D4.6– Intermediate report on OmpSs extensions and storage tuning Version 1.0

Document Information

Contract Number	610402
Project Website	www.montblanc-project.eu
Contractual Deadline	M24
Dissemination Level	PU
Nature	Report
Authors	Ramon Nou (BSC), Xavier Martorell (BSC)
Contributors	Toni Cortes (BSC), Guray Ozen, Diego Nieto (BSC), Javier Bueno (BSC)
Reviewers	Jose Gracia (USTUTT)
Keywords	OmpSs Extensions, resource limits, OpenCL profiler, OmpSs@cluster, Storage tuning

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° 610402.*

Change Log

Version	Description of Change
v0.1	Adding Partial Stripe Avoidance
v0.2	Adding OmpSs extensions
v1.0	Final version

Contents

Executive Summary

For the OmpSs extensions part, in this deliverable we present three developments we have done in OmpSs. First, we have incorporated a resource specification in the programming model to allow programmers to tune the use of cores and devices in the execution of OmpSs tasks. As a result, the programmer can better guide the runtime to use more or less resources of a specific type and get better performance. In the second place, we have extended OmpSs to provide the capability to profile the execution of the OpenCL kernels to determine the most suitable kernel configuration. The Mercurium compiler allows to specify the ranges of values that should be analyzed, and the Nanos++ runtime does the exploration. Finally, in the third place, we have further evaluated the performance of the OmpSs@cluster programming model, with 4 new benchmarks in the Mont-Blanc prototype.

For the storage tuning part, we continued the simulation of the Mont-Blanc environment to explore the effects over I/O of parity on the Parallel File System layer among storage servers. On Exascale systems, the parallel file system should move from a replication configuration to a more space and energy efficient parity-based reliability. The approach presented in this deliverable, is a application guided optimization over realibility based-filesystem. By introducing a new I/O hint, we can write data that only needs to be reliable once it is completed without the performance impact of the parity calculations and updates. Due to the lack of filesystems with the needed reliability characteristics and the lack of the needed storage system, we are still using simulation for the evaluation. The storage part has been published on Europar'15 with the title Performance Impacts with Reliable Parallel File Systems at Exascale Level [9].

1 OmpSs extensions

1.1 Resource extension

We have found that when using various implementations of the same task in both SMP and accelerator architectures, the free scheduling of tasks onto SMP and accelerators may cause imbalance, specially when the difference between the performance of the accelerators and the SMP cores is large.

In order to overcome this problem, we have introduced an additional feature to the `implements` technique, in order to limit the amount of tasks assigned to each type of resource at the same time.

This extension involves the Mercurium compiler and the Nanos++ runtime. Mercurium has been modified to allow the specification of the resources that are consumed by each target architecture, when the tasks are defined. The Nanos++ runtime allows the definition of the resources that are available for each architecture, and the scheduling policies control not to assign more resources of a task onto an architecture than those allowed.

1.1.1 `Task` Directive Syntax Extension

We propose to extend the `task` construct with a new `resources` clause to give hints to the runtime system to appropriately balance the scheduling of tasks to the different devices in the system. Figure 1 shows the proposed syntax extension.

```
#pragma omp task [ clauses ]
                 [ resources(resource-name : resource-amount-expression) ]
     structured-block
```

Figure 1: Syntax of new resources support

The information provided by the programmer in the `resources` clause is passed to the runtime system as a parameter of the task creation.

1.1.2 Runtime support for resources

Available resources are defined at execution time using a Nanos++ call, `nanos_register_resource( resource-name, quantity )`. The name assigned to a resource must agree with the ones used in the task clauses, and there are no special keywords or reserved names, since resources are purely logical and do not map to any OS or hardware component. Regarding the quantity, there are no limitations other than positive integer numbers. This value can be changed dynamically with successive calls to `nanos_register_resource`.

Before the Nanos++ runtime executes a task, it decrements the number of resources consumed by the amount specified in the task directive, if any, from the global value that has been set at the beginning with the `nanos_register_resource( resource-name, quantity )` call. If the resulting value is negative, the execution is postponed, and the task is kept in the ready queue, waiting for other tasks also consuming the same resource to finish. When a task that is consuming a resource finishes, the amount of available resources is increased, and in this way, other tasks trying to consume from the same resource may proceed to execution.

1.1.3 Evaluation

We have used the Jetson TK1 environment for the evaluation of the `resources` extension. Table 1 shows the characteristics of the Jetson platform.

Processor	**Memory**	**Nvidia GPU**
Nvidia Jetson TK1 SoC *4-core Cortex-A15 up to 2.5GHz*	*2 GB*	*1 x GK20A* *(Kepler, 192 cores)*

Table 1: Jetson TK1 system configuration

Figure 2 shows the evaluation of the Nbody benchmark on the Jetson TK1 environment. For this execution, the CUDA version of Nbody is used. In order to use the SMP cores and the GPU, the benchmark spawns a first level of tasks using the `implements` clause. If the GPU is selected for execution, the task is executed in the GPU. If the SMP is selected for execution, the task spawns additional tasks to exploit all SMP cores.

