TITLE
Autoparallelization

BYLINE
David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL
USA
padua@illinois.edu

SYNONYMS
Parallelization

DEFINITION
Autoparallelization is the translation of a sequential program by a compiler into a parallel form that computes the same final 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, processors with multimedia extensions, and multiprocessors including multicores.

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 obfuscation 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.

Another benefit for productivity of programming with conventional languages is that 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 is not a significant limitation for the vast majority of applications.

Requirements for autoparallelization

A parallelizing compiler must analyze the program to detect implicit parallelism and opportunities for restructuring transformations and then applies 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 execution time predictions 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, the dependence analysis tell us what reordering are 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 and dynamic analysis in the next section under the heading of “runtime resolution. “

How close static dependence analysis comes 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 for 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 value 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

```
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 ”runtime resolution” below). For example, the loop

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

can be transformed into an array operation as long as **k** is negative, but the compiler may 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

```
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

```
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,

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, assuming that **+** is associative, 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

x(i)=a(i)*x(i-1)+b(i).

Some compilers have been known to recognize more complex patterns such a matrix-matrix multiplication.

Once the compiler knows the type of operation, it can choose to generate a parallel version by replacing 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 decide 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

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 values 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

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. For example, in strength reduction, multiplication is

replaced by 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 same variable is always used to carry values from one statement to another within the one iteration of the loop. For example, in

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

the use of a single variable, **a**, for the whole loop demands that the iterations be executed in order to guarantee correct results. Clearly, **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. That is, it eliminates some of the relations in the partial order. For the previous loop, this would be the result:

```
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 vectorization 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 memory requirement.

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 also be done using conventional data flow analysis compiler 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,

```
do i=m,n
        a(i+k)=a(i)*2
end do
```

as discussed above, can be vectorized if **k** ≤ **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 for profitability. Thus, if the loop contains an assignment statement that accesses memory through pointers in the right and left hand side, like the loop

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

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

where the **m[i]**'s must be ≤ **i** and all distinct for vectorization to be possible, or all distinct for transformation into a parallel loop. In

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]** ≤ **q[i]** and the **m[i]**'s all distinct for vectorization or **m[i]** ≠ **q[j]** whenever **i** ≠ **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, 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 perform well when the operands are aligned on a double word boundary.

Scheduling transformations

An important class contains those 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. A simple partitioning 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

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

↓

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

Stripmining is useful, for example, 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. Thus, Stripmining the loop

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

into blocks of size **q** and expanding **a** into an array of the size of the block would accomplish the desired result. 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 in use. The goal of partitioning was to organize loop iterations groups so that each group could be schedule 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 of multiprocessors and to enhance memory behavior by reducing the number of cache misses. For example, the loop

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

can be correctly transformed by loop interchange into

for (j=0; j<n; j++) for(i=0; i<n, i++) {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 now 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 floating point unit. The sequence

r1=r2+r3
r4=r4+r5
f1=f1+f2
f3=f4+f5

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

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

once identifies 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:

```
#pragma omp parallel
 {float sp=0;
#pragma omp for
  for (i=0; i <n; i++) {
     sp+=a[i];}
#pragma omp single
  {s+=sp;}
 }
```

or, more simply,

```
#pragma omp parallel for reduction (+: sum)
  for (i=0; i <n; i++) {
     sp+=a[i];}
```

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 its profitability analysis. That is, the compiler incorrectly assumes that transforming into parallel form would slow the program down.

To circumvent these limitations, compilers accept directives from programmer to help the analysis or guide the transformation and code generation process. A few vectorization directives for the Intel C++ compiler and IBM XLC compiler are shown in Table 3. These directives enable the programmer to control some of the transformations applied by a compiler. The programmer can also influence the result by modifying the program into a form that can be recognized by the compiler.

Despite their limitations, autoparallelizers today contribute to productivity by

i. *Saving labor*. As mentioned, manual intervention in the form of directives or rewriting is typically necessary, but programmers can often rely on the autoparallelizing compiler for some sections of code and in some cases all of it.
ii. *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 in general and autoparallelization brings this capability to the parallel realm.
iii. *As a training mechanism.* Programmers can learn about what can and cannot be parallelized by interacting with an autoparallelizer. Thus, the compiler report to the programmer is not only useful for manual intervention, but also for learning.

Future directions

Autoparallelization 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 autoparallelization is still useful and 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

i. *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 autoparallelization 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. There is the concern that this process will only lead to an endless sequence of different situations. However, there is no clear evidence that this will be the case. Furthermore, there are some indications that code patterns repeat across applications [4]. These costs and risks are worthwhile given the importance of the topic and the potential for an immense impact on productivity.
ii. *Study programming notations and their impact on autoparallelization.* 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 co-design is an important and promising direction not only for autoparallelization but for compiler optimization in general.

