Optimization and Profiling on Vector Computer CONVEX 3820

Kandó Kálmán Polytechnic
Institute of Computer Technology, Székesfehérvár
and
University of Athens

Dr. Rezső Nagy, Associate Professor
Magda Kovács, Senior Lecturer

OPTIMIZATION

Optimization improves the performance of programs. To optimize programs, the CONVEX FORTRAN compiler performs these functions:

- Eliminates unnecessary operations
- Arranges operations in the most efficient order
- Replaces slow operations with faster equivalents
- Takes full advantage of CONVEX architectures

OPTIMIZATION OPTIONS

The CONVEX FORTRAN compiler offers five optimization options, which are specified on the `fc` command line. The compiler transforms code according to the optimization option you specify. These transformations are cumulative: each higher-level option retains transformations of the previous option. The optimization option are the following:

Option	Description
-no	Machine-dependent scalar optimization. This option is default
-O0	Basic block machine-independent scalar optimization
-O1	Basic block and program unit machine-independent scalar optimization
-O2	Vector optimization
-O3	Parallel optimization

SCALAR OPTIMIZATION

A scalar value is a single value or entity. A scalar instruction operates on one or a pair of scalar values. There are two types of scalar optimization:

- machine-dependent,
- machine-independent.

Machine-dependent scalar optimization

At the lowest option (-no), the compiler does machine-dependent scalar optimization, which fully exploits the machine's scalar functional units and registers. Because machine-dependent scalar optimization works at the machine-instruction level, you cannot disable it.

Machine-independent scalar optimization

While machine-dependent scalar optimization works at the machine-instruction level, machine-independent scalar optimization works at two levels:

- Local (basic-block) level
- Global (program-unit) level

A basic block is a sequence of statements ending with a conditional or unconditional branch. Branches do not exist within the body of a basic block. At level –O0, optimization is local to a basic block. The compiler does machine-independent optimization within the scope of a basic block.

A program unit is a subroutine, function, or main section. At level –O1, optimization is local to a program unit and is global with respect to basic blocks. This means that the compiler does machine-independent optimizations across multiple blocks in a program unit at one time.

To improve performance, machine-independent optimizations:

- Reduce the number of times memory is accessed
- Simplify expressions
- Eliminate redundant operations
- Replace variables with constants
- Replace slow operations with faster equivalents

VECTOR OPTIMIZATION

Vector optimization, or vectorization, typically improves the performance of programs that manipulate arrays. For example, suppose you write a loop to add the corresponding elements of two arrays. With vector optimization, the CPU can add up to 128 elements of each array with a single instruction.

The compiler also transform many loops that it cannot vectorize into loops that it can vectorize. This increases the number of loops that the compiler can optimize, which minimize execution time dramatically.

The –O2 option enables vector optimization. It also performs scalar optimization on loops that it cannot vectorize and on loops that are not profitable to vectorize.

PARALLEL OPTIMIZATION

Parallel optimization reduces time to solution by spreading work across multiple CPUs.

The actual savings you can achieve with parallel optimization depend on the application, the load on the system when the application is run, and how well suited your algorithm is to parallel optimization. At best, parallelization can improve time to solution by a factor of N, where N is the number of CPUs on you system. Limitations imposed by algorithms prevent some programs from realizing all of this theoretical improvement.

Every program has at least one thread or sequence of instructions that can execute on a single CPU. Parallel programs have more than one thread. On CONVEX C2 and C3 Series computers, threads can execute on multiple CPUs, which are allocated by the **Automatic Self-Allocating Processors (ASAP)** mechanism. ASAP is a way of getting the most work from multiple CPUs, which gives you the benefits of multiprocessing and parallel processing.

The compiler divides a job into tasks that the processors execute as efficiently as possible, using ASAP technology. The compiler does the first step, which is to look for regions of code it can parallelize. The compiler then generates an instruction that causes a request to be posted in a set of registers called communication registers. During execution, idle CPUs check the communication registers for request. If a CPU finds a request, it begins executing that thread of parallel code. At this point, two or more CPUs are working on different threads of the same job.

When you specify –O3 on the `fc` command line, the compiler performs parallel and vector optimization. It also performs scalar optimization on loops that cannot parallelize or vectorize.

With the –O3 option, the compiler automatically performs parallel and vector optimization at the loop level. The compiler divides loop iterations into separate threads and generates code that is independent of the number available CPUs.

To parallelize constructs other then loop, you can use tasking directives.

VECTOR OPTIMIZATION

Appropriate use of vector instructions is the key to high performance on CONVEX C Series architectures. Vectorization converts scalar operations in loops on array elements into equivalent vector operations. The –O2 compiler option instructs the compiler to vectorize loops in a program. For loops that cannot be vectorized, the compiler carries out the global transformations performed at –O1.

Vector operations use vector registers to perform operations on up to 128 pairs of array elements with a single machine instruction. For vector operations on arrays longer than 128 elements, the compiler partitions the operation into groups of no more than 128 elements. This is called strip mining.

Basic operation

Loops typically perform repetitive operations on multiple elements of arrays. The following loop involves at least 700 instruction executions: load on element of B and an element of C; add them, and store the result in corresponding element of A; increment I and the address of A, B, and C; repeat for each of the next 99 elements.

Example:

```
DO I = 1, 100
   A(I) = B(I) + C(I)
ENDDO
```

At optimization level –O2, the compiler generates vector code to load 100 elements of B and 100 elements of C into vector registers, add them simultaneously, and store 100 resulting elements in A. Think of the vector code as a pseudocode statement involving only four instructions:

A(1:100) = B(1:100) + C(1:100)

Where A(1:100) means A(1) trough A(1:100).

Transformations the compiler performs

The compiler reorders the statement and instructions of a program to make the program easier to vectorize. The following subsections explain the most important of these transformations.

A./ Strip mining

The vector registers hold up to 128 elements. When the iterations of a vectorizable loop are unknown or exceed 128, the compiler strip mines the loop before vectorizing it. Strip mining replaces the original loop with two loops. The inner loop

has an iteration count that never exceeds 128. The outer loop controls the number of times the inner loop is executed. The following example shows a loop that can be strip mined.

Original loop:

```
DO I = 1, N
   A(I) = B(I) + C(I)
ENDDO
```

In the vectorized loop code shown below, `IOUT` is a variable that the compiler uses to count the number of elements remaining to be processed, and the vector operations are shown using above. `I` is the starting index for each vector operation.

Vectorized loop:

```
I = 1
DO IOUT = N, 0, -128
   K = I + MIN(127, IOUT -1)
   A(I:K:1) = B(I:K:1) + C(I:K:1)
   I = I + 128
ENDDO
```

If `N` equals 300, `IOUT` is tested four times. For each comparison of `IOUT` to zero, the table below shows values of `I` and `IOUT` and the elements of array `A` that are calculated.

I	IOUT	Elements processed
1	300	1...128
129	172	129...256
257	44	257...300
385	-84	

The fourth test of `IOUT` fails, so the loop is not executed and no elements of `A` are processed.

Caution: Loops with runtime trip counts greater than 2^{32}-1 will yield incorrect results unless they are manually strip mined.

B./ Loop distribution

Vectorization is only done on simple loop nests. A simple loop nest is one in which all calculations are done in the innermost loop. Nested loops, however, can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests. Consider the loop in the following example.

Original loop:

```
DO I = 1, N
```

```
      B(I, 1) = 0
      DO J = 1, M
         A(I) = A(I) + B(I, J)*C(I, J)
      ENDDO
      D(I) = E(I) + A(I)
   ENDDO
```

Three copies of the `I` loop are created, separating the nested `J` loop from the assignments to arrays `B` and `D`. In this way, all three assignments become vector operations, as shown in the following loop.

Vectorized loop:

```
   DO I = 1, N
      B(I, 1) = 0
   ENDDO
   DO I = 1, N
      DO J = 1, M
         A(I) = A(I) + B(I, J)*C(I, J)
      ENDDO
   ENDDO
   DO I = 1, N
      D(I) = E(I) + A(I)
   ENDDO
```

C./ Loop interchange

The compiler interchanges nested loops for the following reasons:

- To make the loop that is the most profitable to vectorize the innermost loop
- To make the loop that is the most profitable to parallelize the outermost loop
- To make memory accesses to consecutive word in memory
- To make a loop with long vector length (iteration count) inside a loop with short vector length

Consider the matrix addition shown below:

Original loop:

```
   DO I = 1, N
      DO J = 1, M
         A(I, J) = B(I, J) + C(I, J)
      ENDDO
   ENDDO
```

To vectorize the original loop, the compiler interchanges the `I` and `J` loops so that contiguous element of `B` and `C` are loaded into vector registers. This optimization, shown the following example, substantially improves performance over the row by row approach of the source code.

