MB3 D7.16 – Final report on runtime support, optimization and programming productivity for compute accelerator and symbiotic cores

Version 1.0

Document Information

<table>
<thead>
<tr>
<th>Contract Number</th>
<th>671697</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td><a href="http://www.montblanc-project.eu">www.montblanc-project.eu</a></td>
</tr>
<tr>
<td>Contractual Deadline</td>
<td>PM39</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>PU</td>
</tr>
<tr>
<td>Nature</td>
<td>Report</td>
</tr>
<tr>
<td>Authors</td>
<td>Xavier Martorell (BSC)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Marc Casas (BSC), Kallia Chronaki (BSC)</td>
</tr>
<tr>
<td>Reviewers</td>
<td>Miguel Tairum Cruz (Arm), Leonardo Fialho (Atos)</td>
</tr>
<tr>
<td>Keywords</td>
<td>Scheduling for big.LITTLE, Heterogeneous computing, OmpSs programming model</td>
</tr>
</tbody>
</table>

Notices: This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 671697.

©Mont-Blanc 3 Consortium Partners. All rights reserved.
## Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.1</td>
<td>31/08/2018. Initialized git with template deliverable</td>
</tr>
<tr>
<td>v0.2</td>
<td>19/10/2018. Contribution on symbiotic cores</td>
</tr>
<tr>
<td>v0.8</td>
<td>17/12/2018. Final updates on symbiotic cores</td>
</tr>
<tr>
<td>v1.0</td>
<td>21/12/2018. Integrated internal comments from reviewers</td>
</tr>
</tbody>
</table>
# Contents

**Executive Summary**  

1. **Introduction**  

2. **Runtime for compute accelerators**  

3. **Runtime support for symbiotic heterogeneous cores**  
   3.1 Experiments on big-LITTLE Odroid Xu3  
   3.1.1 Execution environment  
   3.1.2 Evaluation on fixed number of cores  
   3.2 Adding Little Cores to a symmetric multi-core  
   3.3 Experiments on Dibona (pre-production nodes)  
   3.4 Experiments on Dibona (production nodes)  
   3.4.1 Matrix multiplication  
   3.4.2 BlackScholes  
   3.4.3 Fluidanimate  

4. **Conclusions and Future Work**  

**Acronyms and Abbreviations**
Executive Summary

This document reports the activities planned in Mont-Blanc 3 WP7 under tasks T7.2.5:

<table>
<thead>
<tr>
<th>T7.2.5 Runtime support for symbiotic heterogeneous cores (m1:m36)</th>
</tr>
</thead>
<tbody>
<tr>
<td>We will provide runtime support and extensions within OmpSs for exploiting the symbiotic relationship of heterogeneous cores explored in T4.1. During the first year of the project we will leverage existing platforms with heterogeneous cores, and then move and continue those developments to the simulation models used for heterogeneous cores.</td>
</tr>
</tbody>
</table>

This document presents the work done on the OmpSs programming model to support the compute accelerator (the Arm Scalable Vector Extension (SVE)), and to provide an OmpSs task scheduler oriented to the big.LITTLE architecture. It is based on a previous work done in the Intel architecture, and now ported to Arm. Experiments on an actual ODROID-XU3 hardware, with 4 big and 4 LITTLE cores show no penalty of the new task scheduler implementation. Task-based scheduling improves static threading performance by 23% on average. Experiments on a Dibona node show the performance obtained in Cholesky, STREAM, Matrix Multiplication, BlackScholes, and FluidAnimate. On Matrix Multiplication, the increase in performance when using the DDAST policy is 8%.
1 Introduction

In the Mont-Blanc 3 project, we have developed several extensions to the OmpSs programming model, regarding the use of the compute accelerator and the use of heterogeneous cores.

The structure of the deliverable is as follows. In Section 2, we show how to use the Arm Scalable Vector Extension (SVE) with OmpSs applications. As of today, there is no chip implementing SVE, we use the Arm Instruction Emulator to check that the OmpSs benchmarks run successfully.

In Section 3, we demonstrate how the Odroid XU3 and the Dibona nodes can run OmpSs applications with scheduling support for heterogenous cores, and the performance obtained.

