The Matrix-Vector Product

HPC Assignment 1 - Feb, 2017

Kamyar Nazeri
Student ID: 100633486

Introduction

In this assignment, we are calculating Matrix-Vector product using C++ code and exploring the effect on wall-time on the way the computation is coded, compiled and run. We will examine the wall-time for each configuration, draw some basic conclusions and offer the best practice to calculate Matrix-Vector product based on our empirical results.
The following are the configurations used to compile/code/run the program:

- **Compiler**: Linux GNU compiler (g++); Intel C++ compiler (icc)
- **Optimization**: No optimization (-O0); Aggressive optimization(-O3)
- **Algorithm**: Imperative double loop; LAPACK Matrix-Vector product routine
- **Parallelism**: No multi-threading (1 thread); 2 threads; 4 threads

Note: There's no built-in Matrix datatype in C++, hence the intrinsic function for calculating the Matrix-Vector product in Fortran has no counterpart in C++ and we have eliminated this method in our tests.

The computer software/hardware configuration used for the test:

- **CPU**: Intel Core I7-5600 CPU 2.60GHz x 4 (Physical Cores: 2)
- **RAM**: 8Gb
- **OS**: Linux Ubuntu 16.04 LTS 64-bit

We have fixed the matrix size to **26,000** which would result in the biggest amount of wall-time using the GNU compiler without optimization running double loop method on the above machine. This wall-time is considered the Base Wall-Time (T0) in this report. The amount of wall-time measured for other configurations are relative to T0, hence our empirical conclusions are based on how fast the wall-times are compared to T0.

Compile/Run C++ Code:

- To compile and execute the code with all the possible configuration, run the shell script ***run.sh***.
- To run the code using LAPACK routine, we would need BLAS library. A distinct compiled library of BLAS exists in the \lib directory for each compiler.
  GNU: ***blas_gnu***, Intel: ***blas_intel***
- We use OpenMP to parallelize the computation. To activate OpenMP directive in C++ code, compile the source code using openmp flag.
  GNU: ***-fopnemp***, Intel: ***-qopenmp***
- To run the code using Double-Loop method, pass the argument ***-D*** to the executable; similarly, to run the code using LAPACK method pass the argument ***-L*** to the executable. e.g. *result -D* (runs the code using double-loop method)
- To run the code using OpenMP parallelism, pass the number of threads as an argument to the executable (***-T2***, ***-T4***, ...) e.g. *result -D -T2* (runs the code using double loop method with 2 threads)

Wall-Time Results:

The following table shows the amount of wall-time in seconds, calculating a matrix-vector product, using different settings and compilers (matrix size: 26,000):

		GNU Compiler		**Intel Compiler**	
	optimization:	**-O0**	**-O3**	**-O0**	**-O3**
Double Loop	**1 thread**	2.02 s	0.69 s	2.08 s	0.33 s
	2 threads	1.32 s	0.36 s	1.15 s	0.21 s
	4 threads	1.43 s	0.27 s	1.18 s	0.22 s
LAPACK (BLAS)		0.68 s	0.68 s	0.18 s	0.18 s

Note: The implementation of the Matrix-Vector product in LAPACK is considered internal to the library, hence, we cannot parallelize the LAPACK matrix-vector multiplication routine in this assignment.

Conclusions:

1 The Effect of Optimization:

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program [1].
In our tests, enabling compiler optimizations, significantly improves the performance of the operation and reduces the wall-time. The effect is almost by a factor of 3 for GNU compiler and by a factor of 6 for Intel compiler.
Changing compiler optimization options however, does not have any effect on the wall-time of the LAPACK routine. This is due to the fact that we are using the **compiled BLAS library** in our tests. Optimization options could be used when compiling the BLAS library, however, any optimization option after that, has no effect on the performance of the library.
Note: By default optimization is disabled for GNU compiler, however the default optimization flag for Intel compiler is -O2 when no optimization flag is specified [2].

2 The Effect of Compiler:

The compiler is the single most important element in the toolchain and it directly correlates to the performance of an application without even making changes to the code itself. Execution time is what we are calculating here:
Choosing Intel compiler over GNU with the right optimization options, we experience 210% increase in the performance of the code using double-loop algorithm on a single core. The performance boost is even more drastic (about 370% increase) when using LAPACK matrix-vector product routine. The increase in performance is mainly because our test machine comes with an Intel CPU, which correlates best with a compiler from the same company that is optimized to take take advantage of advanced processor features like multiple cores and wider vector registers for better performance [3]. And as demonstrated in our tests, better performance is acquired only if compiler's optimization is used. The effect of compiler is almost zero when no optimization is specified.

3 The Effect of Using Libraries:

Using standard, low-level routines in scientific programming which are optimized for speed, can significantly improve the performance of the program. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software [4]. We use BLAS in our tests, which results in almost 300% increase in performance with no compiler optimization, and 200% with aggressive optimization on Intel compiler. However, it appears that GNU optimization does a better job with double-loop than with BLAS, as the performance gained by switching algorithms is not considerable when aggressive optimization is used on GNU compiler.

4 The effect of Parallelism:

Large problems can often be divided into smaller ones, this is the base for parallel programming. We use OpenMP to parallelize our double-loop algorithm using declarative notations and not changing the code. In theory, we would expect 100% increase in performance of a program when the number of threads running the code double. However, in our tests we have experienced 55% (Intel) to 65% (GNU) increase in performance running 2 threads instead of 1, which explains the overhead and the communication time required for running multi-threading. Increasing the number of running threads anywhere above 2 would actually have negative impact on the performance (except for GNU compiler with -O3 optimization, which oddly does a better job running 4 threads instead of 2). The reasoning behind that is that, our test machine's CPU has actually 2 physical cores capable of doing parallel tasks, and although the CPU has 4 logical processors, it still can perform only two jobs at the same time unless we use multitasking by time-sharing on a single-core CPU (which our double-loop algorithm does not).

5 Efficient Matrix-Vector Product, Best Practice:

- Choosing a compiler that exploits the best of both processor features and optimization techniques must be the first choice towards HPC. This option is available without even making changes to the code itself.
- All compiled codes must be optimized. We experienced up to 600% increase in performance enabling compiler optimization. Using this option also requires no change to the code. However we have to do step by step performance optimization. That means that we need to make sure the application correctness before starting the performance tuning and then we always start with -O1 and gradually increase the level of optimization. Higher level optimizations may not cause higher performance and have to be used with caution.
- As a general rule in HPC, always choose standard, mature, optimized libraries over an imperative, customized code for any specific scientific/mathematical task. This option requires changing the code and adds extra dependency to a 3rd party library.
- Parallelism should be used as the last resort and only after we have used all the methods described above. The performance gained by running the code in parallel could be negligible and simply not worth it. In our tests, the best performance we got using parallelism is with Intel compiler, using aggressive optimization running on 2 threads which is almost 900% faster than T0 (using all CPU resources). However with Intel compiler, using aggressive optimization and BLAS routine we get 1100% increase in performance using only a single core. We may not need to parallelize the code and consume all system resources when parallelism has minimum or no performance effect. Using parallelism requires changing the code and might lead to massive and unpredictable consequences if not implemented correctly.

References

[1] GNU: Options That Control Optimization,
https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Optimize-Options.html

[2] Step by Step Performance Optimization with Intel® C++ Compiler,
https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[3] Intel® C++ Compiler in Intel® Parallel Studio XE,
https://software.intel.com/en-us/c-compilers/ipsxe

[4] BLAS (Basic Linear Algebra Subprograms),
http://www.netlib.org/blas/