Vectorized loop:

```
DO J = 1, M
   DO I = 1, N
      A(I, J) = B(I, J) + C(I, J)
   ENDDO
ENDDO
```

D./ Paired host and sink

A vector register can sometimes be used as an accumulator, making it possible for the compiler to move loads and stores of the register outside the vector loop. **Hosting** is the movement of an operation (such as loading a register) out of a loop to a basic block preceding the loop.
Sinking is the complement of hosting. The compiler moves an operation, such as a register store, out of a loop to a basic block following the loop. The following example shows a loop nest that is a candidate for hoisting and sinking.

Example:

```
DO J = 1, N
   DO I = 1, N
      A(I) = A(I) * B(I, J)
   ENDDO
ENDDO
```

When this program fragment is compiled at optimization level –O2, the `I` loop is vectorized. Additionally, the load of vector `A` is hoisted above the loop and the store of vector `A` is sunk below the loop. This optimization eliminates the need for repeated vector loads and stores and makes the loop even faster.

The following code shows an example of vector hoisting and sinking.

Example:

```
I = 1
DO IOUT = N, 0, -128
   K = I + MIN(127, IOUT -1)
   V0 = A(I:K:1)
   DO J = 1, N
      V0 = V0 + B(I:K:1, J)
   ENDDO
   A(I:K:1) = V0
   I = I + 128
ENDDO
```

Vector loads and stores are hoisted and sunk only under these conditions:

- The array reference and array assignment has the same subscripts.

- All subscripts of the array are the induction variable of vectorized loop or loop constants.

The compiler sometimes interchanges loops to make a subscript a loop constant so that sinking and hoisting is possible.

E./ IF-DO optimizations

IF-DO optimizations modify loops containing tests to improve vector performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of masked vector instructions that must be execution, thereby improving performance.

There are three types of IF-DO optimization:
- redundant test elimination,
- loop peeling, and
- test promotion.

Each of these is described in detail below.

Redundant test elimination

Redundant test elimination is the simplest of the IF-DO optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code. This optimization is especially relevant when the programs contains DO loops surrounded by IF tests, as shown in the following example.

Original loop:

```
DO I = 1, N
   IF (I .GT. 0) THEN
      DO J = 1, I
         A(I, J) = 0
      ENDDO
   ENDIF
ENDDO
```

Optimized loop:

```
DO I = 1, N
   DO J = 1, I
      A(I. J) = 0
   ENDDO
ENDDO
```

Here the explicit test IF (I .GT. 0) is redundant, since the test is implicit in the DO loop. It is therefore during redundant test elimination.

Redundant test elimination is always performed at optimization levels –O2 and above.

Loop boundary value peeling

Loop boundary value peeling involves removing the first iteration, last iteration, or first and last iteration of a loop to remove conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates to .TRUE. or .FALSE. for the first iteration, last iteration, or first and last iterations.
With the code shown below, the compiler automatically peels off the first and the last tests and rewrites the loop cover the remaining indexes.

Original loop:

```
DO I = 1, 100
   IF (I .EQ. 100) THEN
      A(I) = B(I)
   ELSE IF ( .EQ. 100) THEN
      A(I) = C(I)
   ELSE
      A(I) = -A(I)
   ENDIF
ENDDO
```

Peeled loop:

```
A(1) = B(1)
DO I = 2, 99
      A(I) = -A(I)
ENDDO
A(100) = C(100)
```

In some cases, boundary-value peeling requires replicating large amount of code and can greatly increase the size of the executable file. By default, the computer peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or, if you wish to do so on a loop-by-loop basis, the `PEEL` compiler directive.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option or the `PEEL_ALL` directive. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time an dcan increase the size of code enough to exceed the limits of some of the compiler's internal tables.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL`

compiler directives.
/The compiler does not try to peel unpeelable loops./

Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.
IF-DO interchange is an important special case of test promotion that is performed on perfectly nested DO loops that contain IF statements. In this case, the IF statements are interchanged out of the DO loops, and the DO loops are replicated if necessary. An example is shown below.

Original loop:

```
DO I = 1, 100
   IF (G) THEN
      A(I) = B(I)
   ELSE
      A(I) = C(I)
   ENDIF
ENDDO
```

Interchanged loop

```
IF (G) THEN
   DO I = 1, 100
      A(I) = B(I)
   ENDDO
ELSE
   DO I = 1
      A(I) = C(I)
   ENDDO
ENDIF
```

For loops containing large numbers of tests, loop replication can greatly increase the size of code.

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler promotes tests and replicates code up to a predetermined, conservative limit.

The `-ptst` compiler option increases this limit and can cause a noticeable increase in compile time.
The `-ptstall` option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some compiler's internal tables.

The `-noptst` option disables test promotion.

The `PROMOTE_TEST, PROMOTE_TEST_ALL` and `NO_PROMOTE_TEST` compiler directives provide similar functionality on a loop-by-loop basis.

At optimization levels –O2 and above, the CONVEX FORTRAN compiler automatically performs IF-DO optimizations on DO and hand-rolled loops that contain logical and arithmetic IF statements, and computed GOTO statements. Simple and nested loops, and loops, and loops with exists are handled.

F./ Pattern matching

Pattern matching allows the compiler vectorize certain loops that it cannot otherwise vectorize. At optimization levels –O2 and above, the compiler will recognize a loop that uses an IF test to determine a maximum (or minimum) value stored in an array and replace it with a call to a vectorized subroutine that performs the same task. The following example shows such a loop.

Example

```
CM = 1
XM = A(1)
DO I = 2, N
   IF (A(I) .GT. XM) THEN
      C(M) = I
      XM = A(I)
   ENDIF
ENDDO
```

Similarly, the compiler recognizes loops containing recurrences that can be implemented with a special sequence of vector instructions. The following code shows examples of patterns the compiler matches.

Example

```
DO I = 1, N
   X(I) = X(I-1) + Y(J)
ENDDO
DO I = 1, N
   IF (X(I) .GE. X(M)) M = I
ENDDO
DO I = 1, N
IF (X(I) .EQ. Y(I)) K = I
ENDDO
```

CONDITIONAL INDUCTION VARIABLES

A loop induction variable is a variable whose value is incremented by a constant amount on every iteration of a loop. Loop induction variables that do not change on every iteration are called conditional induction variables. The compiler frequently recognizes these variables and generates vector code for expressions involving them.
In the following example, K is a conditional induction variable, not an upper limit on the vector.

Example

```
K = 0
DO I = 1, 100
   IF (COND(I) .EQ. .TRUE.) THEN
      K = K + 1
      A(I) = B(K)
      C(K) = D(I)
   ENDIF
ENDDO
```

The compiler generates machine instructions that do the following:

- Save values of I for which `COND(I)` is `.TRUE.`
- Count the number of those values.
- Load the vector strip of `B`.
- Expand the vector strip of `B` to the appropriate indexes according to the saved true values.
- Stored expanded vector in `A(1:100).`
- Load the vector `D(1:100).`
- Compress the vector according to the saved true values.
- Store the vector in `C`.

INHIBITORS OF VECTORIZATION

Any of the following conditions can inhibit or prevent vectorization:

- Computed or assigned GOTO statements
- Multiple loop entries or exits
- Function or subroutine calls
- I/O statements
- Equivalenced scalar or array variables

Recurrences

Of these conditions, you may be unfamiliar with recurrence and its variations.

Parallel optimization

At optimization level -O3, the CONVEX FORTRAN compiler performs vector and parallel optimization to enhance program performance. Unlike vector optimization, parallel optimization does not reduce CPU time. Instead, processing of a single program is spread across multiple CPUs, reducing the program's time to solution.

Basic operation

Parallel optimization divides a program into *threads.* A thread is a sequence of instructions that must execute on a single CPU.

The CONVEX FORTRAN compiler finds parallelism at the loop level. The compiler vectorizes inner loops and parallelizes outer loops. Often, the outer loops are the strip-mine loops that the compiler creates when it vectorizes an inner loop.

As with vector optimization, the compiler distributes and interchanges loops to produce the most efficient parallel code. The compiler can parallelize most scalar reductions and assignments with the addition of synchronization code.

As an example of the transformations the compiler performs at optimization level -O3, consider the matrix multiplication shown below.

Example

```
DO I = 1, N
   DO J = 1, N
      C(I, J) = 0.0
      DO K = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The compiler processes this loop nest by distributing the loop nest containing the I and J loops, as shown in the following example.

Example

```
DO I = 1, N
   DO J = 1, N
      C(I, J) = 0.0
   ENDDO
ENDDO
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         C(I, J) =  C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The compiler moves the I loop to the innermost position in each nest, as shown below, so that it can retrieve contiguous elements on successive iterations.

