# D4.2– Preliminary report on OmpSs extensions and storage tuning
# Version 1.0

## Document Information

| | |
|---|---|
| Contract Number | 610402 |
| Project Website | www.montblanc-project.eu |
| Contractual Deadline | M12 |
| Dissemination Level | PU |
| Nature | Report |
| Authors | Alberto Miranda (BSC), Ramon Nou (BSC), Toni Cortes (BSC), Javier Bueno (BSC), Xavier Martorell (BSC) |
| Contributors | |
| Reviewers | Jose Gracia (HLRS), Steffen Brinkmann (HLRS), |
| Keywords | OmpSs, Mercurium, Nanos++, 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$^o$ 610402.*

# Change Log

| Version | Description of Change |
|---|---|
| v0.1 | Adding Partial Stripe Avoidance |
| v0.2 | Adding OmpSs Extensions |
| v0.3 | Incorporating comments from internal reviews |
| v1.0 | Version to be sent to the EU |

# Contents

# Executive Summary

In this deliverable we present the extensions to OmpSs regarding the support of clusters. Within OmpSs we have implemented a caching system to deal with the data that must be sent to remote nodes to be processed there. A remote node can be another node in the cluster or an accelerator attached to it. This implementation has been done in the Intel architecture, and evaluated in a cluster with NVidia GPUs. Deliverable D4.1 presents the porting to the ARM architecture.

For the storage tuning part we simulate 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 we are simulating and presenting on this deliverable is suitable for light-weight Exascale nodes. As the number of clients will be larger, the write request frequency will increase creating more parity update requests. We propose a mechanism to reduce the number of operations to update the parity, improving the performance of writes up to a 200%.

# 1 OmpSs Extensions

The version of OmpSs that we are developing for the Mont-Blanc 2 project is OmpSs@cluster. This version allows the execution of applications in several nodes in a cluster, with or without accelerators. In this section, we describe the OmpSs@cluster extensions developed in this period for Mont-Blanc 2.

## 1.1 OmpSs@cluster

The execution environment that we got as the basis for the OmpSs@cluster for Mont-Blanc 2 is based on a master-worker design. Figure 1 shows a sample cluster consisting of two nodes. The left node is the master, which starts the execution of the application, and connects to the worker node on the right side.

There is a cluster thread that manages the communications between the master and worker nodes. It sends commands and data to the workers. The commands indicate to execute a task or a set of tasks in the remote node. The data is the values over which those tasks will work. Data computed in the remote worker node may be claimed by the master when encountering a `taskwait` OmpSs synchronization primitive.

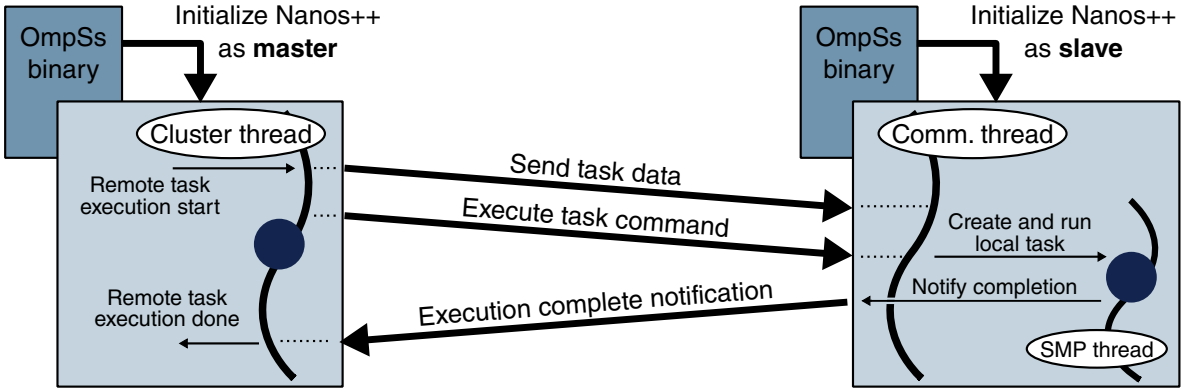



Figure 1: Master-worker scheme used to implement the Nanos++ cluster support

In order to manage the data that is sent to remote nodes, OmpSs@cluster implements a data directory and caching mechanism. The master node is the responsible for keeping the memory consistent with the OmpSs memory consistency model but also for offering the OmpSs single address space view. The master node memory is what OmpSs considers the *host memory* or *host address space*, and it is the only address space exposed to the application. The memory of each slave node is treated as a private device memory and is managed by the master node.

Figure 2 shows how the directory and caches are organized in a small two-node system.

To mantain the consistency of the memory, the master node uses the *data directory*. It contains the information of where the last produced values of a memory reference are located. With it, the system can determine which transfer operations must perform to execute a task in any device of the system. Also, each task execution updates the information of the data directory to reflect the newly produced data.

The current implementation of the caching system is the subject of the next subsection.

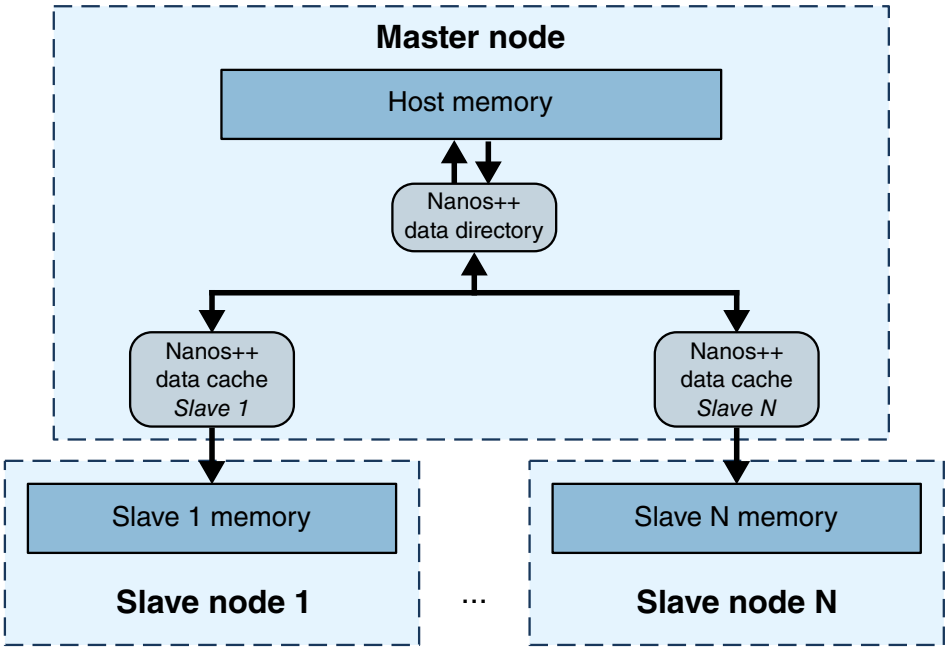



Figure 2: OmpSs@cluster memory management organization

## 1.2 Implementation of the caching system for the OmpSs@cluster version

The part of the OmpSs system managing the data that is available in each node or accelerator is the OmpSs caching system. Such a cache has been implemented in the Nanos++ runtime system, and it is described in the next subsections.

### 1.2.1 Precise regions of data in Nanos++

According to the OmpSs specification, regions are defined using a syntax similar to the one used to declare and access array types in the C programming language. A region is defined using a base object, which we will refer to as a *program-object*. This object will typically be an array with an arbitrary number of dimensions. Array subscripts are used to specify which elements of each dimension are covered by the region. For each dimension, two integer values define the first element and the number of elements covered on that dimension. These values must always represent a range where all of its elements fall within the limits of the corresponding dimension. We will refer to these pair of values as *dimension-access*. Figure 3 shows an example of code which defines a region. A two dimensional array is used as *program-object*, therefore a set of two dimension-access elements must appear in the region definition.

Regions can be dynamically defined using dimension-access containing program variables. This provides a lot of flexibility to the programmer, who can defer the computation of the region size and shape until the execution of the program. Also, it is legal that different regions overlap with each other. By overlap we refer to the fact that different regions access the same elements of a *program-object*.

The internal representation of the regions will allow the Nanos++ runtime library to check if two or more regions overlap with each other. In order to achieve a simple and efficient design,

```
double M[8][8];

void compute() {
   #pragma omp task inout(M[0;6][0;4])
   {
      for (int j=0; j<6; i++) {
         for (int i=0; i<4; i++) {
            process_elem(M[j][i]);
         }
      }
   }
}
```

Figure 3: A task defining a region. The *program-object* of the region is the array `M`. It is a two dimensional array so two *dimension-access* elements are needed to declare the region, `[0;6]` defines the range of accessed elements of the outer dimension and `[0;4]` defines the range of accessed elements of the inner dimension

we have set a few limitations to our system. These limitations are:

- Regions are associated always to a single *program-object*. Regions covering more than one object are not supported.
- All regions that refer to the same program-object must be consistent with each other, that means that all of them must provide the same number of dimensions of the given object and access valid elements on these dimensions.

These limitations have a very low impact to the programmer, since when they do not hold, the situation is considered a programming error. In addition, the runtime detects when a set of tasks does not follow these rules and emits an error message to inform the user.

### 1.2.2 Region identification

*Program objects* are registered at run-time and are identified by their base address. Each region is associated to a single *program-object*. A region is defined by a set of *dimension-access* objects, one for each dimension of its referred object. In order to have a simple way to identify regions an integer value, unique within its *program-object* is assigned to each region. We will refer to this value as the *region identifier*. Figure 4 shows a sample code that contains three tasks and each one declares an access to a region of a program-object. The associated layout of the regions defined on the program-object with the internal identifiers is presented in Figure 5.

The generation of the identifier is done using a dictionary implemented using a *trie* (or *digital tree*). Each node of the tree contains a value of each of the *dimension-access* values (first element and number of elements covered). With this, a region information is stored as the path from the leaf to the root of the tree. The leaf node contains the *region identifier*. Figure 6 illustrates the data structure that results from registering the regions defined in the previous example. The first two levels of the tree contain to the range defined by dimension-access of the outer dimension whereas the two following levels contain the information regarding the outer most dimension.

```
double M[8][8];

void compute() {
   #pragma omp task inout(M[0;4][0;4])
   {
      ...
   }
   #pragma omp task inout(M[0;8][6;2])
   {
      ...
   }
   #pragma omp task inout(M[6;2][0;8])
   {
      ...
   }
}
```

Figure 4: A function that creates three tasks, and each one defines a region using the prorgam object M.