We run the benchmark in four different configurations. The *4 cores* configuration used the 4 A15 cores available in the Jetson TK1 node, with no limitations on the number of resources used by the Nbody tasks. In this configuration all tasks are executed on the SMP cores (`target device(smp)`).

In the *1 GPU* configuration, Nbody executes all tasks in the Nvidia GPU, again with no resource limits (`target device(cuda)`). As it can be observed, the performance obtained by the benchmark is much higher, because the tasks obtain more performance from the GPU than from the A15 cores.

In the *4 cores, 1 GPU* configuration, Nbody is allowed to use both the A15 cores and the GPU. This version is written using the `implements` clause on the GPU tasks (`target device(cuda)`, as an alternative execution to the SMP tasks (`target device(smp)`). Although the programmer may expect an increase in the performance, compared to the version using the GPU only, it is not the case because as soon as an A15 core gets an SMP task, it spends too much time executing it, and the GPU manages to finish with the rest of the tasks of the benchmark. This way, the A15 core delays the end of the execution.

The solution to this problem is to only allow a certain number of tasks to be executed by the A15 cores. The configutation used to achieve this is *4 cores, 1 GPU, Resources*, where the amount of resources assigned to the SMP cores are limited to 5 tasks. In this conditions, the A15 cores do not execute tasks so aggresively as in the previous configuration, and they help the GPU in solving the problem faster.

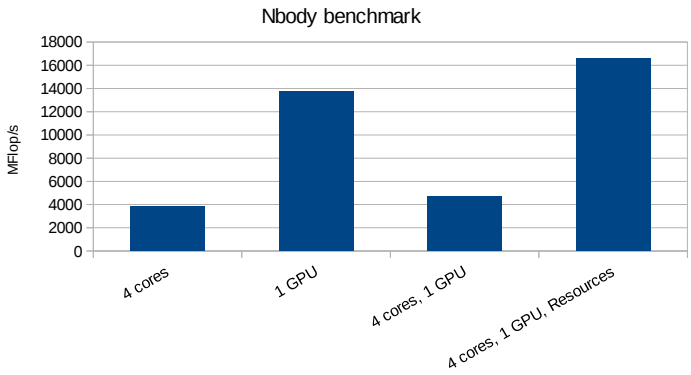


Figure 2: Evaluation of the `resources` extension with Nbody on Jetson

Figure 3 shows the execution timelines of three different configurations of resource assignments to Nbody. The top timeline shows the execution when the resource assignment is free. In this situation, the SMP cores get too many top level tasks, that are then split in smaller tasks, and the GPU is not given the opportunity to execute them. Collecting statistics from an execution, we determined that the GPU was executing 9 top level tasks, and the rest 1006 went to be executed on the SMP cores.

In the intermediate timeline, the resources available on the SMP are limited to the execution of a single top level task, that is still split in inner level tasks. In this situation, the GPU has more opportunities to get tasks, and the overall execution achieves better performance.

In the bottom timeline, we have manually tuned the amount of top level tasks, setting it to 5, which is the amount that achieves best overall performance. As a matter of comparison, collecting statistics from an execution, we determined that the GPU was executing 340 top level tasks, while the rest 344 went to be executed on the SMP cores. Currently, the resource assignment is static. It is part of our future work to investigate how to determine automatically the amount of resources of each architecture that achieves better performance.

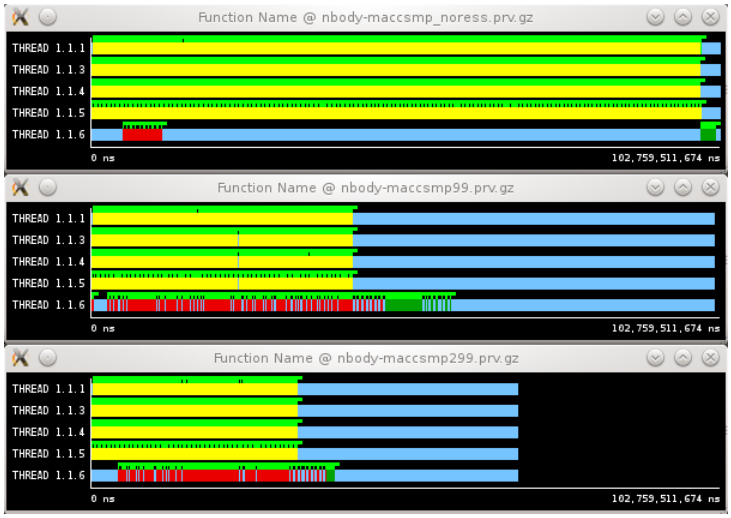


Figure 3: Comparison of the executions of Nbody with and without the `resources` extension on Jetson

1.2 OpenCL dynamic profiling

OpenCL kernels need to be configured appropriately in the GPU to get good performance. This task is usually done by the programmer as a tuning process which is time consuming. In order to alleviate this problem, we have developed a new OmpSs extension to perform the automatic evaluation of OpenCL kernel configurations.

