DOI: 10.15439/2017F230
ISSN 2300-5963 ACSIS, Vol. 11

A Framework for Generating and Evaluating Parallelized Code

Jarosław Bylina
Maria Curie-Sklodowska University
Lublin, Poland
Email: jaroslaw.bylina@umcs.pl

***Abstract*—The work describes a flexible framework built to generate various (parallel) software versions and to benchmark them. The framework is written with the use of the Python language with some support of the gnuplot plotting program. An example of the use of this tool shows the tuning of a matrix factorization on different architectures (Intel Haswell and Intel Knights Corner) with various parameters of parallelization, vectorization, blocking etc.**

I. Introduction

THE OPTIMAL use of the contemporary hardware and software is not an easy and straightforward job. The efficiency and the accuracy of the applications depends on a lot of parameters as: places and manners of parallelization and vectorization, loops' order, block sizes, scheduling, affinity etc. The number of possible (and potentially beneficial) combinations is huge and the choice of the best set of parameters is not always obvious. So, generating different versions, testing and benchmarking them, and then tuning is very time consuming and boring, repetitive task. Thus, it is suitable for automation.

Moreover, the code tuned for one hardware often needs a very different treatment on another machine. The parameters chosen and set for one machine as the most profitable can give a very poor performance of the same code after the change of the memory, the accelerators, or, especially, the central processing unit. Now, the hardware market is full of various parallel machines, processors, and coprocessors. We have multicore architectures (like Intel Haswell — with not too many cores), as well as manycore ones (like Intel Knights Corner and other MIC models) and also very-many-core chips (like various GPU coprocessors). Some of them have hierarchical shared memory — with various sizes and numbers of levels — and vector units — of different sizes. All of them demand a different treatment to acquire the best results in terms of performance and efficiency. On top of that, they can be combined into hybrid machines — which have to be treated differently than their components.

Our framework addresses these problems. It enables developers to rapidly generate and automatically test a lot of versions of the algorithms after a little preparation. It can easily be employed on different architectures and be utilized to find the optimal set of parameters on them.

There is also a number of tools to create parallel versions of an algorithm for various hardware — like OpenMP, MPI, OpenACC, OpenCL and others. Our framework can be used with all of them; although we tested it with the use of OpenMP for now.

The philosophy behind our framework is parametrizing and testing (and tuning) programming units — like functions or classes. The developer provides the template of the unit — with some formal parameters — and some sets of actual parameters with which the function (or class) is to be tested. The software generates all the permitted (by actual parameters) versions of the function (class) and tests them (measuring their computation speed and/or numerical accuracy).

Since our software works on the text of the source code, it is very flexible. We can, for example, enable and disable various directives and pragmas (like OpenMP pragmas or similar), change the sizes of the blocks and also the order of loops. Shortly, any textual parametrization of the investigated function/class can be utilized.

The framework itself is a Python 3 application. However, the source code in any language with separate and named units (like functions in C or functions and classes in C++) can be investigated with it. For now, there are configuration files created to study efficiency and accuracy of the units written in C and C++ and compiled with Intel C++ Compiler (icc) and GNU Compiler Collection (gcc), although it can be easily extended to other languages.

An advantage of such an approach is an automatic generation, compilation, and testing of a large number of versions of the same function/class (or functions/classes) which differ in an organized manner. The output of the software is a set of plots of the desired characteristics, which can be easily compared by the developer. However, we prepare a further facilitation — automatic ordering and selection of the generated versions on the ground of their efficiency and accuracy.

The remainder of this work is following. Section II gives some background of other similar projects. Section III describes the working of our framework. Section IV shows a working example of the testing and tuning with our software. Finally, Section V concludes the work and gives some plans for the future of the project.

II. Related Work

There is a long tradition of software automatic optimization in scientific computing and other computer applications. Thus, there are also a lot of software performing tasks similar to our framework.

One of the approaches to the optimization of the algorithms (or their building blocks) is auto-tuning. It is based on performing many efficiency tests on different versions of building blocks (like BLAS subroutines) and choosing the best for a given architecture. Some examples of the narrow libraries using this approach are ATLAS [10] and FFTW [3]. There are also languages and libraries which employ the approach of auto-tuning to general source codes and use the parameter space (similar to our framework). Their examples are Active Harmony [9], Atune-IL [7], Chapel [2].

On the other hand, we have profilers and similar tools which investigate the code and gather information about the utilization of the architecture and weak points of the implementations. Their examples are PAPI [1], Tau [8], Vampir [5], Scalasca [4], Intel VTune Amplifier [13], Intel Advisor [12].