Finally, in Section 4, we show the conclusions of this work, and our future work.

2 Runtime for compute accelerators

The Arm compute accelerator consists of the new set of Arm vector instructions, known as Arm Scalable Vector Extension (Arm SVE) [1, 2]. As such, our OmpSs programming model can leverage its use through the vectorization of the tasks’ code.

As no actual chip currently supports SVE, we have used the Arm Instruction Emulator [3] to run the vectorized OmpSs applications and obtain the statistics about the number of SVE instructions executed during application runtime.

Code vectorization supporting SVE is currently implemented in GCC version 8 and the Arm HPC Compiler version 18.x. We have used GCC 8 as the backend for OmpSs, to generate the vectorized SVE version of the OmpSs tasks.

We have used the STREAM benchmark to test that OmpSs is actually able to support the execution of applications using SVE. We have executed this benchmark, generated with SVE instructions, in a Dibona node, with the Arm Instruction Emulator 18.4. The emulator supports threaded applications, and this allowed to run the parallel version of STREAM, with 1 to 64 cores. For these executions, the number of elements in the STREAM vectors is 128 million double precision values, for a complete STREAM data set size of 3GB. We have explored SVE vector lengths from 128 to 2048 bits, that is from 16 to 256 double precision values.

The emulator plugin used to gather statistics is counting the number of SVE emulated instructions. For the STREAM benchmark executions, the numbers reported range between 7.2 billion SVE instructions when executing with 1 thread, and 128 – bit vector length, down to 11 million SVE instructions when executing with 64 threads, and 2048 – bit vector length.

Figure 1 shows how the amount of SVE emulated instructions evolves when running STREAM, from 1 to 64 cores in a Dibona node, and also when varying the vector length.

Increasing the number of cores, the number of SVE instructions emulated decreases. We think that this is caused by the fact that the emulation only reports the instructions emulated on a single thread. When increasing the vector length, we can observe how the number of SVE instructions halves. This is the expected behaviour, as now each instruction processes double the number of elements.
3 Runtime support for symbiotic heterogeneous cores

3.1 Experiments on big-LITTLE Odroid XU3

We have evaluated the OmpSs runtime targeting big-LITTLE architectures with new benchmarks from the PARSEC benchmark suite [4]. This suite includes parallel applications from multiple domains such as finance, computer vision, physics, image processing and video encoding. We quantify the performance loss of executing the applications as-is on all cores in the system. These applications were developed on homogeneous platforms and are bound to suffer from load imbalance on parallel regions that statically distribute the work evenly across cores without considering their performance differences.

3.1.1 Execution environment

The experiments in this section have been executed on the Hardkernel Odroid XU3. This processor implements the Arm big.LITTLE architecture [5, 6], a state-of-the-art Asymmetric multi-core (AMC) architecture. The Arm big.LITTLE combines simple in-order cores with aggressive out-of-order cores in the same System-on-Chip (SoC) to provide high performance and low power. Big and little cores support the same architecture so they can run the same binaries and, therefore, easily allow thread migration within the same system.

As shown in Figure 2, the Odroid-XU3 includes an 8-core Samsung Exynos 5422 chip with four Arm Cortex-A15 cores and four Cortex-A7 cores. The four Cortex-A15 share a 2 MB 16-way 64-byte-cache-line L2 cache, while the Cortex-A7 cores share a 512 KB L2 cache. A single memory controller provides access to 2 GB of LPDDR3 RAM with dual 32-bit channels at 1866 MT/s.

The Cortex-A7 cores in this SoC support dual-issue of instructions and their pipeline length is between 8 and 10 stages. The L1 instruction cache is 32KB two-way set associative, with virtually indexed and physically tagged cache-lines that can hold up to 8 instructions. The core supports instruction prefetch by predicting the outcome of branches; the prefetch unit can fetch up to a maximum of four instructions per cycle. The L1 data cache is four-way set associative with physically-indexed and physically-tagged cache lines and uses a pseudo-random...
replacement policy [7]. Dynamic Voltage and Frequency Scaling (DVFS) techniques adjust the frequency of the little cores from 200MHz up to 1.4GHz.