The profiler is based on extensions at the compiler level. The `ndrange` clause in the `target` directive has been extended with a new syntax that allows to express ranges for the *local work group* sizes. The original `ndrange` syntax is the following:

```
#pragma omp target device(opencl) \
     ndrange (dimensions, N, M, ..., lwgs1, lwgs2, ...)
```

where *N, M...* are the complete dimension sizes of the data structure at work. *lwgs1, lwgs2...* are the block sizes in which *N, M...* are split. *lwgs* stands for *local work group size*, and it is

usually set such that it fits well con the GPU cores. With the new syntax, the compiler allows to use a formula to express a range of values for each *lwgs*. This range of values is then passed to the Nanos++ runtime, allowing to do an exploration of the performance obtained with the different combinations of values. The newly incorporated `ndrange` syntax is the following:

```
#pragma omp target device(opencl) \
     ndrange (dimensions, N, M, ..., {/ (1 << v), v = lower0:upper0 /}, \
                                     {/ (1 << v), v = lower1:upper1 /}, ...)
```

When using this new syntax, the Mercurium compiler generates the code that implements the evaluation of the OpenCL kernel with all the combinations of lower and upper values for the number of dimensions provided. As OpenCL does, the number of dimensions is restricted to be between 1 and 3.

As an example of use, Figure 4 shows how to annotate the kernel task of matrix multiply in order to use the OpenCL profiler. With the presented annotation, the combinations of block sizes that will be tried are 16x16, 16x32,32x16, and 32x32.

```
#pragma omp target device(opencl) copy_deps file(matrixMul.cl) \
            ndrange(2, nrows, ncols, \
                    {/ (1 << i), i = 4:5  /}, {/ (1 << i), i = 4:5  /})
#pragma omp task in([nrows*ncols]a,[nrows*ncols]b) out([nrows*ncols]c)
void mxm_kernel(float* c,float* a, float* b, int nrows, int ncols);
```

Figure 4: Use of the OpenCL profiler from the application source code

As a result of executing matrix multiplication with this annotation, the runtime system uses the information provided to explore the four combinations of sizes in the X and Y dimensions, obtaining the results shown in Figure 5.

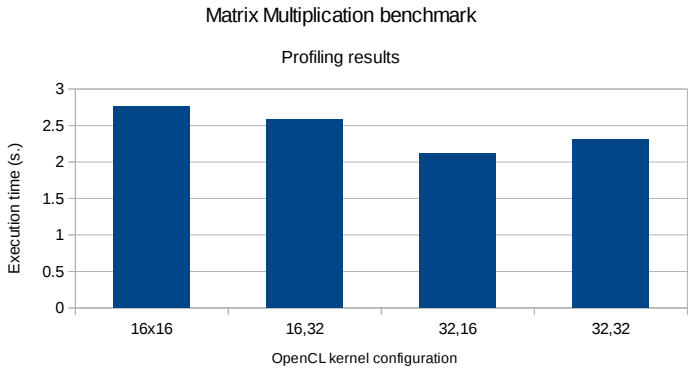


Figure 5: Evaluation of the OpenCL profiler with Matrix Multiplication

After executing the benchmark, the programmer may decide to use the configuration obtaining better results, thus replacing the `ndrange` expressions, with the actual values, (32, 16) in this example.

1.3 Evaluation of OmpSs@cluster

We have evaluated the cluster version of OmpSs with the additional benchmarks NAS EP, SparseLU, PTRANS, and FFT. Executions use up to 32 nodes on the Mont-Blanc Cluster. On each node, one core is used for computing, and the other one is used for supporting the communications. These are the results obtained.

Figure 6 shows the results of the NAS EP benchmark (Class C) when run on the Mont-Blanc cluster, from 1 to 32 nodes.

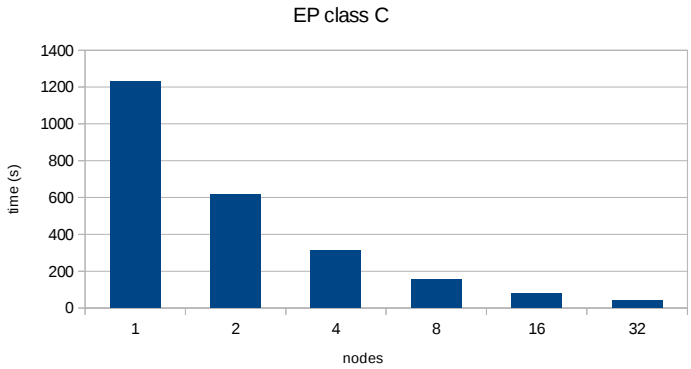


Figure 6: NAS EP benchmark

Figure 7 shows the results of the SparseLU benchmark on a matrix of 8Kx8K elements, with a block size of 256x256 elements, when run on the Mont-Blanc cluster, from 1 to 32 nodes. The plot shows how the benchmark scales. It has limited scalability because of the imbalanced nature of the benchmark and the increase in the amount of data transfers compared to the EP benchmark.

