D5.1– Report on performance and tuning of runtime libraries for ARM Architecture
Version 1.0

Document Information

Contract Number	288777
Project Website	www.montblanc-project.eu
Contractual Deadline	M12
Dissemintation Level	PU
Nature	Report
Coordinator	Alex Ramirez (BSC)
Contributors	Isaac Gelado (BSC), Xavier Martorell (BSC), Judit Gimenez (BSC), Harald Servat (BSC), Bernd Mohr (JUELICH), Daniel Lorenz (JUELICH), Marc Schluetter (JUELICH)
Reviewers	Chris Adeniyi-Jones (ARM)
Keywords	compiler, runtime system, parallelism, instrumentation, performance analysis

Notices: *The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 288777*

Change Log

Version	Description of Change
v0.1	Initial draft released to the WP5 contributors
v0.2	Draft version sent to internal reviewer
v1.0	Final version sent to the EU

Contents

Executive Summary

In this Mont-Blanc deliverable we present the current status of porting to the ARM architecture of the OmpSs (Mercurium compiler and Nanos++ runtime system), the Extrae instrumentation library and the Scalasca instrumentation facilities. In addition, we present an initial evaluation of the overhead observed in the OmpSs programming model when using Extrae instrumentation in the Intel architecture.

1 Introduction

The Mont-Blanc project aims to build the first supercomputer based on low-power pro cessors based on the ARM architecture. Traditionally, those ARM processors have been used in mobile devices and embedded platforms, and not in high performance computing (HPC) systems.

In this document we present the work done on the porting of the Mercurium and Nanos++ infrastructure to support the OmpSs programming model on the ARM architecture. We have also ported the instrumentation facilities of Extrae and Scalasca, to support the same kind of analysis we have in HPC systems, to the Mont-Blanc architecture.

Finally, we have also evaluated the overhead of the Nanos++ and instrumentation facilities, currently on the Intel architecture. Those results will be used as a reference while the evaluation and tuning of the infrastructure on ARM.

2 Runtime Libraries

In this section, we describe the steps followed to port the Mercurium compiler, the Nanos++ runtime system, the Extrae instrumentation library, and the Scalasca instrumentation environment to run on the ARM architecture.

2.1 Mercurium compiler

In the context of the Mont-Blanc project, the Mercurium compiler has been ported to the ARM-v7 architecture, and it is being modified to support Fortran 95.

2.1.1 Porting to ARM-v7

The Mercurium compiler, being a source–to–source code transformation tool has a small number of dependencies on hardware, compared to a compiler that should generate machine code. Nevertheless, it is important to have the proper definitions of sizes for the basic scalar and pointer types for the target architecture.

For this purpose, the ARM environment has been incorporated into Mercurium. Table 1 shows the properties defined on it.

Type	Size	Alignment
bool	1	1
signed short	2	2
unsigned short	2	2
wchar(c99)	4	4
signed int	4	4
unsigned int	4	4
signed long	4	4
unsigned long	4	4
signed long long	8	8
unsigned long long	8	8
half float	2	2
float	4	4
double	8	8
long double	16	16
float128	16	16
pointer to data	4	4
pointer to function	4	4
pointer to data member	4	4
pointer to member function	8	4
builtin variable arg. list	4	4

Table 1: ARM environment incorporated into Mercurium.

2.1.2 Fortran support

The internal representation of the Mercurium compiler has been adapted to support the Fortran 95 language. We have identified the common representation that can be shared with the C

OmpSs construct	Support
omp parallel	ok
omp for	ok
omp task	ok
omp threadprivate	no
in/out/inout task clauses	ok

Table 2: OmpSs support in Mercurium Fortran.

and C++ languages, and we have added the necessary data structures to represent Fortran-only statements. These are mostly, specific Fortran I/O sentences, intrinsic functions, support for modules, and specific Fortran attributes for symbols, like belonging to COMMON blocks. Fortran support is being tried with the applications provided by the end users (BigDFT, Euterpe, MP2C...), to ensure that Mercurium can compile them.

Regarding the support of OmpSs in Fortran, Table 2 shows the current status.

