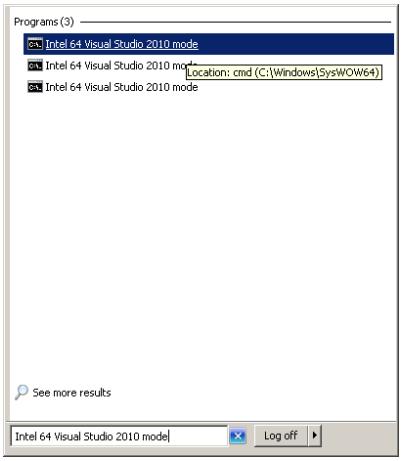
PPCES 2012: Serial Tuning Lab Exercises

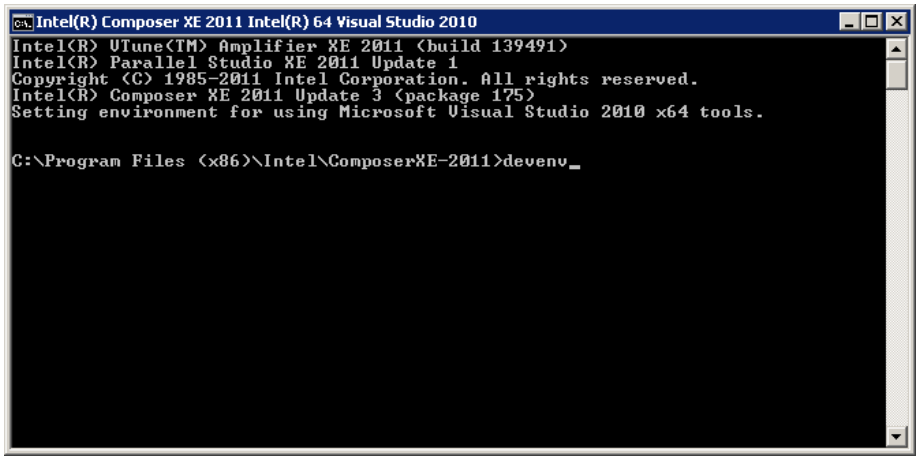
March 19, 2012

Tim Cramer, Frank Robel, Daniel D'Abramo
{cramer,robel,dabramo}@rz.rwth-aachen.de

0. Preparations

- Login to the Windows system (please refer to the additional handout):
  `cluster-win.rz.rwth-aachen.de`
- Download serial_tuning2012.tar.gz from the PPCES Homepage (e.g., into your $HOME directory):
  http://www.rz.rwth-aachen.de/ppces
- Untar the archive. (e.g. with the program 7zip)
- Change into the lab directory:
- Do **not** open the project by double-clicking the Visual Studio Project File! Do the following instead:
  - Click `Start`, type "`Intel 64 Visual Studio 2010 mode`" into the command line and open the first one.
  - Start Visual Studio 2010 by typing `devenv` to the command line.
  - Open the project: `File -> Project/Solution.`





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 $A = (a_{ij})_{i,j=1,\ldots n} \in \mathbb{R}^{nxn}$ is a real matrix. The example is written in C++ and can be found in the directory `norm/C++`. You can specify different dimensions n for the matrix as input parameter: Open `Project / Properties` in the menu. In the next window you can configure the command line parameter in the field: `Configuration Properties / Debugging / Command Arguments`.

1.1. Implement Row-Wise calculated Norm

Please perform the row wise calculation of $\|A\|_1$ by implementing the function `norm1_row_wise(double** const A, const int n)`. How is the performance compared to the calculation of $\|A\|_\infty$ performed by `norm_max(double** const A, const int n)`? What could be the reason for the difference?

1.2. Implement Column-Wise Norm

As we have seen the run time difference between the calculation of $\|A\|_1$ and $\|A\|_\infty$ is significant, although the number of floating point operations are the same. But in the row-wise norm1 calculation the memory access pattern is not optimal in regard to the memory access pattern of C++. Please implement the function `norm1_col_wise(double** const A, const int n)`. Use the helper array `double* colsums` to store the sum for each column. Which performance can you reach now? What is the speedup for different matrix sizes?

1.3. Compare Different Compilers

In many cases the performance gap between the executables built with different compilers can be really big. Try to switch the compiler, rebuild the application and compare the performance for the Intel and Microsoft compiler for different matrix sizes.

Hint: We have an existing Intel project. If you want to use the Visual Studio compiler, the project has to be converted. This can be done by right-clicking the project and selecting the “`Intel C++ Composer XE 2011`” / “`Use Visual C++`”

Write the results into

Table 1. Note: Since you are not alone on the frontends the performance could be distorted by other users! So if there are some really unexpected measurements, please repeat the execution and take the best result.

Table 1: Compiler Performance

Compiler	Dimension N	Max-Norm (MFlops)	1-Norm (row) (MFlops)	1-Norm(col) (MFlops)
Intel 12.0	100			
	5000			
Microsoft 2010	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 should be done:

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

where $\boldsymbol{A} \in \mathbb{R}^{mxn}$, $\boldsymbol{x} \in \mathbb{R}^{m}$ and $\boldsymbol{y} \in \mathbb{R}^{n}$. The example, which can be found in the directory `vecmat\C++`, is now written in C. In contrast to the example in exercise 1 the matrix A is now stored as a contiguous block in the memory (row-by-row). This means that you have to calculate the offset for a new row by multiplying it with n. This has the advantage that you do not need to dereference the pointer twice.

2.1. Memory Access Pattern

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).

2.2. Compiler Optimization

If you switch between different `Solution Configurations` and `Solution Platforms` some important compiler specific variables are set. For example the level of optimization is chosen:

Solution Configuration	Level of Optimization

Debug	Disabled (/Od)
Release	Maximize Speed (/O2)

You can use the `Release` settings together with the `x64` Platform. 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 `/Qopt-report`. You can set this flag in `Project / Properties / Configuration Properties / C/C++ / Command Line`). 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?

2.3. Vectorization

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 case you need to help the compiler to do 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 get a detailed report (e.g., for the Intel Compiler "/Qvec-report3").

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

2.3.2. 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 be the reason (use the compiler feedback!)? How much does the performance change? Hint: For the Intel Compiler you have to add the flags /Q`restrict` and `/fp:fast`.

2.4. MKL

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 cluster's example collection in the directory `P:\psr\usecblas.c`.

2.4.1. Modify the Project Properties such that Intel MKL Sequential can be used for compiling / linking and include the corresponding header in `vecmat.c`. Hint: Open `Project / Properties / Configuration Properties / Intel Performance Libraries` and choose `MKL Sequential`.

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

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

2.4.4. 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 achieve with the optimized versions of the matrix-vector-multiplication? Hint: Some of this information can be found in `System` (click `Windows + Pause`) and in the Task Manager. The Flops/cycle are depending on the use hardware. In our case it is equal 4.

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.
Hint:

- Right click on the file that should be changed and choose `Exclude From Project`. (e.g. `vecmat.c`)
- Right click on the folder `Source Files` and choose `Add / Existing Item …` (e.g. `vecmat.solu.c`)

Notice:
Using vecmat.solu.c also need a couple of changes in Project Properties: You have to use the Intel Compiler with Release Configuration, the Intel MKL, and some additional c flags: `/Qrestrict /fp:fast`