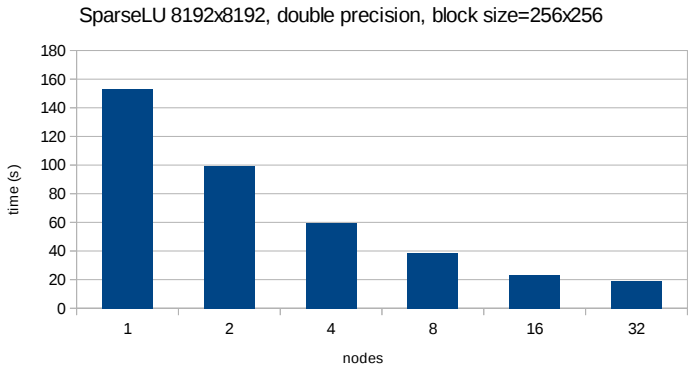


Figure 7: SparseLU benchmark (matrix size 8K x 8K)

Figure 8 shows the results when increasing the matrix size to 16Kx16K and the block size to 512x512 elements. In this case, the results obtained are less uniform due to the increase in the block size, which causes an increase of the imbalance, specially when run on 4 and 32 nodes. We will work more on this particular benchmark to try to solve this issue.

Figure 9 shows the results of the PTRANS benchmark with two different matrix and block sizes, when run on the Mont-Blanc cluster, from 2 to 32 nodes. For both sizes, the scalability of PTRANS is very limited due to the communication pattern.

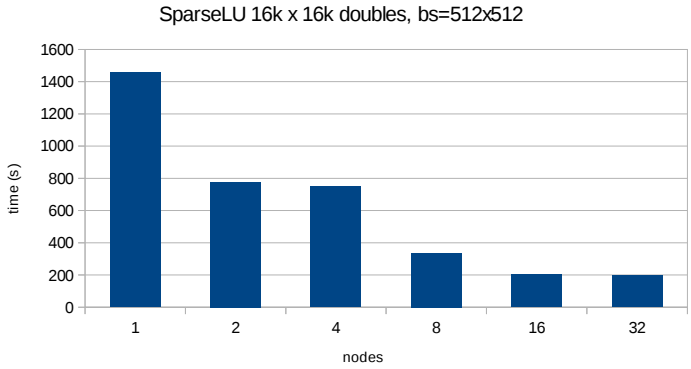


Figure 8: SparseLU benchmark (matrix size 16K x 16K)

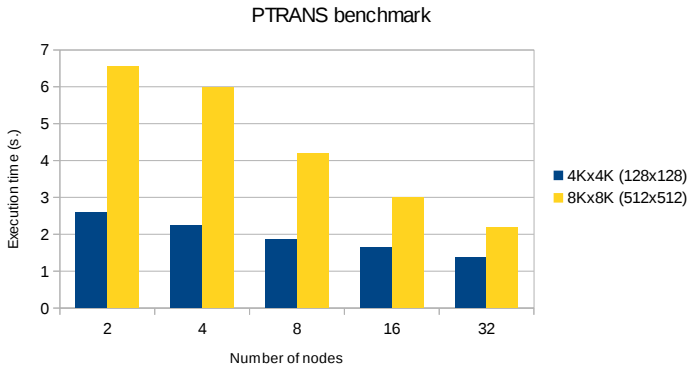


Figure 9: PTRANS benchmark

Figure 10 shows the results of the FFT benchmark with two matrix sizes, when run on the Mont-Blanc cluster, from 1 to 32 nodes. The scalability is also limited as in the case of PTRANS, as FFT involves several matrix transpositions. We will study better the behaviour of PTRANS and FFT to determine if these results can be improved in the Mont-Blanc prototype.

1.4 Conclusions and Future Work

We have presented three OmpSs extensions that we are developing in the context of Mont-Blanc 2 project. The OmpSs `Resource` extension allows to better balance the distribution of tasks to the different devices available (SMP cores, GPUs), and we have shown that it allows to obtain better performance because it allows to balance the progress done by the GPU and the SMP cores on the tasks spawned at the top level.

The second extension, the OpenCL profiler, allows to use OmpSs to profile the OpenCL kernel configuration, and select the one that provides best performance. Finally, we have also evaluated the OmpSs@cluster alternative to execute applications across nodes using a single programming model. We have shown that benchmarks scale, although we have still work to do to analyze their execution, determine the actual bottlenecks that limit their scalability and improve their execution if possible.

As our future work, we plan to introduce the use of the three OmpSs extensions in our trainings of the project, thus allowing the project partners to use them, while we tune their implementations for achieving better performance.

