

PRACTICAL 2 - COMPILER OPTIMIZATIONS AND TIMING ROUTINES*

Tim Stitt Ph.D.

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0[†]

Abstract

In this module you will gain some insight into the effect of Cray compiler optimization options, as well as hand-tuning optimizations, on the execution performance of a simple scientific kernel. You will also discover different techniques for timing the runtime performance of a complete code, or regions of the code.

1 Introduction

In this practical you will experiment with the optimization flags on the Cray compiler, to observe their effect on the runtime performance of a simple scientific kernel. Furthermore, you will be given the opportunity to perform some "hand-tuning" on the source code. You will also be introduced to methods for timing the runtime performance of your complete source code, or individual segments of it. If you require any assistance, please do not hesitate to contact the available support staff.

1.1 Objectives

The objectives of this practical are to gain experience in:

- i. applying compiler optimization flags and observing their effect
- ii. applying "hand-tuned" optimizations and observing their effect
- iii. timing the runtime performance of complete codes and individual instruction blocks

2 Timing

Calculating the time your code requires to execute is beneficial for comparing runtime performance between various code modifications and/or the application of **compiler optimization** flags.

*Version 1.4: Oct 5, 2009 3:14 am -0500

[†]<http://creativecommons.org/licenses/by/3.0/>

2.1 Timing Complete Program Execution

The elapsed **real time** (wallclock) of an executing program can be obtained at the command line using the **time** utility.

Example 1: Invoking The time Utility

```
> time app
real 0m4.314s
user 0m3.950s
sys 0m0.020s
```

The **time** utility returns 3 timing statistics:

real	the elapsed real time between invocation and termination
user	the amount of CPU time used by the user's program
sys	the amount of CPU time used by the system in support of the user's program

Table 1: Statistics Returned By The 'time' Utility

NOTE: Typically the **real** time and **user+sys** time are the same. In some circumstances they may be unequal due to the effect of other running user programs and/or excessive disk usage.

Frequently it is useful to time specific regions of your code. This may be because you want to identify particular performance **hotspots** in your code, or you wish to time a specific **computational kernel**. Both C and Fortran90 provide routines for recording the execution time of code blocks within your source.

2.2 Timing Code Regions in Fortran90

Fortran Language Timers

The Fortran90 language provides two portable timing routines; **system_clock()** and **cpu_time()**.

Example 2: system_clock()

The **system_clock()** routine returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. To get the elapsed time, you must call **system_clock()** twice, and subtract the starting time value from the ending time value.

IMPORTANT: To convert from the tick-based measurement to seconds, you need to divide by the clock rate used by the timer.

```
integer :: t1, t2, rate

call system_clock(count=t1, count_rate=rate)

! ...SOME HEAVY COMPUTATION...

call system_clock(count=t2)

print *, "The elapsed time for the work is ",real(t2-t1)/real(rate)
```

Example 3: cpu_time()

The **cpu_time()** routine returns the processor time taken by the process from the start of the program. The time measured only accounts for the amount of time that the program is actually running, and not the time that a program is suspended or waiting.

```

    real :: t1, t2

call cpu_time(t1)

! ...SOME HEAVY COMPUTATION...

call cpu_time(t2)

print *, "The elapsed time for the work is ",(t2-t1)

```

MPI Timing

To obtain the wallclock time for an individual MPI process, you can use the `mpi_wtime()` routine. This routine returns a double precision number of seconds, representing elapsed wall-clock time since an event in the past.

Example 4: `mpi_wtime()`

```

    DOUBLE PRECISION :: start, end

start = MPI_Wtime()

! ...SOME HEAVY COMPUTATION...

end   = MPI_Wtime()

print *, 'That took ',(end-start), ' seconds'

```

OPENMP TIP: In OpenMP codes you can time individual threads with `omp_get_wtime()`.

2.3 Timing Code Regions in C

C Language Timers

The C language provides the portable timing routine `clock()`.

Example 5: `clock()`

Like the Fortran90 `system_clock()` routine, the C `clock()` routine is tick-based and returns the number of clock ticks elapsed since the program was launched.

IMPORTANT: To convert from the tick-based measurement to seconds, you need to divide the elapsed ticks by the macro constant expression `CLOCKS_PER_SEC`.

```

#include <stdio.h>
#include <time.h>

int main(void)
{
    clock_t t1,t2;
    double elapsed;

```

```

t1=clock();

// SOME HEAVY COMPUTATION

t2=clock();
elapsed=t2-t1;

printf("The elapsed time for the work is %f",elapsed/CLOCKS_PER_SEC);
return 0;
}

```

MPI Timing

Like Fortran90 codes, you can obtain the wallclock time for an individual MPI process, using the `MPI_Wtime()` routine. This routine returns a double precision number of seconds, representing elapsed wall-clock time since an event in the past.