The Cortex-A15 cores in this SoC support triple-issue of instructions and their pipeline length is between 15 and 24 stages [8]. The L1 instruction and data caches of the Cortex-A15 are both 32 KB and 2-way set-associative with 64 byte cache lines. The processor supports speculative instruction execution by maintaining a 2-level global history-based dynamic predictor with a branch target buffer [9]. The instruction decode unit performs register renaming to remove the Write-After-Write and the Write-After-Read hazards, and promote instruction reordering [9]. The instruction dispatch unit analyzes instruction dependences before issuing them for execution. The integer execute unit includes 2 Arithmetic Logical Units with support for operand forwarding. DVFS techniques vary the frequency of the big cores from 200 MHz up to 2 GHz. To avoid machine overheating, we make use of the cpufreq driver to set big cores at 1.6GHz and little cores at 800MHz.

### 3.1.2 Evaluation on fixed number of cores

We measure execution time, power, energy and EDP of nine applications from the PARSEC benchmark suite [10]. Table 1 shows their main characteristics.

We compare these metrics for three different scheduling approaches:
- **Static threading**: scheduling decisions are made at the application level. The OS is not allowed to migrate threads between the clusters of big and little cores.

- **GTS**: dynamic coarse-grained OS scheduling using the GTS scheduler integrated in the Linux kernel [5][11] using the default PARSEC benchmarks.

- **Task-based**: dynamic fine-grained scheduling at the runtime level with the task-based implementations of the benchmarks provided in PARSECs [12].

Figure 3: Execution time speedup over 1 little core for systems that consist of 4 cores in total with 0, 2 and 4 big cores. Different schedulers at the application (static threading), OS (GTS) and runtime (task-based) levels are considered.

Figure 4: Average power measurements on a 4-core system with 0, 2, and 4 big cores.

First, we examine the opportunities and challenges that current AMCs offer to emerging parallel applications. With this objective, we first evaluate a system with a constant number of four cores, changing the level of asymmetry to evaluate the characteristics of each configuration. In these experiments, all applications run with the original parallelization strategy that relies on the user to balance the application (Static threading). We also evaluate the OS-based dynamic scheduling (GTS) and the task-based runtime dynamic scheduling (Task-based) for the same applications. The system configurations evaluated in this section are: i) Four little cores (0+4); ii) Two big and two little cores (2+2); and iii) Four big cores (4+0).

For these configurations, Figure 5 shows the speedup of the PARSEC benchmarks with respect to running on a single little core. Figure 4 reports the average power dissipated on the evaluated platform. Finally, Figure 5 shows the total energy consumed per application for the same configurations. Energy results are normalized to the energy measured with four little cores (higher values imply higher energy consumptions). Average EDP results are also included in this figure.
Focusing on the average performance results, we notice that all approaches perform similarly for the homogeneous configurations. Specifically, applications obtain the best performance on the configuration $4+0$, with an average speedup of $9.5 \times$ over one little core. When using four little cores, an average speedup of $3.8 \times$ is reached for all approaches. This shows that all the scheduling approaches are effective for the homogeneous configurations. In the $2+2$ configuration, Static threading slightly improves performance ($5.0 \times$ speedup), while GTS and Task-based reach significantly higher average speedups: $5.9 \times$ and $6.8 \times$, respectively.

Contrarily, in terms of power and energy, the most efficient configuration is running with four little cores, as the performance ratio between the different cores is inversely proportional to the power ratio $[\cdot]$. On average, all the approaches reach a power dissipation of $0.75 \text{W}$ for the $0+4$ configuration, while Task-based reaches $3.5 \text{W}$ for the $4+0$ configuration which is the one with the highest average power dissipation. In the $2+2$ configuration, average energy values for Static threading and Task-based are nearly the same. In this case, the increase in power from $1.6 \text{W}$ to $2.1 \text{W}$ is compensated by a significant improvement in performance (of around $30\%$).

Finally, also in terms of EDP, using the four big cores provides the optimal results, as the performance improvements compensate the increase in total energy. In the $2+2$ configuration, Task-based achieves the same EDP results as in $0+4$, but with $81\%$ better performance. Furthermore, for the asymmetric configuration, Task-based achieves the best performance-energy combination since its dynamic scheduling is effectively utilizing the little cores.