Finally, there is ELAPS (Experimental Linear Algebra Performance Studies), an interactive multi-platform open source framework [6]. It is designed to build experiments testing dense linear algebra algorithms, functions, libraries. However, that software tests ready subroutines and their combinations and it is very convenient for standard linear algebra building blocks, but it is not very usable for other applications.

III. The Framework Description

Figure 1 shows the workflow of the framework. The dashed arrows represent the steps not requiring the user's intervention (that is, intermediate steps) and the dashed borders represent files not demanding user's direct concern (or even user's view). However, all the work can be repeated from an arbitrary step — for example, after some changes in the configuration.

The flexibility of the framework is provided in two manners — in the preparation of input files and in choosing configuration options.

A. Input files

The first step the user is to make is to prepare input files. The files are source codes of tested units (functions and/or classes) with some of their text parametrized — one file per unit. Both the formal parameters and actual parameters are included in the file. The formal parameters are represented in the code as the Python format strings, that is, `%(name)s` where *name* is an arbitrary name of the parameter. The actual sets of parameter values are given as special comments in the beginning of the file. In C or C++ these must be single-line comments starting with `//`, directly after which there is a character indicating the kind of configuration command.

B. Configuration

There are some configuration options with which we can set other features of the tests. We have, among others:

- the language and the compiler used in tests (C/C++ on icc/gcc for now);
- the compiler options;
- the precision of the computations (like `float`, `double` etc);

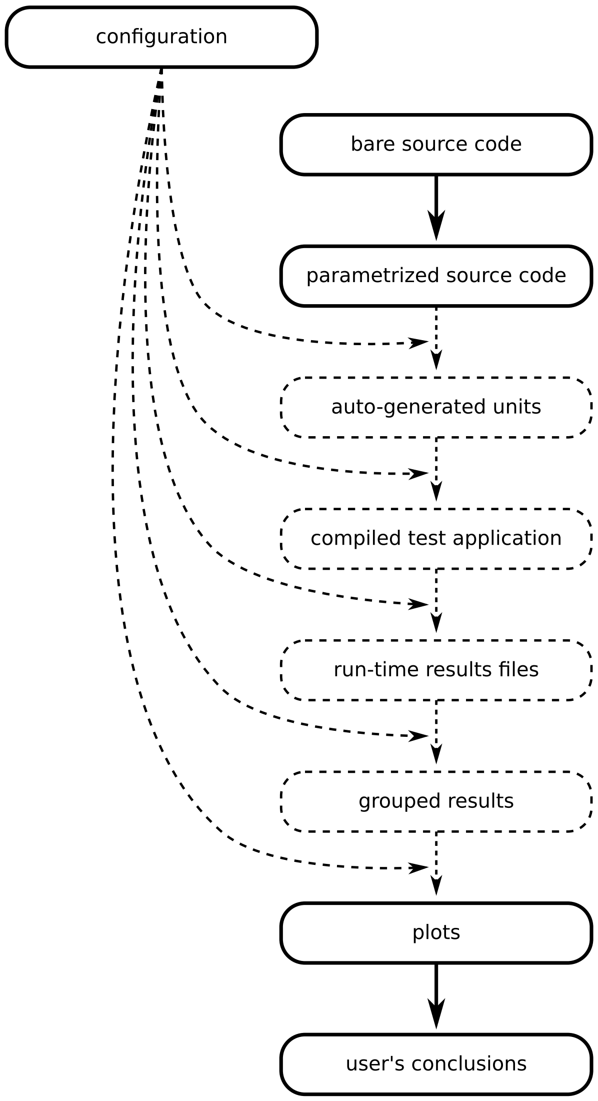

Fig. 1. The framework operation scheme (dashed lines shows the elements processed without the user's awareness)

- the investigated measure (or measures — accuracy, run time and performance for now);
- the number of repetitions of each test (the final value of the performance or accuracy is computed as a mean value from these repetitions);
- the set of the number of threads;
- the set of the problem sizes;
- various parameters of the target plots (groups of plots, ranges etc.).

