program. For some authors, autoparallelization means only translation for multiprocessors. However, this definition is more general and includes translation for instruction level, vector, or any other form of parallelism.

Discussion

Introduction

The compilers of most parallel machines are autoparallelizers, and this has been the case since the earliest parallel supercomputers, the Illiac IV and the TI ASC, were introduced in the 1960s. Today, there are autoparallelizers for vector processors, VLIW processors, microprocessor vector extensions, and multiprocessors.
Autoparallelization is for productivity. Autoparallelizers, when they succeed, enable the programming of parallel machines with conventional languages such as Fortran or C. In this programming paradigm, code is not complicated by parallel constructs and the obsfuscation typical of manual tuning.

Inserting explicit parallel constructs and tuning is not only time-consuming but also produces non-portable, machine-dependent code. For example, codes written for multiprocessors and those for SIMD machines have different syntax and organization. On the other hand, with the support of autoparallelization, conventional codes could be portable across machine classes.

Explicit parallelism introduces opportunities for program defects that do not arise in sequential programming. With autoparallelization, the code has sequential semantics. There is no possibility of deadlock and programs are determinate. The downside is that it is not possible to implement asynchronous algorithms, although this limitation does not affect the vast majority of applications.

Requirements for Autoparallelization
A parallelizing compiler must analyze the program to detect implicit parallelism and identify opportunities for restructuring transformations, and then apply a sequence of transformations.

Detection of implicit parallelism can be accomplished by (1) computing the dependences to determine where the sequential order of the source program can be relaxed, and (2) analyzing the semantics of code segments to enable the selection of alternative parallel algorithms.

The transformation process is restricted by the information provided by this analysis and is guided by heuristics supported by static prediction of execution time or program profiling.

Dependence Analysis
The dependence relation is a partial order between operations in the program that is computed by analyzing variable and array element accesses. Executing the program following this partial order guarantees that the program will produce the same output as the original code. For example, in

```
for (i=0; i < n; i++) {a[i] += 1;}
for (j=0; j < n; j++) {b[j] = a[j]*2;}
```

corresponding iterations of the first and the second loop must be executed in the specified order. However, these pairs of iterations do not interact with other pairs and therefore do not have to execute in the original order to produce the intended result. Only corresponding iterations of these two loops are ordered. By determining what orders must be enforced, dependence analysis tells us which reordering is valid and what can be done in parallel: two operations that are not related by the partial order resulting from the dependence analysis can be reordered or executed in parallel with each other.

Dependence analysis can be done statically, by a compiler; or dynamically, during program execution.

Static analysis is discussed next, while dynamic analysis is discussed below under the heading of “Runtime Resolution”.

How close the dependences generated by static dependence analysis are to the minimum number of ordered pairs required for correctness depends on the information available at compile time and the algorithms used for the analysis. The loops above are examples of loops that can be analyzed statically with total accuracy because (1) all the information needed for an accurate analysis is available statically, and (2) the subscript expressions are simple, so that most analysis algorithms can analyze them accurately. Accuracy is tremendously important because when the set of dependences computed by a test is not accurate, spurious dependences must be assumed and this may preclude valid transformations including conversion into parallel form.

There are numerous algorithms for dependence analysis that have been developed through the years. They typically trade off accuracy for speed of analysis. For example, some fast tests do not make use of information about the values of the loop indices, while others require them. Ignoring the loop limits works well in some cases. The loops above are an example of this situation. The value of n in these loops is not required to do an accurate analysis. However, in other cases, knowledge of the loop limits is needed. Consider the loop

```
for (i=10; i<15; i++) {a[i]+=a[i-8];}
```

The loop limits, 10 and 14, are necessary to determine that no ordering needs to be enforced between loop iterations since, for these values, the iterations do not
interact with each other. A test that ignores the loop limits will report that (some) iterations in this loop must be executed in order.

