Optimisation Workshop Notes

Codes can be optimised in two ways: manual and compiler assisted optimisation. Optimisation requires a good understanding of computational architectures as well compilers. Both techniques are required to get the best performance from scientific codes as users can help assist the compiler make the best optimisations. Some experimentation maybe required to find the most optimal configuration for your code. Commands for timing your application run:

```
time ./program
```

The time command is available on every Linux operating system and is a crude way of measuring application performance. This command does not have a very high precision timing mechanism, but can still be used to determine possible performance gains. Ultimately, the code should run quicker. Explanation of the output of the time command is given below:

1. `real` - wall clock or elapsed time;
2. `user` - user time spent in code. This is usually the CPU time;
3. `sys` - system time spent in code. This is usually the time spent writing to disk.

Practical 1 - Loop re-ordering

If you have not already done so, download the example programs onto your home directory using the command:

```
wget http://grace-head00.uea.ac.uk/grace-docs/dpo_examples.tar
tar -xvf dpo_examples
cd dpo_examples
```

Please request an interactive session from the login node before starting the exercises. This exercise involves multiplying two matrices using the standard equation (`program6.c` or `program6.f90`):

$$A_{i,j} = \sum_{k=1}^{n} B_{i,k} C_{k,j}$$

1. Is the current loop ordering in the right order for the C/Fortran program? It should use column major ordering, namely the left most index should vary more;
2. Sketch the matrix memory access pattern for all three matrices A, B and C;
3. Compile code `program6.c` or `program6.f90` and time it. Note the real time spent for the code;
4. Make the `k` loop first and sketch the memory access pattern. How will this speed up the program?
5. Compile the program and time it. How much improvement has been made?
6. Repeat the above steps using the Intel compiler `ifort` with the `-fast` switch. Load the correct module file:

   ```
   module load icc/intel/11.1
   ```

7. Is there any speed up with the Intel compiler?

Practical 2 - Loop un-rolling

1. Unroll the k loop 4 times using the code:

```
do k = 1, n-3, 4
  do j = 1, n
    do i = 1, n
      a(i,j) = a(i,j) + b(i,k) * c(k,j) &
                      + b(i,k+1) * c(k+1,j) &
                      + b(i,k+2) * c(k+2,j) &
                      + b(i,k+3) * c(k+3,j)
    end do
  end do
end do
```

2. Compile the program again with the GNU Fortran compiler and time it. How much improvement has been made compared to the non-unrolled version?
3. Unroll the loop 8 times and re-compile and time. Is there any improvement?
4. Unroll the loop 10 times and re-compile and time. Is there any improvement?
5. Repeat the above steps using `ifort` with the `-fast` switch.

Practical 3 - Cache tiling

Cache tiling is method to split the problem in sizes that fit entirely within the CPU cache. This usually improves scientific code performance. Below is an example code that uses the cache tiling technique:

```
do j1 = 1, n, block
  j2 = min( n, j1 + block - 1 )

  do k1 = 1, n, block
    k2 = min( n, k1 + block - 1 )

    do i1 = 1, n, block
      i2 = min( n, i1 + block - 1 )

      do k = k1, k2, 4
        do j = j1, j2
          do i = i1, i2
            a(i,j) = a(i,j) + b(i,k) * c(k,j) &
                            + b(i,k+1) * c(k+1,j) &
                            + b(i,k+2) * c(k+2,j) &
                            + b(i,k+3) * c(k+3,j)
          end do
        end do
      end do

    end do
  end do
end do
```

There are two assumptions made in the code above a) n is an exact multiple block b) the unroll factor (four) is an exact multiple of block.

1. Implement the cache tiling technique for your code;
2. Experiment with block sizes of 10, 20 and 40 and measure times using the GNU Fortran compiler;
3. Repeat the above step using the Intel Fortran compiler `ifort` using the `-fast` switch and measure times;
4. Compare the times between the GNU compiler and the Intel compiler;
5. Does the cache tiling technique improve performance?

Please complete the event feedback on the web site:

http://rscs.uea.ac.uk/events/feedback

Thanks.