IV. A Framework Application Example

We show more details on the operation of the framework on an example of parallelizing a numerical problem, namely the WZ factorization [11]. Two sequential versions of this algorithm (in pseudocode) are shown in Figures 2 (the basic version) and 3 (the fission version). In both versions, the matrix a is an input-output data and the matrix w is an output data (the factors of the given matrix a of the size n are stored in matrices a and w after the end of the algorithm).

```
for(k = 0; k < n/2-1; k++) {
    p = n-k-1;
    akk = a[k][k];     akp = a[k][p];
    apk = a[p][k];     app = a[p][p];
    detinv = 1 / (apk*akp - akk*app);
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p]
          - app*a[i][k]) * detinv;
        w[i][p] = (akp*a[i][k]
          - akk*a[i][p]) * detinv;
        for(j = k+1; j < p; j++)
            a[i][j] = a[i][j]
              - w[i][k]*a[k][j]
              - w[i][p]*a[p][j];
    }
}
```

Fig. 2. The pseudocode of the basic WZ factorization algorithm

```
for(k = 0; k < n/2-1; k++) {
    p = n-k-1;
    akk = a[k][k];     akp = a[k][p];
    apk = a[p][k];     app = a[p][p];
    detinv = 1 / (apk*akp - akk*app);
    for(i = k+1; i < p; i++) {
        w[i][k] = (apk*a[i][p]
          - app*a[i][k]) * detinv;
        w[i][p] = (akp*a[i][k]
          - akk*a[i][p]) * detinv;
    }
    for(i = k+1; i < p; i++)
        for(j = k+1; j < p; j++)
            a[i][j] = a[i][j]
              - w[i][k]*a[k][j]
              - w[i][p]*a[p][j];
}
```

Fig. 3. The pseudocode of the fission WZ factorization algorithm

We would like to use our software to parallelize the WZ factorization with the use of the OpenMP standard and to test it on two platforms, namely Intel Haswell (denoted CPU) and Intel Knights Corner (denoted MIC). To achieve that we use our framework, writing our algorithms in C, with some parameters which can help us try various variants and test them on both platforms (CPU and MIC).

The basic versions implemented with the use of our framework is presented in Figure 4, and the fission version — in Figure 5. Here, the matrices are represented in 1D vectors, stored column-wise or row-wise and accessed through the macros INDc and INDr, respectively.

We can see that both are quite straightforward implementations of pseudocodes from previous Figures — apart from first lines (special comments) and % signs (parameters). The meanings of the special comments are following (their order is freeform).

- //= gives the template of the unit (here: function) name. Each file generates a lot of functions, and functions have to possess unique names. We achieve this including parameters into the name template.
- //? describes the header of the function, where a special parameter %s is the name of the function (generated on the basis of the previous line).
- Each //: describes one of the template parameters. Consecutively, we give the name of the parameter (IND, for example) and then its possible values, in two variations each: the first used in the function name (here: col, row — it should be short and adjusted to the function name syntax) and the second used in the function body (here: INDc and INDr, respectively; _ means ‘space’).
- In Figure 5, we can also see //+ which restricts the function names to strings containing the given character sequences (here, we want to test different loop orders, so we use it to disable incorrect loops, that is ii and jj allowing only ij and ji).

A lot of functions were generated, compiled and tested on CPU and MIC for various sizes of the matrix and numbers of threads. The plots for the results were also automatically created.

As we can see, we can quite freely shape our source code and test cases with the use of described above directives. We can easily investigate the influence of

- the matrix storage order,
- OpenMP scheduling,
- vectorization,
- loop order,

and many others — not presented here; however, everything which can be parametrized within the text of the function can be investigated.

V. Conclusion

The aim of our work was to create a software which can support a program developer in his attempts to utilize the hardware, the compiler, and the libraries the best.

The advantage of the framework is generating a lot of versions of parallel (for example, but not only) code. These versions differ in an organized way. Moreover, they are automatically compiled and run, then some measures are taken and plots are drawn. The results in such form can be easily compared by the code developer.

Our framework can be easily used to test not only programs written with the use of OpenMP on CPU and MIC, but it can also be adapted to tests on GPU with the use of CUDA compilers, OpenCL and OpenACC — for example.