This data structure can be used with any number of dimensions, the tree will be created with the appropiate number of depth levels to contain the complete region information. The usage of a tree eases the implementation of operations to check the existence of a given region or registering a new one, while keeping them efficient even when the tree holds a high number of entries.

The *region identifier* provides a simple way to identify a region within an object, and it also is used to access the region meta-data, which is stored using an array of meta-data entries. The region meta-data contains the location information of the related data, that is, in which memory address spaces (GPUs, nodes) is stored.

### 1.2.3 Region overlaps

Regions may overlap. To detect which regions overlap with another region, we maintain an *intersection map*. An *intersection map* is actually composed by a set of maps, one for each dimension. Each of these contains the information about how regions cover the elements of the given dimension. They map *dimension-access* elements to sets of *region identifiers*. These maps can be seen as a projection of the present regions onto a one dimensional space. Figure 7 depicts the contents of the intersection map corresponding to the regions declared on the given matrix.

A mapping from *dimension-access* to a set of *region identifiers* that overlap along that dimension. Figure 8 shows the example of the *intersection map* created by the *program-object* A, and its four regions.

To compute the set of regions that overlap with a given region, we must first obtain the sets of *region identifiers* using each *dimension-access* object of the region. The intersection of all obtained sets is the set of overlapping region identifiers.

Example: assuming the state described by figure 8, a new region, `A[0;2][2;2]`, is registered and the overlap detection process goes as follows. The first dimension, `[2;2]`, is used to access the *Dimension 0* map, which returns the set `{2,4}`. Then the second dimension, `[1;2]`, is used on

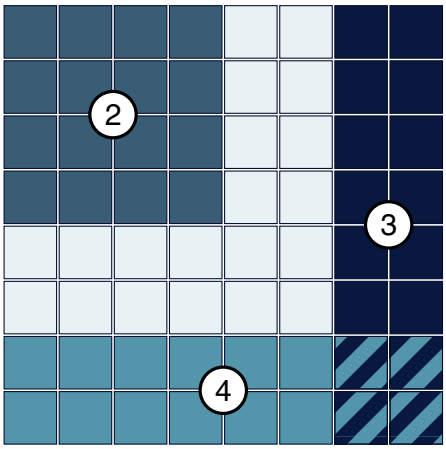



Figure 5: The representation of the regions defined by the function on the left, with their corresponding internal identtifiers.

the *Dimension 1* map and the result is `{1,2,3,4}`. The intersection of both result sets is `{2,4}`, meaning that regions 2 and 4 overlap with the new region.

### 1.2.4 Distributed region information

Region meta-data is generally accessed by the run-time when it starts the execution of a task since it requires to know where its needed data is located. Since the described design is a centralized one, it can suffer from contention problems when running with several threads at the same time, since all of them will try to access to the *intersection maps* if they access the same *program-objects*.

To mitigate this contention, we exploit the dependence subsystem of OmpSs. When a task depends on another one, it is likely that the regions that were accessed by the preceding task will be also accessed by the successor task. We transfer the regions meta-data of a task to its successors hoping that they will be able to use it instead of having to request it to the centralized data structures.

### 1.2.5 Memoizing overlaps

A very common operation that the OmpSs run-time has to perform is to compute if two given regions overlap, and which is the resulting region of this overlap, if it exists. Because regions are always given unique identifiers we can apply memoization in order to perform the overlap check and store the result to avoid computing the same values in the future.

### 1.2.6 Region based memory management

The main subsystem of Nanos++ that had to be adapted was the memory management part. Moving from a management based on contiguous memory, to a one based on regions with potentially non-contiguous data caused important changes in the way Nanos++ allocates and transfers data among the different address spaces of the system.

The allocation of regions brings a few questions. When it comes to allocating non-contiguous memory regions of data, the actual allocated size will always be bigger than the used size by the application, because the shape of the data must remain the same. This can be a problem when we

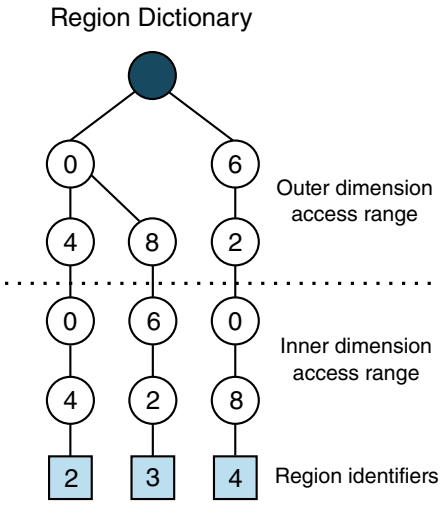



Figure 6: Representation of the internal tree that holds the region information and the leaf nodes corresponding to the region identifiers. Starting from a region identifier node, the information of the region can be retrieved by navigating to each parent node until reaching the root.

are working with accelerators with limited memory, but there is nothing that can be done from the run-time library to solve this, unless other techniques like data *reshaping* are implemented. For the case of this work targeting the cluster environment, this is not a problem because each node has plenty of memory available on the Intel architecture, and the decision here has been to allocate whole arrays in remote nodes. This reduces the costs of allocating data on demand, which can potentially cause reallocations with severe performance penalties. For the ARMv7 architecture, where we know there will be not more than 2-4 GBytes of total memory, we will consider alternatives. The more simple one being use blocking at the application level in order to reduce the size of the data that should be accessed at the same time in remote nodes.

Data movement operations of non-contiguous data were originally implemented on a very naive way. For each region that had to be moved, a list of contiguous data chunks was generated, and then a movement operation was issued for each chunk. While this approach fulfills the requirements of correctness, its performance may vary depending on the architecture system that is the responsible for moving the data.

## 1.3 Optimizing for clusters

The main problem of the first implementation was the number of data movement operations that were generated for non-contiguous regions. In the cluster architecture these operations were translated to network messages. While sending a network message is an expensive operation by itself, some regions caused hundreds or thousands of these calls, which resulted in a poor application performance.

To overcome this problem, we implemented a simple mechanism to minimize the number of network messages that were needed to transfer regions of non-contiguous data. The idea is to use a temporal buffer where the non-contiguous data is placed contiguously prior to be sent through the network. Once the data arrives to its destination, the run-time library can reverse the operation and place it with its original shape. Figure 9 shows a sample transfer using the described mechanism.

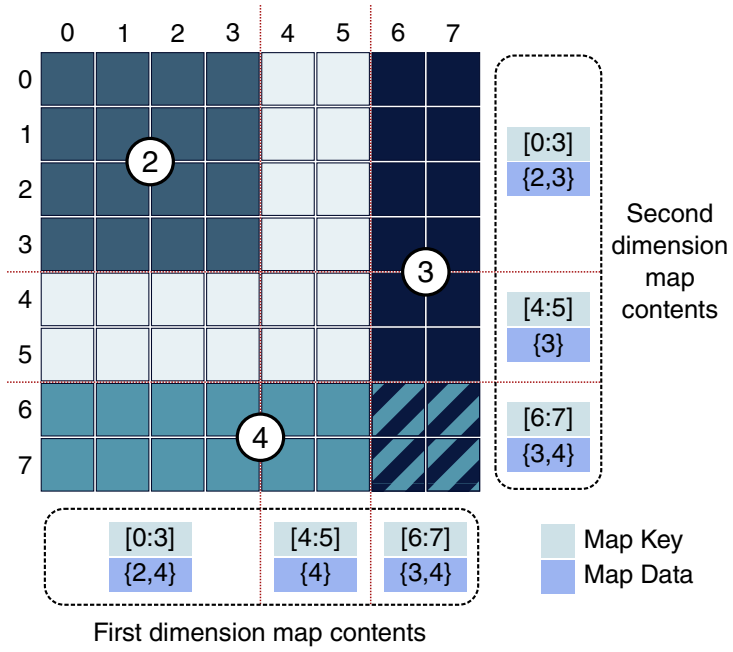



Figure 7: Intersection maps

An important implementation detail that had to be taken into account when we were implementing this mechanism is to control which thread is responsible for packing and unpacking the data. In Nanos++ there is a thread in each node which is responsible of monitoring the network for incoming messages, this is a requirement of the underlying communications library used (GASNet [4]). Delegating the task of unpacking the data to the thread which is actually selected to execute a parallel task was needed to keep a good network latency, if not, the communication thread could spend too much time dealing with this packing mechanism, stalling the execution of the whole application. In the case of data packing, the problem is more complex because it usually is the communication thread the one starting the transfers of data, so these tasks can not be delegated to another thread as easily as the unpacking. However, a mechanism was also implemented where idle threads could collaborate on sending data to remote nodes, with this the communication thread work load was lowered and more network efficiency was obtained.

## 1.4 Affinity scheduler

A new scheduling policy has been developed to use when running applications on a cluster. The main idea is to execute tasks on nodes where the data they access is located, this strategy reduces the amount of data transferred during the execution. The scheduler also takes care of distributing the data and avoiding inbalance.

The scheduling process first classifies the tasks in two classes: *data producer* tasks and *computation* tasks. Data producer tasks are those that will only write new data, that is, their data usage clauses only include `out` or `copy_out`. These kind of tasks typically appear at the beginning of applications and they are in charge of generating large amounts of data that will be used later by other tasks. The scheduler distributes these tasks according to the amount of data that they generate, trying to balance also the amount of data generated on each node of the cluster.