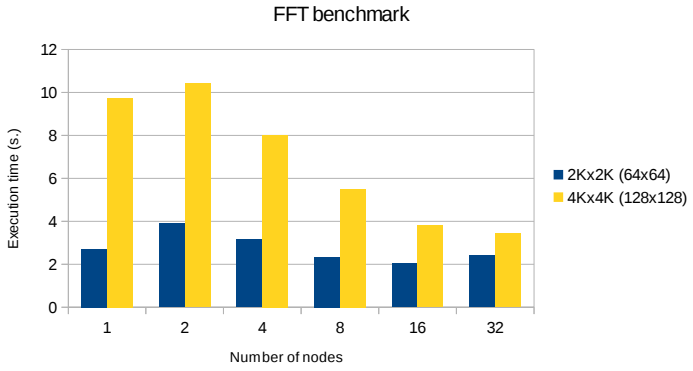


Figure 10: FFT benchmark

2 Storage Optimizations

Given the strong constraints on energy efficiency imposed on Exascale clusters [15, 11, 1], the current replication techniques used by parallel file systems (PFSs) to increase reliability and availability (where several copies of each datum are kept in independent storage nodes) can represent a huge penalty, since they multiply the investment and energy costs in the storage layer.

Parity based-reliability, where mathematical checksums are computed and stored to recover failed data, is a more suitable method in this scenario since it uses less storage resources than replication. As such, as we explained in the previous deliverable D4.2 [7] , there is an increasing interest to support node-wide RAID-5/6 reliability schemes in current PFSs. For instance, Lustre [2] is planning to support file-level replication, and this technique can already be found in Gluster [4]. On the other hand, Panasas (with PanFS [8]) supports object/file level RAID configurations using triple parity data [10], and GPFS [13] also supports a similar configuration with the *declustered array* technique.

Unfortunately, an increase in the number of I/O requests will also affect traditional parity-based reliability techniques. Increasing the number of data writes will accentuate the *partial stripes* and *small writes* problems [14] that typically affect these strategies: a small change to a datum will force the parity checksum to be recomputed and stored, which requires additional I/O operations as well as computation. Thus, introducing these node-wide reliability strategies into Exascale storage can cause a performance impact: updating a datum in RAID-5 requires four I/O requests (reading the original datum and the old parity and writing the new datum and the new parity) and six I/O requests in RAID-6 (as it uses two parity checksums). As we will show later, even though these additional requests can be distributed between storage servers to be processed in parallel, they can represent a loss of performance of up to 85% for update operations when compared to storing raw data.

In this situation, it seems clear that optimizations over RAID parity calculations are needed to remove or alleviate this performance penalty and provide Exascale storage systems with alternatives for reliability. In this deliverable we propose a novel method (Delayed Parity) that takes advantage of the collaboration between the PFS and the clients and, using this layer, allows applications to delay parity computations. The analysis and design is evaluated with a simulator, using a write-only workload to focus on the issue that we are solving (read operations are not affected negatively).

2.1 Partial Stripe Avoidance

In this section, we will explain the advanced strategy proposed to reduce the Partial Stripe Avoidance problem. As a change over the previous deliverable we renamed the Write Cache Server (WCS) to Write Cache Layer (WCL) as it better suits how it works.

As a summary of the previous deliverable [7], Figure 11 shows the parity update workflow of the basic avoidance technique when compared to vanilla RAID-5. The technique reduces the congestion in Data OSSs by removing one (or two in RAID-6) data reads, hence reducing the number of operations when writing new data by 25% in RAID-5 and 16% in RAID-6 and obtaining a similar performance gain.

