SPP Language and Programming Overview Course Notes

Order No. TRN-XXXX

Preliminary Edition
January 1995

CONVEX Education Center
Richardson, Texas
United States of America

SPP Language and Programming Overview Course Notes

Order No. TRN-XXXX

Printed in the United States of America

Automatic parallelism

9

Topics:

- -O3 optimization
- Inhibitors of parallelization
- Data dependence in loops
- Automatically parallelized loops
- Exercises

-O3 optimization

- Primary goal of -O3 optimizations is *parallelization*
  - Divides a program into threads
  - A *thread* is a sequence of instructions that execute on a single CPU

- Parallelism can exist at both the **loop** and **task** level
  - Compilers automatically exploit *loop* level in which all loops are examined:
    - Explicitly coded loops
    - Fortran 90 array expressions
  - Task level parallelism can be created by the user using the **BEGIN_TASK**, **NEXT_TASK** and **END_TASKS** directives discussed later

-O3 optimization - *cont.*

- Requirements for **automatic** loop parallelization:
  - Loop must **not** contain any **data dependencies**
  - Loop must have a known iteration count at runtime
  - Loop nest has sufficient parallel work to be done
  - Loop must not contain any I/O statements
  - Loop must not contain multiple entries or exits (includes STOP and RETURN statements)
  - Loop must not contain any procedure calls other than intrinsic functions
  - Loop must not contain potentially aliased scalar or array variables
- When a loop is automatically parallelized:
  - The loop is divided up into several smaller iteration spaces parceled out to be run simultaneously on all available processors
  - Compiler will try to parallelize the outermost loop
  - Compiler takes care of privatization of loop variables as needed, for example, the loop index is always privatized

-O3 optimization- *cont.*

- Compiler creates following types of parallelism:
  - thread-way (one-dimensional parallelism)
  - node-way (two-dimensional parallelism)
- *-or all* compiler flag shows the complete optimization report including the variables privatized

- Executable code generated will automatically run on as many processors as are available at runtime without recompilation
  - Normally all processors of the subcomplex
  - Smaller number of processors specified via:
    - **mpa(1)** utility
    - **LOOP_PARALLEL(max_threads=m)** and **PREFER_PARALLEL(max_threads=m)** compiler directives and pragmas which limit threads
    - **+min/+max** loader options

-O3 optimization- *cont.*

- Thread activity:
  - Shared memory program runs as a collection of threads on multiple processors
  - At program initiation, a separate thread of execution is started on each of the processors in the subcomplex
  - All threads spin for a default amount of time and then go idle except for thread 0 which runs all of the serial code of the program
  - When thread 0 encounters a parallel loop or task, it wakes-up or "spawns" the other threads signalling them to begin execution of the parallel code
  - The spawned threads then become active acquiring spawned thread IDs (1 to numprocs-1), run until their portion of the parallel code is finished or until they get time-sliced out, and then spin and go idle once again. Thread 0 also participates running its portion of the parallel code.
  - All spawned threads execute to completion of their spawned context before thread 0 continues

-O3 optimization - *cont.*

- **Thread-way parallelism** - Encountered by thread 0, it causes all threads available to the application to participate. Each of these threads is assigned a portion of the loop iteration space to execute.

  Consider the following loop:

  ```
  DO I = 1, 1024
     A(I) = B(I) + C(I)
  ENDDO
  ```

  - If run on 8 processors, each thread will execute:

    1024/8 = 128 iterations

  - If run on 128 processors, each thread will execute:

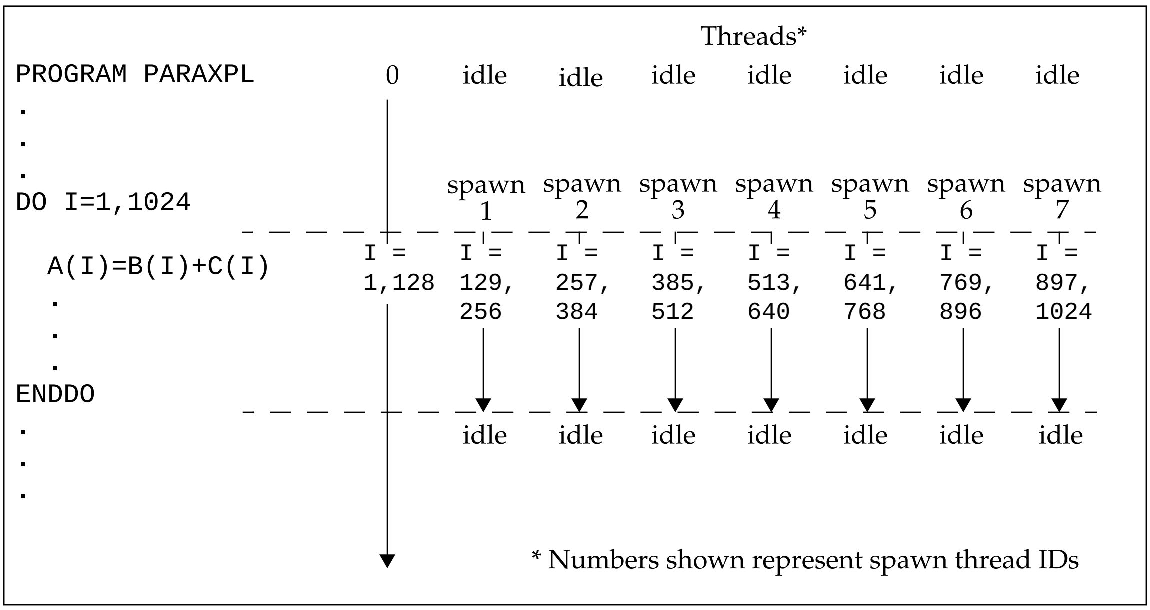
    1024/128 = 8 iterations

  - Compiler transforms the loop such that the starting and stopping iteration values for each thread are determined at runtime based on the number of available processors

  - If the iteration count is not evenly divisible by the number of threads, some threads perform fewer iterations than others

-O3 optimization - *cont*.

This figure shows **thread activity** and the **parceling of loop iterations** for the previous loop running on 8 processors.



-O3 optimization - *cont.*

This figure shows the loop parallelized running on 8 processors.

```
DO I = 1, 128
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 0

```
DO I = 129, 256
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 1

```
DO I = 257, 384
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 2

```
DO I = 385, 512
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 3

```
DO I = 513, 640
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 4

```
DO I = 641, 768
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 5

```
DO I = 769, 896
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 6

```
DO I = 897, 1024
  A(I) = B(I) + C(I)
ENDDO
```

THREAD 7

-O3 optimization - *cont.*

- **Node-way parallelism** - Encountered by thread 0, it causes 1 thread per hypernode available to the application to participate. Each of these threads is assigned a portion of the node-way loop iteration space to execute. However, node-way automatic parallelism will never be created, unless the compiler finds a thread-way parallel inner loop or an opportunity for thread-way parallelism via a function call.

  Thread-way loop parallelism encountered within a node-way parallel construct, causes each thread within the hypernode to be assigned a portion of the thread-way loop iteration space to execute.

  Consider the following loop:

```
DO J = 1, 1024
  DO I = 1, 1024
    A(I,J) = B(I,J) + C(I,J)
    .
    .
  ENDDO
ENDDO
```

  - Assuming no inhibitors and there is enough work, the compiler automatically parallelizes the J loop across hypernodes and the I loop across threads within those hypernodes.

-O3 optimization - *cont.*

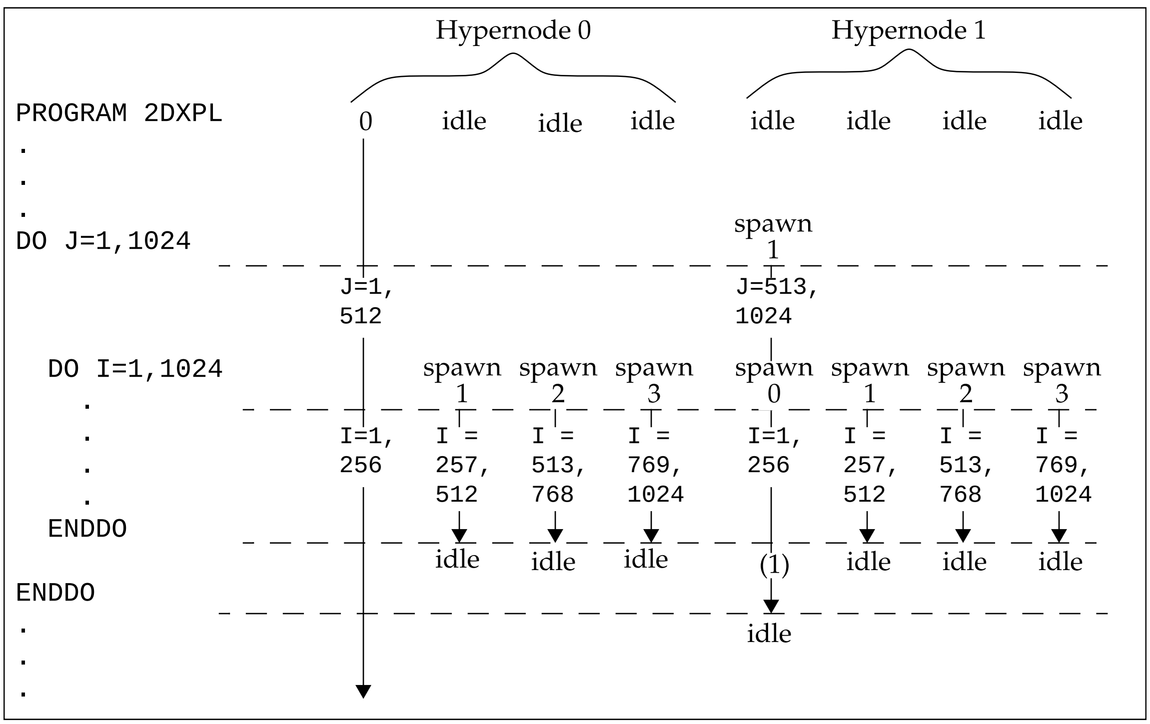
- If run on a 2 4-processor hypernode subcomplex:

  The node-way J loop spawns 2 threads, one on each hypernode. Each of these threads will execute:

  `1024/2 = 512` iterations of the J loop

  The I loop then spawns thread-way parallelism within each hypernode. Each of these threads will execute:

  `1024/4 = 256` iterations of the I loop



-O3 optimization - *cont.*

- Node-way parallelism can be disabled by specifying the *-nonodepar* compiler flag
  - Disables automatic and directive-specified node-way parallelism
  - Automatic and directive-specified thread-way parallelism still enabled
  - Eliminates node-way parallel overhead on a single node subcomplex
- Node-way and thread-way automatic parallelization can be disabled by specifying the *-noautopar* compiler flag:
  - Directive-specified node-way and thread-way parallelism still enabled through the usage of PREFER_PARALLEL, LOOP_PARALLEL, and BEGIN_TASKS compiler directives.
  - All other loops treated as if the NO_PARALLEL directive was specified for them

Inhibitors of parallelization

Most constructs that inhibit data localization also inhibit parallelization for the same reason. Specifically:

- Loop carried dependencies (LCDs). More categories of LCDs can inhibit parallelization than data localization:
    - Backward LCDs (B-LCD)
    - Forward LCDs (F-LCD)
    - Output LCDs (O-LCD)
    - Apparent LCDs (A-LCD)

  Examples of each will be given in the following **Data dependence in loops** section
- Potential for aliased scalar or array variables
- Multiple loop entries or exits (includes STOP and RETURN statements)
- Procedure calls other than intrinsic functions
- I/O statements

More inhibitors:

- Insufficient amount of parallel work to be done in loop
- Loop iteration count unknown at runtime

Data dependence in loops

- A loop-carried-dependence (LCD) results from an address being assigned a value in one loop iteration and the same address being assigned or referenced in another iteration.

  Example:

  ```
  DO I = 2,N
     A(I) = A(I-1) + B(I)
  ENDDO
  ```

  An example of a **Backward LCD (B-LCD)**:

  - The **A(I-1)** reference on iterations 3 through **N** was assigned on the previous iteration as **A(I)**
  - Each iteration must execute to completion before the next can begin. Therefore it is fruitless to assign parallel threads of execution to compute different iterations
  - B-LCDs cannot be automatically parallelized

Data dependence in loops - *cont.*

Example:

```
DO I = 2,N
   A(I) = A(I+1) + B(I)
ENDDO
```

An example of a **Forward LCD (F-LCD)**:

- The **A(I+1)** referenced on iterations 2 through **N-1** is assigned by the following iteration
- If parallel threads of execution attempt to execute different iterations of the loop, it is quite possible, for example, that **A(3)** might be assigned in iteration 3 before **A(3)** is referenced by iteration 2
- F-LCDs cannot be automatically parallelized
- The example can be automatically parallelized by making an extra copy of array **A**:

```
DO I = 2,N
   OLDA(I+1) = A(I+1)
ENDDO

DO I = 2,N
   A(I) = OLDA(I+1) + B(I)
ENDDO
```

Creating the 2nd loop clearly adds overhead. Therefore, it must be used with care.

Data dependence in loops - *cont.*

Example:

```
DO I = 2,N
   A(J(I)) = B(I)
ENDDO
```

An example of a *potential* **Output LCD (O-LCD)**:

- If **J(I)** contains repeated values (i.e., **J(3)** = **J(7)** = 4), then 2 different iterations are attempting to assign a value to the same address
- If parallel threads are executing the iterations, then the values *output* by the loop into array **A** depend on the order in which the iterations are executed
- The compiler will not automatically parallelize such loops

Data dependence in loops - *cont.*

Example:

```
DO I = 1,N
   A(I) = A(J(I)) + 1.0
ENDDO
```

This is an example of an **Apparent LCD (A-LCD)**:

- Since the value assigned to **A(I)** in one iteration *might* be used in a later iteration, the compiler lacks sufficient information to determine whether an LCD exists or not
- Rather than risk wrong answers, the compiler will not automatically parallelize such loops

Summary: the compiler does not automatically parallelize loops containing actual or apparent array based LCDs.

Automatically parallelized loops

Following are examples of loops that are automatically parallelized by the compiler at -O3. All examples assume that there is no aliasing among the arrays.

```
SUBROUTINE MYSUB(A,B,C,N)

REAL*4 A(N,N), B(N,N), C(N,N)

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

RETURN

END
```

The above nested loop will automatically parallelize, since there are no LCDs.

Automatically parallelized loops - *cont.*

The compiler can handle some *scalar* LCDs.

```
SUBROUTINE MYSUB(A,B,Y,N)

REAL*4 A(N), B(N), Y(N), X(N), S

J = 5
DO I = 1, N
   S = A(I) * B(I)
   J = J + 1
   X(I) = S * Y(J)
ENDDO

RETURN

END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **S** (O-LCD)
- The compiler avoids this problem by providing a private version of **S** for each thread
- If the value of **S** is needed after the loop, the processor executing the **N**th iteration stores its private value in **S**
- **J,** a loop induction variable like **I**, has the same problem as **S** (O-LCD) plus is a B-LCD. A private version of **J** is provided to each thread and its value is determined as a function of **I**:
  (**J** = **Jinit** + **I**, where **Jinit** = **J** at loop invocation)