The support for *threadprivate* is not planned in the context of the Mont-Blanc project, as the source–to–source code transformation in Fortran does not allow us to represent *threadprivate* variables. The reason is that in Fortran there is no way to express that a variable has to be allocated onto the *thread local storage* data segment (that is achieved with the _ *thread* attribute, in C/C++).

2.2 Nanos++

The Nanos++ runtime system was already running on the Linux OS for the Intel *x86*, *x86_-64*, *ia64*, the IBM *PowerPC* (*ppc32*, *ppc64*), and the Tilera *Tile64PRO* architectures. In the context of Mont-Blanc, the library has been ported to the ARM architecture, by writing the machine–dependent part. This consists of three routines dealing with the machine register of the ARM-v7 architecture:

- switchStacks(arg0, arg1, newsp, helper), in arch/smp/armv71/stack.s. This routine has the purpose of switching the execution between two user–level threads. It saves the general purpose registers of the processor in the current stack, changes the stack pointer to be *newsp*, and calls the helper function, in order to free resources from the previous thread. It finally restores the general purpose register from the new stack, and it returns to the new thread.

- startHelper(userArg, userFunction, cleanupArg, cleanup), in arch/smp/armv71/stack.s. This function ensures that after invoking the user function, the cleanup function is called, to clean the environment left by the old thread. Usually, the exit() function of the current scheduling policy is used for this purpose.

- initContext(stack, stackSize, userFunction, userArg, cleanup, cleanupArg), in arch/smp/armv71/stack.cpp. This function initializes the context of a new user–level thread in the stack provided. This initialization makes the new thread start by running the userFunction, with the provided argument userArg. The cleanup function and its parameter will be invoked afterwards to cleanup the environment of the thread upon termination. The startHelper function is used to ensure the execution of userFunction first, and then the cleanup function.

2.3 Extrae instrumentation library

We ported the instrumentation library (Extrae) when the prototype of Tibidabo was accesible although not all the functionalities of Extrae are currently available. This is because some of the packages on which Extrae depends on are not available yet in the ARM architecture:

- DynInst, a library that allows dynamic instrumentation of binaries, is not ported into ARM. Despite DynInst is not necessary to instrument applications, it would allow instrumenting user routines without changing the source code of the application.
- The libunwind, which is responsible for tracking the code location at MPI calls or sampling points, is under development. Once these libraries become available, we would recompile Extrae to use them.

During this period we have used Extrae to instrument MPI (either using MPICH or Open-MPI), OpenMP (using the GNU implementation), OmpSs and pthread applications by linking them with the Extrae libraries or by using the LD_PRELOAD mechanism. Extrae also benefits from PAPI, a library that accesses the hardware performance counters, to enrich the tracefile and provide insightful information regarding the application execution.

2.4 Scalasca support for the ARM platform

In the context of Mont-Blanc, the Jülich Supercomputing Centre will extend Scalasca to support the ARM platform as well as the OmpSs programming model. Because the new community measurement Score-P will replace Scalasca's native instrumentation and measurement system it implies porting Score-P to the ARM platform and implement OmpSs support in Score-P.

Scalasca is a free software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XE, but is also well suited for small- and medium-scale HPC platforms. The analysis identifies potential performance bottlenecks - in particular those concerning communication and synchronization - and offers guidance in exploring their causes. The user of Scalasca can choose between two different analysis modes: (i) performance overview on the call-path level via profiling and (ii) the analysis of wait-state formation via event tracing. Wait states often occur in the wake of load imbalance and are serious obstacles to achieving satisfactory performance. Performance-analysis results are presented to the user in an interactive explorer called Cube (Figure) that allows the investigation of the performance behavior on different levels of granularity along the dimensions performance problem, call path, and process. The software has been installed at numerous sites in the world and has been successfully used to optimize academic and industrial simulation codes.

Score-P is a common instrumentation and measurement infrastructure for jointly developed by the developers of the performance analysis tools of:

- Scalasca from Forschungszentrum Jülich, Germany and the German Research School for Simulation Sciences, Aachen, Germany,
- Vampir at Technische Universität Dresden, Germany,
- Persicope group at Technische Universität München, Germany, and
- TAU at University of Oregon, Eugene, USA.

