Available on-line at www.prace-ri.eu
Partnership for Advanced Computing in Europe

Auto-tuning of OpenMP Applications on the IBM Blue Gene/Q

Maciej Cytowski[a,*], Maciej Szpindler[a]

[a]Interdisciplinary Centre for Mathematical and Computational Modeling, University of Warsaw, Poland

Abstract

Modern high performance computing architectures are based on multi-core and multi-threaded computing nodes. The mixed MPI and OpenMP programming is currently a reference model for obtaining high scalability on large computing systems. In such a model, MPI processes contain many OpenMP parallel regions. Scalability and performance of those parallel regions may differ between various computing systems and between each run of the code. The control of the number of threads used by different OpenMP regions, by users of the HPC systems, is very often limited to setting a single environment variable - OMP_NUM_THREADS. In this work we present a tool called SOMPARlib which is based on OpenMP Monitoring Interface (POMP) and is capable of controlling the execution of various OpenMP parallel regions introduced in computational codes during run time. The tool is particularly useful in the case of architectures that introduce the multithreading mechanisms like Simultaneous multithreading (SMT) or Hyper-Threading (HT).

1 Introduction

It is predicted that future HPC systems, available by the end of this decade, will be based on computing nodes with hundreds of cores. Additionally each core will provide support for multiple hardware threads, resulting in overall fine-grained concurrency of few thousands per node [1]. As a result, applications that will run on future HPC systems will need to demonstrate very high thread based intra-node scalability.

Many modern HPC applications nowadays use both MPI based and thread based parallelization. Especially mixed MPI and OpenMP model has become the recommended programming model for most novel supercomputing architectures currently available. In this model, MPI processes executed on different computational nodes contain many OpenMP parallel regions. The number of threads used by those regions is very often limited to setting only a single switch, i.e. the OMP_NUM_THREADS environment variable. Moreover, many general purpose processor architectures available nowadays introduce the multithreading mechanisms like Simultaneous multithreading (SMT) or Hyper-Threading (HT). It creates an additional factor that may affect the overall scalability of different OpenMP regions within a single node. In fact, as we have identified in our previous work [2] [3] not all applications and algorithms may benefit from using SMT or HT mechanisms.

The main objective of this work was to create a tool which is capable of selecting appropriate number of threads for each OpenMP parallel region individually. The SOMPARlib tool is designed to support scientific applications whose computations are performed in multiple iterations. It detects all OpenMP parallel regions used in every iteration and performs scalability measurements during run time. Based on the collected results it selects the best parallel mode for each OpenMP region.

The SOMPARlib implementation is based on the OpenMP Monitoring Interface (POMP) proposed by B. Mohr et. al. [4] [5]. Although POMP is not officially part of the OpenMP standard it is available for usage on number of HPC platforms. The IBM compilers for the Blue Gene/Q architecture provide a prototype implementation of some of the POMP functionality.

In Section 2 we describe the design, implementation and functionality of SOMPARlib. We also discuss the timing and memory overhead of the tool. Usage details are discussed in Section 3. Simple examples on how to link the application on Blue Gene/Q system are presented therein. In Section 4 we present a benchmark program which was developed especially for the purpose of performance testing of SOMPARlib on different platforms. The benchmark was executed on IBM Blue Gene/Q architecture. The results of these tests are all collected in Section 4. Finally, in Section 5 we summarize our results and disscuss possible further improvements.

[*]m.cytowski@icm.edu.pl

2 Somparlib - design and implementation

The library in it's current version provides support for codes of a specific computational structure. The main object of our interest are those simulation packages for which computations are divided into a large number of iterations, e.g. molecular dynamics packages, cosmological simulation codes and many others. The required structure of the code is shown in Scheme 1. The main simulation loop may contain numerous OpenMP parallel regions. It may also call an external function (e.g. library calls) which may contain it's own OpenMP parallel regions.