Computation tasks are the rest of the tasks that are not data producers. Generally compu-

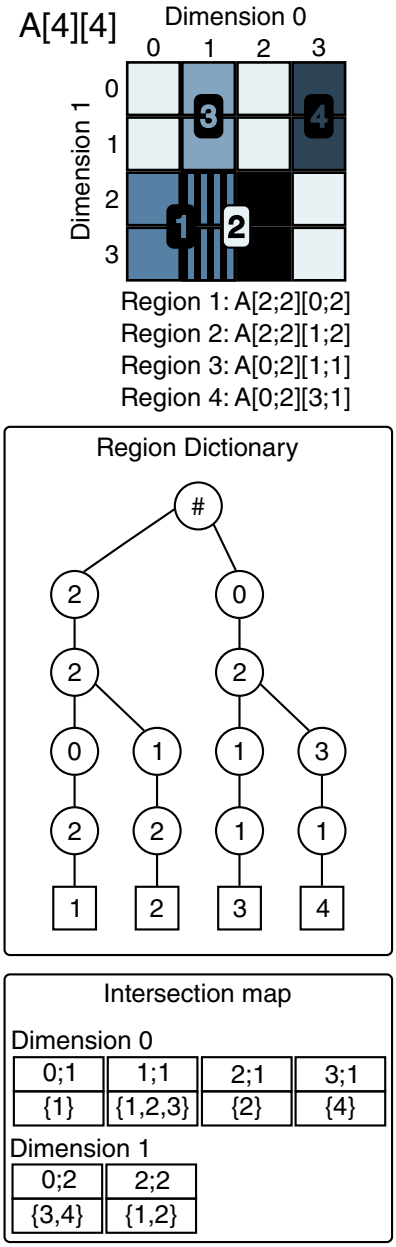



Figure 8: An array of 4x4 dimensions with 4 different regions and the data structures that hold the corresponding *region identifiers* and the *intersection map*

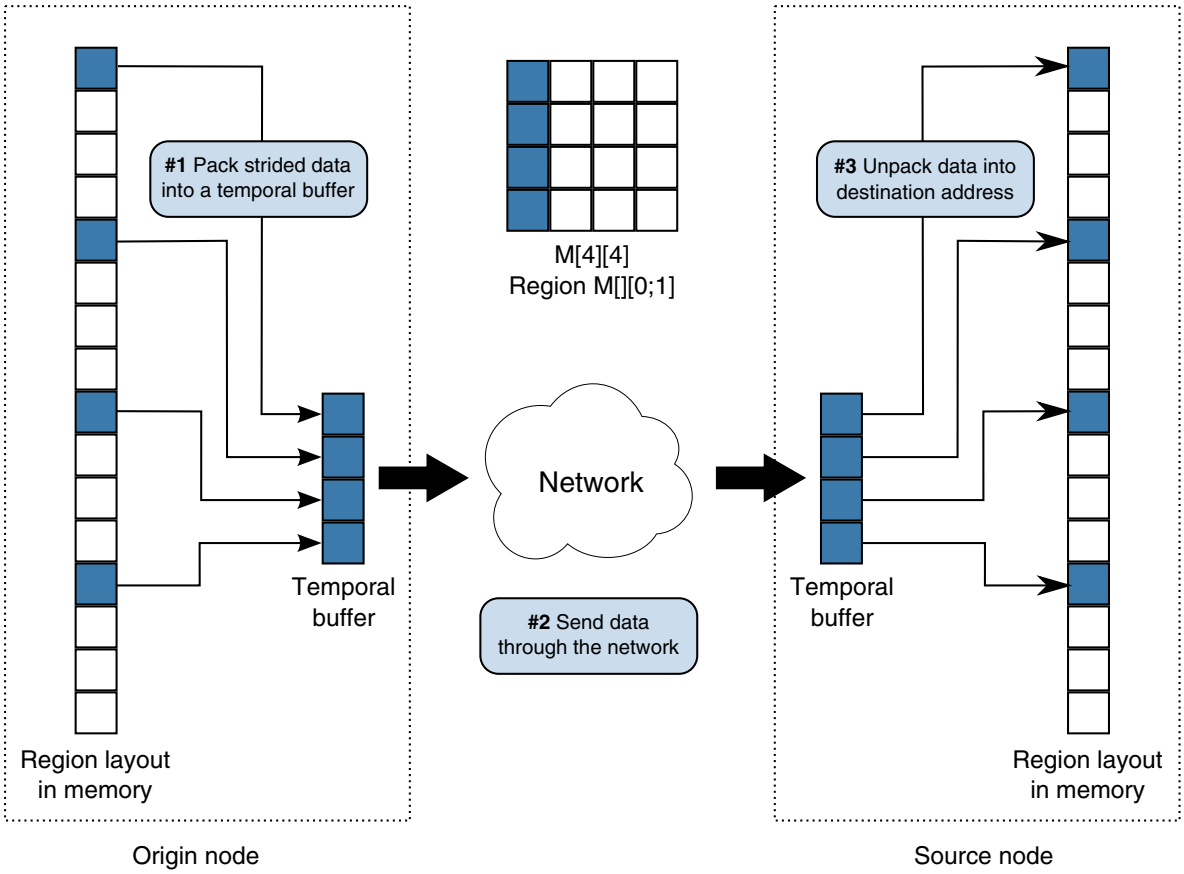



Figure 9: Region transfer with data packing

tation tasks read certain pieces of data and update them, or produce data in other memory locations, so they are declared using several `in/copy_in` clauses and some `inout/copy_inout` or `out/copy_out` clauses. For each computation task, the scheduler assigns it to the node that contains the largest amount of data required. This is done computing an *affinity score* for each node on the cluster. The score is the amount of data, in bytes, available on the given node. The information of where the data is located is provided by the data directory subsystem. The scheduler assigns a task to the node with the highest affinity score.

Different nodes may tie for the highest affinity score. This may happen because data may be replicated on different nodes, or because data produced with a computation task is later used as input data of other tasks. The scheduler keeps a set of counters, one for each node, that keep track of the number of tied tasks that have been scheduled to each node. Whenever a task ties for two or more nodes the scheduler picks the node with the least number of tied tasks scheduled, and updates the counter. If several nodes also tied for the tied tasks counter value, the node with the lowest identifier is picked.

The assignation of tasks to threads is implemented using task queues. Each node has a specific queue where the corresponding cluster thread will check for new tasks to execute.

As a final mechanism to avoid inbalance between nodes, *task stealing* may be enabled by the user. Task stealing allows a cluster thread to pick tasks from queues of other nodes. This provides an automatic balancing mechanism since when a node runs out of tasks to execute, its

cluster thread will pick tasks originally scheduled for other nodes. This mechanism, while probably disrupting the original purpose of the scheduler, will obtain a better balance of work and end up in a more efficient use of the parallel resources.

The aforementioned scheduling policy is used only when scheduling top level tasks. Top level tasks are those that are created by the main program, that is, not created by any other parallel task. Children tasks are always executed on the same node where they were created. This favors locality since the data accessed by children tasks is always a subset of the data accessed by the parent task, thus it makes sense to schedule the children tasks where the parent is running.

## 1.5 Evaluation

This section covers the evaluation performed to measure the performance of the OmpSs applications using the mechanisms presented on a cluster of multicore processors.

### 1.5.1 Methodology and environment

To evaluate our environment we selected a set of OmpSs applications and measured their scalability with our run-time environment. These applications exploit the support for non-contiguous regions of data, and we also compare them to well known MPI implementations of the same applications.

The machine used to run the evaluation is a cluster with nodes consisting of two Intel Xeon E5649 @ 2.53GHz multicore processors with six cores each. Each node also has 24 GB of memory divided in two NUMA nodes. Each node has two Infiniband QDR network interface controllers connected to a non-blocking network.

All OmpSs benchmarks were compiled using the Mercurium C/C++ source-to-source compiler, using GCC 4.4.4 as a the final C compiler, and using -O3 optimization level. Benchmark kernels are compiled with the Intel C Compiler (ICC) version 12.0.4, also with -O3 optimization level enabled, or invoked from the Intel Math Kernel Library (MKL) version 10.3.4. The MPI distribution (Bull MPI 1.1.11.1) is provided by the hardware vendor, derivedfrom the OpenMPI implementation.

For each one of the selected applications, we have performed tests with different configurations on the number of nodes and threads used on each node. The metric measured for each application is the execution time.

### 1.5.2 Benchmarks

**DGEMM.** The DGEMM experiments have been done using a matrix size of 16384x16384. Two different OmpSs implementations have been tested. The first one is the more naive approach, it uses a single level of parallelism to perform the computation. The schema used corresponds to the code shown in figure 10, and it does not require the infrastructure implemented in this paper. The second one uses multilevel parallelism to improve locality and improve the distribution and creation of the computation tasks. The multilevel version required the support of the non-contiguous data regions implemented in this paper. The schema followed to implement this version can be seen in figure 11. We have compared both versions of OmpSs with the purpose of showing the performance benefits that can be achieved by using non-contiguous data regions. The parallelization is implemented by dividing the main algorithm in tasks that perform the matrix multiplication of blocks of 512x512 elements.

```
double A[NBLOCKS][NBLOCKS][BS BS];
double B[NBLOCKS][NBLOCKS][BS BS];
double C[NBLOCKS][NBLOCKS][BS BS];

//matmul main loop
for ( i=0; i < NBLOCKS; i++ )
   for ( j = 0; j < NBLOCKS; j++ )
      for ( k = 0; k < NBLOCKS; k++ )
         #pragma omp target device (smp) copy_deps
         #pragma omp task in(A[i][k], \
                             B[k][j]),\
                        inout(C[i][j])
         matmul_blk(A[i][k], B[k][j], C[i][j], BS);
```

Figure 10: DGEMM OmpSs implementation without regions support

```
double A[N][N], B[N][N], C[N][N];

//matmul main loop
for ( i=0; i < N; i += BS )
   for ( j = 0; j < N; j += BS )
      #pragma omp target device (smp) copy_deps
      #pragma omp task in(A[i;BS][], \
                          B[][j;BS]),\
                     inout(C[i;BS][j;BS])
      for ( k = 0; k < N; k += BS )
         #pragma omp target device (smp) copy_deps
         #pragma omp task in(A[i;BS][k;BS], \
                             B[k;BS][j;BS]),\
                        inout(C[i;BS][j;BS])
         matmul_blk(&A[i][k], &B[k][j], &C[i][j], BS, N);
```

Figure 11: DGEMM OmpSs implementation, with regions and multilevel parallelism

The MPI implementation used has been the ScaLAPACK [3] one provided by the Intel Math Kernel Library (MKL). The OmpSs versions have also used the Intel MKL.

