PPCES 2013: Serial Tuning Lab Exercises

March 11, 2013
Tim Cramer, Frank Robel
{cramer,robel}@rz.rwth-aachen.de

0. Preparations

- Login to one of our frontend nodes, e.g.:
  ```
  $ ssh cluster.rz.rwth-aachen.de
  ```

- Change into the lab directory:
  ```
  $ cd serial/serial_tuning
  ```

Since we all work on the same frontends the performance measurements can differ between two runs. Execute the examples several times to get proper results or connect to another frontend.

1. Norm Calculation of a Matrix

This first small exercise performs the calculation of $||\cdot||_1$ and $||.||_\infty$, which are defined by

$$\|A\|_1 := \max_{j=1\ldots,n} \sum_{i=1,\ldots n}^{n} |a_{ij}| \qquad \text{("maximum column sum")}$$

$$\|A\|_\infty := \max_{i=1\ldots,n} \sum_{j=1,\ldots n}^{n} |a_{ij}| \qquad \text{("maximum row sum"),}$$

where $A = \left(a_{ij}\right)_{i,j=1,\ldots n} \in \mathbb{R}^{nxn}$ is a real matrix. The example is written in C++ or Fortran and can be found in the directory `norm`. You can specify different dimensions n for the matrix as input parameter:

```
$ cd norm
$ cd C++
or
$ cd Fortran
$ make
$ ./norm n
```

We have prepared the exercises in C/C++ and Fortran. Please choose your favorite language. Begin with section 1.1 for C++ and with section 1.2 for Fortran.

1.1. C++

1.1.1. Implement the 1-Norm (using Column Wise Memory Access)

Please realize the column wise calculation of $\|A\|_1$ by implementing the function `norm1_col_wise(double** const A, const int n)`. What is the performance compared to the calculation of $\|A\|_\infty$ realized in `norm_max(double** const A, const int n)`? What could be the reason for the difference?

1.1.2. Implement the 1-Norm (using Row Wise Memory Access)

As we have seen the run time difference between the calculations of $\|A\|_1$ and $\|A\|_\infty$ is significant, although the number of floating point operations are the same. But in the column-wise norm1 calculation the memory access pattern is not optimal concerning the memory access pattern of C++. Please implement the function `norm1_row_wise(double** const A, const int n)`. Use the helper array `double* colsums` to store the sum for each column. What performance can you reach now? What is the speedup for different matrix sizes?

1.2. FORTRAN

1.2.1. Implement the Maximum Norm (using Row Wise Memory Access)

Please realize the row wise calculation of $\|A\|_\infty$ by implementing the function `max_norm(A, n)` in norm.f90 What is the performance compared to the calculation of $\|A\|_1$ realized in `norm1(A,n)`? What could be the reason for the difference?

1.2.2. Implement the Maximum Norm (using Column Wise Memory Access)

As we have seen the run time difference between the calculations of $\|A\|_1$ and $\|A\|_\infty$ is significant, although the number of floating point operations are the same. But in the column-wise max_norm calculation the memory access pattern is not optimal concerning the memory access pattern of Fortran. Please implement the function `max_norm1_col_wise(A, n)`. Use the helper array `col_wise_row_sum` to store the sum for each column. What performance can you reach now? What is the speedup for different matrix sizes?

1.3. Compare Different Compilers (C++ and Fortran)

In many cases the performance gap of the executables build with different compilers can be really big. Try to switch the compiler, rebuild the application and compare the performance for the Intel, GNU, PGI and Oracle (Sun) compiler for different matrix sizes, e.g.:

```
$ module switch intel gcc
$ make clean
$ make
$ ./norm 5000
```

Hint: To get a list of the available compiler type `module avail`. All compilers are installed in different versions. To find out which compiler is actually loaded type `module list`. The modules you need are gcc, intel, pgi and studio. Please use `module switch` to change the compiler!

Write the results into

Table 1 if you used the C++ version and in Table 2 for Fortran. Note: Since you are not alone on the frontends it could be that the performance is distorted by the other users! So if there are some very unexpected measurements, please repeat the execution and take the best result.