Scheme 1 Structure of the code required by SOMPARlib

```
$iter \leftarrow 0$
...
{main simulation loop}
while $iter \leq niters$ do
    ...
    {OpenMP loop}
    #pragma omp parallel for
    for all loop iterations do
        {body of the parallel loop}
    end for
    ...
    {function call}
    {func1() may contain OpenMP constructs}
    func1()
    ...
    {OpenMP parallel region}
    #pragma omp parallel
    {body of the parallel region}
    ...
    $iter \leftarrow iter + 1$
end while
```

When code is linked against SOMPARlib and executed the following occurs:

1. **SOMPARlib initialization phase**
   - memory required by SOMPARlib is allocated
   - maximum number of threads available for a single process is found
   - parallel modes (i.e. different number of threads used during testing phase) are defined
2. **First iteration - detection phase**
   - structure of the main simulation loop is detected
   - all OpenMP parallel regions within a single iteration are detected
3. **Next N iterations (where $N =$ number of parallel modes) - testing phase**
   - performance of parallel regions in all defined parallel modes is measured
   - the best parallel mode for each of the OpenMP regions is selected
4. **All remaining iterations - computing phase**
   - all remaining calculations are carried out with number of threads set individually for each of the OpenMP parallel regions
5. **SOMPARlib cleanup phase**

During the initialization phase SOMPARlib allocates memory required for storing performance measurements data. The maximum number of threads available for current process is checked by calling the function `omp_get_thread_limit()`. Based on this number the available parallel modes are determined. The first parallel mode is always related with the use of all available threads. If the number of threads assigned to a given parallel mode is divisible by two, the next parallel mode will be selected by dividing the number of threads by two. Otherwise, the number of threads is reduced by one. For example, if the maximum number of threads available to a single process is equal to 64, there will be 6 parallel modes considered during the testing phase, corresponding to 64, 32, 16, 8, 4 and 2 threads per process. However, if the maximum number of threads available to a single process is equal to 24, SOMPARlib will select 5 parallel modes corresponding to 24, 12, 6, 3 and 2 threads per process. The most effective setup is achieved when the maximum number of threads available to a single process is a power of two, which is also a very natural choice for many of today's HPC platforms.

In the next phase, the computations are started and during the first iteration of the main simulation loop all OpenMP parallel regions within the loop need to be detected. SOMPARlib is able to automatically detect a loop structure, but only for simple loops, wherein each OpenMP parallel region is called at most once during each iteration. In such cases the detection of the loop structure ends when SOMPARlib encounters the first OpenMP parallel region again, i.e. in the beginning of the second iteration. In all other cases, the SOMPARlib

Scheme 2 Usage of the SOMPARlib API.

```
$iter \leftarrow 0$
sompar_init()
...
{main simulation loop}
while $iter \leq niters$ do
    sompar_loop_start()
    ...
    {body of the simulation loop}
    ...
    sompar_loop_end()
end while
```

API need to be used in order to mark the beginning and end of the iteration. The usage of the API is shown in Scheme 2.

Next N iterations of the main simulation loop, where N is equal to the number of parallel modes, are used to gather necessary performance measurements of all OpenMP parallel regions in the loop. Starting from the parallel mode with the largest number of threads SOMPARlib measures execution time of all parallel regions. These measurements are repeated for subsequent parallel modes with decreasing number of threads. To prevent a significant slowdown of the testing phase we have decided to implement an additional checkpoint. If in two consecutive parallel modes the execution time of a given OpenMP parallel region increased 1.5 times, then we recognize that this indicates scalability limit has been achieved. Therefore, in such situations we do not continue to search for the best parallel mode for the given OpenMP region, since we expect that consecutive time measurements will be worse than the previous ones.

When appropriate parallel modes have been selected for all regions, the simulation is continued, and before each subsequent OpenMP region the appropriate number of threads is set by calling the `omp_set_num_threads()` function.

SOMPARlib was created primarily as a case study and therefore has some important limitations. In the current implementation, we assume that the amount of work corresponding to each OpenMP parallel region is at the same level in all the iterations. Furthermore, each iteration of the main simulation loop must always contain the same OpenMP parallel regions executed in the same order. Assumptions about the structure of the program are so restrictive mainly due to technical limitations of the POMP interface available on the Blue Gene/Q architecture. This topic is disscussed in the next Section.

When it comes to handling different OpenMP standard functionalities, currently only classical parallel regions and parallel loops are supported. In particular, SOMPARlib does not support OpenMP tasking.

SOMPARlib overhead

The amount of memory allocated by SOMPARlib is of the order of $O(Nmodes \times Nregions \times 16\ bytes)$ per process, where $Nmodes$ is the number of parallel modes to be tested and $Nregions$ is the number of OpenMP parallel regions in a single iteration of the simulation. Therefore SOMPARlib does not create a significant memory overhead.