Example

```
DO J = 1, N
   DO I = 1, N
      C(I, J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
   DO K = 1, N
      DO I = 1, N
         C(I, J) = C(I, J) + A(I, K) * B(K, J)
      ENDDO
   ENDDO
ENDDO
```

The compiler strip mines both I loops to the optimal vector length, a function of the loop upper bound (N). In the following examples, MVSL represents that function, and V0 and V1 represent vector registers that can contain up to 128 64-bit elements.

Example

```
M = MVSL(N)
DO J = 1, N
   DO IOUTER = 1, N, M
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
```

```
   DO K = 1, N
      DO IOUTER = 1, N, M
         V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
         V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
         V0 = V0 + V1 * B(K, J)
         C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
       ENDDO
    ENDDO
ENDDO
```

In the second nest, the compiler interchanges the IOUTER strip-mine loop outside of the K loop.

Example

```
M = MVSL(N)
DO J = 1, N
   DO IOUTER = 1, N, M
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
DO J = 1, N
   DO IOUTER = 1,N, M
      DO K = 1, N
         V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
         V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
         V0 = V0 + V1 * B(K, J)
         C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
      ENDDO
   ENDDO
ENDDO
```

In the following example, PARALLEL DO represents a loop that can be processed by multiple CPUs.

Example

```
M = MVSL(N)
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
PARALLEL DO J = 1, N
```

```
   DO IOUTER = 1, N, M
      DO K = 1, N
         V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
         V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
         V0 = V0 + V1 * B(K, J)
         C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
      ENDDO
   ENDDO
ENDDO
```

As shown below, the compiler hoists a vector load and sinks a vector store out of the K loop. The remaining reference to vector V1 chains with the vector addition and vector multiplication in the next statement, resulting in even faster execution.

Example

```
M = MVSL(N)
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = 0.0
   ENDDO
ENDDO
PARALLEL DO J = 1, N
   DO IOUTER = 1, N, M
      V0 = C(IOUTER:MIN(N, IOUTER + M - 1), J)
      DO K = 1, N
         V1 = A(IOUTER:MIN(N, IOUTER + M - 1), K)
         V0 = V0 + V1 * B(K, J)
      ENDDO
      C(IOUTER:MIN(N, IOUTER + M - 1), J) = V0
    ENDDO
ENDDO
```

The combination of these optimizations results in generated code that performs at a level similar to that of hand-tuned assembly code.

Inhibitors of parallelization

Parallelization and vectorization are so closely related that most things that prevent vectorization can prevent parallelization.
The following specific factors can inhibit or prevent automatic parallel optimization:

- Multiple entries or exits
- Function or subroutine calls
- I/O statements
- Equivalenced scalar or array variables
- Nondeterminism of parallel execution
- Loop-carried dependencies (LCDs)

Loops with subroutine calls

The compiler does not automatically parallelize a loop containing a subroutine call. You can force it to parallelize such a loop by inserting the FORCE_PARALLEL directive before the loop. This directive allows parallelization regardless of potential dependencies that the compiler detects. Certain actual dependencies, such as from one scalar to another, cause the compiler to ignore the directive.

If you use FORCE_PARALLEL, you must recompile the called subroutine (or any routines called indirectly) for re-entrancy with the -re option. Each invocation of a subroutine compiled with -re maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables.

The call to SUB in the following example prevents the compiler from automatically parallelizing the loop. FORCE_PARALLEL overrides the compiler's decision, and the compiler generates parallel code for the loop.

Example

```
      .
      .
C$DIR FORCE_PARALLEL
      DO I = 1, N
         CALL SUB(A, B, I, N)
      ENDDO
       .
       .
       END
      OPTIONS -re ! compile for reentrancy
      SUBROUTINE SUB(A, B, I, N)
      REAL A(N), B(N)
       A(I) = B(I) * 3.14
       RETURN
      END
```

The way the code is written guarantees that SUB does not contain any operations violating data independence, so the code can execute safely in parallel.

If a subroutine is called only from within a parallelized loop, compile the subroutine at a level lower than -O3. Only one loop at a time can be run is parallel. Code that can be parallelized within the subroutine cannot execute in parallel. Additional code generated to parallelize the called routine is useless overhead.

Caution

If you use FORCE_PARALLEL to parallelize a loop containing an actual recurrence, the behavior of

the loop may change from one execution to the next. Errors may result at runtime, but no amount of testing can guarantee that an error will be revealed. Analyze your data and algorithms to ensure that your code can be safely parallelized before using this directive. A good test is to run the loop with iterations in reverse order, for example, DO I = N, 1, -1 .

Another way to allow loops containing subroutine calls to parallelize (or vectorize) is to inline the subroutine.

Loop-carried dependency

While only backward dependencies interfere with vectorization, forward and backward dependencies affect parallelization.

The loop in the following example has no dependencies. The compiler can strip mine and vectorize the inner loop and parallelize the strip-mine loop.

Example

```
DO I = 1, N
   A(I) = A(I) + 3.14
ENDDO
```

The compiler transforms the outer strip-mine loop so that it runs in parallel on a multiprocessor machine. The result is a parallel vector loop.

The loop below has a backward loop-carried dependency (LCD) caused by the assignment to A(I+1) . The loop cannot be vectorized or parallelized by the compiler. The loop remains in scalar form.

Example

```
DO I = 1, N - 1
   A(I + 1) = A(I) + 3.14
ENDDO
```

The following loop has a forward LCD. Because forward LCDs do not interfere with vectorization, the compiler strip mines and vectorizes the loop. It is not safe to parallelize a loop that has an LCD, however. The result is a strip-mine vector instead of a parallel vector loop.

Example

```
DO I = 1, N - 1
```

```
    A(I) = A(I + 1) + 3.14
  ENDDO
```

If a loop has dependencies that prevent the compiler from automatically parallelizing it, you can instruct the compiler to insert synchronization code to honor the dependencies. The compiler can then parallelize the loop. Synchronization code causes execution of a thread to halt momentarily, if necessary, until an operation in another thread, on which the halting thread depends, has been performed. The SYNCH_PARALLEL directive instructs the compiler to generate synchronization code.

The overhead of synchronization code often outweighs performance gains from parallelization. Synchronized parallel loops are advantageous only if the amount of code that contains dependencies is small compared to the amount of code that does not contain dependencies.

The compiler can handle most scalar assignments and reductions within parallel loops. For example, the compiler can generate parallel code for the following loop.

Example

```
  DO I = 1, N
    IF (A(I) .LE. 0.0) THEN
      S = S + B(I) * C(I)
      X = B(I)
    ENDIF
  ENDDO
```

Parallelizing code outside of loops

The compiler does not automatically parallelize code outside a loop. You can use tasking directives to instruct the compiler to parallelize such code.
The BEGIN_TASKS directive tells the compiler to begin parallelizing a series of tasks.
The NEXT_TASK directive marks the end of a task and the start of the next task.
The END_TASKS directive marks the end of a series of tasks to be parallelized.

The following example shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel.

Example

```
C$DIR BEGIN_TASKS
        statement 1
C$DIR NEXT_TASK
        statement 2
C$DIR NEXT_TASK
        statement 3
C$DIR END_TASKS
```

The compiler transforms the above code into a parallel loop and creates machine code equivalent to that shown below.

```
 C$DIR FORCE_PARALLEL
         DO I = 1, 3
            GOTO (10,20,30) I
10            statement 1
               GOTO 40
20            statement 2
               GOTO 40
30            statement 3
               GOTO 40
         ENDDO
40       CONTINUE
```

Note

If the task contains a subroutine call and variable passed to the subroutine is referenced within the task, the compiler will issue a warning and fall to parallelize the task. If possible, moving the variable reference to before the BEGIN_TASKS directive will allow parallelization.

Profilers and profiling

A profiler is a software tool that measures the run-time performance of a program. Profiling typically measures the total time it takes each portion of your program to execute. Additional information may be measured as well, such as the number of times routines or loops are executed.
Running your program under a profiler's control enables the profiler to collect performance measurements. After the measurements are analyzed, the profiler reports the performance data.
Cxpa measure a program's entire execution time, reporting the total time spent in individual routines, loops, parallel regions, and basic blocks.
With Cxpa the program's executable must be modified for use with the profiler.
When a program is compiled with a special compiler option, the resulting executable contains extra information that the profiler uses to collect performance data.
Once your program is compiled, you can run it under the profiler's control. The profiler gathers performance data during execution and reports the results.

Why use a profiler?

Profiling a program is different from most other development steps because it typically happens after a program is working correctly.
A good profiler can help you learn how your program executes and where that execution is spent. Using the information Cxpa provides, you can learn which routines and loops are slowest, and which are fastest. Profiling your program compiled at different optimization levels can provide insight into the types of optimizations that work best for given situations.