Some of the most popular dependence tests require for accuracy that the subscript expressions be affine expression of the loop indices and the values of the coefficients and the constant be known at compile time. For example, a test that requires knowledge of the numerical values of coefficients would have to assume that iterations of the loop

```c
if (m >0) {
    for (i=0; i < n; i+=2){
        a[m*i]+=a[m*i+1];
    }
}
```

must be executed in order, while a test with symbolic capabilities would be able to determine that the iterations do not have to be executed in any particular order to obtain correct results. Table 1 presents the main characteristics of a few dependence tests.

When the needed information is not available at compile time or the analysis algorithm is inaccurate, the decision can be postponed to execution time (see “Transformations for Runtime Resolution” below). For example, the loop

```c
for (i=0; i <n; i++) {a[i+k]+=a[i];}
```

cannot be transformed into an array operation as long as k is negative, but the compiler will not know that this is the case if k happens to be a function of the input to the program or if the value propagation analysis conducted by the compiler cannot decide that k is negative. A similar situation arises in the loop

```c
for (i=0; i < n; i++) {a[m[i]]+=a[i];}
```

where m[i] must be ≤ i or ≥ n and all the m[i]'s be different for a transformation into vector operation to be valid. But this will only be known to the compiler, if it can propagate array values and these values are available in the source code. Otherwise, dependences must be assumed or the analysis postponed to execution time.

### Semantic Analysis

Semantic analysis identifies operators or code sequences that have a parallel implementation. A good example is the analysis of array operations. For example, in the Fortran statement

```fortran
a(1:n) = sin(a(2:n+1))
```

the n evaluations of sin can proceed in parallel since their parameters do not depend on each other.

Although array operations like this can be interpreted as parallel operations, most Fortran 90 compilers at the time of the writing of this entry do not parallelize directly array operations, but instead translate them into loops which are analyzed by later passes for parallelization. So, in effect, they rely on semantic analysis.

The compiler can also apply semantic analysis to sequence of statements with the help of a database of patterns. For example,

```c
for (i=0; i <n; i++) {s+=a[i];}
```

cannot be parallelized by relying exclusively on dependence analysis, because this analysis will only state the obvious: that each iteration requires the result of the previous one (the values of sum) to proceed. However, accumulations like this can be parallelized, if assuming that + is associative is acceptable, and are frequently found in real programs. Therefore, this pattern is a natural candidate for inclusion in this database.

Other frequently found patterns include: finding the minimum or maximum of an array, and linear recurrences such as

```c
x[i]=a[i]*x[i-1]+b[i]
```

### Parallelization, Automatic. Table 1 Characteristics of a few dependence tests

<table>
<thead>
<tr>
<th>Test name</th>
<th># of loop indices in subscript</th>
<th>Subscript expressions must be affine?</th>
<th>Uses loop bounds?</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZIV</td>
<td>0 (constant)</td>
<td>Y</td>
<td>N/A</td>
<td>[5]</td>
</tr>
<tr>
<td>SIV</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>[5]</td>
</tr>
<tr>
<td>GCD</td>
<td>Any</td>
<td>Y</td>
<td>N</td>
<td>[2]</td>
</tr>
<tr>
<td>Banerjee</td>
<td>Any</td>
<td>Y</td>
<td>Y</td>
<td>[2]</td>
</tr>
<tr>
<td>Access Region</td>
<td>Any</td>
<td>N</td>
<td>Y</td>
<td>[9]</td>
</tr>
</tbody>
</table>
Some compilers have been known to recognize more complex patterns such as matrix–matrix multiplication.

Once the compiler knows the type of operation, it can choose to replace the code sequence with a parallel version of the operation.

**Program Transformations**

Program transformations are used to

1. Reduce the number of dependences
2. Generate code for runtime resolution, that is, code that at runtime decides whether to execute in parallel
3. Schedule operations to improve locality or parallelism