**PTRANS.** The PTRANS benchmark measures the rate of transfer for larges arrays of data by implementing a parallel matrix transpose. It exercises the communications of the cluster heavily on a realistic problem where pairs of processors communicate with each other simultaneously. The arrays data used has also been two-dimensional arrays of 16384x16384 elements.

Following the same parallelization schema as the one used in the DGEMM experiments, the parallelization of the PTRANS is implemented using tasks that process sub-blocks 512x512 elements. We have tested the OmpSs version of this benchmark with two Nanos++ configurations, depending on whether the optimization technique of packing the non-contiguous data prior to send it over the network was enabled or not. This shows how effective this optimization can be

on applications that stress the usage of the network.

The MPI implementation of this benchmark comes from the HPC Challenge Benchmark suite [21], [13].

While the benchmark scales with the increase in the number of nodes, it does not scale when increasing the number of threads per node. This is because of the memory bandwidth available in each node getting saturated by the parallel transposition operations.

**FFT1D.** The FFT1D application measures the floating point rate of execution of the double precision complex one-dimensional Discrete Fourier Transform (DFT). The OmpSs implementation uses the Cooley-Tukey algorithm to implement the one-dimensional DFT.

The data is distributed in a two-dimensional array of $16384 \times 16384$ complex double precision elements. The first step of the algorithm is to perform a in-place transposition of the data, after this, a FFT1D round is applied to each of the 16384 rows of the data. The next step is to transpose again the data and to apply a *twiddle* factor, to follow with a second round of FFT1D on each row. Finally, a last in-place transpose obtains the final result.

The parallelization of the transpose and the twiddle+transpose are also implemented using tasks that operate on sub-blocks of the matrix, the dimension of these blocks is also 512. The row-FFT1D processes are parallelized by creating tasks that process blocks of 512 rows of the main matrix, each of these tasks also create more parallel tasks to perform the final computation. This structure can be seen in figure 12. For this application, we have also evaluated the impact of the data-packing optimization implemented in Nanos++, to demonstrate its impact on a more complex application.

As for the selected MPI implementation, we have used the one provided by the HPC Challenge benchmark suite [21], [13], which is a FFT1D algorithm specifically tailored for MPI. We have also used the FFT1D kernel

### 1.5.3 Results

**DGEMM.** Figure 13 shows the performance obtained by the DGEMM benchmark. As stated in the previous section, we have implemented two versions of this benchmark, the one labeled "1L OmpSs" refers to the naive implementation (the schema used by corresponds to the source code shown in figure 10) which can be coded without the need of the work described in this paper. The second version, labeled "ML OmpSs" is the implementation using multilevel parallelism enabled by the data region specifications (same schema as the one shown in figure 11). The ScaLAPACK results are under the "MPI" label. Five charts are shown depending on the number of worker threads used by the run-time. Because the ScaLAPACK version is implemented using OpenMP, this could be controlled using the `OMP_NUM_THREADS` environment variable. In the last chart of the series, titled "Max threads per node", the OmpSs run-time environment uses only eleven threads as an extra thread is created to perform network polling and send control messages. However in the ScaLAPACK execution, twelve OpenMP threads are requested. The X axis represents the number of nodes used during the different executions, the Y axis shows the execution time in seconds. It must be noted that the Y axis scale is different for each of the charts. This has been done to be able to better appreciate the differences on the cases with a high number of threads per node.

The charts show how the version implemented using multilevel parallelism is capable of achieving the same scalability as the ScaLAPACK version, even being able to obtain a slightly better performance due to other available optimizations of Nanos++ [8], [9] (task pre-send, affinity scheduler) that increase the overlapping of communication and computation. The simple OmpSs

```
double ( A)[N][N];
size_t BS = getBlockSize();
A = malloc( N N sizeof(double) );

// first transpose
for ( i=0; i < N; i += BS )
   for ( j = 0; j < N; j += BS )
      #pragma omp target device (smp) copy_deps
      #pragma omp task inout( A[i;BS][j;BS], \
                              A[j;BS][i;BS])
      transpose_blk(A[i][j], A[j][i], BS, N);

// first row fft
for ( i=0; i < N; i += BS )
   #pragma omp target device (smp) copy_deps
   #pragma omp task inout( A[i;BS][0;N])
   for ( j = i; j < i + BS; j++ )
      fft1d_row(A[j][0], N);

// transpose and twiddle
for ( i=0; i < N; i += BS )
   for ( j = 0; j < N; j += BS )
      #pragma omp target device (smp) copy_deps
      #pragma omp task inout( A[i;BS][j;BS], \
                              A[j;BS][i;BS])
      transpose_tw_blk(A[i][j], A[j][i], BS, N);

// second row fft
for ( i=0; i < N; i += BS )
   #pragma omp target device (smp) copy_deps
   #pragma omp task inout( A[i;BS][0;N])
   for ( j = i; j < i + BS; j++ )
      fft1d_row( A[j][0], N);

// last transpose
for ( i=0; i < N; i += BS )
   for ( j = 0; j < N; j += BS )
      #pragma omp target device (smp) copy_deps
      #pragma omp task inout( A[i;BS][j;BS], \
                              A[j;BS][i;BS])
      transpose_blk(A[i][j], A[j][i], BS, N);
```

Figure 12: FFT1D OmpSs implementation

version performs well with a low number of nodes, but it fails to scale properly with 16 or more

nodes, this is caused by the fact that in this version, task creation is centralized on the master node, which creates a bottleneck when distributing the tasks among the rest of the threads of the cluster. This is why the problem is more noticeable with an increasing number of nodes and threads. The multilevel version is capable of delegating part of the task creation to the remote nodes, which reduces the overhead of task distribution from the master.

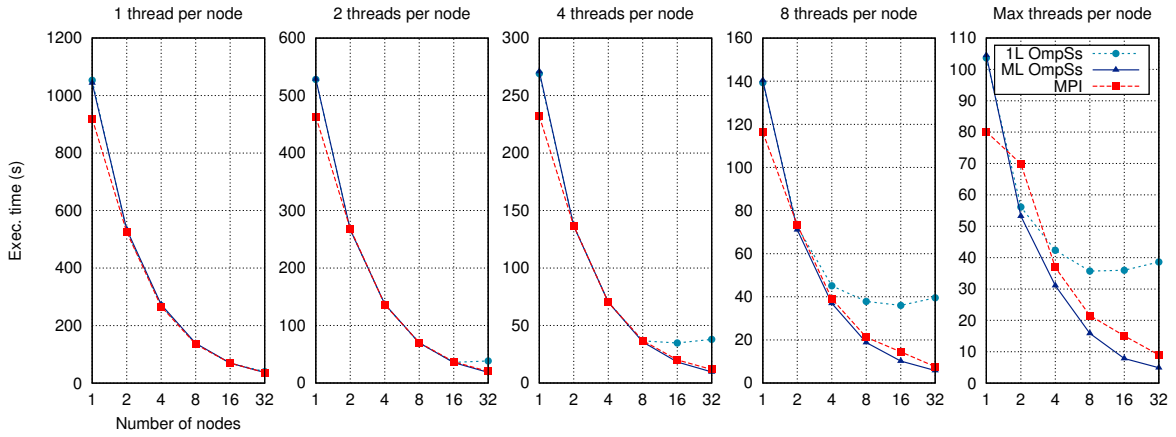



Figure 13: DGEMM performance comparison, OmpSs vs. MPI (different Y axis scale on each chart)

**PTRANS.** The PTRANS results are shown in Figure 14. The results are also organized in different charts depending on the number of threads per node used on each execution. In this case we have tested a single OmpSs implementation of the benchmark but we have run it with two Nanos++ configurations, the first without enabling the technique of packing the non-contiguous data prior to send it through the network, labeled as "NP OmpSs", and the second with this technique enabled, labeled as "OmpSs". The "MPI" label in these charts refers to the HPCC implementation of the benchmark. It must be noted that this benchmark is not implemented using OpenMP, so to enable the usage of more than one core per node, we executed it spawning the appropriate number of MPI tasks instead of threads. Again the X axis shows the number of nodes and the Y axis shows the measured execution time in seconds.

The results show how effective is the packing mechanism in this application. It greatly improves the performance of the OmpSs implementation in every situation and it actually makes OmpSs to be at the same level, or even better in some cases, than the HPCC version. The reasons for this are that the PTRANS performance relies on achieving a good network usage. Without the packing technique this is not possible because non-contiguous accesses are transformed to one network message per contiguous data, which leads to a very high number of messages. By packing the data the run-time library reduces this number of messages and communications are more efficient, and the execution time is significantly reduced. In some cases MPI can outperform OmpSs since task distribution is a cost that can not be avoided easily. In the PTRANS, all tasks are distributed from the master, in addition, the PTRANS tasks are not very computation intensive, which leaves few opportunities of overlapping communication and computation, exposing the distribution overhead. That is why the results of 32 nodes show a low scaling compared to the 16 node cases. Still, OmpSs performance is comparable to the one achieved by MPI. The cases

where both OmpSs versions perform much better than MPI are explained because of the extra thread that the Nanos++ run-time library creates, since it helps when it comes to sending more messages and therefore, it can provide a better usage of network bandwidth, which is critical in this benchmark. This can make the comparison unfair from the resource point of view, however, the last case (the chart titled "Max threads") can be considered fair from this perspective, and it shows that the performance is almost on par.