Cxpa and profiling

Cxpa is an interactive profiler, capable of profiling programs compiled by the CONVEX FORTRAN (Ada, C) compilers.
Cxpa enables you to profile your program in great detail. You can profile at the routine level, or you can profile the loops within the routines. Cxpa also optimized loops, including parallel regions created by compiler. Finally, Cxpa can profile at the block level, including the number of times each basic block executed.

Cxpa provides a variety of performance statistics. For each routine, Cxpa reports:

Total CPU time, not including routines called
Total CPU time, including routines called
Total number of times routine was called
Minimum, maximum, and average time spent in a single call to the routine

For each loop, Cxpa reports:

Total CPU time, not including nested called

Total CPU time, including nested called
Minimum, maximum, and average number of iterations for the loop
Optimization performed on the loop
The loop's nesting level
Vector floating-point performance metrics for vectorized loops

For each parallel region, Cxpa reports:

Total CPU time
Total WALL-clock time
Total process Virtual Time (PVT). Process virtual time is time that at least one thread of a process is executing.
Concurrency factor (parallel efficiency)
Information about each thread created
Chore counts for each thread (representing the distribution of the work loads among threads)

For each block, Cxpa reports:

Total number of times executed
Percentage of routine's total CPU time
Program counter address for the start of the basic block

General steps to profiling with Cxpa

Tuning application performance is an iterative process. Each iteration identifies another area of the program whose performance you may want to improve. After making a change to the program and profiling again you can determine if the change improved performance, and perhaps identify the next area of the program to improve.
In general, profiling consist of the following steps:

Compile your source code for use with Cxpa.
Invoke Cxpa with the name of the executable.
Select the areas of your program to profile.
Run the program to collect performance data.
Generate a performance report.
Interpret the profile report.

After viewing the profile report, you may choose to profile different areas or make changes to the source file. Or, you may want to recompile with a different optimization and profile the program again.

Compiling for profiling with Cxpa

There are three compiler options you can use to prepare your program for profiling. Each option enables you to profile your program in different ways. These options are summarized in Table 1.

Compiler option	Enables Cxpa to profile
-pa	Routines, loops, and parallel regions
-par	Routines only
-pab	Blocks only

Table 1

Compiling considerations

Each source file that you compile for use with Cxpa should be compiled with only one the above options.
Compiling to profile routines and loops. To do this, compile all of your source files with the –pa option:

%fc –pa prog.f –c
%fc –pa sub1.f –c
%fc –pa sub2.f –c
%fc –pa prog.o sub1.o sub2.o

In above example, the compiler generates profiling information for all routines and loops in the source files prog.f, sub1.f, and sub2.f. The –c compiler option generates object files from the source files. These object files are linked together with Cxpa's support routines into an executable in the last step.

Invoking Cxpa

Once you have compiled your program, you can invoke Cxpa using the cxpa command followed by the name of your executable.
If your executable is named a.out and is located in the current directory, you do not need to specify it when invoking Cxpa.
If a.out does not exist, you can still to look at reports generated in previous session, but you will not unable to profile a program and generate new reports.

After Cxpa processes any additional options used to invoke it, Cxpa executes any commands found in two initialization files:

$HOME/.cxpainit - Your default initialization file
./.cxpainit – Your local initialization file

These two initialization files can be used to automatically set up the Cxpa

environment.
After Cxpa executes the commands in the initialization files, the Cxpa window appears.

There are two interfaces to Cxpa:

CRT interface that can run on any ASCII terminal
X interface that runs under the X Window System

Enabling monitor points

A monitor point is a location in your executable where performance data can be collected. Before profiling a program with Cxpa, you must first specify the areas of your program that you want to profile. You do this by enabling the monitor points at these regions.
A monitor point's type determines the kind of data it collects. There are four types of monitor points, corresponding to the four types of areas that you can profile:

Routine -Collects data for the entire routine, including CPU time and number of times called.
Loop - Collects data for the loop, including CPU time and iteration count.
Parallel region - Collects data for the parallel region, including CPU time, process virtual time and information for each thread executing in the region.
Block -Counts the number of times the block was executed.

Initially, all monitor points are disabled. A disabled monitor point is ignored by Cxpa during execution, and incurs virtually no everhead.
Before profiling the program, enable the monitor points at the regions you want to profile.
To enable all monitor points, select the Monitor All option from the Setup Menu.
Profiling the program

When you run your program under Cxpa, Cxpa collects profiling data at all enabled monitor points.
This data is collected by Cxpa and stored in a binary performance data file (.PDF file).
By default the PDF is named cxpa.pdf and is located in the directory from which you invoked Cxpa.
Using the X interface, you control the profiling of the program by using the Control Center dialog box. You can start, pause, continue, and stop profiling using the Control Center.
To open the Control Center dialog box, select the Run option from the Process menu. To begin program execution and performance data collection, click the start button.
After you press the start button, Cxpa opens an xterm window for the program's input and output.
Pausing profiling enables you to view performance reports based on the collected data, without having to wait for process completion.

When the process is finished executing, Cxpa stops data collection and terminates the process.

Analyzing performance data

Once profiling has been paused or completed, you can create performance reports. You create these reports from the data, or individual reports. Generating all reports ensures that you get all reports possible from the PDF. Generating a particular report is useful if you are only interested in a particular region (for example, if you only want to look at loop performance).

Reading performance reports

Cxpa can generate several reports. Cxpa only generates a report if profiling data has been collected for that type of report. The report are:

Routine Report
Loop report
Parallel Region Report
Basic Block Report

Routine Report

The Routine Report consists of the following tables:

Routine Performance Analysis table: -Consist of the following two sections:

Summary – Providing an overview of the performance of each routine including the number of times the routine was called and the total time spent in the routine, including called routines.
Details: - Lists the minimum, maximum, and average time spent for a single call to the routine.

Dynamic Call Graph – Displays an outline of the routine calls in your program. It can be used to trace the program's execution.

Loop Report

The Loop Report consists of a Loop Performance Analysis table for each executed routine that has monitored loops. Each Loop Performance Analysis table can contain three section:

Summary – Lists summary information such as the number of times each loop was executed, the iteration counts for each loop, and the CPU time spent in each loop, including time spent in inner loops
Details -Lists detailed information such as the resulting nesting level of each loop after optimization, the compiler optimization performed on each loop, and the CPU time spent in each loop.
Vectorized Loop - Lists information such as the number of vector spills, number of

vector floating point arithmetic instructions, chime count, and estimated number of millions of floating point operations per second.

These measurements can be evaluate the efficiency of a vectorized loop on that machine.

Parallel Region Report

The Parallel Region Report provides performance data for monitored parallel regions. There is one Parallel Region Performance Analyzis table for each executed routine with a monitored parallel region. The table is divided into two section:

Summary – Lists the total time spent in each parallel region. The total CPU time, wall-lock time, and process virtual time (PVT) is given. The process virtual time is defined as the total CPU time during which at least one thread is executing in a parallel region.
Cxpa also lists the concurrency factor (CPU time divided by PVT time) for the parallel region. The concurrency factor measures the efficiency of code executing in parallel.

Details -Lists the CPU time, wall-clock time, and chore count of each thread in each parallel region.
A chore is one unit of work executed by a thread in a parallel region, typically a single iteration of a parallelized loop. The chore count is the number of chores a particular thread executed in the region.

Basic Block Report

The Basic Block Report provides performance data for the basic blocks of each monitored routine. A basic block is a section of code that has a single entry point and a single exit point. This report consists of one Basic Block Performance Analysis table for each executed routine with monitored basic blocks.
The Basic Block Performance Analysis table provides the percentage of CPU time spent executing each basic block. This percentage is calculated by dividing total basic blocks executed by a basic block's times executed. The total CPU time of the routine is multiplied by this percentage to give the % CPU time for each basic block.

Using monitor points

Cxpa collects performance data at the enabled monitor points in your program. The more points you have enabled, the more performance data you collect, and the longer it takes for profiling to complete.

Before you run your program under Cxpa, you must specify the regions of your program that you want to profile.
When you compile your program for use with Cxpa, the compiler inserts special instructions in the executable around the routines, loops, blocks,, and parallel regions of your program. A region with these instructions is said to be instrumented.

A monitor point is the set of instructions used to monitor a particular region. The type of region that monitor point monitors determines its type. There are four types of monitor points:

Routine – Monitors CPU time spent inside of the routine. Also counts the number of times a routine is called.
Loop - Monitors CPU time spent inside of the loop, and the minimum, maximum, and average iteration count. Loop monitor points will not exiist in your program unless it was compiled at optimization level –O1 or above.
Parallel region - Monitors CPU time spent inside of the parallel region, including measurements for each thread executing inside the region.
Block – Monitors the number of times the basic block was executed

When listing monitor points with Cxpa, your source code is displayed with special annotations.