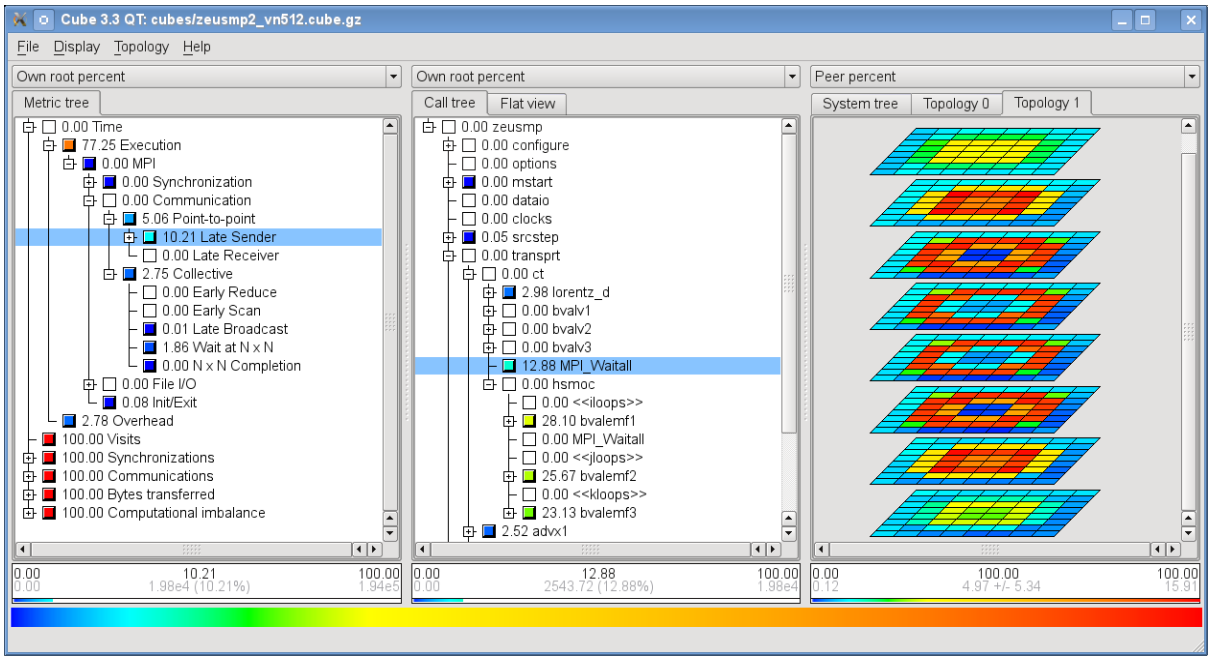


Figure 1: Interactive exploration of performance behavior in Scalasca along the dimensions performance metric (left), call tree (middle), and process topology (right).

Further partners are:

- the Computing Center at RWTH Aachen, Germany, and
- the GNS mbH, Braunschweig, Germany, as industry partner.

The advantages of a common measurement system are:

- Interoperability between the tools: The common infrastructure implies the usage of common data models. A user does not need to learn multiple tools usage, and perform multiple builds and measurement runs in order to analyze his code with different tool, But instrument and measure his application only once and then can use the measurement with all available tools.
- Before using Score-P every tools group had to implement its own instrumentation and measurement framework. Thus, a lot of work was spent to implement similar functionality. When using a common measurement framework, the individual tools groups save resources for the development of the instrumentation and measurement system.

Score-P supports serial, MPI and OpenMP applications and also the combination of MPI and OpenMP. CUDA support is already implemented and awaits its release. The Score-P instrumenter can insert instrumentation with various methods, e.g. compiler instrumentation, MPI library wrapping, source-to-source instrumentation by OPARI2 and the TAU instrumentor, and manual user instrumentation.