**Transformations for Reducing the Number of Dependences**

This class of transformations aims at reducing the number of ordered pairs to improve parallelism and enable reordering. Induction variable substitution and privatization are two of the most important examples in this class. Induction variables are those that assume values that form an arithmetic sequence. Their computation creates a linear order that must be enforced. In addition, using induction variables in subscripts hinders the dependence analysis of other computations. For example, the loop

```c
for (i=0; i<n; i++){j+=2; a[j]=a[j]*2;}
```

cannot be parallelized in this form since j+=2 must be executed in order. Furthermore, dependence analysis cannot know that each iteration of the loop accesses a different element unless it knows that j takes a different value in each iteration. Fortunately, in this example, as in most cases, the induction variable can be eliminated to increase parallelism and improve accuracy of analysis. Thus, here j may be represented in terms of the loop index and forward substituted:

```c
for (i=0; i<n; i++) {a[j+2*i+1]=a[j+2*i+1]*2;}
```

The effect of this transformation is that the chain of dependences resulting from the j++ statement goes away with the statement. Also, the removal of the increment makes j a loop invariant and this enables an accurate dependence analysis at compile time.

The identification of induction variables was originally developed for strength reduction, which replaces operations with less expensive ones. A typical strength reduction is to replace multiplications with additions. For parallelism, the replacement goes in the opposite direction. For example, additions are replaced by multiplications as shown in the last example. Induction variable identification relies on conventional compiler data-flow analysis.

Privatization can be applied when the a variable is used to carry values from one statement to another within the one iteration of the loop. For example, in

```c
for(i=0;i<n; i++){a=b[i]*2; c[i] = a*c[i]}
```

the use of a single variable, a, in all iterations demands that the iterations be executed in order to guarantee correct results because, a should not be reassigned until its value has been obtained by the second statement of the loop body. The privatization transformation simply makes a private to the loop iteration and thus eliminates a reason to execute the iterations in order.

An alternative to privatization is expansion. This transformation converts the scalar into an array and has the same effect on the dependence as privatization. For the previous loop, this would be the result:

```c
for (i=0;i<n;i++) {
    a1[i]=b[i]*2;
    c[i]=a1[i]*c[i]
    }
    a=a1[n-1];
```

Privatization is applied when generating code for multiprocessors, and expansion is necessary for vectorization. The main difficulty with expansion is the increase in memory requirements. While privatization increases the memory requirements proportionally to the number of processors, expansion does so proportionally to the number of iterations, a number that is typically much higher.

However, expansion can be applied together with a transformation called stripmining to reduce the amount of additional memory.

Privatization and expansion require analysis to determine that the variable being privatized or expanded is never used to pass information across iterations of the loop. This analysis can be done using conventional data flow analysis techniques.
Transformations for Runtime Resolution

In its simplest form, runtime resolution transformations generate if statements to select between a parallel or serial version of the code. For example,

```plaintext
do i=m,n
    a(i+k)=a(i)*2
end do
```
as discussed above, can be vectorized if k \( \leq 0 \). The compiler may then generate a two-version code

```
if (k<=0) then
    a(k+m:k+n)+=a(m:n)
else
    do i=m,n
        a(i+k)=a(i)*2
    end do
end if
```

Two-version code can be also be used in other situations. Thus, if the loop contains an assignment statement that accesses memory through pointers in the right- and left-hand sides, such as the loop

```plaintext
for (i=0; i <n; i++) {*(a+i)=*(b+i)+2;}
```

the if statement should check that address a is either less than address b or greater than address (b+n-1).

More complex runtime resolution would be needed for loops like

```plaintext
for (i=0; i <n; i++) {a[m[i]]+=a[i];}
```

where the m[i]'s must be \( \leq i \) or \( \geq n \) and all distinct for vectorization to be possible, or m[i] either = i or outside the values in the iteration space and all distinct for transformation into a parallel loop. In

```plaintext
for (i=0; i <n; i++) {a[m[i]]+=a[q[i]];}
```

the m[i]'s and q[i]'s must be such that m[i] \( \leq q[i] \) and the m[i]'s all distinct for vectorization, or m[i] \( \neq q[j] \) whenever i \( \neq j \) for parallelization. Two-version loops can be generated also in this case, but the if condition is somewhat more complex as it must analyze a collection of addresses. In this last case, the technique is called inspector-executor. Another approach to runtime resolution is speculation, which attempts to execute in parallel and optimistically expects that there will be no conflicts between the different components executing in parallel. During the execution of the speculative parallel code or at the end, the memory references are checked to make sure that the parallel execution was correct.