The main overhead of SOMPARlib is related to the testing phase, when all OpenMP parallel regions are executed with the use of different number of threads. However, testing phase takes only few iterations and therefore the resulting overhead is not significant in the case of long time simulations consisting of a large number of iterations.

2.1 Implementation on the Blue Gene/Q system

The IBM compilers for the Blue Gene/Q architecture provide a prototype implementation of the POMP interface. The compiler automatically instruments the binaries by adding the POMP API calls. Currently, only the functions included in the first version of the POMP interface (POMP1) are supported, of which SOMPARlib uses only the following calls: `POMP_Init`, `POMP_Parallel_enter` and `POMP_Parallel_exit`.

Unfortunately the POMP1 interface does not allow to execute a user-defined code before entering the parallel region. Instead the `POMP_Parallel_enter` is called by the master OpenMP thread at the time of entering a parallel region. This was the main issue during the implementation, since the `omp_set_num_threads()` function, which sets the number of threads for execution of a given OpenMP region, had to be called while exiting the previous OpenMP parallel region. For this reason, the structure of the loop must be identified apriori. This is not an issue in the case of the implementation based on the Opari2 translator. Opari2 is a tool that automatically instrument OpenMP code using POMP2 API calls [12]. Therefore, a question could be raised as to why we have decided to create a special implementation for the Blue Gene/Q architecture. The main reason for this was related to the strong integration of the SOMPARlib with the compiler, which allows for very convienient usage with both programmer-defined code and libraries available on the system. Details on the usage are presented in Section 3.

3 Usage details

3.1 Usage on the Blue Gene/Q system

Usage of the SOMPARlib on the IBM Blue Gene/Q system is rather simple. Recompilation of the code is only necessary in the case of applications that use the SOMPARlib's API. Programs need to be also linked with the library, e.g.:

```
1: $ bgxlc_r -qsmp=omp program.o -o program.x -lxlsmp_pomp -lsompar
```

Thanks to the POMP implementation available in IBM compilers, SOMPARlib is able to control OpenMP parallel regions included within external libraries. For example, in the case of the benchmark program presented in Section 4 we were using shared memory versions of the FFTW and ESSL libraries. SOMPARlib was able to detect and control OpenMP parallel regions defined in both libraries. Most importantly, no additional code modifications or recompilation of those libraries was required. All of this was achieved simply by performing following linking step.

```
1: $ bgxlc_r -qsmp=omp program.o -o program.x \
             -lfftw -lesslsmp -lxlsmp_pomp -lsompar
```

3.2 Debug mode

SOMPARlib can be also used in a debug mode under which it prints informations on the time measurements for a different threading modes selected for each of the OpenMP parallel regions. Figure 1 presents an example log file created under the debug mode.

```
somparlib: starting
somparlib: following 6 configurations will be checked:
64t 32t 16t 8t 4t 2t
somparlib: start of the simulation loop marked
somparlib: end of the simulation loop marked
somparlib: number of OpenMP parallel regions detected: 5
somparlib: meas for pr:0 64t:0.000393s
somparlib: meas for pr:1 64t:0.006385s
somparlib: meas for pr:2 64t:0.006422s
somparlib: meas for pr:3 64t:0.006354s
somparlib: meas for pr:4 64t:0.006363s
somparlib: meas for pr:0 64t:0.000393s
somparlib: meas for pr:1 64t:0.006385s 32t:0.010875s
somparlib: meas for pr:2 64t:0.006422s 32t:0.010901s
somparlib: meas for pr:3 64t:0.006354s 32t:0.010884s
somparlib: meas for pr:4 64t:0.006363s 32t:0.010900s
...
somparlib: meas for pr:0 64t:0.000393s ... 2t:0.000025s
somparlib: meas for pr:1 64t:0.006385s ... 2t:0.167025s
somparlib: meas for pr:2 64t:0.006422s ... 2t:0.165719s
somparlib: meas for pr:3 64t:0.006354s ... 2t:0.165714s
somparlib: meas for pr:4 64t:0.006363s ... 2t:0.165716s
somparlib: best modes for parallel regions:
pr0:m5:t2 pr1:m0:t64 pr2:m0:t64 pr3:m0:t64 pr4:m0:t64
somparlib: exiting
```

Fig. 1. Example log file generated by SOMPARlib in debug mode.

