INTRODUCTION TO CODE OPTIMIZATION AND PARALLELIZATION TECHNIQUES

I.T.U.
Institute of Informatics
Computational Science and Engineering Program
Dr. Serdar ÇELEBİ
STEPS FOR PARALLELIZING A PROGRAM
Sequential Optimization Techniques

Before we analyze the steps for parallelizing a program, let’s focus on optimization techniques applicable to the serial programs.

Here, most of the optimization techniques are loop oriented.

In certain cases, different techniques can be used to achieve the same effect.

Code specific details and the underlying processor/system architecture may determine which one is best.
Array Indexing

- Compilers analyze the data dependencies in a loop to find opportunities for transformations
- Expressing dependencies in a clear way, will help compilers to do a good job

DO J=1, N
DO I=1, M
... A(I,J) ...

DO J=1, N
DO I=1, M
k = k + 1
... A(k) ...

DO J=1, N
DO I=1, M
... A(INDX(I,J)) ...

DO J=1, N
DO I=1, M
... A((I+(J-1)*M) ...

DO J=1, N
DO I=1, M
... A(INDX(I,J)) ...

DO J=1, N
DO I=1, M
... A(I+(J-1)*M) ...

DO J=1, N
DO I=1, M
... A(I,J) ...
Loop Interchange: Example

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

If all arrays fit in the cache, it doesn’t matter how they are accessed (in general).

- If \( N \) is large, only one element per cache line will be used.
- Loop interchange will solve both problems.
Loop Interchange: Example

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

- It is common to have cache misses, but they are now minimal
- This is the best one can do for this problem
- Recall that the storage order is language dependent
Loop Interchange

- It can be very beneficial for performance to interchange the order of loops

- Benefits:
  - Improved memory access
  - Eliminate data dependency in innerloop to generate more efficient instruction scheduling
Loop Unrolling: Example 1

\[
\text{DO } I = 1, N \\
\quad Y(I) = X(I)^{**2} + B \times X(I) + C \\
\text{END DO}
\]

Ax=&X(1); Ay=&Y(1)
d = d – 8
start_loop:
d = d + 8
\begin{align*}
\text{r1 = load@ (Ax+d)} \\
\text{r2 = r1 * r1} & \quad \text{r3 = B * r1} \\
\text{r4 = r2 + r3} & \quad \text{r5 = r4 + C} \\
\text{store r5=Y(I)@(Ay+d)} \\
\text{I < N ?} \\
\text{I = I + 1} \\
\text{if not done: GO TO start_loop}
\end{align*}

1 int add
2 load
2 fp muls
2 fp adds
1 store
1 compare
1 int incr
1 branch

- These 10 instructions (6 int and 4 fp) use 5 fp registers and can be executed in 3 cycles at best
- The execution speed is therefore 3 cycles/iteration
- Loop unrolling can improve this by better using the instruction
Loop Unrolling: Example 1

DO $I = 1, N, 3$
  $Y(I) = X(I) \times X(I) + B \times X(I) + C$
  $Y(I+1) = X(I+1) \times X(I+1) + B \times X(I+1) + C$
  $Y(I+2) = X(I+2) \times X(I+2) + B \times X(I+2) + C$
END DO

Loop unroll
Factor is 3
(cleanup loop omitted)

Start_loop:
3 int adds  $d = d + 24$  $d1 = Ax + d$  $d2 = Ay + d$
3 loads  $r1 = ld@d1$  $r2 = load@(d1+8)$  $r3 = load@(d1+16)$
6 fp muls  $r4 = r1 \times r1$  $r5 = r2 \times r2$  $r6 = r3 \times r3$
  $r7 = B \times r1$  $r8 = B \times r2$  $r9 = B \times r3$
6 fp adds  $r10 = r4 + r7$  $r11 = r5 + r8$  $r12 = r6 + r9$
  $r13 = r10 + C$  $r14 = r11 + C$  $r15 = r12 + C$
3 stores  $st r13@d2$  $st r14@(d2+8)$  $st r15@(d2+16)$
1 compare  $I < N$?
1 int incr  $I = I + 3$
1 branch  If not done : GO TO start_loop
• These 24 instructions (12 int + 12 fp) use 15 fp registers and can be
  executed in 6 cycles at best on the UltraSPARC processor
• The execution speed is improved to 2 cycles/iteration (1.5 times faster)
Loop Unrolling: Example 2