Auto parallel

Automatically parallelized loops - *cont.*

The compiler can handle some *scalar* LCDs known as *reductions*. Generally, a reduction has the form:

X = **X** *operator* **Y**

where:

X - variable not assigned or used elsewhere in the loop
Y - a loop constant expression not involving **X**
operator is +, -, *, .AND., .OR., .EQV. or .NEQV.

```
SUBROUTINE MYSUB(A,B,C,D,E,N,SUM)

REAL*8 A(N),B(N),C(N),D(N),E(N),SUM
INTEGER*4 N

DO I = 1, N
   A(I) = E(I) / C(I)
   SUM = SUM + A(I) * B(I)  !reduction
   D(I) = A(I) / B(I)
ENDDO

RETURN
END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **SUM** (O-LCD)
- The compiler avoids this problem by providing a private version of **SUM** for each thread in which each thread's partial sum is computed. The private **SUM**s are added to compute final **SUM** by the compiler.

Automatically parallelized loops - *cont.*

The compiler also recognizes scalar reductions of the form:

X = *function* **(X, Y)**

where:

X - variable not assigned or used elsewhere in the loop
Y - a loop constant expression not involving **X**
function is the intrinsic MAX or MIN function

```
SUBROUTINE MYSUB(A,B,C,D,E,N)

REAL*8 A(N),B(N),C(N),D(N),E(N)
REAL*8 MAX, X
INTEGER*4 N

X = 0.0

DO I = 1, N
   A(I) = E(I) / C(I)
   X = MAX(X,A(I))       !reduction
   D(I) = A(I) / B(I)
ENDDO
RETURN
END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **X** (O-LCD)
- The compiler provides a private version of **X** for each thread. Thread 0 uses private **X**s to set original **X**.

Automatically parallelized loops - *cont.*

Even entire arrays can be privatized as needed for the array **Q** in order to parallelize the **J** loop in the example below:

```
SUBROUTINE MYSUB(U,M)

PARAMETER (N = 99)

REAL*4 P(N), Q(N), R(N), U(M), T(N,M)

DO J = 1, M
   DO I = 1, N
      Q(I) = P(I) * R(I)
      T(I,J) = Q(I) * U(J)
   ENDDO
ENDDO

RETURN

END
```

- If parallel threads execute different iterations of **J**, they can overwrite each other's values of **Q(I)** (O-LCD)
- The compiler avoids this problem by providing a private version of the **Q** array for each thread, since it knows **Q**'s size at compile time
- Parallelized, each thread sets a unique subarray of **T**
- Compiler also provides a private version of **I** for each thread

Automatically parallelized loops - *cont.*

```
SUBROUTINE MYSUB

PARAMETER (M = 50, N = 10)

REAL*4 X(100,100), Y(100,100)

Y(2:M+1:2, 2:N+1) = 1.0
X(1:M:2, 1:N) = Y(2:M+1:2, 2:N+1)

RETURN

END
```

The above Fortran 90 array assignments will automatically parallelize, creating 2 parallel loops, since by definition there are no dependencies.

Exercises

[1] Can the compiler automatically parallelize any of these loops?

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

```
(b) DO I = 1, N
       IF (A(I) .LT. 0.0) THEN
          J = J + 1
          A(I) = B(J)
       ENDIF
    ENDDO
```

```
(c) DO WHILE (A(I) .LT. Z)
       I = I + 1
       A(I) = B(I)
    ENDDO
```

```
(d) DO WHILE (I .LT. N)
       I = I + 1
       A(I) = B(I)
    ENDDO
```

```
(e) DO I = 1, N
       S = C(I) * B(I)
       IF (S .GT. 0.D0) THEN
          T = S + 5.D0
       ELSE
          T = S + 4.D0
       ENDIF
       A(I) = A(I) + T
    ENDDO
```

```
(f) DO I = 1, N
       A(J(I)) = A(K(I)) + 1
    ENDDO
```