To enable Scalasca to be used in conjunction with the Score-P community measurement system, its automatic trace analysis component has to support event traces in OTF2 format, which is ongoing work. Currently, a working prototype providing the same functionality as Scalasca 1.x is available for internal testing, and will subsequently be enhanced with additional trace analysis capabilities.

The instrumentation system Score-P has already been ported to the ARM platform. This required extensions of the build system to detect the platform and set appropriate defaults for the configuration. Furthermore, to enable the compiler-based instrumentation with the gcc compiler on ARM, we needed to extend the implementation of the compiler adapter in the Score-P measurement system. When enabling compiler instrumentation, the gcc compiler inserts function calls at enter/exit points of every function and passed the function address as a parameter to these function. The compiler adapter of the Score-P measurement system implements these functions. In order to identify the function and associate them with information that is meaningful to the user, we read debug information for the given function address. However, on ARM platforms the function addresses provided to the compiler function call are sometimes off by one, because the gcc compiler uses the least significant bit to encode information whether the function is compiled to thumb code or ARM binary code. Thus, we introduced a special handling for the least significant bit of the function address on ARM platforms to make the compiler instrumentation work.

The OmpSs support for Scalasca consists of two components. The first part is a plug-in to the OmpSs instrumentation interface and the second part integrates the OmpSs events into the Score-P task model. The shared build approach will allow for the use of different plug-ins for the same instrumented application if necessary. The entire process can be described as a translation from the OmpSs event model into the region based Score-P task model, which will also be used for other tasking approaches. The effort to support the OmpSs model in Score-P is still in progress. Currently, an early instrumentation plug-in has been created, using an initial set of OmpSs events to define corresponding Score-P events enabling the creation of task regions within the Score-P measurement system. First results can be viewed in the CUBE result browser. The ongoing efforts include realization of the complete OmpSs event model and integration of the OmpSs GPU support.

3 Performance Analysis and Tuning

Meanwhile we have ported Nanos++ and Mercurium to work on ARM, we have also evaluated the overhead of Nanos++ and our Extrae instrumentation facilities on the Intel architecture. All the experiments described in this section have been executed on a system containing four hexa-core Intel® Xeon® E7450 processors running at 2.4GHz and 48 GBytes of RAM. The applications have been compiled using a development branch of the Mercurium version 1.3.5.8 and Nanos++ version 0.7a, which relies on the GNU C compiler version 4.3.4, using `-O3` as optimization flags.

In order to analyze the overhead of our instrumentation facilities, we have chosen some benchmarks from the Barcelona OpenMP Task Suite (BOTS [D+09]), and the Hydro benchmark. The BOTS benchmarks are based on OpenMP tasks and worksharing constructs. They include a cut-off mechanism that is used to avoid the generation of too many levels of tasks, and handle task granularity. When the cut-off level is reached, tasks are serialized. The benchmarks description is as follows:

- *SparseLU* computes a LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to smaller submatrices (blocks) that may not be allocated. In each of the SparseLU phases, a task is created for each block of the matrix that is allocated. We have used a matrix of $50 * 50$ blocks, and several block sizes from $25 * 25$ up to $200 * 200$. Due to space restrictions, we show the results for blocks of $50 * 50$ and $200 * 200$.

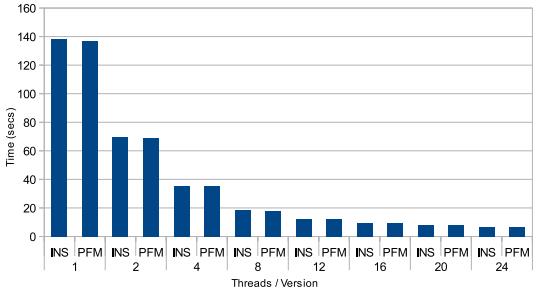
- *Floorplan* kernel computes the optimal floorplan distribution of a number of cells (each cell with its own shape description) which has been provided as a parameter. The algorithm calculates the minimum area size which includes all cells through a recursive branch and bound search. Floorplan application implements a cut-off based on the depth of the tree. In our experiments we have tested 20 cells input with three cut-off values (4, 5 and 6).