Monitor point annotations in source listings:

Type	Enabled	Disabled
Routine	R	r
Loop	L	l
Parallel region	P	p
Block	B	b

Table 2

Choosing the monitor points to enable

You can enable all of the monitor points in your program, or a subset. You do not have to recompile your program to enable or disable monitor points.
Typically, you will want to enable all monitor points the first time you profile your program. This provides a complete picture of your program's performance. Using this information, you can then identify the regions that take the longest to executable. After you have identified the regions of your program whose performance you want improve, you may want to enable the monitor points at just these regions.
If you want to profile only parallel regions, enable only parallel region monitor points. This provides a more accurate representation of the performance of parallel regions containing nested loops. If Cxpa profiles a parallel region and a nested loop within that region, the time spent monitoring the nested loop affects the parallel virtual time of the parallel region.
Enabling all monitor point types in all routines

It is good idea to enable all monitor points the first time you profile an application. This ensures that you will collect performance data at all available regions of your program.
To enable every monitor point in your program, use the Monitor All option from the Setup menu.
Enabling one monitor point type in all routines

To enable all monitor points of a particular type, use the corresponding Monitor option from the Setup menu.
Enabling all monitor point types in select routines

To enable monitor points in specific routines, you use the Set Monitor Points dialog box. Use the Monitor … option from Setup menu.
Listing monitor points

When enabling a subset of monitor points, it is helpful to get a list of the available monitor points in your program. Cxpa displays your original source code with special annotations indicating the location, type, and status of all the monitor points in your program.
To display the monitor points in a routine, select the Monitor List option under the Info menu. The Monitor Point Information dialog box appears.
Select the routine whose monitor points you want to display from the list of routines. When you select a routine, any monitor points in that routine are displayed, along with the source line they are monitoring, in the box to the right.

Generating and interpreting reports

Cxpa builds performance reports by analyzing the data in the performance data file (PDF). The reports provide performance measurements on the monitored regions of your program.
Cxpa produces a report for each type of region you can monitor.(routines, loops, parallel regions, and basic blocks).
Cxpa can generate a particular report if data for that region was collected during profiling. Cxpa only collects data at executed regions with enabled monitor points.
You can generate all reports available from the PDF or individual reports.
Reports are generated using the Analysis menu on the Cxpa window.
To generate all possible reports from the PDF, select the Analyze All option from the Analysis menu.

Printing reports

Once you generated the reports, you can print them to a file or printer. Reports can be printed to a printer or to an ASCII file.

To print the reports being displayed in the Cxpa window of the X interface, select the Print option from the File menu.

Routine report

The Routine Report gives performance measurements for every monitored routine that was executed, including library functions. The report can contain three tables:

Routine Performance Analysis table: - Display the total time in each routine, both with and without time spent in called routines. The table also displays timing

statistics for a single call to the routine and the number of times the routine was executed.

Dynamic Call Graph **-** Display the parents (calling routines) and children (called routines) for each routine. Routines are listed in the order they were executed.

Routine Performance Analysis table: -Consist of the following two sections:

Summary section **-** Provides an overview of routine performance for all calls.
Details section: - Provides data on a single call basis.

Summary section

The Summary section provides the following information for each monitored routine:
Total CPU time (both with and without time spent in called routines)
Number of times the routine was executed
Profiling status and additional notes about the routine.

The routines with the largest CPU times (not including children) are listed first. This helps to quickly identify the routines that take the greatest portion of the total execution time, often called the hot routines.
The Summary section is divided into four major columns for each routine of your program and any library function called:

CPU time - Lists the total CPU time and percentage of total execution time taken by the routine. The column is divided into two categories:
-(less children) - Does not include time spent in the children of the routine.
-(plus children) - Includes the total time spent in all the children of the routine.

Times Exec - Lists the number of times each routine was executed
PS - Lists the profiling status of the routines. Cxpa uses this column to note the state of a routine or its instrumentation.
Routine name - The name of each monitored routine that was called, listed in order of highest CPU Time (less children)

Detail section

A Details section lists the minimum, maximum, and average time spent in each monitored routine. The data indicates how execution time varies between different calls to a routine.

CPU time - Lists the minimum, maximum, and average length of time spent in the routine. The column is divided into two categories:
-(less children) - Does not include time spent in the children of the routine. Children are called routines.
-(plus children) - Includes the total time spent in all the children of the routine.

PS - Lists the profiling status of the routine. Cxpa uses this column to note the state of a routine or its instrumentation.
Routine name - The name of each monitored routine in your program (including monitored library functions).

Dynamic Call Graph table

The Dynamic Call Graph table lists the routines that were monitored in the order they were executed. Cxpa lists the parents and children for each routine, along with the number of times the routine was called.

If you would like to omit the Dynamic Call Graph in the Routine Report, invoke Cxpa with the -ncg option.

Loop Report

The Loop Report consists of a Loop Performance Analysis table for each executed routine that has monitored loops. Each Loop Performance Analysis table can contain three section:

Summary – Lists summary information such as the number of times each loop was executed, the iteration counts for each loop, and the CPU time spent in each loop, including time spent in inner loops
Details -Lists detailed information such as the resulting nesting level of each loop after optimization, the compiler optimization performed on each loop, and the CPU time spent in each loop. By comparing the loops in this table with those in the summary table, you can compare the loops actually executed to the original source code.
Vectorized Loop - provides a detailed account of the performance of vectorized loops (compiled at optimization level -O2 or -O3). These measurements can help you determine the efficiency of vectorized loops.
.

If the -nc option is used invoking Cxpa, then any loops of a routine that are monitored, but not executed, are listed in the Monitored Loops Not Executed table. During optimization, the compiler can significantly alter the order and flow of a loop to improve performance. In some cases, the compiler will interchange nested loops, separate complex loops into multiple smaller and simpler loops, or remove loops entirely.
Cxpa enables you to track the performance of the original loops, even after all of the modifications listed in the Details section.

Summary section

The Summary section details the execution time and iteration count of each monitored loop in each routine. Each routine identified by source file line number on which it begins.
The columns are following:

Line - Lists the source line number of the loop.

Times Exec - Lists the number of invocations of the loop.
Iteration Count - Lists the minimum, maximum, and average iteration counts.
CPU time (plus inner loops) - Lists the total CPU time for the loop, including time spent inner loops.
PS - Lists the profiling status and any additional notes for the loop.

Detail section

A Details section describes the loops actually executed. Due to optimizations this may differ from the loops in the original source code. Loops may have been interchanged, unrolled or even split into multiple loops due to distribution or dynamic selection.
Line - Lists the beginning line number of the loop. For compiler-generated loops (distributed loops), Cxpa places a letter next to the line number to distinguish multiple loops derived from a single source loop.
NL - Lists the resulting nesting level of the loop after optimization. 0 indicates an outer loop, with each inner loop receiving a higher nesting level.
Optimization Report - Lists the optimizations performed on the loop. The optimizations are abbreviated, and are listed in Table 3.
Times Exec - Lists the number of invocations of the loop.
Iteration Count - Lists the minimum, maximum, and average number of iterations for loop.
CPU time (less inner loops) - Lists the time spent executing the loop, not including inner loops.
PS - Lists the profiling status and any additional notes for the loop.

Mark	Transformation	Description of optimization performed by the compiler
D	Distributed	Split loop into multiple loops.
Ds	Dynamic section	generated multiple versions of the loop, with the best version chosen at run time when the trip count is known.
Hs	Hoisted	Moved a portion of the loop to the block before the loop.
I	Interchanged	Swapped an outer loop with an inner loop.
V	Fully vectorized	Vectorized the entire loop (the loop executes as a single vector instruction).
P	Parallel	Enabled the loop to execute in parallel.
PS	Parallel strip-mined	Enabled the loop to execute in parallel and strip-mined.
P/v	Parallel vectorized	Enabled the loop to be executed as vector instructions in parallel.
pV	Partially vectorized	Vectorized a part of the loop (but loop includes code that cannot be vectorized).
S	Scalar	Scalar optimization performed (as at -no -O0, and -O1).
SM	Strip-mined vector	Converted loop to iterate through vector instructions.
UL	Uninstrumentable	Loop was not instrumentable by Cxpa.
Ur	Unrolled	Replaced loop with a block of code for each iteration

Table 3

Vectorized loop section

The Vectorized loop section of the Loop Performance Analysis table summarizes the performance of vectorized loops in your program.