From the average energy-delay product (EDP) results in Figure 5, we can determine the configuration that achieves the highest performance with the best possible energy consumption. The configuration with the lowest EDP is the one with four big cores ($4+0$) since all the scheduling approaches manage to compensate the additional energy consumption with the speedup that can be achieved on this set-up. However, it is interesting to note that for the asymmetric configuration, Task-based achieves the best combination of performance and energy since its dynamic scheduling approach is effectively utilizing the little cores.

Next, we focus on the obtained results per benchmark. For applications with an extensive use of barriers (blackscholes, facesim, fluidanimate, streamcluster and swaptions) or with a memory intensive pattern (canneal), the extra computational power offered by the big cores in configuration $2+2$ is not exploited. In fact, the Static threading performance is only slightly improved by $1\%$ on average when moving from $0+4$ to the $2+2$ configuration. This slight improvement comes at the cost of much more power and energy consumption ($79\%$ and $77\%$ respectively).

These results are explained three-fold: i) load is distributed homogeneously among threads in some applications; ii) extensive usage of barriers force big cores to wait until little cores reach
Figure 6: Average results when running on 4 to 8 cores with 4 of them big. Speedup is over 1 little core. Static threading on 4 little cores is the baseline of energy consumption and EDP (see legend in Figure 7).

Figure 7: Speedup over 1 little core when running on 4 to 8 cores and 4 of them are big.

the barrier; and iii) high miss rates in the last-level cache cause frequent pipeline stalls and prevent to fully exploit the computational power of big cores. To alleviate these problems, the programmer should develop more advanced parallelization strategies that could benefit from AMCs, as performed in the remaining applications, or rely on dynamic scheduling at OS or runtime levels.

The three remaining applications (bodytrack, dedup, and ferret) are parallelized using a pipeline model with queues for the data-exchange between pipeline stages and application-specific load balancing mechanisms designed by the programmer. As a result, Static scheduling with these applications benefits from the extra computational power of the big cores in the configuration 2+2. These mechanisms are not needed in the Task-based code; in this approach the code of the application is simplified and the runtime automatically allows the overlapping of the different pipeline stages. Thus, on the asymmetric configuration, Task-based further improves the obtained performance, reaching a 13% average improvement over GTS. Clearly, these applications benefit in performance by the increased number of big cores, while power and energy are increasing since the big cores are effectively utilized.

Generally, relying on the programmer to statically schedule asymmetric configurations does not report good results, as it is very hard to predict the system’s behaviour at application-level. Only applications that implement advanced features with user-level schedulers and load balancing techniques, can benefit from asymmetry, at the cost of programmability effort. Relying on the OS scheduler is a suitable alternative without code modifications, but relying on the runtime to dynamically schedule tasks on the asymmetric processor achieves much better performance, power and energy results.
3.2 Adding Little Cores to a symmetric multi-core

In the following experiments, we explore if an application, running on a symmetric multi-core (SMC) with big cores, can benefit from adding small cores that help in its execution. Having more computational resources increases the ideal speedup a parallel application can reach, but it also introduces challenges at application, runtime and OS levels. Thus, we examine how many small cores have to be added to the system to compensate the cons of having to deal with AMCs.

To evaluate this scenario, we explore configurations 4+0, 4+1, 4+2, 4+3 and 4+4. In these experiments, the number of big cores remains constant (four), while the number of little cores increases from 0 to 4. First we focus on the average results of speedup, power, energy and EDP, shown in Figure 6.

The speedup chart in Figure 6 shows that Static threading does not benefit from adding little cores to the system. In fact, this approach brings an average 6% slowdown when adding four little cores for execution (4+4). This is a result of the static thread scheduling; because the same amount of work is assigned to each core, when the big cores finish the execution of their part, they become idle and under-utilized. GTS achieves a limited speedup of 8% with the addition of four little cores to the 4+0 configuration. The addition of a single little core brings a 22% slowdown (from 4+0 to 4+1) and requires three additional little cores to reach the performance of the symmetric configuration (4+3). Finally, the Task-based approach always benefits from the extra computational power as the runtime automatically deals with load imbalance. Performance improvements keep growing with the additional little cores, reaching an average improvement of 15% over the symmetric configuration when 4 extra cores are added.