RELATED ENTRIES

Dependences
Dependence analysis
Banerjee's test
GCD test
Omega test
Semantic independence
ILP test
Trace Scheduling
Modulo scheduling
Software pipelining
Vectorization: The Allen and Kennedy Algorithm
Basic block parallelization
Loop nest parallelization
The Wolfe and Lam algorithm
Unimodular transforamtions
Scheduling algorithms
Parallel code generation
Run time parallelization
Thread-level Speculation
Loop level Speculation
High Performance Fortran
Vectorization
Autovectorization
Instruction–level parallelization

BIBLIOGRAPHIC NOTES AND FURTHER READING

As mentioned in the introduction, work on autoparallelization started in the 1960s with the introduction of Illiac IV and the Texas Instrument Advanced Scientific Computer (ASC). The *Paralyzer*, and autoparallelizer for IlliacIV developed by Massachusetts Computer Associates, is discussed in [10]. This is the earliest description of a commercial autoparallelizer in the literature. Since then, there have been numerous papers and books describing commercial autoparallelizers. 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 autoparallelization features.

Many of the autoparallelization 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 co-workers [1] at Rice University have been particularly influential.

There have been only a few papers evaluating the effectiveness of autoparallelizers. In [7] different vectorizing compilers are compared in terms of a collection of snippets and in [3] the effectiveness of parallelizing compilers is discussed using the Perfect Benchmarks.

More information on autoparallelization, can be found in the related entries or in books devoted to this subject [2, 5, 12, 14]. Reference [5] also contains a discussion of compiler techniques for High-Performance Fortran.

BIBLIOGRAPHY

1. Allen, R. and Kennedy, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987), 491-542. DOI= http://doi.acm.org/10.1145/29873.29875
2. Banerjee, U. K. 1996 *Dependence Analysis*. Kluwer Academic Publishers.
3. Bik A.J.C. The Software Vectorization Handbook. Intel Press. May 2004.
4. Eigenmann, R., Hoeflinger, J., and Padua, D. 1998. On the Automatic Parallelization of the Perfect Benchmarks®. *IEEE Trans. Parallel Distrib. Syst.* 9, 1 (Jan. 1998), 5-23. DOI= http://dx.doi.org/10.1109/71.655238
5. Kennedy, K. and Allen, J. R. 2002 *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc
6. Kuck D.J. Parallel Processing of Ordinary Programs. Advances in Computers 15. 119-179 (1976)
7. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia, January 26 - 28, 1981). POPL '81. ACM, New York, NY, 207-218. DOI= http://doi.acm.org/10.1145/567532.567555
8. Levine D., Callahan, D., and Dongarra, J. A comparative study of automatic vectorizing compilers. Parallel Comput. 17, 1223-1244. 1991.
9. Paek, Y., Hoeflinger, J., and Padua, D. 2002. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.* 24, 1 (Jan. 2002), 65-109. DOI= http://doi.acm.org/10.1145/509705.509708
10. Presberg, D. L. 1975. The Paralyzer: Ivtran's Parallelism Analyzer and Synthesizer. In *Proceedings of the Conference on Programming Languages and Compilers For Parallel and Vector Machines* (New York, New York, March 18 - 19, 1975)., 9-16. DOI= http://doi.acm.org/10.1145/800026.808396

11. Scarborough, R. G. and Kolsky, H. G. 1986. A vectorizing Fortran compiler. *IBM J. Res. Dev.* 30, 2 (Mar. 1986), 163-171.
12. Wolfe M. High performance compilers for parallel computing Addison-Wesley 1996
13. Zhang G., Unnikrishnan P., Ren J. Experiments with Auto-Parallelizing SPEC2000FP Benchmarks. LCPC 2004. 348-362
14. Zima, H. and Chapman, B. 1991 *Supercompilers for Parallel and Vector Computers*. ACM Press

Test name	# of loop indices in subscript	Subscript expressions must be affine?	Uses loop bounds?	Ref.
ZIV	0 (constant)	Y	N/A	[5]
SIV	1	Y	Y	[5]
GCD	Any	Y	N	[2]
Banerjee	Any	Y	Y	[2]
Access Region	Any	N	Y	[9]

Table 1. Characteristics of a few dependence tests

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

Table 2. An incomplete list of transformations for autoparallelization

Vectorization directive	Purpose
#pragma vector always *(ICC)*	Vectorize the following loop whenever dependences allow it, disregarding profitability analysis.
#pragma nosimd *(XLC)* **#pragma novector** *(ICC)*	Preclude vectorization of the following loop
__assume_aligned(A, 16); *(ICC)* **__alignx(16, A);** *(XLC)*	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)

Table 3. Vectorization directives for the IBM (XLC) and Intel (ICC) compilers,