Very important — but missing for the present — feature is the next step in the automatic analysis, namely, automatic selection of the best (that is the fastest or the most accurate) sets of parameters. We are working on such a feature to be included in the future versions of the framework.

```
//=wz_bas_%(IND)s_%(sch)s_%(vec)s
//?void %s(int n, double * a, double * w)
//:IND     col INDc          row INDr
//:sch     d10 dynamic,10    d1 dynamic,1       \
           s static    g guided
//:vec     v0 _              v1 #pragma_simd
{
    int p, k, i, j;
    double det;
    for (k = 0; k < n/2-1; k++) {
        p = n-k-1;
        det = a[%(IND)s(p,k)]*a[%(IND)s(k,p)]
            - a[%(IND)s(k,k)]*a[%(IND)s(p,p)];
#pragma omp parallel for default(shared)        \
              private(i, j) schedule(%(sch)s)
        for (i = k+1; i < p; i++) {
            w[%(IND)s(i,k)] =
                (a[%(IND)s(p,k)]*a[%(IND)s(i,p)]
              - a[%(IND)s(p,p)]*a[%(IND)s(i,k)])
              /det;
            w[%(IND)s(i,p)] =
                (a[%(IND)s(k,p)]*a[%(IND)s(i,k)]
              - a[%(IND)s(k,k)]*a[%(IND)s(i,p)])
              /det;
%(vec)s
            for (j = k+1; j < p; j++)
              a[%(IND)s(i,j)] =
                a[%(IND)s(i,j)]
                - w[%(IND)s(i,k)]*a[%(IND)s(k,j)]
                - w[%(IND)s(i,p)]*a[%(IND)s(p,j)];
        }
    }
}
```

Fig. 4. The basic algorithm for the WZ factorization implemented in our framework

References

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl., 14(3):189–204, Aug. 2000.

[2] R. S. Chen and J. K. Hollingsworth. Towards fully automatic autotuning: Leveraging language features of chapel. Int. J. High Perform. Comput. Appl., 27(4):394–402, Nov. 2013.

[3] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. Proceedings of the IEEE, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[4] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. Concurr. Comput.: Pract. Exper., 22(6):702–719, Apr. 2010.

[5] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. Supercomputer, 12:69–80, 1996.

[6] E. Peise, P. Bientinesi. The ELAPS Framework: Experimental Linear Algebra Performance Studies. arXiv:1504.08035, 2015.

[7] C. Schaefer, V. Pankratius, and W. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In H. Sips, D. Epema, and H.-X. Lin, editors, Euro-Par 2009 Parallel Processing, volume 5704 of Lecture Notes in Computer Science, pages 9–20. Springer Berlin Heidelberg, 2009.

[8] S. S. Shende and A. D. Malony. The tau parallel performance system. Int. J. High Perform. Comput. Appl., 20(2):287–311, May 2006.

```
//=wz_fiss_%(i)s%(j)s_%(IND)s_%(sch)s_%(vec)s\
_%(for1v)s_%(for2v)s
//?void %s(int n, double * a, double * w)
//:IND     col INDc    row INDr
//:sch     s static    d10 dynamic,10          \
           d1 dynamic,1     g guided
//:i       i i         j j
//:j       i i         j j
//:vec     v0  _       v1 #pragma_simd
//:for1v   f1v0  _     f1v1 simd
//:for2v   f2v0  _     f2v1 simd
//+wz_fiss_ij
//+wz_fiss_ji
{
    int p, k, i, j;
    double det;
    for (k = 0; k < n/2-1; k++) {
        p = n-k-1;
        det = a[%(IND)s(p,k)]*a[%(IND)s(k,p)]
            - a[%(IND)s(k,k)]*a[%(IND)s(p,p)];
#pragma omp parallel for %(for1v)s             \
    default(shared) private(i) schedule(%(sch)s)
        for (i = k+1; i < p; i++) {
            w[%(IND)s(i,k)] =
                (a[%(IND)s(p,k)]*a[%(IND)s(i,p)]
              - a[%(IND)s(p,p)]*a[%(IND)s(i,k)])
              /det;
            w[%(IND)s(i,p)] =
                (a[%(IND)s(k,p)]*a[%(IND)s(i,k)]
              - a[%(IND)s(k,k)]*a[%(IND)s(i,p)])
              /det;
        }
#pragma omp parallel for %(for2v)s             \
  default(shared) private(i,j) schedule(%(sch)s)
        for (%(i)s = k+1; %(i)s < p; %(i)s++) {
%(vec)s
            for (%(j)s = k+1; %(j)s < p; %(j)s++)
              a[%(IND)s(i,j)] =
                a[%(IND)s(i,j)]
                - w[%(IND)s(i,k)]*a[%(IND)s(k,j)]
                - w[%(IND)s(i,p)]*a[%(IND)s(p,j)];
        }
    }
}
```

Fig. 5. The fission algorithm for the WZ factorization implemented in our framework

[9] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards automated performance tuning. In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[10] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[11] P. Yalamov and D.J. Evans. The WZ matrix factorisation method. Parallel Computing 21 (7), pages 1111–1120, 1995.

[12] `https://software.intel.com/en-us/intel-advisor-xe`

[13] `https://software.intel.com/en-us/intel-vtune-amplifier-xe`