Example 6: `mpi_wtime()`

```

double t1, t2;

t1 = MPI_Wtime();

// SOME HEAVY CALCULATIONS

t2 = MPI_Wtime();

printf("MPI_Wtime measured an elapsed time of: %1.2f\n", t2-t1);
fflush(stdout);

```

OPENMP TIP: Also like Fortran90, C-based OpenMP codes can be timed with `omp_get_wtime()`.

3 The "Naïve" Matrix Multiplication Algorithm

Matrix multiplication is a basic building block in many scientific computations; and since it is an $O(n^3)$ algorithm, these codes often spend a lot of their time in matrix multiplication.

The most naïve code to perform matrix multiplication is short, sweet, simple and very very slow. The naïve matrix multiply algorithm is highlighted in Figure 1.

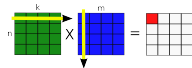


Figure 1: The "naïve" Matrix-Multiplication Algorithm

For each corresponding row and column, a **dot product** is formed as shown in Figure 2.

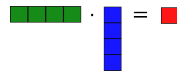


Figure 2: Matrix-Multiplication is composed of repeated dot-product operations

The **naïve** matrix-multiplication algorithm can be implemented as follows:

```

for i = 1 to n
  for j = 1 to m
    for k = 1 to m
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end for
  end for
end for

```

end for

Naïve Matrix-Multiplication Implementation

In the following exercises you will use the naïve matrix-multiplication implementation to experiment with various compiler optimization options, as well as "hand-coded" tuning, to deliver the best performance on this simple scientific kernel.

4 Compiler Optimization Flags

Fortran90 Template

For this practical, use the template code `matmul.f90` provided in `../Practicals/Practical_2/Fortran90`

C Template

For this practical, use the template code `matmul.c` provided in `../Practicals/Practical_2/C`

Exercise 1: Compiler Flag Optimizations

Read the section on compiler optimization flags in the Cray compiler manpages i.e.

Fortran Compiler Manpages

`man crayftn (line 678)`

or

C Compiler Manpages

`man craycc (line 509)`

LISTING OPTIMIZATIONS: If you want to know what compiler optimization options are applied at levels **-O0**, **-O1**, **-O2** and **-O3** then compile your code with the additional option **-eo** e.g. **ftn -O2 -eo -o foo foo.f90**

Exercise 2: Applying Optimization Flags *(Solution on p. 8.)*

Compile and execute a separate copy of the naïve matrix-multiplication implementation for each compiler optimization flag; **-O0**, **-O1**, **-O2** and **-O3**. Record your observed timings in a table like the one shown in Table 2.

TIMING TIP: Use the **time** utility in your batch script to request the elapsed time for each calculation. The timings reported by the **time** utility will be displayed in the standard error logfile e.g. jobOutput.e12345

SUBMISSION TIP: Request a maximum of 10 minutes for your batch jobs

Optimization Flag	Wallclock Time (sec)
-O0	
-O1	
-O2	
-O3	

Table 2: Timings Table for Matrix-Multiply Kernel

Exercise 3: Timing The Matrix-Multiplication Kernel *(Solutions on p. 8.)*

By using the **time** utility to record the timing statistics for the entire code, we are including the overhead time it takes to populate the matrices **A** and **B** with initial values. For large matrices, this overhead time could be quite significant and hence skew the recorded time for the matrix-multiply kernel calculation.

To ensure we are only recording the time for the matrix-multiplication kernel, we should wrap the matrix-multiply code block with source-level timing routines.

Using the language-level timing routines discussed earlier, record the time taken for the matrix-multiply kernel **only**. How do these times compare to the overall execution time?

TESTING TIP: Only record results for **-O0** and **-O2** compiler optimization flags

5 "Hand-Tuned" Optimizations

Sometimes it is possible to generate further performance by manually applying optimizations to your source code instructions. In the following exercises you will gain some experience in hand-coding simple optimizations into the naïve matrix-multiply implementation.

5.1 Fortran90 Programmers Only

The element order in which 2D arrays are traversed can have a significant performance impact between Fortran and C languages. In C, 2D arrays are stored in memory using **row-major** order. In Fortran, arrays are stored in memory using **column-major** order.

Exercise 4: Loop Re-ordering

(Solutions on p. 8.)

The naïve matrix-multiply Fortran90 implementation suffers in performance because its inner-most loops traverse array rows and not columns (this prevents the **cache** from being used efficiently).

Try to improve the performance of the Fortran90 implementation by maximizing column traversals. What performance gains do you achieve for **-O0** and **-O2** compiler flags? What order of indices I, J and K gives the best performance?

TIP: Modern compilers are very good at detecting sub-optimal array traversals and will try to reorder the loops automatically to maximize performance.

Exercise 5: Loop Unrolling (Advanced)

(Solutions on p. 9.)

Manually unrolling loops can sometimes lead to performance gains by reducing the number of loop tests and code branching, at the expense of a larger code size. If the unrolled instructions are independent of each other, then they can also be executed in parallel.

TIP: Review loop unrolling by consulting the course slides here.