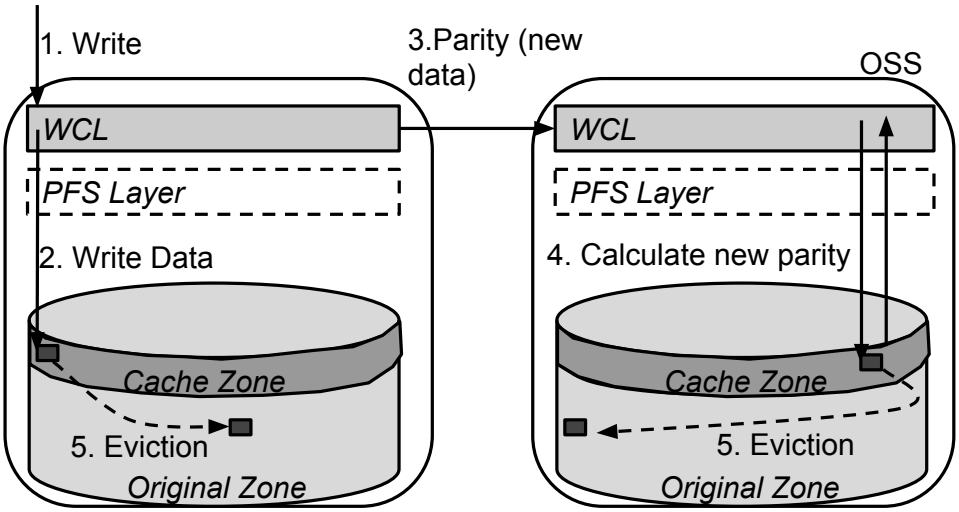


Figure 11: Basic Avoidance Technique

2.2 Advanced Avoidance: Delayed Parity

One of the most used fault tolerance mechanism in HPC applications is checkpointing, an operation that stores the current status of all processes creating an opportunity to restore the application in case of failure. Due to the continuous writes needed to save the state, this particular operation issues many parity update requests to maintain the reliability. However, is such reliability really needed? Consider for instance that the system fails in the middle of the checkpointing: the application could recover using a previous checkpoint and delete the partial-checkpoint file, rendering the parity computations for the partial checkpoint useless. This also applies to long computations like matrix multiplications (e.g., MADCAP, on MADBench2 [3]). A failure in the middle of the computation would require it to be restarted again from the beginning, hence the partially stored data would be discarded and the parity calculations would become an avoidable overhead. We can envision a lot of HPC applications that could make use of such functionality, since reprocessing a chunk would be less costly than the cost over all the system to store *all the data* in a reliable way compared to doing it when the process completes.

Using this idea, we propose the `START_DELAYED` and `END_DELAYED` hints to mark this kind of candidate operations, delaying the parity calculations until all the writes are completed. When the WCL receives a `START_DELAYED` hint, redirects all write operations to the caching zone and disables the parity computations for this data. Once the corresponding `END_DELAYED` hint is received, the WCL computes the parities of all the full stripes affected by the hints, and moves (rewrites) the data to its original location in the OSS. As a result, the PFS can avoid using the

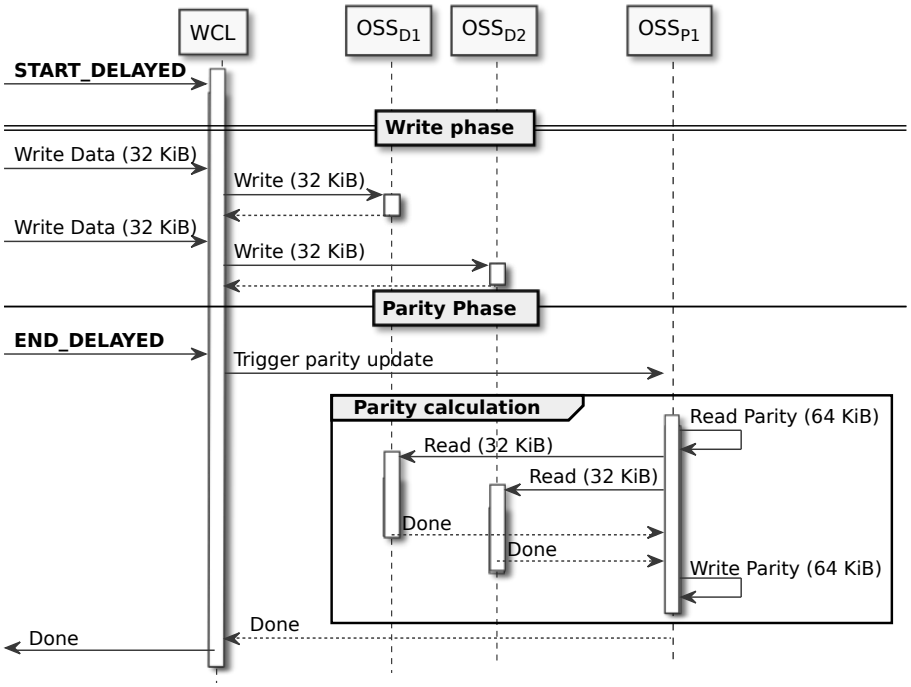


Figure 12: Delayed Parity: Clients issue two writes and then the parity is calculated. The writes are aligned so they end at the same Parity OSS but at sequential positions. Due to this alignment, the parity calculation can be consolidated.

Parity OSSs during the creation of the data and, when parities are calculated, it only needs to send them the consolidated writes.

Figure 12 shows a simplified sequence diagram for the delayed parity technique. In that example, 2 clients issue a 32 KiB write and then their parity is calculated. The writes are directed to different data OSS, but the same Parity OSS. Parity blocks are sequential, and hence parity updates can be consolidated.

The WCL supports synchronous parity calculations, but can also advance the calculation in the background to reduce time. However, advancing the calculation may generate extra work if the calculation is not needed (i.e., application cancellation or error in the client). In short, the delayed parity technique can be represented as a collective operation between all clients, but without increasing intermediate memory requirements since partial data will be written to disk.