If it was not, the execution is undone and the components executed at a later time either in the right order or again speculatively, in parallel.

Run time resolution is also used to check for profitability, i.e., that parallel execution will make execution faster. For example, if the number of iterations of a parallel loop is not known at compile time, runtime resolution can be used to decide whether to execute a loop in parallel as a function of the number of iterations. Also, runtime resolution can be used to guarantee that vector operations are only executed if the operands are or can be properly aligned in memory when this is required for performance. For example, SSE vector operations sometimes perform better when the operands are aligned on double word boundaries.

Scheduling Transformations

An important class are the transformations that schedule the execution of program operations or partition these operations into groups. To enforce the order, the compiler typically uses the barriers implicit in array operations or multiprocessor synchronization instructions. One such transformation is stripmining. It partitions the iterations of a loop into blocks by augmenting the increment of the loop index and adding an inner loop as follows:

```plaintext
for (i=0; i <n; i++) {a[i]=a[i]+1;}
```

```plaintext
↓
```

```plaintext
for (i=0; i < (n/q)*q; i+=q)
    for (j=i; j< i+q, j++) {
        a[j]=a[j]+1;
    }
```

```plaintext
for (i=(n/q)*q; i< n, i++) {a[i]=a[i]+1;}
```

Stripmining is useful to enhance locality and reduce the amount of memory required by the program. In particular, it can be used to reduce the memory consumed by expansion. If the goal is vectorization and the size of the vector register is q, this transformation will not reduce the amount of parallelism.

Another type of loop partitioning transformation is that developed for a class of autoparallelizing compilers targeting distributed memory operations. These compilers, including High-Performance Fortran and Vienna Fortran, flourished in the 1990s but are no longer used. The goal of partitioning was to organize loop iterations groups so that each group could
be scheduled in the node containing the data to be manipulated.

An important sequencing transformation is loop interchange, which changes the order of execution by exchanging loop headers. This transformation can be useful to reduce the overhead when compiling for multiprocessors and to enhance memory behavior by reducing the number of cache misses. For example, the loop

\[
\text{for } (i = 0; i < (n/q) \times q; i += q) \\
\text{for } (j = i; j < i + q, j +=) \\
\quad a[j] = a[j] + 1;
\]

can be correctly transformed by loop interchange into

\[
\text{for } (j = 0; j < n; j++) \\
\text{for } (i = 0; i < n, i++) \\
\quad a[i][j] = a[i-1][j] + 1;
\]

The outer loop of the original nest cannot be executed in parallel. If nothing else is done, the only option of the compiler targeting a multiprocessor is to transform the inner loop into parallel form and while this could lead to speedups, the result would suffer of the parallel loop initiation overhead once per iteration of the outer loop. Exchanging the loop headers makes the iteration of the outer loop independent so that the outer loop can be executed in parallel and the overhead is only paid once per execution of the whole loop. Furthermore, the resulting loop has a better locality since the array is traverse in the order it is stored, so that the elements of the array in a cache line are accessed in consecutive order, improving in this way spatial locality.

A third example of sequencing transformation is instruction level parallelization. Consider, for example, a VLIW machine with a fixed point and a floating point unit. The sequence

\[
\begin{align*}
\text{r1} &= \text{r2} + \text{r3} \\
\text{r4} &= \text{r4} + \text{r5} \\
\text{f1} &= \text{f1} + \text{f2} \\
\text{f3} &= \text{f4} + \text{f5}
\end{align*}
\]

contains two fixed point operations (those operating on the r registers) and two floating point operations. Exchanging the second and the third operation is necessary to enable the creation of two (VLIW) instructions each making use of both computational units.