Table 1: Compiler Performance C++

Compiler	Dimension N	Max-Norm (MFlops)	1-Norm (col) (MFlops)	1-Norm(row) (MFlops)
Intel 12.1	100			
	5000			
PGI 13.1	100			
	5000			
GNU 4.7	100			
	5000			
Oracle(Sun) 12.3	100			
	5000			

Table 2: Compiler Performance Fortran

Compiler	Dimension N	1-Norm (MFlops)	Max-Norm (row) (MFlops)	M-Norm (col) (MFlops)
Intel 12.1	100			
	5000			
PGI 13.1	100			
	5000			
GNU 4.7	100			
	5000			
Oracle(Sun) 12.3	100			
	5000			

Note: It is always a good idea to compare different compilers in different versions, but the results of this measurements do not allow a general statement like “Compiler of vendor A is better than compiler of vendor B”. It strongly depends on many different parameters (e.g., the compiler flags, the compiler version, the used hardware etc.).

2. Performance of a Vector-Matrix-Multiplication

In the second exercise a Vector-Matrix-Multiplication

$$\boldsymbol{y}' = \boldsymbol{x}' * \boldsymbol{A}$$

for the **C** example and a Matrix-Vector-Multiplication

$$\boldsymbol{y} = \boldsymbol{A} * \boldsymbol{x}$$

for the **Fortran** should be done, where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{mxn}}$, $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{m}}$ and $\boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$. The example can be found in the directory `vecmat/{C++|Fortran}`, depending on the language you want to use.

In the C version the matrix A is now stored as contiguous block in the memory (row-by-row), in contrast to the example in exercise 1. This means that you have to calculate the offset for a new row by multiplying with n. This has the advantage that you do not need to dereference the pointer twice.

Continue with section 2.1 if you prefer C or go to section 2.2 for Fortran.

2.1. C

2.1.1. Memory Access Pattern (C)

In the function `vxm_ref(int m, int n, const double* A, const double* x, double* y)` in the file `vecmat.c` the column-wise Vector-Matrix-Multiplication has to be done. Add the missing line using the offset for A to make this operation work. What is the performance impact of this access order concerning the storage order of C / C++? Implement the function `vxm_order(int m, int n, const double* A, const double* x, double* y)` to reach a better memory access pattern by interchanging the loops and compare the performance (MFlop/s).

2.1.2. Compiler Optimization (C)

If you load a compiler in the RWTH Environment (e.g., `$ module load intel`) some important compiler specific variables are set. One of them is FLAGS_FAST. You can use this flag to use compiler optimizations during the build process. Modify the C_FLAGS variable in the Makefile to enable this. What is the performance impact? Is the optimization of `vxm_order()` still more efficient compared to `vxm_ref()`? Hint: To see what the compiler is doing you can get very detailed compiler feedback (e.g., for the Intel compiler with the flag `–opt-report)`. Most modern compilers are very smart nowadays, but in some cases we expect them to do some optimizations they cannot do as we will see in the exercise. What optimization techniques are used for the performance relevant parts?

2.1.3. Vectorization (C)

A very powerful compiler optimization is vectorization. This means that in case of loop independent iterations several operations can be executed in parallel by using special CPU instructions. Briefly spoken the compiler tries to unroll the loop and combines this technique with the generation of packed SIMD (Single Instruction Multiple Data) instructions. But in some cases you need to help the compiler to reach this. This is especially true in C / C++ because of pointer aliasing. To determine if the compiler was able to vectorize one of the loops you can add a c flag to C_FLAGS in the Makefile to get a detailed report (e.g., for the Intel Compiler "-vec-report3").

a. Find out in which of the vxm_* functions the compiler is successful! What could be the reason why in one case the compiler is able to vectorize and in the other not?

b. The C99 standard defines a special `restrict` keyword to limit the effects of pointer aliasing. Please implement the column-wise function `vxm_restrict(…)` using the restrict keyword. Can the loop be vectorized now? What could the reason (use the compiler feedback!)? How does the performance change? Hint: For the Intel Compiler you have to add the `-restrict,` for the gcc `-std=c99` flag as well. We recommend using the Intel compiler.

2.1.4. MKL (C)

For many basic operations highly optimized libraries are available. In the area of HPC the LAPACK- / BLAS-Routines are widely used. The reference manual to the Intel implementation (MKL) can be found here:

http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/mklxe/mkl_manual_win_mac/index.htm.

These routines are implemented in Fortran. Although you can call these functions directly from C / C++, it is strongly recommended to use corresponding interface called CBLAS. An example of using the CBLAS can be found in the clusters example collection in the directory `$PSRC/psr/usecblas.c`.

2.1.5. Modify the Makefile such that the environment variables FLAGS_MATH_INCLUDE and FLAGS_MATH_LINKER are used for compiling / linking and include the corresponding header in `vecmat.c`. Hint: Since the Intel Complier comes with the MKL included this step might be not necessary, but keep it in mind, if you want to use a different compiler! For the other compiler you additional need to load the intelmkl module.