The vectorized loop section is only generated for executed monitored loops that include vector instructions. The Mflops data is only generated for loops containing vector floating point operations. If the vectorized loop section does not appear in the table, no vectorization was done by compiler for the loop or nest of loops.

The parts of the Vectorized loop section are briefly discussed below:
Line - Source line number of the loop.
NL -Nesting level of the loop.
Vector spill - Number of times a vector was written out of a register only to be read in again later.
Vector flop - number of arithmetic vector floating point instructions in the loop. This count does not include vector load or store operations.

Chime count - number of chimes occuring within a given loop. Chime is an acronym for chained vector instruction measurement. The greater the chime count, the more contention for available vector functional units, and therefore the less vector chaining.

Vector chaining
The CONVEX C series architecture enables certain vecor operations to be performed concurrently, or to be overlapped with other vector operations. this ability, known collectivitily as vector chaining, can enable vector operations to occur simultaneously, rather than one after another.
This concurrency offers significant performance improvement. This chime count (chained instruction measurement) for a loop is a theoretaical measure of the efficiency of vector chaining.

Vector chaining and functional unit

On a C Series machine, the vector processor enables the compiler to load 128 elements of an array into a vector register, and then operate on all 128 elements as a unit. Because of the speed at which vector register operations are performed, such vector operations are often assumed to be happening to all 128 elements simultaneously.
A vector functional unit operates on one element of the vector register at a time. A vector functional unit is an independent part of a CPU that performs a designated operation, such as addition or multiplication.
Beacuse of this, the first element from the result of a vector is ready before the second. wich is ready before the third, and so on.
Vector chaining is adventage of this by using are output or one functional unit as the input to the next functinal unit.
Consider the following tw vector instructions:

V2 = V1 + V0
V4 = V3 * V2

In this example, each element of V0 is added to the corresponding element of V1, and the result is stored into the corresponding element of V2. Then each element of V2 is multiplied with the corresponding element of V3, and the result is stored in V4. Without vector chaining, all of the elements of the first operation (the addition) wold have to be completed before the multiplication operation could begin.

With vector chining, The output of the addition operation can be fed directly into the multiplication operation. this is possible because the add operation and the multiply operation are calculated by two different functional units. This could be represented as:

V4 = V3 * (V1 + V0)

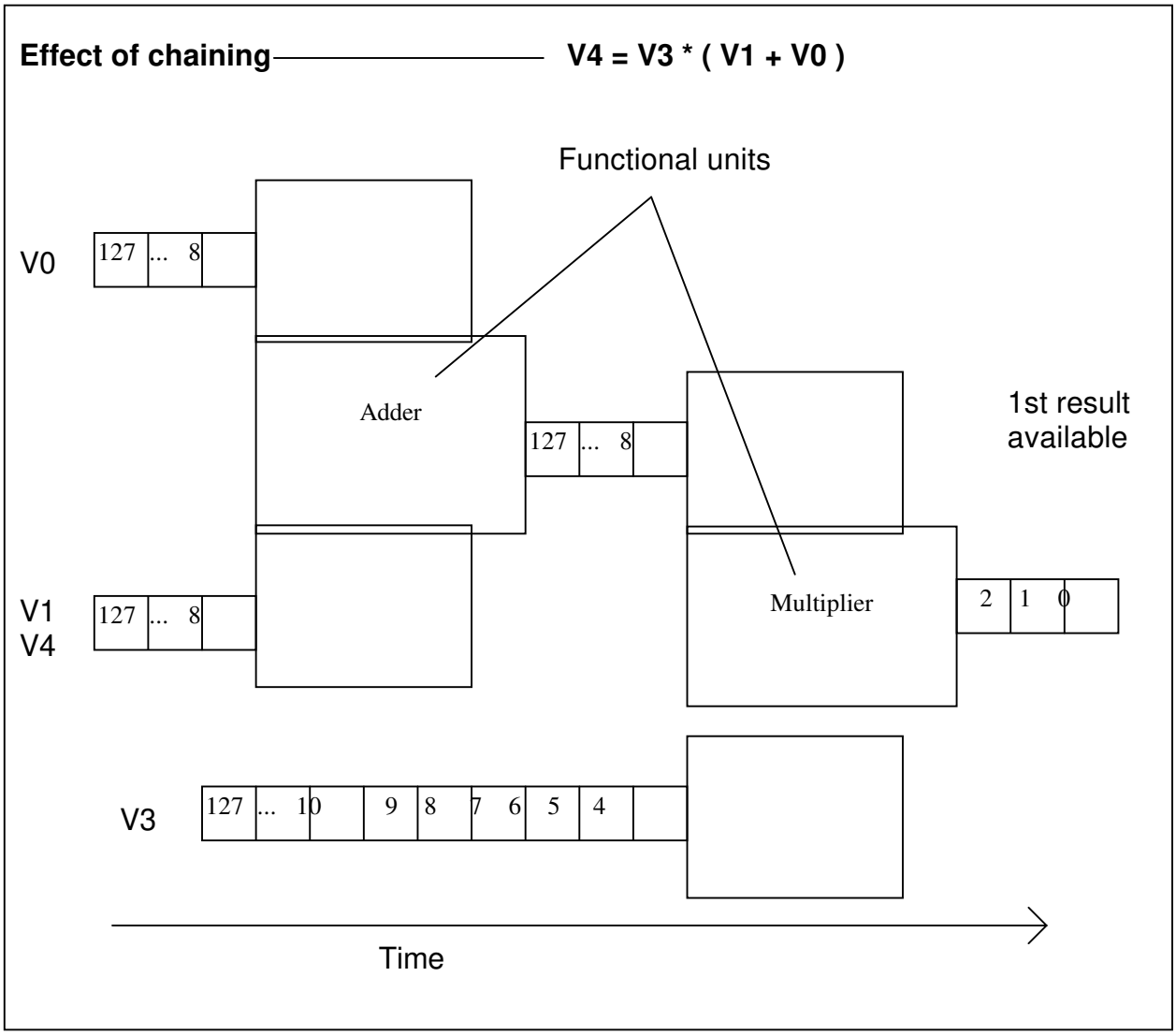


Table 1

V0 and V1 are both passed to the Adder functinal unit. As the results are output, they are fed directly into the Multiplier functional unit along with V3 as the results are generated, they are stored into V4.

If the values in V4 were to be stored into an array, this store could be chained with the previous two instructions. On the CONVEX C Series architecture, loading and storing of vector registers is handled by a third functional unit, called the Loader. The amount of vector chaining available depends on the underlying architecture.

Functional unit reservation

Vector chaining is only possible when a functional unit is available. CONVEX C Series machines have three vector functional units per CPU. Each functional unit has a specific set of vector instructions it is capable of processing.

When a vector instruction cannot be chained with the previous instruction because the appropriate functional unit is not available, functional unit reservation occurs, and chaining is prohibited.
On the C200 and C3200 architectures, the following vector instructions could not be chained:

V2 = V1 + V0
V4 = V3 * V2

For these architectures, the add operation can only be performed by the Adder functional unit. The first add operation occupies the Adder functional unit. The second addd operation must wait until the Adder is again available before beginning.
The abaility of the CPU to chain vector instructions together depends on what vector operations can be performed by each vector functional unit of the CPU.

C Series vector functional units:

	C200 and C3200 Vector func. units			**C3400 Vector func. units**			**C3800 Vector func. units**		
Generic operation	Adder	Multiplier	Loader	Adder	Multiplier	Loader	Adder	Multiplier	Loader
Add/substract	x			x	x		x	x	
Logical compare	x			x	x		x	x	
Bit count	x			x	x		x	x	
Shift	x			x	x		x	x	
Multiply		x			x			x	
Divide		x			x		x		
Type conversion		x			x		x	x	
Edit	x			x	x		x	x	
Load/store			x			x			x
Square root		x			x		x		

If an operation can be performed on two functional units, it is said to be redundant. For example, the add operation is redundant on C3400 and C3800 series CPUs.

Register bank reservation

Vector chaining can also be prevented when a vector instruction accesses a vector register that is not available. This is known as register reservation.

A register bank holds a pair of vector registers. The pairs are as follows:

V0 and V4
V1 and V5
V2 and V6
V3 and V7

At most, two reads and one write can be made to a register bank during a single clock cycle. Vector instructions cannot be chained together if this would cause more read and writes than possible to a single register bank.
On C3400 and C3800 Series CPUs, two reads to the same register bank count as a single read.
For example, the following two instructions cause register bank reservation on all CONVEX C Series architectures:

```
V0 = V2 + V1
V4 = V3  * V1
```

The first instruction requires one write to the V0/V4 register bank. The second instruction cannot be chained with the first because of the additional write to the V0/V4 register bank.