This loop unrolled by 4:

DO I = 1, M, 4
    DO J = 1, N
        A(I) = A(I) + B(I,J)*C(J)
        A(I+1) = A(I+1) + B(I+1,J)*C(J)
        A(I+2) = A(I+2) + B(I+2,J)*C(J)
        A(I+3) = A(I+3) + B(I+3,J)*C(J)
    END DO
END DO

• Loop interchange would give Stride 1, but 3 memory references
• Unroll length should match Cache line size
• Note the re-usage of C(J)
Loop Unrolling

- With loop unrolling, more than one iteration of the loop gets executed within the body

- Benefits:
  - Get better instruction slot usage
  - Improved memory access
  - Increased data re-usage

- Disadvantages:
  - Clean-up code needed
  - More register needed
Loop Unrolling - Structure

Loop unroll factor is “unroll”

DO I = 1, N, 1
  ...(I)...
END DO

DO I = 1, N-mod(N,unroll), unroll
  ...  (I)  ...
  ...  (I+1)  ...
  ...
  ...
  ... (I+unroll-1)  ...
END DO

Cleanup loop

DO I = N-mod(N,unroll)+1, N, 1
  ...(I)...
END DO
Loop Unrolling - Comments

- In reality, the loop may be unrolled deeper than seems to be necessary
- This is because instructions have a certain latency (cycles to get the result) and this puts another constraint on the schedule
- This mechanism that deals with these latencies is called *modulo scheduling*
- Unrolling increases the need for registers and this is often the limiting factor
Loop Blocking – Example

- Consider $A = B \times C$, for square $N \times N$ matrices.
- Floating point: $2 \times N^2 \times 3$ (opportunity for blocking).
- Memory: $3 \times N \times N$.
- Matrix multiply can be written as:
  - Innerproduct – 2 memory operations, with non-unit stride on one vector.
  - Vector Update $a() = a() + c \times b()$ – unit stride, but 3 memory operations.
Loop Blocking – Example

- Recall that matrix multiply can be written as the product of smaller matrices.
- We will use this to block the algorithm.

\[
\begin{array}{ccc}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{array}
\begin{array}{ccc}
B_{11} & B_{12} & B_{13} \\
B_{21} & B_{22} & B_{23} \\
B_{31} & B_{32} & B_{33}
\end{array}
= \begin{array}{ccc}
C_{11} & C_{12} & C_{13} \\
C_{21} & C_{22} & C_{23} \\
C_{31} & C_{32} & C_{33}
\end{array}
\]

- For a sufficiently small blocksize, \(B_{11}, B_{12}\) and \(B_{13}\) can be kept in the cache and are re-used when calculating \(A_{12}\) and \(A_{13}\).
- Note that in general, one also needs to block for \(A\) and \(C\).
Loop Blocking

- The most important benefit from blocking is to improve temporal locality (re-use of data)
- Rule of thumb: consider blocking if more arithmetic than memory operations
- The blocking size chosen depends on the:
  - size of cache
  - Total storage needed in the blocked loop
- It is good practice to keep about 10% available for other data
Loop Fission - Example

Long latency operation
Bad memory access

Can be scheduled more efficiently or be replaced by a vector operation

Optimal memory access

```
DO I=1, N
    A(I)=EXP(-L*B(I))
    DO J=1, N
        C(I,J) = C(I,J) + A(I)*D(J)
    END DO
END DO
```
Loop Fusion - Example

Do I = 1, N
   A(I) = 2 * B(I)
End Do
Do I = 1, N
   C(I) = A(I) + D(I)
End Do

For a large value of N, only the last part of array A() will be in the cache when the loop is finished

Array A() needs to be reloaded again

In this case, the gain is multi-fold:
• No reloads of A needed
• Perform a multiply and add in parallel
• Less loop overhead
Loop Fission and Fusion

Fission: split one loop into multiple loops
Fusion: merge multiple loops into one

**Fission:**
- Reduce register pressure
- Enable loop interchange
- Isolate dependencies
- Increase optimization opportunities e.g. “vectorization”