In some cases, the partitioning and sequencing of the operations is not completely determined at compile time. For example the sum reduction

\[
\text{for } (i = 0; i < n; i++) \{s += a[i];\}
\]

once identified as such by semantic analysis, may be transformed into a form in which subsets of iterations are executed by different threads and the elements of a are accumulated into different variables, one per thread of execution. These variables are then added to obtain the final sum. The number of these threads can be left undefined until execution time. In OpenMP notation, this can be represented as follows:

\[
\begin{align*}
\&\text{#pragma omp parallel} \\
\&\text{float sp=0;} \\
\&\text{#pragma omp for} \\
\&\text{for } (i = 0; i < n; i++) \{ \\
\&\quad \text{sp} += a[i]; \} \\
\&\text{#pragma omp single} \\
\&\quad \{s += sp;\}
\end{align*}
\]

or, more simply,

\[
\begin{align*}
\&\text{#pragma omp parallel for reduction (+: sum)} \\
\&\text{for } (i = 0; i < n; i++) \{ \\
\&\quad \text{sp} += a[i]; \}
\end{align*}
\]

It should be pointed out that in this example, it has been assumed that floating point addition is associative, but because of the finite precision of machines, it is not. In some cases, it is correct to do this transformation, even if the result obtained is not exactly the same as that of the original program.

However, this is not always the case and transformations like this require authorization from the programmer. Table 2 contains a list of important transformations not discussed above.

### Autoparallelization Today

Most of today’s compilers that target parallel machines are autoparallelizers. They can generate code for multiprocessors and vector code. Although autoparallelization techniques have become the norm, the few empirical studies that exist as well as anecdotal evidence indicate that these compilers often fail to generate high-quality parallel code. There are two reasons for this. First, sometimes the compiler fails to find parallelism due to limitations of its dependence/semantic analysis or transformation modules. In other cases, it is unable to generate good quality code because of limitations in
Parallelization, Automatic. Table 2  An incomplete list of transformations for autoparallelization

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>Reorganizes computation so that values produced in one iteration are consumed by the same iteration</td>
<td>Reduce synchronization costs</td>
</tr>
<tr>
<td>Distribution</td>
<td>Partitions a loop into multiple loops</td>
<td>Separates sequential from parallel parts</td>
</tr>
<tr>
<td>Fusion</td>
<td>Merges two loops</td>
<td>Reduce parallel loop initiation overhead</td>
</tr>
<tr>
<td>Skewing</td>
<td>Partitions the set of iterations into groups that are not related by dependences (i.e. are not ordered)</td>
<td>Enhance parallelism</td>
</tr>
<tr>
<td>Node Splitting</td>
<td>Breaks a statement into two</td>
<td>Reduce dependence cycles and thus enable transformations</td>
</tr>
<tr>
<td>Software pipelining</td>
<td>Reorders and partitions the executions of operations in a loop into groups that are independent from each other</td>
<td>Enhance instruction level parallelism</td>
</tr>
<tr>
<td>Tiling</td>
<td>Partitions the set of iterations of a multiply nested loop into blocks or tiles</td>
<td>Enhance locality</td>
</tr>
<tr>
<td>Trace scheduling</td>
<td>Reorder and partition the executions of operations in a loop into groups that are independent from each other</td>
<td>Enhance instruction level parallelism</td>
</tr>
<tr>
<td>Unroll and Jam</td>
<td>Partitions the set of iterations of a multiply nested loop into blocks or tiles with reuse of values</td>
<td>Enhance locality</td>
</tr>
</tbody>
</table>

Despite their limitations, autotransformers today contribute to productivity by

1. Saving labor. As mentioned, manual intervention in the form of directives or rewriting is typically necessary, but programmers can often rely on the autotransforming compiler for some sections of code and in some cases the whole program.

2. Portability. Sequential code complemented with directives is portable across classes of machines with the support of compilers. Portability is after all one of the purposes of compilers and autotransformation brings this capability to the parallel realm.

3. As a training mechanism. Programmers can learn about what can and cannot be parallelized by interacting with an autotransformer. Thus, the compiler reports to the programmer is not only useful for manual intervention, but also for learning.