The power chart in Figure 6 shows oppositional benefits among the three approaches. We can see that Static threading and GTS benefit from asymmetry, effectively reducing average power consumption. Static threading reduces power consumption when moving from the 4+0 to the 4+4 system by 23% while GTS does so by 6.2%. On the other hand, the task-based approach keeps the big cores busy for most of the time so it maintains the average power nearly constant.

The reduction in power, results in reduced average energy in the case of Static threading in configuration 4+4, as shown on the energy chart in Figure 6. Little cores are more energy efficient than big cores, at the cost of reduced performance. In all the approaches, at least two extra little cores are needed to reduce energy. In configuration 4+4, energy is reduced by 14% for Static threading, 15% for GTS, and 16% for Task-based. Consequently, we can state that asymmetry reduces overall energy consumption.

To see the impact on both performance and energy efficiency we plot the average EDP on the rightmost chart in Figure 6. In this chart, lower values are better. The task-based approach...
is the one that has the best performance-energy combination for the asymmetric configurations since it maintains the lowest EDP for all cases. Static threading manages to reduce the average EDP by 6% while GTS and task based approaches do so by 24% and 36%, respectively.

Figure 7 shows a more detailed exploration of the performance results. As Table 1 shows, the applications with barrier synchronization are blackscholes, facesim, fluidanimate, streamcluster and swaptions. For these applications, the most efficient system configuration with the Static threading approach is the 4+0. Little cores increase execution time due to load imbalance effects. Since the big cores reach barriers earlier, power is reduced for these applications, as shown in Figure 8. Energy reduction is less significant with a few extra little cores as the performance degradation is higher, but as the number of little cores increases, energy is reduced.

Applications with more advanced load balancing techniques, like pipelined parallelism (bodytrack, dedup and ferret), take benefit from the asymmetric hardware and balance the load among all the cores. As a result, performance improves as the number of little cores increases. In the case of bodytrack, GTS reduces performance by 15% when adding four little cores. We attribute this to the cost of the thread migration from one core to the other in contrast to the Static threading approach that does not add such overheads. In the case of dedup, results show more variability. This benchmark is very I/O intensive and, depending on the type of core that executes these I/O operations, performance drastically changes. In order to deal with this problem, a smarter dynamic scheduling mechanism would be required. Finally, canneal does not scale as it has a memory intensive pattern that limits performance.

Figure 8 shows the average power. The barrier-synchronized applications (blackscholes, facesim, fluidanimate, streamcluster and swaptions) reduce power because of their imbalance; since big cores have long idle times with the Static threading approach, they do not spend the same amount of power as GTS and Task-based. For pipeline-parallel applications, both bodytrack and ferret maintain nearly the same power levels among the configurations for each scheduling approach. Dedup is an exception, as the results highly depend on the core that executes the aforementioned I/O operations. In all the benchmarks, the effect of the lower power for Static threading is observed, and is due to the big cores being under-utilized.

This section proves that adding little cores to an SMC with big cores presents significant challenges at the application, OS and runtime levels. Little cores increase load imbalance and can degrade performance as a result. Relying on the programmer to deal with this asymmetry is complex, but a dynamic OS scheduler such as GTS helps in mitigating these problems, providing an average performance increase of 10%. However, the optimal performance results are obtained with the Task-based approach, as they improve static threading by 23% on average. In terms of power and energy, the AMC provides significant benefits, although the SMC with little cores remains the most energy-efficient configuration. The answer to the question of which system configuration provides the best power-performance balance, can be found on the average EDP chart of Figures 5 and 6, and is the use of the entire 8-core system with the Task based approach.

3.3 Experiments on Dibona (pre-production nodes)

We evaluated the implementation of the distributed thread scheduler (DDAST) with OmpSs in the pre-production nodes of Dibona that were made available at the beginning of 2018. Pre-production nodes A0 had up to 27 cores. Pre-production nodes B1 reached 64 cores.

DDAST implements the task submission, and computation of dependences, both for adding and removing tasks from the task graph, out of the critical path. In this distributed version, there is no specific thread executing the services, as they are executed by any thread at idle points.
We present the evaluation of STREAM and Cholesky. Executions have been done using NUMA control, and at 2 different frequencies for the cores, 2 GHz, and 960 MHz.