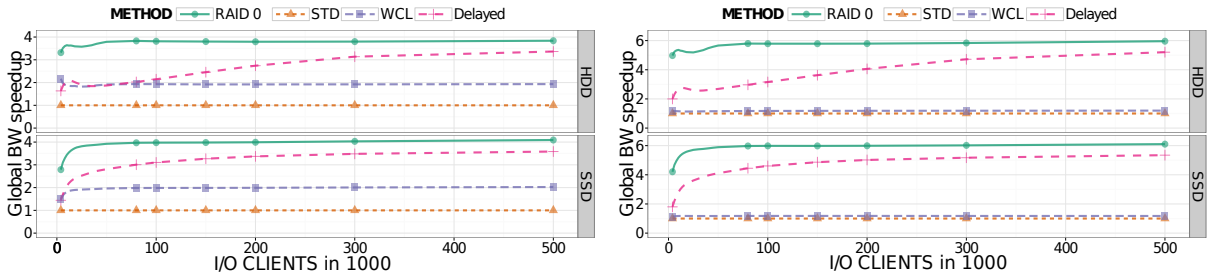
2.3 Evaluation

The simulated application uses a set of clients issuing writes to the I/O layer. During a single simulation run, the datum size of write operations is fixed for all the clients and the writes are distributed along a different file per application to avoid overwrites. Each client writes enough data to produce a statistically representative number of parity calculations. The behaviours of the applications simulated mimic that of the FLASH application [6], a computational tool for simulating and studying thermonuclear reactions, that periodically outputs large checkpoint files and smaller plot files. For the Delayed Parity evaluation, one process of each application acts as master issuing the new hints and all the clients wait until the parity is calculated.

Workloads with mixed block sizes were only tested with a low number of devices, as the cost of generating the statistical device model was too high (it is necessary to attempt all

the possible combinations of request sizes, which grows exponentially). This is an important drawback of modelling the storage devices without a simulator. Nevertheless, we did not find significant differences between these simulated workloads in systems with the large number of clients targeted in the paper. The same also applies to different stripe sizes.

We decided to use this workload in order to concentrate on the effect of typical HPC writes over the proposed reliability techniques. Nevertheless, we also checked other workloads, which showed similar results. In particular, our preliminary results using mixed workloads (with reads and writes), showed improvements in the performance on read operations since the proposed techniques favoured a reduction of the overhead on the storage devices. Due to space limitations, we will not discuss these results further.



(a) Weak-Scaling

(b) Weak-Scaling

(c) Strong-Scaling

(d) Strong-Scaling

Figure 13: RAID-5 (left) and RAID-6 (right) write bandwidth w.r.t. STD-RAID.

To assess the effectiveness of the novel strategies, we repeat each measurement with different seeds, that control how requests are mapped into each storage device. Each simulation run stops when we have a minimum of 1000 seconds of simulated time, other variables are tracked to assess that the results are representative. In the following experimental results, RAID-0 represents the performance obtained from the OSSs when there are no parity calculations. STD-RAID represents a standard striped RAID with parity and without optimizations. Finally, our two proposals are WCL (already presented on D4.2 [7], and that we will not explain on this one) and Delayed Parity. If not specified, the results are scaled relative to STD-RAID for each x-axis point. We use 512 OSSs in all the experiments with a stripe unit of 1MiB and a width of 512 OSSs.

We measure the results in two different ways: using Weak-Scaling and using Strong-Scaling. Weak-Scaling means that the problem size increases with the number of clients and thus, regardless of the number of clients, each one will process the same amount of data (128 KiB per data write). Conversely, in Strong-Scaling the number of clients is increased, but the problem size is kept. Consequently, this increases the network and storage congestion due to I/Os when the number of clients increases, due to the higher number and smaller size of requests. In this

scenario, the data writes grow from 8 KiB to 1MiB according to the number of clients in the simulation. For all the scenarios, the data is partitioned to avoid overwrites according to their block size. We analyse the outcome of the new techniques using RAID-5 in Subsection 2.3.1 and RAID-6 in Subsection 2.3.2.

2.3.1 RAID-5 Results

This section describes the measured results of our simulations when reliability is implemented using RAID-5 (i.e., one parity checksum per stripe).

Weak-Scaling results. Each client writes 128 KiB per write distributed along a file avoiding overwrites with other clients. As we can see on the Delayed Parity proposal, the strategy reduces the number of writes on the corresponding Parity OSS for each row of the file ($m-1$ vs 1, where m are the OSSs involved) w.r.t. the WCL strategy, but we increase the reads in the data OSSs (0 vs $m-1$) as we need to transfer the data to the parity node to calculate the parity (see Figure 12). If we take a look at the number of accesses per Data OSS and Parity OSS, we observe that only one access is needed to the Data OSS with WCL (to write) and two with the Delayed Parity approach (to write and to read in order to calculate the parity). On the other hand, using the WCL approach requires a parity for each write (a read and a write to the Parity OSS), whereas with the Delayed Parity approach the parity only needs to be generated when all operations have completed (i.e., a maximum of two accesses to the OSSs).