**Fusion:**
- Minimize cache reloads
- Improve data re-usage
- Reduce loop overhead
Procedure Inlining

- A function or subroutine is indispensable for modular programming
- Unfortunately, this may introduce a performance bottleneck:
  - The calling overhead for small modules can be relatively costly (mostly a minor issue)
  - Calls to a function/subroutine are an optimization bottleneck, as the compiler will not know the side-effects
- Inlining replaces the call by the corresponding function/subroutine
- This solves the problem, but will increase the size of the executable
Array Padding

- As we have seen, data may conflict in the cache if memory addresses are the same, modulo the cache size and the capacity of the cache is not sufficient to store all data.
- By changing the memory addresses, this can be solved.
- The Operating System may do this for you through a technique called page/cache colouring.
- Through padding, compilers can avoid trashing.
- Also, mapping on physical addresses is less likely to cause such conflicts.
- Still, it may happen and array padding is a technique to solve it.
Array Padding - Example

```
COMMON /BAD/A(1024),B(1024)
DO I = 1, N
   SUM=SUM + A(I)*B(I)
END DO
```

- The cache line with B(I) will overwrite the cache line of A(I)
- Therefore A(I+1) will have to be reloaded, but then also B(I+1), etc.
Array Padding - Example

```plaintext
COMMON /BETTER/A(1024),PAD(16),B(1024)
DO I = 1, N
    SUM = SUM + A(I) * B(I)
END DO
```

- Now, the cache line with A(I) and B(I) are one cache line size apart and no longer conflict
- The program will require 64 bytes more virtual memory to run
Array Padding

- The previous example is rather straightforward
- The problem can also show up in a more hidden way

```
REAL A(1024,1024)
DO J = 1, 1024
  DO I = 1, 1024
    SUM = SUM + A(I,J) * A(I,J+1)
  END DO
END DO
```

- Column J and J+1 are 1024 elements apart and will conflict in our 4 KB cache
- Changing the leading dimension of A to 1040 will solve the problem
- The program will need more memory to run though
- Note that these conflicts will only take place under “special” circumstances
De-vectorization

- Cache memory and cache bandwidth are scarce resources.
- Compilers know this, but sometimes they have no choice than to store data that need not be stored ...
- This will unnecessarily cost space in the cache and a store slot in the instruction sequence.
- The programmer may know the data is only scratch data and need not be stored.
- For the compiler this may be very hard to determine.
- Also the way it has been coded could influence the compiler optimizations in this respect.
Because TMP is re-written in the second loop, it does not have to be stored in the first loop.

But how about the last loop in which TMP gets modified?

Because TMP is in a COMMON block, it may be needed elsewhere.

Therefore the compiler has to store it back!

Only YOU will know it doesn’t have to do this.

Dr. Serdar ÇELEBİ
De-vectorization - Example

In this modified loop, we keep the last values of TMP because they are needed later on in the same subroutine.

If array TMP is not used any further, we can replace it by a scalar variable TMPL. The compiler will try to keep this variable in a register.
SUMMARY

- Powerful optimization techniques are available
- Most, but not all, are memory oriented
- Compilers can do quite a bit today
- However, they are limited by:
  - Lack of knowledge of the application
  - Existing data structures
- It is up to you to write code such that the compiler can find Optimization opportunities
- Better may even be to do it yourself and structure your code appropriately
- Leave the low level details to the compiler
Introduction into Parallelization
What is Parallelization?

- Parallelization is simply another optimization technique to get your results sooner.
- To this end, more than one processor is used to solve the problem.
- The “elapsed time” (also called wallclock time) will come down, but the total CPU time will probably go up, because:
  - The newly introduced parallel portions in your program need to be executed.
  - Processors need time talking to each other and synchronizing (“communication”).
Typically it also gets worse when increasing the number of processors.