For STREAM, we compare the performance obtained on A0 nodes, with up to 27 cores (see Figure 9), and the current B1 nodes, up to 64 cores. The overall bandwidth achieved in the B1 nodes is 27% lower, compared to A0 nodes, probably due to the fact that B1 nodes need to implement the cache coherency between the two chips of 32 cores.

![Figure 9: Performance obtained from the STREAM benchmark on Dibona A0 nodes (in MB/s).](image)

On B1 nodes (Figure 10), we observe that the performance of the benchmark increases when the frequency of the cores is high (2 GHz), compared to when using 960 Mhz. And that the benchmark scales while using the cores on a single chip, but its bandwidth gets reduced when using the two chips in the node (visible after the 32-core mark).

It is also observed that, despite variations, the DDAST implementation gets comparable performance when using a single chip, and it is performing a little worse when using the 2 chips. We will investigate the reasons for this loss of performance when using 2 chips.

Regarding the cholesky evaluation, we present the performance obtained when using 2 GHz (Figure 11), and 960 MHz (Figure 12) configurations. We show the Cholesky decomposition of a matrix of 4096x4096 elements in blocks of 256x256 and 128x128.

![Figure 10: Performance obtained from the STREAM benchmark on Dibona B1 nodes (in MB/s).](image)

In both 2GHz and 960 MHz configurations, Cholesky tends to perform better with the DDAST implementation, than with the standard OmpSs implementation. In particular, at 2GHz, the higher performance is obtained when using a single chip, and the 256x256 DDAST
implementation. Again, when using the 2 chips in the node, the performance reduces, probably due to the cache coherence overhead. When executing at 960 MHz, the performance degradation with the second chip is not so high, and the fine grain implementation with 128x128 blocks achieves the highest performance at 64 cores.

![Figure 11: Performance obtained from the Cholesky benchmark on Dibona B1 nodes (in GFlop/s), when running at 2 GHz.](image1)

![Figure 12: Performance obtained from the Cholesky benchmark on Dibona B1 nodes (in GFlop/s), when running at 960 MHz.](image2)

### 3.4 Experiments on Dibona (production nodes)

We have evaluated matrix multiplication (Matmul), BlackScholes and Fluidanimate in the production nodes of Dibona.
3.4.1 Matrix multiplication

We have run this benchmark with different data sizes in 32 cores of one Dibona node. Using one of the chips of the node only, we avoid the heterogeneity caused by the NUMA architecture, and we concentrate the evaluation on the effects of having more or less cores with higher frequency.

The execution environment is set in such a way that we emulate having 1 to 32 big cores by changing the frequency of them. Big cores in production nodes run at 2.5GHz, and little cores run at 1GHz.

Figure 13 shows the performance obtained from DDAST, compared to the baseline OmpSs scheduler, for matrices of size 2048x2048 single precision floating point elements, with a block size of 128x128 elements. Results show that the block size is too small to get benefits from the parallel scheduler, with the two techniques tested (OmpSs scheduler, DDAST).

![Figure 13: Performance obtained from the Matmul benchmark (2048x2048 with block size of 128x128) on Dibona (GFlop/s).](image)

However, as soon the block size on which the global matrix is partitioned reaches a larger size, we can see the benefit of DDAST compared to the baseline OmpSs scheduler. Figure 14 shows how the baseline OmpSs scheduler cannot efficiently use the additional big cores, obtaining a flat performance. Instead, the DDAST scheduler is able to adapt the execution of the tasks to the presence of big and little cores, achieving better performance when more big cores are present in the execution. Both schedulers get similar results when all cores are little, or big.

![Figure 14: Performance obtained from the Matmul benchmark (2048x2048 with block size of 256x256) on Dibona (GFlop/s).](image)

When increasing the size of the matrix, the application creates more tasks for the same block partitioning. This provides more opportunities for the DDAST scheduler to reduce the
overhead. This is shown when increasing the matrix sizes to 4096x4096 elements. Figures 15 and 16 show these results. For these experiments, we have selected 256x256 (Figure 15), and 512x512 (Figure 16) elements per block. In the 256x256 case, there is enough work to be distributed appropriately across the big and little cores, as we increase the number of big cores (left to right in the Figure).