Try to achieve performance improvement on the **original** naïve matrix-multiplication implementation by applying the loop unrolling technique. Compare your unrolled version against the results obtained with the **-O0** and **-O2** compiler flags.

What performance improvement do you get when you unroll **8** times?

Solutions to Exercises in this Module

Follow-Up to Exercise (p. 6)

Are the results exactly the same for each flag?

Fortran Solution A to Exercise (p. 6)

```

      real :: t1,t2

...MATRIX INITIALIZATION...

call cpu_time(t1)

! Perform the matrix-multiplication

do I=1,N
  do J=1,N
    do K=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    end do
  end do
end do

call cpu_time(t2)

print *, "The time (in seconds) for the matrix-multiply kernel is ",t2-t1

```

C Solution B to Exercise (p. 6)

```

      clock_t t1,t2;
double elapsed;

... MATRIX INITIALIZATION ...

t1=clock();

// Perform Matrix-Multiply Kernel

for( i = 0; i < n; i++ )
  for( j = 0; j < n; j++ )
    for( k = 0; k < n; k++ )
      c[i][j] = c[i][j] + a[i][k] * b[k][j];

t2=clock();
elapsed=t2-t1;

printf("The time (in seconds) for the matrix-multiply kernel is %f\n",elapsed/CLOCKS_PER_SEC);

```

Hint A to Exercise (p. 7)

Try to nest deeper the loop over I.

Solution B to Exercise (p. 7)


```

! Perform the matrix-multiplication

do K=1,N
  do J=1,N
    do I=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    end do
  end do
end do

```

Hint A to Exercise (p. 7)

Step 1. Unroll the outer loop **I** 4 times

Step 2. Initialize 4 accumulating variables at the start of inner loop **J** e.g. C0, C1, C2 and C3

Step 3. Within the inner-most loop **K** do the following:

- i. Create a temporary variable equal to **B(K,J)**
- ii. Replace the matrix-multiply statement with 4 separate accumulators

Step 4. After the inner-most loop is completed, update **C** with the accumulated totals.

Fortran90 Solution B to Exercise (p. 7)

```

! Perform the matrix-multiplication

do I=1,N,4
  do J=1,N
    C0=0.D0
    C1=0.D0
    C2=0.D0
    C3=0.D0
    do K=1,N
      TEMP=B(K,J)
      C0=C0+A(I,K)*TEMP
      C1=C1+A(I+1,K)*TEMP
      C2=C2+A(I+2,K)*TEMP
      C3=C3+A(I+3,K)*TEMP
    end do
    C(I,J)=C(I,J)+C0
    C(I+1,J)=C(I+1,J)+C1
    C(I+2,J)=C(I+2,J)+C2
    C(I+3,J)=C(I+3,J)+C3
  end do
end do

```

C Solution C to Exercise (p. 7)

```

// Perform Matrix-Multiply Kernel

for( i = 0; i < n; i=i+4 )
{
  for( j = 0; j < n; j++ )
  {

```

```

c0=0;
c1=0;
c2=0;
c3=0;

for( k = 0; k < n; k++ )
{
    temp=b[k][j];
    c0=c0+a[i][k]*temp;
    c1=c1+a[i+1][k]*temp;
    c2=c2+a[i+2][k]*temp;
    c3=c3+a[i+3][k]*temp;
}

c[i][j]=c[i][j]+c0;
c[i+1][j]=c[i+1][j]+c1;
c[i+2][j]=c[i+2][j]+c2;
c[i+3][j]=c[i+3][j]+c3;

}
}

```

Glossary

Definition 1: Cache

Area of high-speed memory that contains recently referenced memory addresses.

Definition 2: Column-Major Ordering

Array elements are stored in memory as contiguous columns in the matrix. For best performance (and optimal cache use) elements should be traversed in column order.

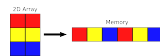


Figure 3: Column-Major Ordering

Definition 3: Dot Product

Also known as the scalar product, it is an operation which takes two vectors over the real numbers \mathbb{R} and returns a real-valued scalar quantity. It is the standard inner product of the orthonormal Euclidean space.

Definition 4: Hotspot

A block of source code instructions that account for a significant amount of the CPU execution time.

Definition 5: Kernel

A block of source code instructions that represent a particular algorithm or calculation.

Definition 6: `omp_get_wtime()`

```
use omp_lib

DOUBLE PRECISION START, END

START = omp_get_wtime()

! ...HEAVY COMPUTATION...

END = omp_get_wtime()

PRINT *, 'That took ', &
      (END - START), ' seconds.'
```

Definition 7: Compiler Optimizations

Transformations to the source code which are applied by the compiler to improve the runtime performance of the executing code e.g. loop unrolling, instruction reordering, in-lining etc.

Definition 8: Real Time

The elapsed time between the invocation of a program and its termination.

Definition 9: Row-Major Ordering

Array elements are stored in memory as contiguous rows in the matrix. For best performance (and optimal cache use) elements should be traversed in row order.

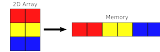


Figure 4: Row-Major Ordering