Future Directions

Autotransformation has only been partially successful. As previously mentioned, in many cases today’s compilers fail to recognize the existence of parallelism, or having recognized the parallelism, incorrectly assume that transforming into parallel form is not profitable. Although autotransformation is useful and
Parallelization, Automatic. Table 3 Vectorization directives for the IBM (XLC) and Intel (ICC) compilers

<table>
<thead>
<tr>
<th>Vectorization directive</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma vector always (ICC)</td>
<td>Vectorize the following loop whenever dependences allow it, disregarding profitability analysis</td>
</tr>
<tr>
<td>#pragma nosimd (XLC) #pragma nvector (ICC)</td>
<td>Preclude vectorization of the following loop</td>
</tr>
<tr>
<td>__assume_aligned (A, 16); ICC __alignx(16, A); (XLC)</td>
<td>The compiler is told to assume that the vector (A in the examples) start at addresses that are a multiple of a given constant (16 in the examples)</td>
</tr>
</tbody>
</table>

effective when guided by user directives, there is clearly much room for improvement. Research in the area has decreased notably in the recent past, but it is likely that there will be more work in the area due to the renewed interest in parallelism that multicores have initiated. Two promising lines of future studies are

1. Empirical evaluation of compilers to improve parallelism detection, code generation, compiler feedback, and parallelization directives. Evaluating compilers using real applications is necessary to make advances in autotrace Scheduling of conventional languages. Although there has been some work done in this area, much more needs to be done. This type of work is labor intensive since the best and perhaps the only way to do it is for an expert programmer to compare what the compiler does with the best code that the programmer can produce. This process is likely to converge since code patterns repeat across applications [8]. These costs and risks are worthwhile given the importance of the topic and the potential for an immense impact on productivity.

2. Study programming notations and their impact on autotrace Scheduling. Higher level notations, such as those used for array operations, tend to facilitate the task of a compiler while at the same time improving productivity. Language-compiler codesign is an important and promising direction not only for autotrace Scheduling but for compiler optimization in general.

Related Entries

- Banerjee’s Dependence Test
- Code Generation
- Dependence Analysis
- Dependences

Bibliographic Notes And Further Reading

As mentioned in the introduction, work on autotrace Scheduling started in the 1960s with the introduction of Illiac IV and the Texas Instrument Advanced Scientific Computer (ASC). The Paralyzer, an autotrace Scheduling for Illiac IV developed by Massachusetts Computer Associates, is discussed in [10].

This is the earliest description of a commercial autotrace Scheduling in the literature. Since then, there have been numerous papers and books describing commercial autotrace Schedulers. For example, [11] describes an IBM vectorizer of the 1980s, [3] discusses Intel’s vectorizer for their multimedia extension, and [13] describes the IBM XLC compiler autotrace Scheduling features.

Many of the autotrace Scheduling techniques were developed at universities. Pioneering work was done by David Kuck and his students at the University of Illinois [6, 7]. The field has benefited from the contributions of numerous researchers. The contributions of Ken Kennedy and his coworkers [1] at Rice University have been particularly influential.
Parallelization, Basic Block

Utpal Banerjee
University of California at Irvine, Irvine, CA, USA

Synonyms
Parallelization

Definition
A basic block in a program is a sequence of consecutive operations, such that control flow enters at the beginning and leaves at the end without halt. Basic block parallelization consists of techniques that allow execution of operations in a basic block in an overlapped manner without changing the final results.

Introduction
The operations in a basic block are to be executed in the prescribed sequential order. This execution order imposes a dependence structure on the set of operations, based on how they access different memory locations. A new order of execution is valid if whenever an operation $B$ depends on an operation $A$ in the block, execution of $B$ in the new order does not start until after the execution of $A$ has ended. The basic assumption is that executing the operations in any valid order will not change the final results expected from the basic block.

To reduce the total execution time of the block, one needs to find a new valid order where operations are overlapped. Among all such orders, one must choose only those that are compatible with the physical resources of the given machine. Even when the simultaneous processing of two or more operations is permissible by dependence considerations alone, there may not be enough resources available to process them simultaneously.

There are many algorithms for basic block parallelization. This essay presents four of them: two for a hypothetical machine with unlimited resources, and two for a machine with limited resources. It starts with a section on basic concepts, and after developing the algorithms, ends with a simple example that compares

Bibliography