In order to use the debug mode the applications needs to be linked with debug version of the SOMPARlib, i.e. with `libsompar_debug.a` instead of the standard `libsompar.a`. In the case of the Blue Gene/Q system this can be done by performing following linking step:

```
1: $ bgxlc_r -qsmp=omp program.o -o program.x -lxlsmp_pomp -lsompar_debug
```

4 Example of usage

4.1 Benchmark code

Functionality of the SOMPARlib will be shown based on the benchmark program specially written for this purpose. A schematic program diagram is presented in Scheme 3. Our main goal was to illustrate the usefulness and versatility of the SOMPARlib.

Benchmark code is made up of 512 successive iterations each consisting of five steps with different computational footprint. In the first step of the main simulation loop a N-body type computation is carried out. For all of 32768 particles in three-dimensional space distance and interactions between them are calculated. The loop over particles is parallelized with single OpenMP pragma. In the second step, the matrix multiplication is calculated with two square matrices of size 256×256. On the Blue Gene/Q system we use the OpenMP parallelized DGEMM available in the ESSL SMP library. Third step of the benchmark is an OpenMP implementation of a sorting algorithm applied to randomly generated sequence of 1048576 floating point numbers. The implementation is based on the `qsort` function available in the standard C library, which is applied in its thread-safe version to equal subsequences of the original data. The resulting sorted sequences are then merged into the final result. In the fourth step of the benchmark we use the FFTW (v.3.3.2) library compiled with OpenMP support to compute the 3D FFT of a $512 \times 512 \times 512$ grid data. The last step of the main simulation loop is the LU factorization of a sparse matrix. For this purpose we use the SuperLU-MT [9] [10] library and an example sparse matrix Fidapm11 of size 22294×22294 and 623554 non-zero elements obtained from the Matrix Market [11].

Scheme 3 Diagram of the benchmark code

```
$iter \leftarrow 0$
$niters \leftarrow 512$
{main simulation loop}
while $iter \leq niters$ do
    nbody(nparticles=32768)
    mxm(n=256)
    sorting(nelements=1048576)
    fft3d(fft_size=512)
    sparseLU(matrix="fidapm11.rua")
end while
```

4.2 Results on Blue Gene/Q platform

The most important result of presented test was the decrease of overall walltime for benchmark runs. As Table 1 shows, the SMT mechanism is used for almost all parallel regions on the Blue Gene/Q system.

To compare performance we have executed two benchmark runs with one of these runs compiled and controlled by the SOMPARlib. We compare the results obtained with the use of SOMPARlib with standard executions in available SMT modes. On the Blue Gene/Q platform we have executed the program on 16 (SMT1), 32 (SMT2) and 64 (SMT4) threads. The actual number of threads used by the application was chosen by setting the OMP_NUM_THREADS environment variable. In this way we try to mimic a situation where in order to obtain the highest scalability computations are executed with the use of a single MPI process per node and maximum number of threads available within node. Results of this comparison are presented in Table 2.

Table 1. Results of the benchmark test with different parallel modes.

Parallel region ID (step)	**Parallel modes selected by SOMPARlib (# threads / max. threads)**
	Blue Gene/Q
pr:0 (nbody)	4 / 64
pr:1 (nbody)	64 / 64
pr:2 (mxm)	-
pr:3 (mxm)	-
pr:4 (mxm,essl)	32 / 64
pr:5 (sorting)	64 / 64
pr:6 (sorting)	64 / 64
pr:7 (sorting)	16 / 64
pr:8 (fft3d)	64 / 64
pr:9 (fft3d)	64 / 64
pr:10 (fft3d)	16 / 64
pr:11 (fft3d)	64 / 64
pr:12 (fft3d)	64 / 64
pr:13 (sparse)	32 / 64
pr:14 (sparse,essl)	4 / 64

The presented test was created to be an example of a real application, which contain multiple OpenMP regions of different computational footprint. It is not difficult to produce a simple example that demonstrates a much higher increase in performance using SOMPARlib, for instance by introducing short loops which do not scale to large number of threads. However, our main intention was to demonstrate the usefulness of the tool to support real world applications.

Table 2. Overall improvement for different parallel modes tested.