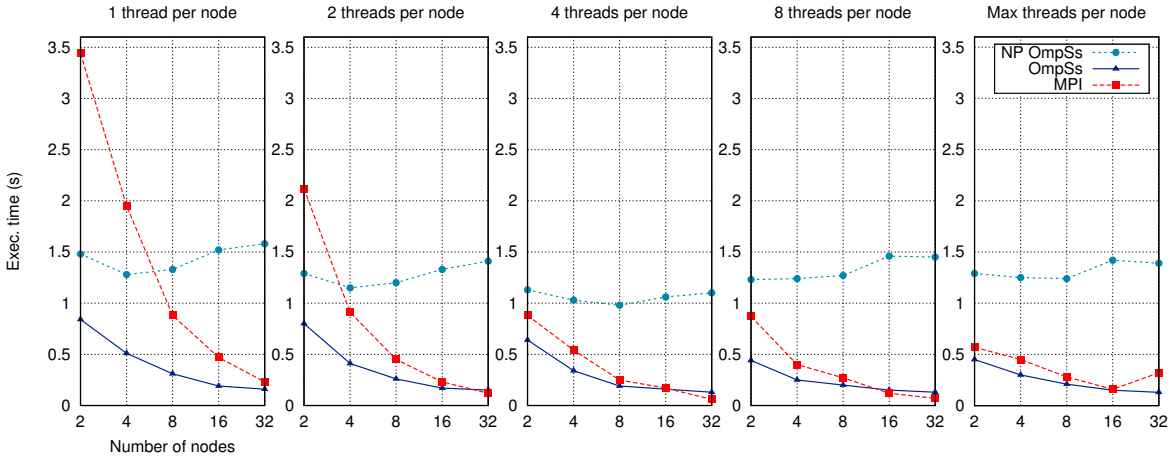



Figure 14: PTRANS performance comparison, OmpSs vs. MPI

**FFT1D.** Figure 15 shows the performance results of the FFT1D benchmark. Again the experiments are grouped by the number of threads used in each execution. X axis shows the number of nodes and Y axis the execution time in seconds of the experiments. As with the previous benchmark, we have executed the OmpSs version with the packing technique disabled and enabled to measure its impact on the FFT1D benchmark. The packing enabled data series is labeled with the "OmpSs" title, and the packing disabled is referred by the "NP OmpSs". The MPI version of the FFT1D benchmark provided by the HPC Challenge suite is implemented using MPI and OpenMP so in this case we can control the number of threads created on each node using the same way as in the DGEMM benchmark.

FFT1D is a demanding benchmark since combines some phases of high bandwidth requirements with phases of intense computation. Because of this, we see the same behavior of PTRANS when it comes to the effect of the packing optimization. Enabling the technique boosts performance in almost all cases and it is absolutely necessary to achieve a proper scalability with a large number of nodes and threads per node. In addition Nanos++ is capable of overlapping computation and communication, and this provides an extra performance edge over the MPI version on the majority of the cases. With 32 nodes the performance is almost identical, OmpSs losing the extra advantage because of the same problems of the PTRANS case, cost of task distribution with a large number of nodes and tasks with short execution time. It must be noted that the fact of having an extra thread can be the cause of performing better in some cases, specially with few threads per node, as it happened in the PTRANS benchmark. However OmpSs is capable to outperform the MPI version in the "Max threads per node" case, when the same resources are used in both versions. In this case, the asynchronous parallelism is capable of overlapping parts of the transposition with the execution of the row FFTs, which leads to a better overlapping of

computation and communication.

The difference of performance on the 1 node cases between OmpSs and MPI can be explained because the former is actually a sequential algorithm which still while the latter is a SPMD program running on a single node.

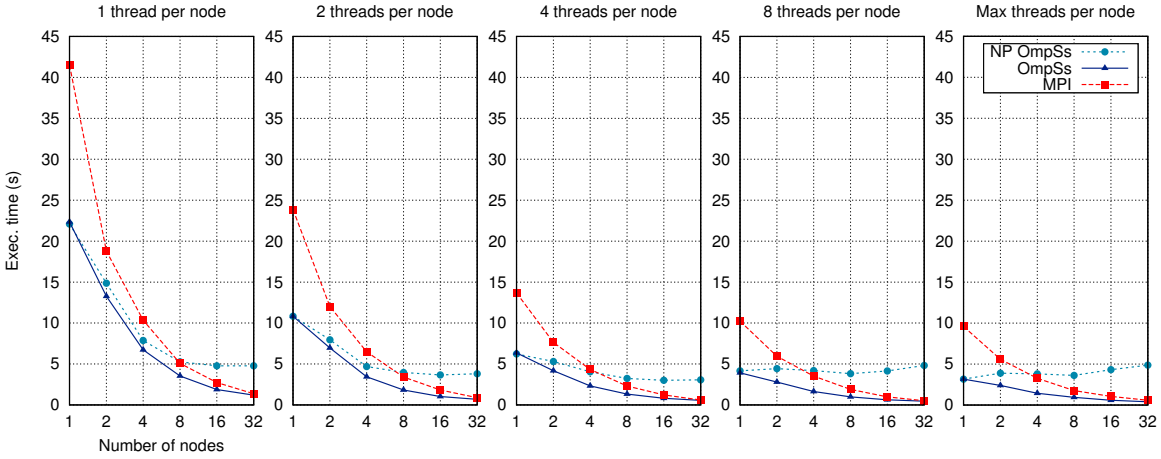



Figure 15: FFT1D performance comparison, OmpSs vs. MPI

# 2 Storage Optimizations

The Mont-Blanc project needs to increase the number of clients to provide enough computing power and as a result of that, we have an undesired congestion on all I/O levels (e.g.,, Network, OS, Disk, etc.) as more requests and messages go through the I/O layer. This needs to be solved either with hardware (increasing the number of I/O nodes and the network) and/or software (as our proposal or collective I/O).

Using collective I/O operations [29], where buffering and coordination reduces the number of clients doing I/O, can mitigate the problem, but additional solutions to keep the performance levels and the reliability of the system while keeping the energy requirements can be pursued.

On the performance plane, we will need to increase the number of storage servers to maintain the clients requirements, however adding more storage servers will need some work on the reliability plane if we want to keep the energy used under the desired levels. Traditionally, full file-replication (or two-way fail-over) has been used, but at Exascale levels this is too costly as it doubles the investment costs of the storage layer. We propose, as some of the parallel file system (PFS) trends show, to apply RAID-5 or RAID-6 directly, using a storage server (Object Storage Server or OSS using Lustre nomenclature) as an abstraction of a disk. With those RAID schemes the recover capability of data is improved: The system can recover from up to $n$ storage nodes failure. The value of $n$ is 1 for a RAID-5 and 2 with RAID-6, but can be higher. The recover with erasure codes needs more effort than replication, however it is more versatile in several aspects.

Parity schemes have an additional problem, they suffer from the Partial Stripes problem: writes generating many parity requests. Those parity requests require a lot of I/O operations to be processed, as for example a write in a RAID-5 [19] needs two reads and two writes while RAID-6

needs three reads and three writes to keep the reliability. Those extra operations increase the utilization of the disks, decreasing their performance.

Analyzing the different PFS as Gluster [15], Lustre [6] or PVFS [27], they only allow striping configurations (to increase parallel performance) along different storage servers. However, there is an increasing interest (for example, in the Gluster mailing list) to support RAID 5/6 schemes. Lustre is planning to support file-level replication, and we can already find in Gluster replication and striping configurations. Panasas with their PanFS [24] supports object/file level RAID configurations using triple parity data [25], also GPFS [28] is supporting a similar configuration with their declustered array [12] as it can visualize all the JBOD disks individually to apply better schemes. With the need for Exascale storage systems, such configurations will start to be more frequent with faster parity computation as replication is neither space nor energy efficient.

We will show that naively introducing reliability configurations come with a huge performance impact loosing about 75%-85% of the performance (as Figures like Figure 20 show), as each write issues a parity operation to the corresponding stripe where the data is located (a stripe is a complete set of data and parity elements related). Finally, we analyze the overhead of different RAID schemes and propose an I/O layer to reduce the performance penalty. The analysis and design is evaluated using a simulator.

Our contribution in this deliverable is:

- The design of a transparent I/O layer reducing the number of parity updates for arbitrary reliability configurations.

## 2.1 Partial Stripe Avoidance

In this section, we will explain the two strategies studied to reduce the overhead of parity update operations, but first of all we will explain the access patterns that benefit from our I/O layer proposal, named Write Cache Server (WCS).

### 2.1.1 Targeted Access Patterns

Patterns with write-only access (as reads are not affected) can be classified according to their sequentiality and whether they append or overwrite data:

1. **Sequential / No overwrite**
2. **Sequential / Overwrite**
3. **Random / No overwrite**
4. **Random / Overwrite**

This categorization of file access does not cover all possibilities, but only the extreme ones we need to show the benefit and/or limits of our proposal. Other file accesses can be inferred from the previous ones.

Analyzing the first category, **sequential / No overwrite**, it is easy to see that computing the parity can be done with less operations than in the non-modified RAID-5 or RAID-6 case: We do not need to read of previously written blocks because the data is new. Thus, updating the parity only implies to XOR the new data with the current parity block (and the previous one, as it may have been modified by another OSSs in the same stripe).

In the second category, **sequential / Overwrite**, this optimization cannot be done immediately, but we can interpose the WCS layer which converts this kind of accesses into the first one.

Finally, if we are on a Random scenario, we need also to keep a maintain a bitmap of used space to confirm that we are not overwriting and know which blocks need to have their parity calculated.

### 2.1.2 Basic Avoidance : Write Cache Servers

The idea of the Write Cache Servers (WCS) is to always access full stripes and cache the data blocks for subsequent updates. As the WCSs layer will convert the overwriting applications into non-overwriting, we can modify the parity update workflow (Figure 16) removing the old data read. Although the removed operation is done in parallel, we are reducing the congestion of the Data OSSs.

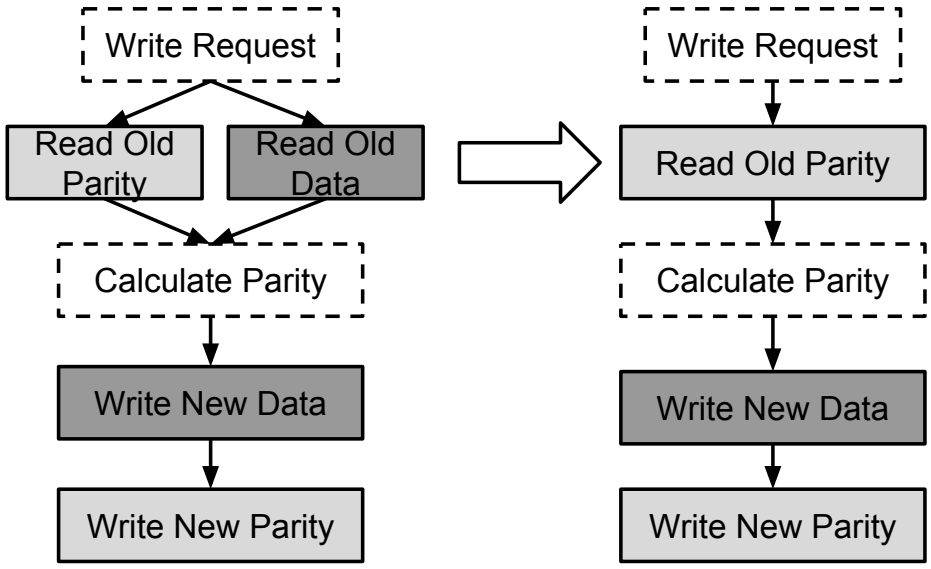



Figure 16: RAID-5 (left) to optimized RAID-5 (right), shading represent different OSSs

However, we still need to read the old parity because other OSSs from the same stripe may have issued a parity update on the same parity block. Removing that read would reduce the reliability of the system as we would need to buffer the requests.