In the 512x512 case, the number of tasks is not so big, and the OmpSs scheduler suffers from having to execute a several chains of block matrix multiplications on the slow cores at the end. This can be seen in Figure 17.

On the contrary, the task scheduling achieved by DDAST in the same matrix configuration (size 4096x4096 and block size 512x512) achieves much better load balancing, as shown in Figure 18. Although this was not the primary objective of DDAST, the fact that the tasks are made available faster to the cores, also contributes to the possibility of achieving a better task scheduling. The time duration of the two traces shown in Figures 17 and 18 is the same.

3.4.2 BlackScholes

We have run the PARSEC BlackScholes benchmark with 10 million options and a block size of 1000 elements, on 32 cores, in one Dibona node, and the results are shown in Figure 19.
In this benchmark, the DDAST scheduler only provides a minor advantage when running with a small number of big cores, up to 8. After this number, the big cores are enough to compensate for the overhead of the baseline OmpSs scheduler, and the DDAST policy has a minor performance penalty.

### 3.4.3 Fluidanimate

We have run the PARSEC Fluidanimate benchmark with the native input of 500K particles, and different block sizes ranging from 32 blocks to 240 blocks. Figure 20 shows the results for 240 blocks, showing that the OmpSs default scheduler and DDAST perform very similarly.

### 4 Conclusions and Future Work

In this deliverable we have done the evaluation of the OmpSs runtime system developed for the Mont-Blanc 3 project. OmpSs now supports the execution of Arm SVE applications, and the DDAST scheduler, supporting task scheduling oriented to asymmetric cores.

Regarding the support for Arm SVE support, we have used the Arm Instruction Emulator in Cavium Thunder-X nodes that had no native SVE support. We have shown that applications with SVE instructions can run using OmpSs, and that the emulation environment provides proper statistics on the execution of the instructions. In particular, we observe that when the number of threads increase, the number of instructions emulated per fault decreases. This is expected, as with more cores, there are more instruction sequences executing SVE instructions, that the emulator cannot join as they are coming from different threads.

Regarding the scheduling on big-LITTLE architectures, we show the results obtained in the platforms Odroid XU3 and Dibona. On the Odroid XU3, we have executed the PARSEC benchmarks and obtained performance and power results using static threading, the GTS scheduler...
from Linux, and the task based approach with OmpSs. Results show that for the homogeneous configurations the three approaches get similar performance and the best power consumption is obtained when running on the 4 LITTLE cores. When evaluating asymmetric architectures, the task based approach gets more benefit from the use of heterogeneous cores than the other scheduling approaches, when accounting for EDP.

We have evaluated several of the benchmarks in one node of Dibona, showing the benefits that can be obtained from the DDAST scheduler approach. Benchmarks like matrix multiply and cholesky get additional performance when executed with the DDAST scheduler. This is because the scheduler can adapt better to heterogeneous cores configurations. Other applications like BlackScholes have minor advantages when using a reduced amount of cores (up to 8). For different applications, like STREAM and Fluidanimate, both schedulers perform nearly the same.

In the future, we will keep evaluating the DDAST scheduler with more benchmarks and applications, and study the effects of the NUMA memory hierarchy in their execution. Also, as soon as we can access a platform with an Arm processor supporting SVE natively, we will perform the evaluation of the applications with SVE instructions.
Figure 19: Performance obtained from the BlackScholes benchmark on Dibona (number of options: 10 million, block size: 1000 elements).

Figure 20: Performance obtained from the Fluidanimate benchmark on Dibona (500K particles).

**Acronyms and Abbreviations**

- AMC: Asymmetric Multi-Core
- BSC: Barcelona Supercomputing Center
- CFD: Computational Fluid Dynamics
- DDAST: Distributed DAS Thread
- DLB: Dynamic Load Balancing
- EDP: Energy-Delay Product
- GTS: Global Task Scheduling
- HPC: High-Performance Computing
- LPDDR3: Low-Power Double Data Rate
- SMC: Symmetric Multi-Core
- SVE: Scalable Vector Extension
References