- *NQueens* computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an $n * n$ chessboard such that none of the queens attack any other. The benchmark uses a backtracking search algorithm with pruning, creating a task for each step of the solution. The algorithm has a cut-off mechanism based on the recursion level of the kernel. In our experiments we have tested a chessboard of $13 * 13$ and four cut-off values (2, 3, 4 and 5).

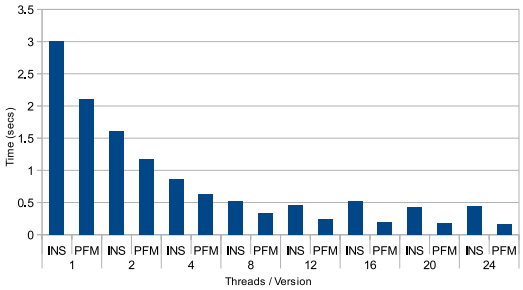
- *Strassen* algorithm uses hierarchical decomposition to multiply large dense matrices. Decomposition is done by dividing each dimension of the matrix into two sections of equal size and a task is created for each decomposition step. In our experiments we have used a matrix $4K * 4K$ of complex numbers and three cut-off levels (3, 4 and 5).

- *Hydro* is a proxy benchmark of the RAMSES [RAM] application, that solves a large scale structure and galaxy formation problem using a rectangular 2D space domain split in blocks. It combines OmpSs tasking constructs with data dependencies and MPI. The application input requires to execute using 4 MPI processes and works with matrices of $2K * 2K$ elements split in blocks of $256 * 256$ elements. We have compiled the application using OpenMPI version 1.4.4.

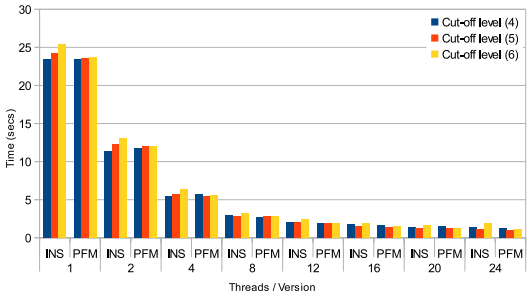
Figure 2 shows the results obtained when executing the aforementioned BOTS benchmarks and Hydro. The Figure shows the comparison of the execution times obtained in the instrumented (INS), and the performance (PFM) versions. The X-axis shows the number of threads



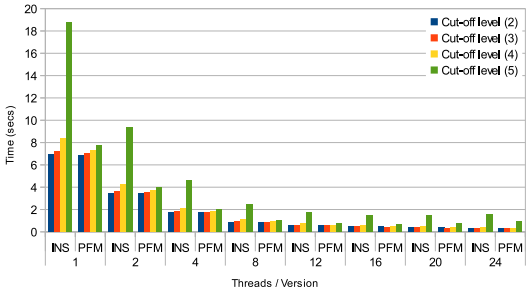
(a) SparseLU (M=50, N=200).



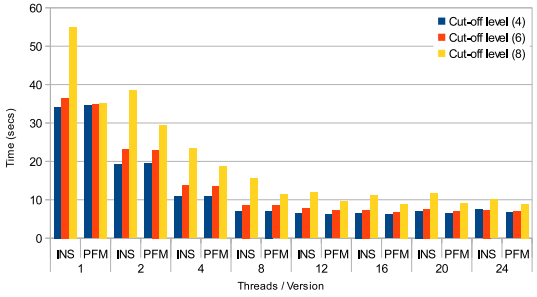
(b) SparseLU (M=50, N=50).



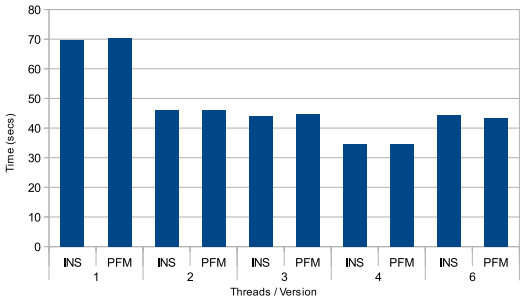
(c) Floorplan (20 cells).