Chimes

A loop whose vector instructions are chained together executes more quickly because chaining enables multiple instructions to occur simultaneously.
A Chained Instruction Measurement, or chime, represents a vector instruction that could not be chained with the previous instructions, due to functional unit reservation or register bank reservation. Everey vectorized loop vill have a minimum of one chime.
Ideally, each vector instruction would be chained together, thus resulting in a single chime. However, this is seldom possible. The higher the chime count for a loop, the less vector chaining achieved. Therefore, the number of chimes in a loop, called the chime count, can be an affective performance indicator for vectorized loops.

Calculating chimes

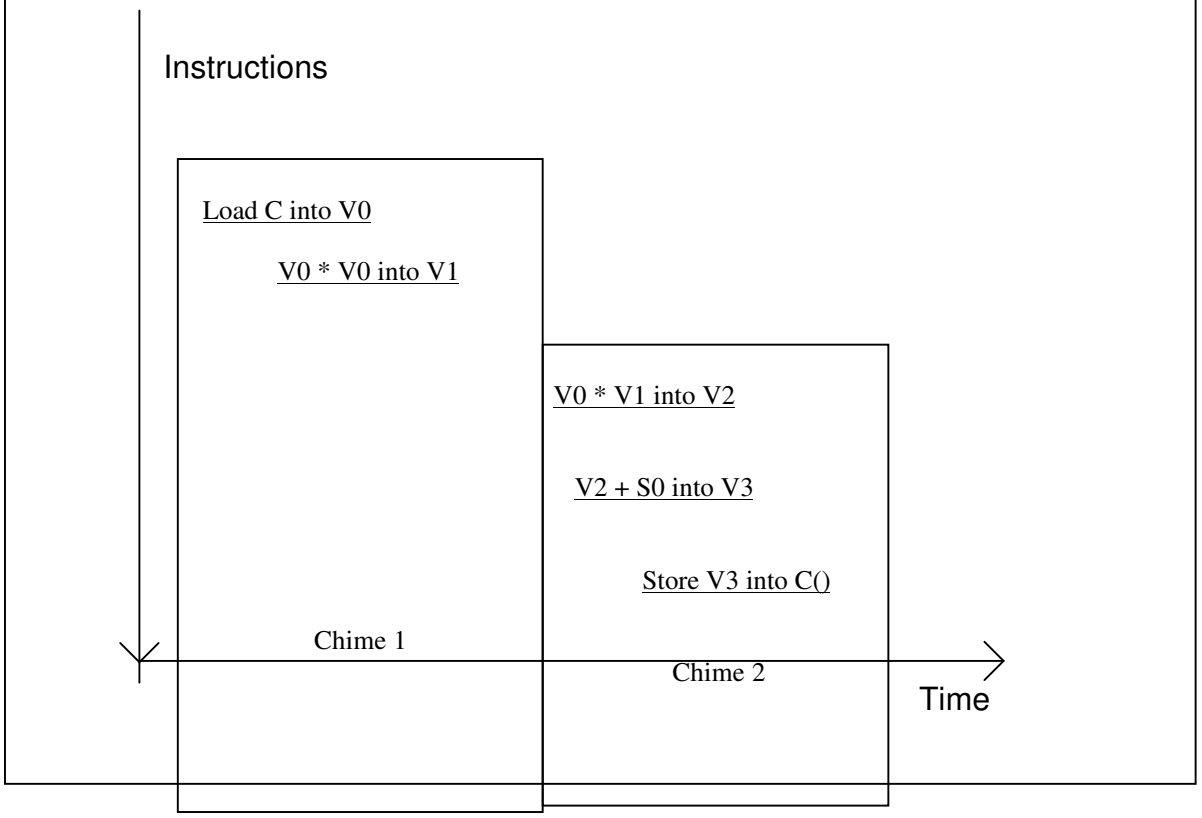
To demonstrate the calculattion of a loop's chime count, consider the following FORTRAN loop

```
DO  I = 1, 256
        C ( I )  =  C ( I ) * C ( I ) * C ( I ) + 3.0
ENDDO
```

Chime calculation on C3200 Series

Assembly code for DO loop

```
                    L20:  mov.w    s5,v1          ;
Load begins new           ld.w     0 (a2 ), v0    ;  new
chime                                               chime
Multiply chains with      mul.s    v0, v0, v1     ;
load
Functional unit           mul.s    v0, v1, v2     ;  new
reservation                                         chime
Add chains with           add.s    v2, s0, v3     ;
multiply
Store chains with         st.w     v3, 0 (a2 )    ;
add
                          add.w    #-128, s5      ;
                          add.w    #512, a2       ;
                          lt.w     #0, s5         ;
                          jbrs.t   L20            ;
```



The first multiply instruction mul.s v0, v0, v1
can chain with the load instruction. The second multiply cannot chain with the first, because the C3200 Series CPU has only one functional unit perform the multiply operation. The second multiply cannot begin until the first multiply completes and the multiply functional unit is again available.

Vector Mflops

Most scientific and engenieering applications use floating point numbers. Due to differences in hardvare and software, the performance of such code is bettet represented by millions of floating point operations performed per second (megaflops or Mflops) than by millions of instructions per second (Mips).
A floating point operation is simply an arithmetic instruction operating on one or more floating point numbers.
Cxpa distinguishes two types of floating point operations:
Scalar - Operating on a single floating point number. For example, taking the square root of 3.14.
Vector - Operating on a vector of floating point numbers. For example, subtracting from the first 128 elements of a array of real numbers.

Cxpa reports both the estimated vectorMflops and the peak vector Mflops. These values are reported for the loop with and without nested loops.

The estimated number of vector Mflops indicates how well the loop ran on that machine. The closer the estimated Mflops is to the peak, the greater the utilization of the vector functional units available on the machine.

Calculating estimated Mflops

The estimated flops for a loop is calculated as:

estimated_flops= floating_point_operation(flop) * trip_count / total_time(sec)

Returning to the previous example, you can calculate the estimated Mflops for the loop:

```
DO  I = 1, 256
        C ( I )  =  C ( I ) * C ( I ) * C ( I ) + 3.0
ENDDO
```

There are 256 iterations of the loop, so the trip_count is 256.
You can determine the number of vector floating point operations in the loop by looking at the assembly code.

Assembly code for DO loop

```
                    L20:  mov.w     s5,v1          ;
                          ld.w      0 (a2 ), v0    ;   new chime
Vector instruction 1      mul.s     v0, v0, v1     ;
Vector instruction 2      mul.s     v0, v1, v2     ;   new chime
Vector instruction 3      add.s     v2, s0, v3     ;
                          st.w      v3, 0 (a2 )    ;
                          add.w     #-128, s5      ;
                          add.w     #512, a2       ;
                          lt.w      #0, s5         ;
                          jbrs.t    L20            ;
```

Calculating peak Mflops

Peak Mflops represents the maximum floating point performance for a particular loop on a particular machine. To calculate the peak Mflops, use the following equation:
peak_flops= (floating_pt_ops/chime_count) * (1/ machine_clock_cycle(sec))

The clock rate of the machine rate per head is as follows:
C200 - 40 ns.
C3400- 40 ns
C3800- 16.67 ns

Parallel Region Report

The Parallel Region Report provides performance data for monitored parallel regions. There is one Parallel Region Performance Analysis table for each executed routine with a monitored parallel region. The table is divided into two section:

Summary section – Lists the total time spent in each parallel region. The total CPU time, wall-lock time, and process virtual time (PVT) is given. The process virtual time is defined as the total CPU time during which at least one thread is executing in a parallel region.
Cxpa also lists the concurrency factor (CPU time divided by PVT time) for the parallel region. The concurrency factor measures the efficiency of code executing in parallel.

Details section -Lists the CPU time, wall-clock time, and chore count of each thread in each parallel region. A chore count for a thread is the number of chores it performed inside the parallel region. Typically, a chore is a single iteration of a parallelized loop. The chore count is the number of chores a particular thread executed in the region.