Blue Gene/Q			
Parallel modes	16 (SMT1)	32 (SMT2)	64 (SMT4)
Improvement	49.91%	21.37%	7.62%

5 Summary and future work

In this paper we have presented a tool for automatic runtime tuning of OpenMP regions. We have shown the usefulness of the tool by running example benchmark application on the IBM Blue Gene/Q architecture. We have reported the improvement in performance.

The IBM Blue Gene/Q architecture version of the tool is based on the POMP1 interface implemented within IBM XL compilers. The main advantage of this version of the library is it's extreme ease of usage. Both user defined code and external libraries can use SOMPARlib simply by adding library during the linking step. On the other hand POMP1 interface has limitations, which forced us to impose a very important assumptions about the structure of the code being analysed by the tool. The POMP2 interface overcomes those limitations, however in order to use it we would needed to introduce Opari2 source-to-source translator in the compilation and linking process. This creates difficulties especially when external libraries are being used. One of the biggest advantages of SOMPARlib is that it can support applications written in C/C++ or Fortran and parallelized with pure OpenMP or MPI and OpenMP mixed programming model.

The SOMAPRlib tool would benefit greatly if there were POMP2 interface implementation available within the compiler. Therefore, we strongly support the idea of inclusion of the POMP2 interface to the OpenMP standard. In our future work we will create a new version of SOMPARlib which will be based only on the POMP2 standard. In this version, we will try to remove most of the existing assumptions about the structure of the code. With this change, the tool will be able to support a broader class of applications.

Our tool would also be an interesting addition to the mechanisms of automatic parallelization of loops available in some compilers (e.g IBM XL compilers). By using SOMPARlib, parallelization of broader class of loops could be considered. However, this would require a mechanism for automatic parallelization to produce modified code and objects that uses OpenMP standard.

6 Accessing SOMPARlib

To access SOMPARlib please visit the webpage:
http://software.icm.edu.pl/projects/somparlib/

Acknowledgements

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no RI-283493. The work was achieved using the ICM, University of Warsaw computing resources. Implementation and computations were carried out with the support of the "HPC Infrastructure for Grand Challenges of Science and Engineering", "Centre for Preclinical Research and Technology" and "PL-Grid" projects, co-financed by the European Regional Development Fund under the Innovative Economy Operational Programme.

References

1. The International Exascale Software Project Roadmap, J. Dongarra et. al., *Int. J. High Perform. Comput. Appl.*, 25(1): 3-60 (2011)
2. Performance Analysis of Parallel Applications on Modern Multithreaded Processor Architectures, M. Cytowski, M. Filocha, J. Katarzynski and M. Szpindler, *PRACE Whitepaper*, available on-line at www.prace-ri.eu (2013)
3. Towards Autotuning of OpenMP Applications on Multicore Architectures, J. Katarzynski, M. Cytowski, *CoRR*, abs/1401.4063 (2014), http://arxiv.org/abs/1401.4063
4. A Performance Monitoring Interface for OpenMP, B. Mohr, A. Mallony, H-C. Hoppe, F. Schlimbach, G. Haab, and S. Shah, *In Proceedings of the fourth European Workshop on OpenMP - EWOMP'02*, September 2002
5. Design and Prototype of a Performance Tool Interface for OpenMP, B. Mohr, A. D. Malony, S. Shende and F. Wolf, *The Journal of Supercomputing*, 23(1): 105-128 (2002)
6. The Scalasca Performance Toolset Architecture, M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker and B. Mohr, *Concurr. Comput. : Pract. Exper.*, 22(6): 702-719 (2010)
7. VAMPIR: Visualization and Analysis of MPI Resources, W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe and K. Solchenbach, *Supercomputer*, 12: 69-80, (1996)

8. P. W. Coteus Packaging the IBM Blue Gene/Q supercomputer. *IBM Journal of Research and Development*, **57**: 2:1–2:13, 2013.
9. An Overview of SuperLU: Algorithms, Implementation and User Interface, Xiaoye S. Li, *Transactions on Mathematical Software*, 31(3): 302-325, 2005
10. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, J. W. Demmel, J. R. Gilbert and Xiaoye S. Li, *SIAM J. Matrix Analysis and Applications*, 20(4):915-952, 1999
11. Martrix Market webpage, *http://math.nist.gov/MatrixMarket*
12. Opari2 User Manual, *http://www.vi-hps.org/projects/score-p/*, version 1.1.1, (revision 1150)