(d) NQueens $13 * 13$ cheessboard.



(e) Strassen $4K * 4K$ matrix.



(f) Hydro $2K * 2K$ matrix with block size of $256 * 256$ elements.

Figure 2: Execution time for instrumented (INS) and performance (PFM) version of the different benchmarks.

and also the runtime version, while the Y-axis shows the execution time. The plots also show the execution time with different cut-offs where applicable.

In all cases, the reader can see that the instrumented version adds some overhead with respect the performance one. Overhead depends on the task granularity, which is also related to cut-off values as in Floorplan, NQueens and Strassen, or block size like in SparseLU. While on large task granularity the overhead is less than 1% we find that reducing the task granularity increases the overhead. For example in Figure 2(b) we see that the overhead for SparseLU is about 43% with 1 thread, in Figure 2(e) we observe up to a 56% and Figure 2(d) shows the largest overhead by using 1 thread (243%).

In order to understand the reasons of the large overhead experienced in the NQueens benchmark, we have executed an instrumented version of the benchmark using one thread with the different cut-off values and adding hardware performance counters metrics into the tracefile. Table 3 shows on each row the TLB miss ratio analysis (i.e. number of TLB misses per instruc-

Cut-off	.00−.01	.01−.02	.02−.03	.03−.04	.04−.05	.05−.06	.06−.07	.07−.08	.08−.09	.09−.10
2	99.99%	0.01%								
3	99.99%	0.01%								
4	99.99%	0.01%								
5	44.02%	8.51%	8.55%	10.76%	11.52%	6.94%	4.30%	2.75%	1.69%	0.96%

Table 3: TLB miss ratio of NQueens with different cut-off values.

tion) for each cut-off value shown in Figure 2(d). Each column in the table represents the TLB miss ratios ranging from 0 to 0.1 in buckets of 0.01. The value in each cell is the percentage of time that the thread has been experiencing that TLB miss ratio. Using a cut-off level of 2 we observe that 99.99% of the total time the number of TLB misses has been between 0 and 0.01, whereas the remaining time a TLB miss ratio between 0.01 and 0.02. By comparing cut-off levels between 2 and 4 we do not observe a fluctuation on this behavior. This is no longer true when the cut-off value is set to 5 where 44.02% of the total time experiences a TLB miss ratio between 0 and 0.01. More than 25% of the execution time shows a TLB miss ratio between 0.03 and 0.06. This large proportion of TLB misses per instruction is clearly increasing the overhead of the instrumentation. Our preliminary analysis of the situation indicates that the issue may appear because the large number of tasks created consumes more memory pages because of the generation of the events.

For Hydro, Figure 2(f) shows that instrumentation adds a minimal overhead, typically close to 1% and that the application does not scale linearly with respect to the number of threads/process used. At 6 threads/process, the application has a performance penalty. Further analysis of the execution of Hydro has revealed that dependencies limit the number of tasks that can be executed in parallel inside each process to 4, being this the cause for the limitation found in the performance.

The results obtained in the evaluation of the overhead of Nanos++ on the Intel architecture, make us confident that the infrastructure is ready to be evaluated on ARM.

4 Conclusions

In this document, we have presented the current status of the porting of the Mercurium compiler, the Nanos++ runtime system, and the Extrae and Scalasca support for instrumentation to the ARM architecture.

At the same time, we have performed an initial evaluation of the overhead introduced by the instrumentation facilities on the execution of parallel applications on the Intel architecture.

The results obtained in this evaluation suggest that the infrastructure is ready to be evaluated on the ARM architecture. We will use the results obtained on Intel as a reference for comparison with the new results obtained on ARM, to detect performance penalties that may have been introduced during the porting. If performance penalties are detected, we will proceed to correct them.

References

[D+09] Alejandro Duran et al. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP'09. International Conference on Parallel Processing*, pages 124–131. IEEE, 2009. `https://pm.bsc.es/projects/bots` - Accessed May, 2012.

[RAM] RAMSES. `http://web.me.com/romain.teyssier/Site/RAMSES.html` - Accessed May, 2012.