2.1.6. Change the `vxm_mkl(…)` routine such that it calls `cblas_dgemv`() from the Intel MKL. Hint: This routines performs a matrix-vector operation. So you need to transform the matrix for performing a vector-matrix-multiplication.

2.1.7. How much better is the Intel implementation compared to your own?

2.1.8. The theoretical peak performance of a machine can be calculated with:

$$\mathrm{MFLOPS} = \mathrm{CPU\_RATE} * \mathrm{FLOPS\_PER\_CYCLE} * \mathrm{NUMBER\_OF\_CORES} * \mathrm{NUMBER\_OF\_SOCKETS}/1E^{-6}$$

What is the theoretical peak performance of the machine you are using? How much do you reach with the optimized versions of the matrix-vector-multiplication? Hint: Use the `/proc/cpuinfo` file to find the needed information.

NOTE: The Intel MKL can run in parallel depending on the value of the environment variable OMP_NUM_THREADS. If set to a value bigger than one, the comparison to your serial vxm version might not be fair.

2.2. Fortran

2.2.1. Memory Access Pattern (Fortran)

In the function `mxv_ref (m,n,A,x,y)` in the file `vecmat.f90` the row-wise Matrix-Vector-Multiplication has to be done. What is the performance impact of this access order concerning the storage order of Fortran? Implement the function `mxv_order(m, n, A, x, y)` to reach a better memory access pattern by interchanging the loops and compare the performance (MFlop/s).

2.2.2. Compiler Optimization (Fortran)

If you load a compiler in the RWTH Environment (e.g., `$ module load intel`) some important compiler specific variables are set. One of them is FLAGS_FAST. You can use this flag to use compiler optimizations during the build process. Modify the F_FLAGS variable in the Makefile to enable this. What is the performance impact? Is the optimization of `mxv_order()` still more efficient compared to `mxv_ref()`? Hint: To see what the compiler is doing you can get very detailed compiler feedback (e.g., for the Intel compiler with the flag `-opt-report)`. Most modern compilers are very smart nowadays, but in some cases we expect them to do some optimizations they cannot do as we will see in the exercise. What optimization techniques are used for the performance relevant parts?

2.2.3. Vectorization (Fortran)

A very powerful compiler optimization is vectorization. This means that in case of loop independent iterations several operations can be executed in parallel by using special CPU instructions. Briefly spoken the compiler tries to unroll the loop and combines this technique with the generation of packed SIMD (Single Instruction Multiple Data) instructions. But in some cases you need to help the compiler to reach this. Find out in which of the mxv_* functions are vectorized successful for different optimization levels. For the vectorization of `mxv_ref()` a combination of optimization is necessary. Which is it?

Hint: You can use `-vec-report3` for more compiler Feedback

2.2.4. MKL (Fortran)

For many basic operations highly optimized libraries are available. In the area of HPC the LAPACK- / BLAS-Routines are widely used. The reference manual to the Intel implementation (MKL) can be found here:

http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/mklxe/mkl_manual_win_mac/index.htm.

These routines are implemented in Fortran. An example of using the BLAS within a C program can be found in the clusters example collection in the directory `$PSRC/psr/usecblas.c.`

2.2.5. Modify the Makefile such that the environment variables FLAGS_MATH_INCLUDE and FLAGS_MATH_LINKER are used for compiling / linking. Hint: Since the Intel Complier comes with the MKL included this step might be not necessary, but keep it in mind, if you want to use a different compiler! For the other compiler you additional need to load the intelmkl module.

2.2.6. Change the `mxv_mkl(…)` routine such that it calls `dgemv`() from the Intel MKL.

2.2.7. How much better is the Intel implementation compared to your own?

2.2.8. The theoretical peak performance of a machine can be calculated with:

$$\text{MFLOPS} = \text{CPU\_RATE} * \text{FLOPS\_PER\_CYCLE} * \text{NUMBER\_OF\_CORES} * \text{NUMBER\_OF\_SOCKETS}/1E^{-6}$$

What is the theoretical peak performance of the machine you are using? How much do you reach with the optimized versions of the matrix-vector-multiplication? Hint: Use the `/proc/cpuinfo` file to find the needed information.

NOTE: The Intel MKL can run in parallel depending on the value of the environment variable OMP_NUM_THREADS. If set to a value bigger than one, the comparison to your serial mxv version might not be fair.

3. Solutions

For all source file you have to modify there is a *.solu or *.solution file in the corresponding directory. Feel free to compare it with your own solution.