As we can see in Figure 13.a, the benefit of the Delayed Parity strategy depends on the cost of the operations in the device. In that Figure, with less than 80,000 clients, the required time to complete the described operations on the HDD devices surpasses the reduction on the number of operations. As a result, the performance with Delayed Parity with a low number of clients is comparable to the WCL strategy, but without the reliability of RAID-5 (as the parity is calculated at the end). Despite this result, the general behaviour is for performance to grow up close to RAID-0, as we remove a big number of parity updates producing a higher throughput.

Strong-Scaling Results. The experiments done with Strong-Scaling (see Figure 13.c.) are similar to the Weak-Scaling ones. Using the Delayed Parity technique, we observe performance improvements due to the fact that the number of parity updates is reduced greatly as the number of clients increases. For instance, with 250K clients, we do not issue a parity update until all the clients of an application have stored their 16KiB to the devices, which means that we go from 250K parity updates (clients $\times$ appl. iterations) to 1.7K parity updates ($\frac{\text{clients}}{\text{clients per appl.}} \times$ appl. iterations). Actually, the performance obtained differs when using HDD or SSD technology (as in Weak-Scaling). On HDDs, the Delayed Parity achieves less performance, thus it may be preferable to avoid using this strategy for a small number of clients.

2.3.2 RAID-6 Results

In this experiments we have selected two horizontal parity devices per stripe for RAID-6, as we did in the previous deliverable.

Weak-Scaling results. As we can observe in Figure 13.b, we found that the Delayed Parity option offers a bigger performance boost on RAID-6 since we now have two parity devices, and

we move from two parity updates per write to two parity updates per application iteration (`START-END` hint).

Strong-Scaling results. For Strong-Scaling results with RAID-6 we obtain similar results to RAID-5 (see Figure 13.d). Like with RAID-5 results, the block size is important on the simulated scenarios for Delayed Parity. In that particular experiment, we can see the same performance loss found on RAID-5 with HDD devices. However, the performance improvement compared to the WCL proposal is bigger even with a lower number of clients since parity updates are more expensive in RAID-6 than in RAID-5. Thus, removing them (more precisely, grouping and delaying them) produces bigger improvements. In general, since the performance loss of STD-RAID w.r.t. RAID-0 is much higher for RAID-6, the potential gain of the Delayed Parity strategy is higher.

2.3.3 Related work

Delayed Parity Calculation. About our delayed parity proposal, a similar approach is found in NetApp [5] where writes are buffered to issue an improved write operation. Also, at AFRAID [12] they move the parity calculation to idle periods to obtain a performance boost. The main difference of our proposal with the previously mentioned works is that the lower reliability mode is selected by the user (via hints) when he decides that the data is not useful until it is completed (i.e., check-pointing or partial results that will need to be recalculated). All writes are persisted to the disk, so it may recover from failures, at the same rate than the used PFS.

2.4 Conclusions and Future Work

Under Exascale constraints, reliability will be needed on the PFS layer if we want to keep the storage costs and the energy used under control. Especially, when we use a high number of clients the number of parity updates will increase.

We proposed on the previous deliverable a transparent cache layer that is able to reduce the number of operations needed to update the parity on such environments. To do that, we ensure that the writes are not overwriting so we can drop the read of old data from the parity update workflow. This proposal improves the write performance of the standard workflow by a 1.18x to a 2x depending on the RAID level (6 or 5, respectively). In this deliverable, we show that applications gain substantial performance controlling the parity calculation as in the Delayed Parity Proposal. Using reliability oriented application hints, we can improve the write performance up to levels near a RAID-0. This behaviour is useful when partial data does not need to be reliable until all the data writing is finished, e.g. big partial matrices.

References

[1] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Technical report, DARPA, 2008.

[2] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[3] J. Carter, J. Borrill, and L. Oliker. Performance characteristics of a cosmology package on leading hpc architectures. In *HiPC 2004*, pages 176–188. Springer.

[4] Gluster. Glusterfs web page. `http://www.gluster.org/`, 2014.

[5] S. Kleiman, R. Sundaram, D. Doucette, S. Strange, and S. Viswanathan. Method for writing contiguous arrays of stripes in a RAID storage system using mapped block writes, 2007. US Patent 7,200,715.

[6] R. Latham, C. Daley, W.-k. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific I/O: Improving the FLASH astrophysics code. *Computational Science & Discovery*, 5(1):015001, 2012.

[7] Montblanc. D4.2 - Preliminary report on OmpSs extensions and storage tuning.

[8] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC'04*.

[9] R. Nou, A. Miranda, and T. Cortes. Performance impacts with reliable parallel file systems at exascale level. Vienna, Austria, 2015-08-24 2015.

[10] Panasas. PanFS RAID. `https://www.panasas.com/products/panfs/PanFS_RAID`.

[11] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439 – 443, 2013.

[12] S. Savage and J. Wilkes. AFRAID: a frequently redundant array of independent disks. In *USENIX ATC'96*.

[13] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST'02*, volume 2, page 19.

[14] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 64–75, 1993.

[15] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, et al. Scaling the power wall: a path to exascale. In *SC'14*.