Tools of high performance computing 2024

Exercise 4
Return by Monday 12.2.2024 23:59 to Moodle.
Exercise session: Tuesday 13.2.2024

Note: When measuring CPU times of programs do it many times and calculate the average of the results. (Many times ~ from tens to hundreds).This can be easily done using bash scripts.

Problem 1. *(6 points)*

Optimize the following loops (they are written in Fortran but converting to C++ should be a piece of cake). Measure the CPU time spent in loops for original and optimized versions and for both `-O0` and `-O3` optimization options.

a)

```
integer,parameter :: n=10000000 ! Definition of a constant
real :: a(n),b(n)               ! Declaration of arrays
!… fill arrays a and b with data before executing the loop …
do i=1,n-1
      if (i<500) then
            a(i)=4.0*b(i)+b(i+1)
      else
            a(i)=4.0*b(i+1)+b(i)
      end if
end do
```

b)

```
integer,parameter :: n=3000
real :: a(n,n),b(n,n),c(n)
!… fill arrays a, b, c with data before executing the loop …
do i=1,n
      do j=1,n
            a(i,j)=b(i,j)/c(i)
      end do
end do
```

Problem 2. *(6 points)*

Write a program that calculates matrix products (using `do`/`for` loops, not any library or intrinsic function)

a) $\mathbf{C}=\mathbf{A}\,\mathbf{B}$,
b) $\mathbf{C}=\mathbf{A}^T\mathbf{B}$ and
c) $\mathbf{C}=\mathbf{A}\,\mathbf{B}^T$,

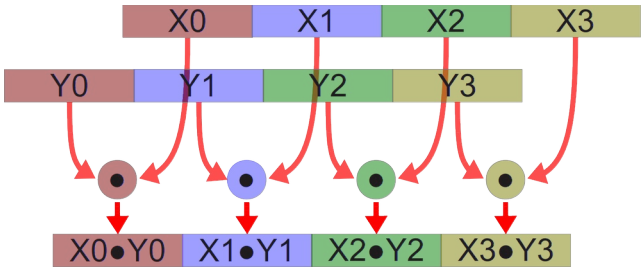
where $\mathbf{A}^T$ is the transpose of $\mathbf{A}$ (i.e. $[\mathbf{A}^T]_{ij}=[\mathbf{A}]_{ji}$). Compare the CPU time consumed for these products. Suitable matrix size is of the order of 1000×1000 . Test with optimization options `-O0` and `-O3`. Comment the results.

Problem 3. *(6 points)*

Measure the CPU time consumed when a simple loop that accesses a 32-bit integer array once per iteration is unrolled (manually, not by compiler; generating code by using a scripting language like `gawk` or `python` is a good idea) to different

depths (e.g. 1, 2, 4, 8,...,32). Test with options `-O0` and `-O3`.

Problem 4. *(6 points)*



Modern Intel and AMD processors have SIMD[1] extensions where simple operations can be done on many data elements concurrently. On Intel hardware these are named as MMX (no more used), SSE and AVX. These extensions introduce new wide registers where you can load integer or floating point numbers and perform e.g. addition and multiplication for all elements at the same time.

These extensions are useful if you have loops in your code that perform the same simple operation to a large amount of data (vectorization of loops); e.g. vector and matrix operations. SIMD features can be switched on by the compiler option

```
-ftree-vectorize
```

in case of `gfortran/gcc/g++`[2].

Compile and run the attached code[3] both enabling and disabling the vectorization. Run multiple times and with varying vector sizes (starting from e.g. 32). Compare the CPU time used with and without SIMD extensions. Calculate average time and its standard deviation for each vector size. The SIMD extensions can be switched off by `gfortran/gcc/g++` option

```
-fno-tree-vectorize
```

For the general optimization option use `-O2`.

Note that you only need to analyze the last CPU time (expression `v4-v3`) the code prints out.

1 https://en.wikipedia.org/wiki/SIMD
2 Note that you don't find this option in the `gfortran` man page. However, `gcc` man page has it. Many `gcc` options work for `gfortran`, too (but not all).
3 File `ex4p4.cpp`. It's written in C++ but conversion to Fortran is straightforward.