As a result of this new data-flow, the number of operations when writing new data is reduced by 25% using RAID-5 and by 16% using RAID-6. As we will see in the evaluation, the system has less overhead so the performance impact will be higher.

The new I/O layer, built using a set of WCSs, will track the original block position and their place inside the cache zone. With that layer, we can write all data as if it were written on a new file, having a tight control of the overwrites. Once a whole stripe is written, it can be moved to the original placement without having to compute any parity. With this technique we reduce the number of reads in the critical path of a write operation and replace it with a whole stripe read (probably cached) and write, all done out of the critical path. The original block position is also used to support partial movements (evictions) from the cache when the cache is full.

The WCSs can be deployed inside the original OSSs or using different nodes. Nevertheless,

the best performance is obtained sharing the OSSs. An example of the workflow for two writes is shown in Figure 17. In this example, we can see how the writes are going to different OSS but the parity is calculated on the same parity OSS. Using the proposed technique we do not need to read the original data of the OSS, avoiding one read operation per write. The example has the WCS layer simplified, as the default setup consists of the I/O library and 3 physical WCSs, integrated inside the OSSs (sharing the devices). Finally, the parity read can be cached in the corresponding WCS, however we are using the worst performance scenario that does not allow the parity to be cached (as we may have it evicted from the cache, due to that we have more I/O).

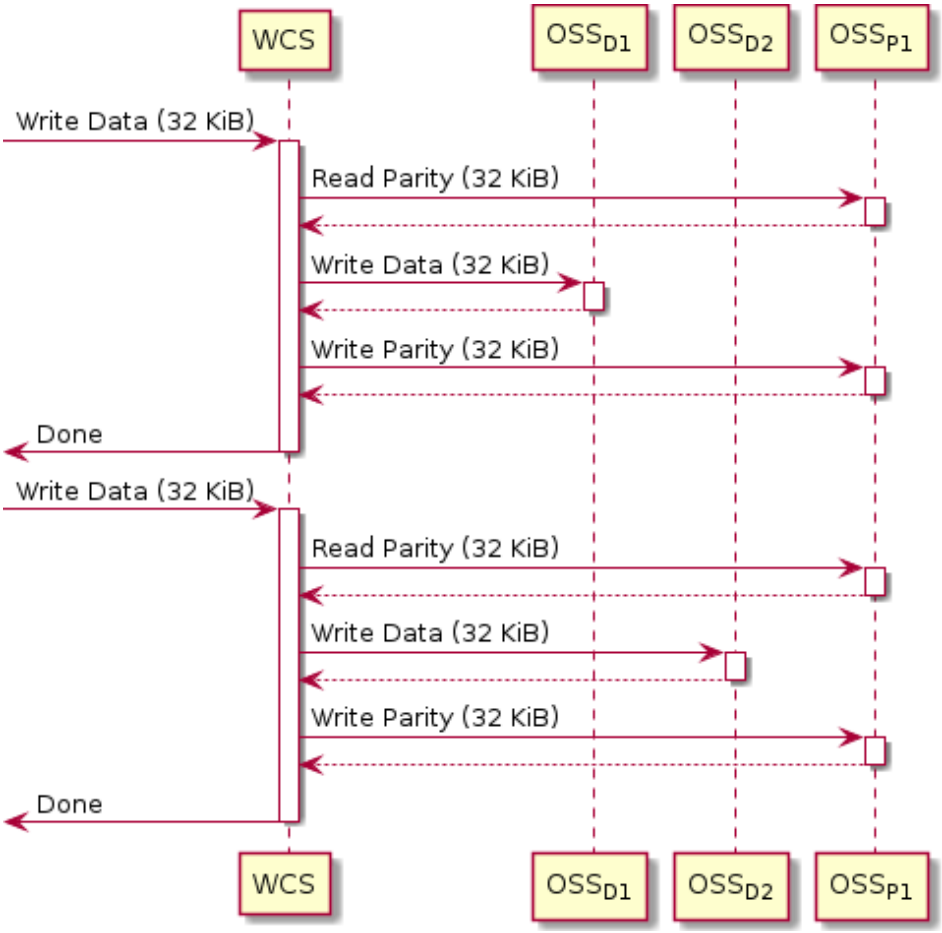



Figure 17: Sequence diagram of Basic Avoidance, with two writes of 32 KiB issued.

## 2.2 Simulation Design

The simulation is created using the OMNet++ event-driven framework [30] and has the following entities (see Figure 18):

- I/O Clients simulating I/O requests.
- A set of OSSs each of them representing a set of devices (SSDs or HDDs).

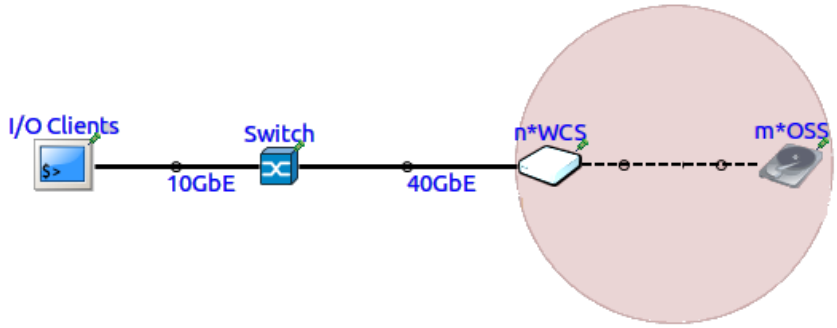



Figure 18: System Layout - OMNet++

- The optional WCSs are deployed on the existing OSSs, but other layouts are available (e.g.,, Hybrid approaches [23] using HDDs and SSDs).
- Network switches connecting client and the PFS (i.e.,, 10 GbE) and for decoupled WSS-OSS experiments, they also connect them.

The following software aspects are covered by the simulation:

- The resulting message size of I/O requests. The actual header size of Lustre RPC messages has been determined and is added to data payloads.
- The distribution of data among servers: Inside the simulation the number of OSSs ($m$) and the test number define how the different files are distributed. By default, each file is divided through all the available OSSs (increasing parallel performance) but we can setup them using Lustre [20] parameters: number of OSSs used, starting OSS per file and the size of the stride (1 MiB as default).
- The CPU time for serving I/O requests or calculating parity are not covered by the simulation. On RAID-6 produce non neglectable CPU overhead; of course, in a real system the vendor would ensure that the deployed CPU is fast enough to handle the data volume without reducing throughput. Therefore, this effect is expected to play a minor role and, thus, is not simulated here.
- Evictions, movements of data from WCSs to OSSs due to that the cache space is full, are supported.

In the next subsections we will explain the decisions taken on the disk simulation and on the simulated application.

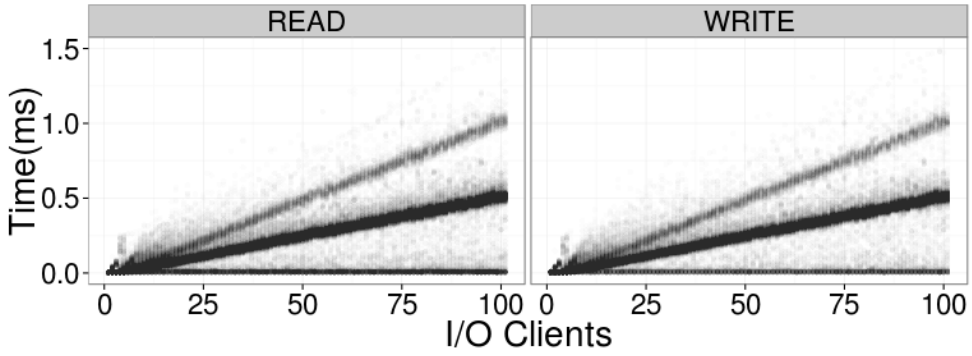
### 2.2.1 Disk Simulation

We started our efforts using DiskSim [7], but we faced important problems when we added SSD capabilities. When we started using a larger number of devices (i.e.,, more or less 64), we had memory corruption problems. In particular, we had found that some part of the code still uses 32 bits pointers and the simulation stops working when the memory space of the new device exceeds the 4GiB zone. Nevertheless, we were unable to find such bug on the simulator code, and moved to another solution.

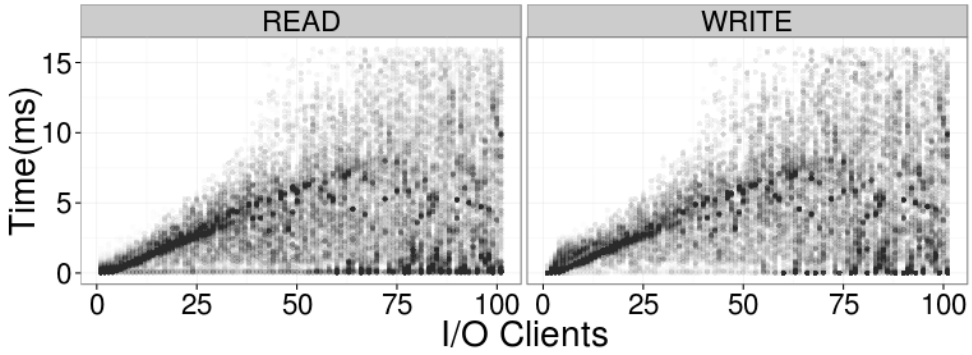
After analysing how the simulation requests I/O to the disk, we observed that only a small set of parameters are needed to obtain the needed request time. For instance, with the level of detail of our simulation and the high number of I/O, we can drop the block position information from the different requests. These reductions, make possible to gather the needed response time from the real devices.