**Efficient** parallelization minimizes this overhead.
Data Dependency Analysis
Watch this logic

Parallelism

Independence

No Fixed Ordering

No Notion of Time

“Something” that does not obey this rule, is not parallel (at that level...)
Example #1

Loop is independent from the other iterations

DO I=1, N
    A(I) = A(I) + B(I)
END DO

<table>
<thead>
<tr>
<th>Proc</th>
<th>T = 1</th>
<th>T = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(1)=A(1)+B(2)</td>
<td>A(5)=A(5)+B(5)</td>
</tr>
<tr>
<td>2</td>
<td>A(2)=A(2)+B(2)</td>
<td>A(8)=A(8)+B(8)</td>
</tr>
<tr>
<td>3</td>
<td>A(3)=A(3)+B(3)</td>
<td>A(12)=A(12)+B(12)</td>
</tr>
<tr>
<td>4</td>
<td>A(4)=A(4)+B(4)</td>
<td>A(7)=A(7)+B(7)</td>
</tr>
</tbody>
</table>

Time

Dr. Serdar ÇELEBİ
Example #2

DO $I=1$, $N$
\[ A(I) = A(I) + B(I+1) \]
END DO

Arrays overlap

<table>
<thead>
<tr>
<th>Proc</th>
<th>$T = 1$</th>
<th>$T = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A(1)=A(1)+B(2)$</td>
<td>$A(5)=A(5)+B(6)$</td>
</tr>
<tr>
<td>2</td>
<td>$A(2)=A(2)+B(3)$</td>
<td>$A(8)=A(8)+B(9)$</td>
</tr>
<tr>
<td>3</td>
<td>$A(3)=A(3)+B(4)$</td>
<td>$A(12)=A(12)+B(13)$</td>
</tr>
<tr>
<td>4</td>
<td>$A(4)=A(4)+B(5)$</td>
<td>$A(7)=A(7)+B(8)$</td>
</tr>
</tbody>
</table>

Proc

Time
Example #3

It can not be run in parallel

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

<table>
<thead>
<tr>
<th>Proc</th>
<th>T = 1</th>
<th>T = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(1) = A(2) + B(1)</td>
<td>A(5) = A(6) + B(6)</td>
</tr>
<tr>
<td>2</td>
<td>A(2) = A(3) + B(2)</td>
<td>A(8) = A(9) + B(8)</td>
</tr>
<tr>
<td>3</td>
<td>A(3) = A(4) + B(3)</td>
<td>A(12) = A(13) + B(12)</td>
</tr>
<tr>
<td>4</td>
<td>A(4) = A(5) + B(4)</td>
<td>A(7) = A(8) + B(7)</td>
</tr>
</tbody>
</table>

Time
Example #3 – Let’s run it!

Values of A(:) are calculated in parallel

We print \( \text{SUM} (A(1:N+1)) \)

The same program was run 4 times on 1, 8, 32 and 64 processors

With the exception of the first test, the results are:

- Wrong
- Inconsistent
- NOT Reproducible

This is called a **Race Condition**

<table>
<thead>
<tr>
<th>Results for P=1</th>
<th>12512501.0</th>
<th>12512501.0</th>
<th>12512501.0</th>
<th>12512501.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results for P=8</td>
<td>12512508.0</td>
<td>12512508.0</td>
<td>12512508.0</td>
<td>12512508.0</td>
</tr>
<tr>
<td>Results for P=32</td>
<td>12512526.0</td>
<td>12512530.0</td>
<td>12512528.0</td>
<td>12512527.0</td>
</tr>
<tr>
<td>Results for P=64</td>
<td>12512548.0</td>
<td>12512545.0</td>
<td>12512549.0</td>
<td>12512547.0</td>
</tr>
</tbody>
</table>
Race Condition

- Loosely described, this means that the update of a shared variable is not well protected.

- It typically shows up in a very nasty way:
  - Numerical results are different from run to run
  - Difficult to tell this is a numeric effect or a bug
  - Changing the number of processors can cause the problem to seemingly (dis) appear
The real Problem

Run in parallel

```
DO I=2, N
    A(I) = A(I+M) + B(I)
END DO
```

M = -1: Not Parallel
M = N: Parallel

Now, what if you were a compiler?
The Pattern

- As long as the data (section) that we read is independent of what we write, all is well:

  ```fortran
  DO I=1, N
    A(I) = A(I) + B(I)
  END DO
  ```

- Care needs to be taken, if there is (a potential of) overlap of data being read and written:

  ```fortran
  DO I=2, N
    A(I) = 2 * A(I - 1) + B(I)
  END DO
  M = ?
  DO I=2, N
    A(I) = 2 * A(I + M) + B(I)
  END DO
  ```
The Analysis

DO I=3, 10
  A(I) = A(I-2) + B(I)
END DO

 Compiler view:
 R: TMP(I) = A(I-2)+B(I)
 W: A(I) = TMP(I)

I = 3: Read A(1) Write A(3)
I = 5: Read A(3) Write A(5)

• If we would execute I=3 after I=5, the result for I=5 would be wrong
• Remember, that parallel means “no notion of ordering”
• Therefore we have to assume this happens
Is this parallel?