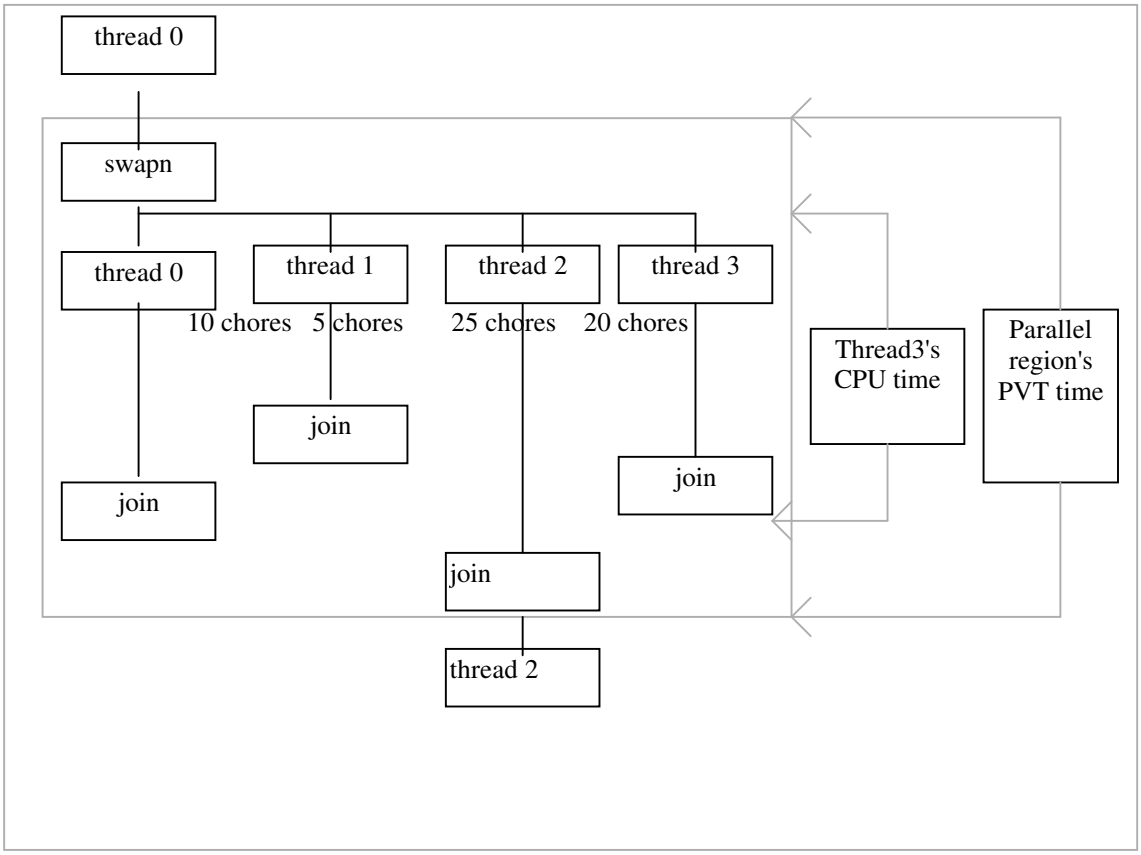
A thread is an independent sequence of instruction that are executed by a single CPU.

Parallel optimizations performed by the compiler at the -O3 level generate an executable that can be multithreaded. Each thread in a parallel region executes an identical set of instructions This type of parallel processing is called symmetric and typically occurs when the compiler parallelizes a loop.

Parallel regions

A loop that the compiler executes in parallel (typically around a nested vectorized loop) represents a set of chores. A chore is a single unit of work in a parallel region. Typically, a parallel region consists of a parallel loop, and each chore is a single iteration of that loop. In this case, the number chores for the region is equal to the iteration count of the loop.
On a machine with multiple processors, the chores of a parallel region are divided among multiple threads. The number of threads available is dependent upon fixed scheduling, the number of CPU's, and CPU availability.
The number of iterations of a parallel loop represents the total number of chores for a parallel region. The more evenly these chores are distributed across the threads executing in the parallel region, the faster the time to solution. An uneven distribution indicates the total CPU time for the region, and therefore total wall lock time is longer because one thread is taking longer than the others.
Chore distribution is controlled by the processors themselves, you cannot directly affect the number of chores a given thread executes. Cxpa reports this information to give you a better understanding of parallel region performance.

Parallel region with total chore count of 60



In the above example, the parallel region has a chore count of 60. At the beginning of the parallel region, three additional threads are spawned, and the chores are divided up among four threads.
In this example, the chores are not evenly distributed between the threads. (Ideally, you would hope for a distribution of 15 chores per thread.)
The total CPU time for the region is the sum of all the CPU times for all threads executing in the parallel region.
The process virtual time (PVT) represents the CPU time where one or more threads were executing. If all threads execute simultaneously, the PVT time will be very close to the CPU time of a single thread.

Profiling with fixed scheduling

Fixed scheduling ensures that all CPU's of a machine are made avalable for each timeslice of the process.
In a time-sharing environment, you may to enable fixed scheduling for two reasons:
To make the profile of parallel programs more independent from resource conflicts with other processes.
To eliminate register bank conflicts between single-threaded programs (non-parallel) and other running applications.
Fixed scheduling does ensure that multiple threads are created. A single thread may complete all of the chores of a parallel region before another CPU could spawn a thread.
To invoke Cxpa with fixed scheduling enabled, iclude the -f option on the Cxpa command line:

%cxpa -f a.out

Summary section

The Summary section provides performance measurements for an entire parallel region. It is a summary of the time spent by individual threads executing in the region. The section including:

Line number at which the parallel region begins
Number of times parallel region was executed
CPU time
Wall-lock time
Process virtual time (PVT)

Line - Lists the line number for the beginning of the parallel region. This is the original line number of the source code.
Time Exec - Lists the number of executions of the region.
CPU time - The total time spent in the parallel region.
Wall Clock Time - The total wall-clock time during execution of the parallel region.
Process Virtual Time - Total PVT for the parallel region. The PVT represents total time spent in user mode, notincluding time spent in kernel mode or waiting for I/O.
CPU/PVT - The concurrency factor for teh parallel region. The concurrency factor represents the degree to which the CPUs are being utilized. The closer the concurrency factor is the number of processors on the maschine, the better.

Detail section

The Details section describes the performance of the threads that executed each parallel region. Each line represents a single thread. For a two-processor maschine, each parallel region would have two lines describing thread performance for that region.

The columns of the Details section are described below:
Line - The starting line number for the parallel loop as it exists in the original source code.
CPU time -The total CPU time spent by each thread in the parallel region.
Wall Clock Time - The total wall-clock time spent by each thread in the parallel region.
Chore count - The total number of chores executed by the thread.
PS -The process state of the parallel region.

Parallel efficiency theory

There are two methods of calculating parallel efficiency. The first method can be used when you have profiled the program at optimization level -O3 and do not want to profile the program at a lower optimization level.
This method uses the concurrency factor (CPU time for the parallel region divided by the PVT). The concurrency factor is divided by the number of processors to get an approximation of the parallel efficiency.

Concurrency per processor method
The following equation demonstrates how to calculate the approximate pqarallel efficiency:

Concurrency factor/Number of processing * 100% = Parallel efficiency

Serial CPU time parallel PVT method

The second method requires you to profile the program without parallel optimization (at optimization level -O2 or lower). This compares the serial CPU time to the parallel PVT time to determine an approximation of the parallel efficiency.
The following formula can be used after you have compiled and profiled a parallel region (such as a loop) at optimization level -O2 or lower:

Serial CPU tim/Parallel PVT *100% = Parallel efficiency

Basic Block Report

If you compile your program with the -pab option, you can profile the basic blocks of your program.
The Basic Block Report provides performance data for the basic blocks of each monitored routine. A basic block is a section of code that has a single entry point and a single exit point. This report consists of one Basic Block Performance Analysis table for each executed routine with monitored basic blocks.
The Basic Block Performance Analysis table provides the percentage of total block executions spent executing each basic block.
The Basic Block Performance Analysis table is made up of two sections:
Summary - Lists the percentage of basic blocks in the routine that were executed at least once.
Detail - Lists the line number, PC Value, and number of times executed for each basic block.

The columns of the Details section are described below:
Line - Lists the starting line number for the basic block.
PC Value - Lists the starting address of the basic block
Times Exec - Lists the total number of times the basic block was executed
% of Total Exec - Lists the percentage of total block executions in this routine
PS - Lists the profiling status of routine.

Bibliography

1. CONVEX Architecture Reference Manual (C Series)
/Sixth Edition, April, 1992 – Convex Computer Corporation/
/Convex Press, Richardson, Texas United States of America – DHW-300/

1. CONVEX FORTRAN Optimization Guide
/Fourth Edition, November, 1992 – Convex Computer Corporation/
/Convex Press, Richardson, Texas United States of America – DSW-034/

CONVEX CXpa User's Guide 3rd edition March 1993
CONVEX Press
Richardson, Texas USA

CONVEX CXpa Reference 1st edition March 1993

Etelka Kovács - Ágnes Burián:
Architecture and Operation of Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94

Dr. Rezső Nagy - Etelka Kovács:
Digital Image Processing Exercises in the Spatial Domain on Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94

Magda Kovács - Ágnes Burián:
Digital Image Processing Exercises in the Fourier Domain using VECLIB on Vector Computer CONVEX 3820
TEMPUS Project S_JEP 8333-94