Then for each number of clients and each block size, we measure the response time for each I/O request. This results in a cloud-like plot as illustrated in Figure 19. This information is stored and used in the simulator: Since the different variables are well defined in the simulation, the simulator randomly pick the response time from one of the observations.

On short, to acquire such information we run a set of FIO [1] tests with a specific request size (4KiB, 8KiB...1MiB) and a number of threads (1..100) doing parallel and random R/W operations. The modelled devices are a Western Digital Black Hard Disk with 750GB and 7200 RPM and an Intel SSD 320 Series with 160 GB.




(a) **SSD** response time.




(b) **HDD** response time.

Figure 19: Response time distribution using 32K - Write/Read random mixed workload. Axis X increases the number of parallel requests issued to the device.

We believe that this models the different caches or read/write inefficiencies in a statistical way that is more than sufficient for our simulation's level of detail. As a positive point over DiskSim, the devices modelled are newer and the number of devices can be increased due to a reduction of the simulation CPU and memory cost. However, we expect that new developed methods based on machine learning can produce such disk models easily in the future.

Table 1: Simulation parameters for Weak-Scaling results for RAID-5 and RAID-6.

| Method | OSS | WCS | Write Size | Stripe unit | RAID |
|---|---|---|---|---|---|
| RAID-0 | 512 | N/A | 128 KiB | 1MiB | 0 |
| STD-RAID | 512 | N/A | 128 KiB | 1MiB | 5/6 |
| WCS | 512 | Yes | 128 KiB | 1MiB | 5/6 |

### 2.2.2 Client Simulation

The simulated application initiates writes, using a set of clients issuing writes to the I/O layer. During a single simulation run, write size is fixed for all the clients and the writes are distributed along a different file per application to avoid overwrites. The behaviour per application is similar to the checkpointing creation of FLASH [18].

For the **Delayed Parity** evaluation, one process of each application acts as master issuing the `START_DELAYED_PARITY` and the `END_DELAYED_PARITY` hints and all the clients waits (barrier) until the parity is calculated.

## 2.3 Evaluation

To assess the effectiveness of the novel strategies, we repeat each measurement with different seeds. Each simulation run stops when we have a minimum of 1000 seconds of simulated time and the bandwidth values have an error lower than 0.1 MB/s (95% c.i.).

We analyse the outcome of our proposal using RAID-5 in subsection 2.3.1 and a simplified RAID-6 evaluation in subsection 2.3.2.

The simulation is setup to avoid evictions of data to simplify the analysis. We also avoid using memory caches as we assume that it is stored to the disk to obtain the worst baseline. Finally, the clients does not generate reads operations on the presented experiments, but, as expected, the performance of the system also improves with reads as we are reducing the pressure on the different devices.

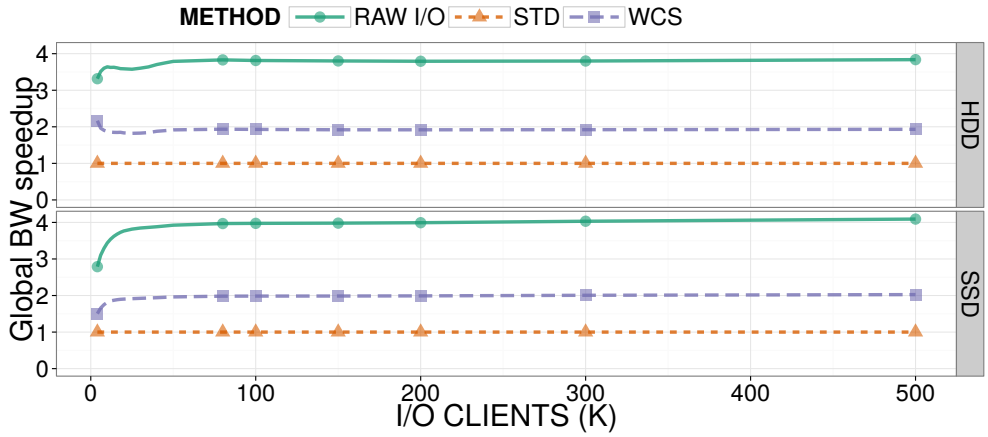
In short, and to describe the next results, **RAID-0** is the performance obtained from the OSSs when we do not have parity calculations. **STD-RAID** (Figure 16, left) is a standard RAID with parity and without optimizations. Finally, our proposal is **WCS**. If not specified, the results are scaled relative to STD-RAID for each x-axis point.

### 2.3.1 RAID-5 results

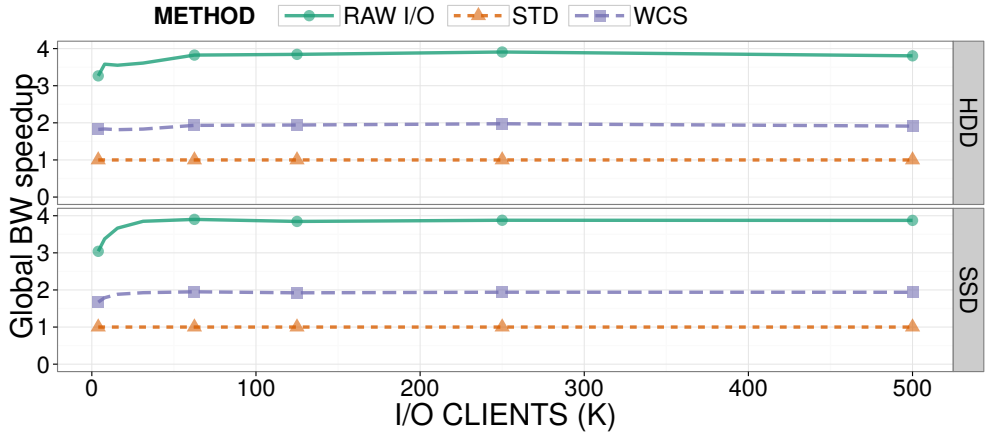
We will present results using Weak-Scaling and Strong-Scaling: Weak-Scaling means that the problem size increases with the number of clients, thus regardless of scaling a client will process the same amount of data. In Strong-Scaling we increase the number of clients, but keep the problem size. Consequently, Strong-Scaling increases the network and disk congestion when we increase the number of clients, due to the higher number of requests.

For all the scenarios, the data is partitioned to avoid overwrites according to their block size.

**Weak-Scaling results** We can find the values simulated in Table 1, each client writes 128 KiB per write distributed along a file and coordinated with the other clients to avoid overwrites. As we can see on the simulation results in Figure 20.a, write performance is four times slower on the **STD-RAID** with respect to the **RAID-0** both in HDD and SSD. Since we go from one operation to four




(a) **Weak-Scaling**, write size per request is 128 KiB.




(b) **Strong-Scaling**, $\frac{\text{IO Clients}}{\text{block size}}$ is constant.

Figure 20: **RAID-5** Write Bandwidth results w.r.t. STD-RAID.

operations per write to two different OSSs, the performance degradation is expected. Moreover, the **WCS** scenario results in a speed up of 2x with respect to the **STD-RAID**, by eliminating the read operation of the old data. This reduces the overhead on the OSSs and network links.

Apart from the results presented with a 128 KiB write size, Figure 21 shows the relative performance between the different methods when we use different block sizes and a fixed number of clients (250,000).

We can observe that the relative performance is stable using HDDs. However, for only a 4KiB blocks and SSDs, the STD-RAID achieves a better performance, thus the benefit of the other schemes is lower.

As opposed to HDDs, SDDs offer a better performance of parallel operations so a reduction of them on 4 KiB block sizes does not produce the same improvement with respect the STD-RAID.

**Strong-Scaling Results** The experiments done are described in Table 2, with their results on the Figure 20.b. We can see how the results, normalized to **STD-RAID**, are similar to the Weak-

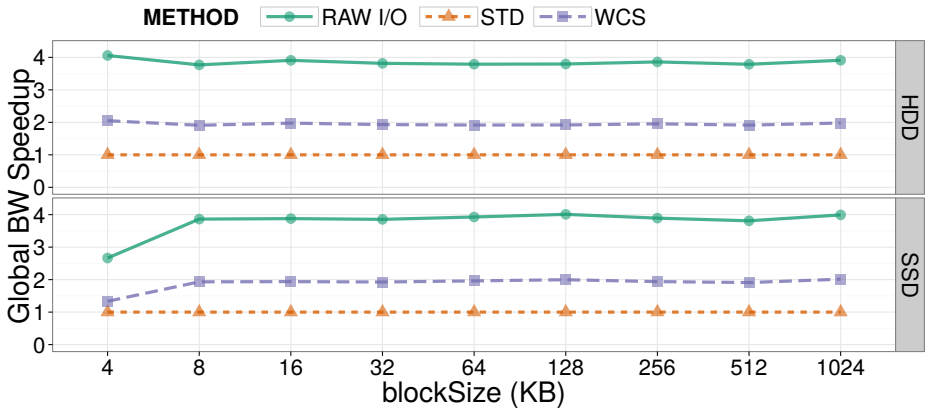



Figure 21: **RAID-5** performance w.r.t. STD-RAID when we have different block sizes with 250,000 clients.