DO I=1, 100
  C(I) = A(3*I+1) + 1
  A(2*I+7) = B(I) - 3
END DO

We could try to solve the problem in this way

I = 1 : C( 1) = A( 4) + 1     A( 9) = B( 1) - 3
I = 2 : C( 2) = A( 7) + 1     A(11) = B( 2) - 3
I = 3 : C( 3) = A(10) + 1     A(13) = B( 3) - 3
I = 4 : C( 4) = A(13) + 1     A(15) = B( 4) - 3
I = 5 : C( 5) = A(16) + 1     A(17) = B( 5) - 3
I = 6 : C( 6) = A(19) + 1     A(19) = B( 6) - 3
I = 7 : C( 7) = A(22) + 1     A(21) = B( 7) - 3
I = 8 : C( 8) = A(25) + 1     A(23) = B( 8) - 3
I = 9 : C( 9) = A(28) + 1     A(25) = B( 9) - 3
I =10 : C(10) = A(31) + 1     A(27) = B(10) - 3
I =11 : C(11) = A(34) + 1     A(29) = B(11) - 3
I =12 : C(12) = A(37) + 1     A(31) = B(12) - 3
Or like this ...

\[ W(x) = R(y) \]
\[ 2x + 7 = 3y + 1 \]
\[ x = \frac{(3y - 6)}{2} \]

For \( x, y \) in \([1, 100]\)

Solution needs to be integer too!

I = 3 and I = 4
I = 6 and I = 6
I = 8 and I = 9
I = 10 and I = 12

... ETC ...

Note the cross-over point for I = 6
About the analysis

- Compilers will try to solve equations of the type we’ve just seen.

- If they fail to prove there is no data dependency, they have to take the safe way and not parallelize.

- A well-know problem is indirect addressing:

```fortran
DO I=1, N
   A(INDX(I)) = A(INDX(I)) + B(I)
END DO
```

- If INDX() is a permutation of the indices only, this operation is parallel, otherwise it isn’t.

- Compilers typically can’t tell ...
Another example

```plaintext
DO J=1, P
   DO I=1, N
      SUM = SUM + A(I,J)
   END DO
END DO
```

<table>
<thead>
<tr>
<th>Proc</th>
<th>T=1</th>
<th>T=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUM = SUM + A(1,1)</td>
<td>SUM = SUM + A(1,1)</td>
</tr>
<tr>
<td>2</td>
<td>SUM = SUM + A(1,1)</td>
<td>... etc ...</td>
</tr>
<tr>
<td>3</td>
<td>SUM = SUM + A(1,1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SUM = SUM + A(1,1)</td>
<td></td>
</tr>
</tbody>
</table>

Race Condition

Time
About this example

- As written by the user, we can not parallelize this problem
- After a code transformation, we have a version that is much more suitable for parallelization
- We exploit the observation that we can perform the summation in two steps:
  - Sum the individual columns of the array
  - Accumulate these individual values into the global sum
Automatic Parallelization

- The compilers can auto-parallelize a program
- There are two key issues with this:
  1. The analysis is limited to individual loops
  2. Compilers are restricted in their knowledge
- From a practical point of view, this means that often the user has to assist the compiler
- This is achieved with special comment lines called “directives” (Fortran)” and #pragma constructs (C)
- The amount of work involved is often modest
Decisions, Decisions, ...

User directives or pragma’s Found?
- Yes
- No
  - Analyze the loop for data dependencies
    - Is a data dependency Found?
      - Yes
      - No
        - Can the dependency be broken?
          - Yes
          - No
            - Generate parallel code
            - Maybe
  - Generate parallel code