Table 2: Simulation parameters for Strong-Scaling results for RAID-5 and RAID-6.

| Method | OSS | WCS | Write Size | Stripe unit | RAID |
|---|---|---|---|---|---|
| RAID-0 | 512 | N/A | 8 KiB - 1 MiB | 1MiB | 0 |
| STD-RAID | 512 | N/A | 8 KiB - 1 MiB | 1MiB | 5/6 |
| WCS | 512 | Yes | 8 KiB - 1 MiB | 1MiB | 5/6 |

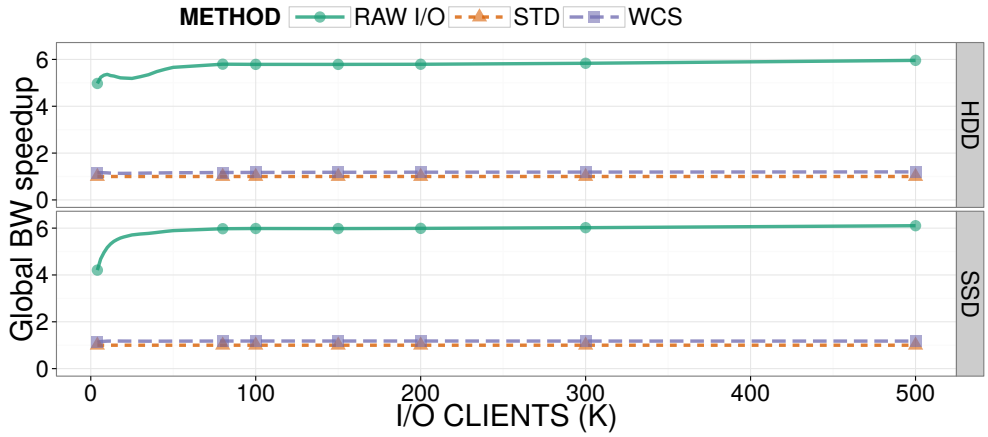
Scaling ones. Specifically, we obtain the same results for **RAID-0**, **STD-RAID** and **WCS**, so the difference on block size (i.e.,, from 128 KiB for Weak-Scaling to 8 KiB for Strong-Scaling in the 500,000 client scenario) is not important at the scale we are working.

### 2.3.2 RAID-6 results

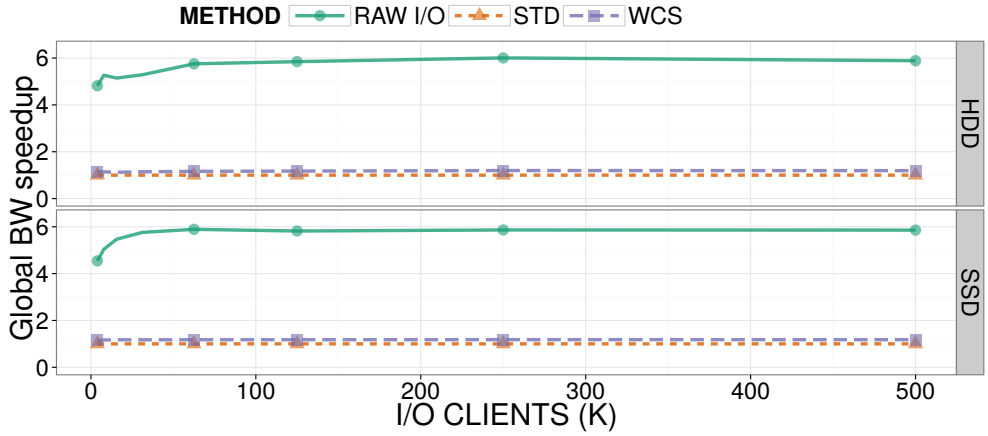
The complexity and variability of RAID-6 deployments make it more difficult to simulate than RAID-5. Using RAID-6 with two or more parity devices, where location and modification rules depend on the erasure codes used, adds too many variables into the evaluation and is out of the scope. The position of the parity devices can modify the number of operations of each OSS and the network communication patterns, but in any situation, our proposal will always optimize them. We have selected two horizontal parity devices per stripe for RAID-6. The different parameters are the same as in the RAID-5 scenario, found in Table 1 and Table 2.

**Weak-Scaling results** As we can observe in the Figure 22.a, we found that the **STD-RAID** performance of RAID-6 is lower than RAID-5 and the improvement on performance using **WCS** is a bit lower (1.18x speedup) as we only remove a 16% of the operations instead of the 25%.

**Strong-Scaling results** Testing Strong-Scaling with the same parameters described on Table 2 but using a RAID-6, we obtain similar results (Figure 22.b). Like the RAID-5 results, the block size is not important on the simulated scenarios for all the experiments.




(a) **Weak-Scaling**, write size per request is 128 KiB.




(b) **Strong-Scaling**, $\frac{\text{IO Clients}}{\text{block size}}$ is constant.

Figure 22: **RAID-6** Write Bandwidth results w.r.t. STD-RAID. RAID-6 setup with 2 parity devices in horizontal.

## 2.4 Related work

RAID-5 or RAID-6 are redundant systems that provide a way to recover from a data loss using the remaining disks, so for RAID-5 we can recover using $n-1$ disks and with RAID-6 we can recover using only $n-2$ disks. RAID-6 can use a huge range of erasure codes offering different performance and recoverability values. One such example is *Reed-Solomon*, which maps to a polynomial equation so the missing data recovers using interpolation, therefore we can use extra devices to extend the recovery capabilities (for example 12 data disks using 4 redundant disks, will be able to recover any failed disk from 12 correct disks of the 15 available that are still working).

### 2.4.1 Node-local Redundacy

Inside RAID-6 research at local layer, we can find optimizations of the erasure codes as in *P-Code* [17] and *HDP-Code* [31], and optimizations of the erasure codes using specialized hardware

(FPGAs and GPUs) as in Gilroy [14] and Curry [11]. Moreover, we can find improvements with SSD RAIDs [26] taking into account the wear-levelling of the parity stripe. Directly related to the partial write on stripes problem, we found H-Code [32] with a performance improvement of 15.54% and 22.17% compared to *RDP* and *EVENODD* erasure codes. Finally, there are also patents that solve the problem in the hardware level as Lyons [22] and Baylor [2], reducing the reads and writes done at the controller level.

Those proposals may not be fully usable or become inefficient at the storage server layer, as it involves network communication and bigger latencies. Despite of this, some of them may improve the performance due to the different parity layouts or calculations. Our work is transparent to the reliability configuration used and will improve their performance.

### 2.4.2 Distributed Redundancy

Inside distributed redundancy, with can find TickerTAIP [10] showing how to build a RAID system using the network as transport method. It can be seen as a preliminary approach to support RAID parity schemes over PFS, finally RAID-x [16] presents how to optimize the previous proposal using a mirroring mechanism. For small writes, this mechanism reduces the number of operations by 1/4.

On the PFS plane, GlusterFS supports mirroring schemes and some requests have been done to support RAID 5/6 schemes. Hadoop (HDFS [5]) supports file replication. Finally, PanFS supports file level RAID configurations using triple parity data [25].

## 2.5 Conclusions and Future Work

It is a well-known fact that reliability is needed since as the number of disks are increasing, the global MTBF (Mean time between failures) decreases. 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 light-weight clients the number of parity updates will increase.

We propose 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 the old data from the parity update workflow. This proposal, WCS, improves the write performance of the standard workflow by a 1.18x to a 2x depending of the RAID level (6 or 5, respectively).

In this work, we used simulation to predict the impact of these strategies. The implementation in existing file systems is non-trivial and out of scope of this deliverable; nevertheless this theoretical consideration steers the direction of a beneficial implementation in the future.

For the next deliverable we will explain a new proposal using collaboration between clients/runtime and PFS providing more benefits to this scenario.

# References

[1] J. Axboe. FIO-flexible IO tester. 2008.

[2] S. Baylor, P. Corbett, and C. Park. Efficient method for providing fault tolerance against double device failures in multiple device systems, 1999. US Patent 5,862,158.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[4] D. Bonachea. GASNet Specification, v1.8 . Technical report, U.C. Berkeley, 2006.

[5] D. Borthakur. Hadoop - HDFS Design. `http://hadoop.apache.org/docs/r1.2.1/hdfs\_design.html`, 2014. Apache, accessed 2014.

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

[7] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[8] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with OmpSs. *Euro-Par 2011 Parallel Processing*, pages 555–566, 2011.

[9] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, 2012.

[10] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes. The tickertaip parallel raid architecture. *ACM Transactions on Computer Systems (TOCS)*, 12(3):236–269, 1994.

[11] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell. Accelerating reed-solomon coding in RAID systems with GPUS. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6. IEEE, 2008.

[12] V. Deenadhayalan. GPFS Native RAID slides, Invited talk at LISA'11. `http://www.usenix.org/events/lisa11/tech/slides/deenadhayalan.pdf`, 2011.

[13] J. J. Dongarra, I. High, and P. C. Systems. Overview of the HPC Challenge Benchmark Suite.

[14] M. Gilroy and J. Irvine. RAID 6 hardware acceleration. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006.

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

[16] K. Hwang, H. Jin, and R. Ho. Raid-x: A new distributed disk array for i/o-centric cluster computing. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 279–286. IEEE, 2000.

[17] C. Jin, H. Jiang, D. Feng, and L. Tian. P-Code: A new RAID-6 code with optimal properties. In *Proceedings of the 23rd international conference on Supercomputing*, pages 360–369. ACM, 2009.

[18] 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.

[19] P. Luse. Understanding RAID-5 and I/O Processors. `http://www.dell.com/content/topics/global.aspx/power/en/ps2q03_luse`, 2003. Dell Power Solutions, accessed 2014.

[20] Lustre. Lustre openSFS web page. `http://www.opensfs.org/lustre`, 2014.

[21] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[22] G. Lyons. Method and apparatus for improving sequential writes to RAID-6 devices, Aug. 8 2000. US Patent 6,101,615.

[23] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, and L. Zeng. Hpda: A hybrid parity-based disk array for enhanced performance and reliability. *Trans. Storage*, 8(1):4:1–4:20, Feb. 2012.

[24] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[25] Panasas. PanFS RAID. `https://www.panasas.com/products/panfs/PanFS_RAID`, 2014. Panasas, accessed 2014.

[26] K. Park, D.-H. Lee, Y. Woo, G. Lee, J.-H. Lee, and D.-H. Kim. Reliability and performance enhancement technique for SSD array storage system using RAID mechanism. In *Communications and Information Technology, 2009. ISCIT 2009. 9th International Symposium on*, pages 140–145. IEEE, 2009.

[27] R. B. Ross, R. Thakur, et al. PVFS: A parallel file system for linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000.

[28] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[29] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[30] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, volume 9, page 185. sn, 2001.

[31] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie. HDP code: A Horizontal-Diagonal parity code to optimize I/O load balancing in RAID-6. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 209–220. IEEE, 2011.

[32] C. Wu, S. Wan, X. He, Q. Cao, and C. Xie. H-Code: A hybrid MDS array code to optimize partial stripe writes in RAID-6. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 782–793. IEEE, 2011.