Journal Title
XX(X):1–11
©The Author(s) 2016
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Dynamic Reconfiguration of Non-iterative Scientific Applications: A Case Study with HPG-aligner

Sergio Iserte[1], Héctor Martínez[1], Sergio Barrachina[1], Maribel Castillo[1], Rafael Mayo[1] and Antonio J. Peña[2]

Abstract

Several studies have proved the benefits of job malleability, i.e., the capacity of an application to adapt its parallelism to a dynamically changing number of allocated processors. The most remarkable advantages of executing malleable jobs as part of a high performance computer workload are the throughput increase and the more efficient utilization of the underlying resources.

Malleability has been mostly applied to iterative applications where all the processes execute the same operations over different sets of data and with a balanced per process load. Unfortunately, not all scientific applications adhere to this process-level malleable job structure. There are scientific applications which are either non-iterative or present an irregular per process load distribution. Unlike many other reconfiguration tools, the Dynamic Management of Resources Application Programming Interface (DMR API) provides the necessary flexibility to make malleable these out-of-target applications.

In this paper we study the particular case of using the DMR API to generate a malleable version of HPG-aligner, a distributed-memory non-iterative genomic sequencer featuring an irregular communication pattern among processes. Through this first conversion of an out-of-target application to a malleable job, we both illustrate how the DMR API may be used to convert this type of applications into malleable, and test the benefits of this conversion in production clusters. Our experimental results reveal an important reduction of the malleable HPG-aligner jobs completion time compared to the original HPG-aligner version. Furthermore, HPG-aligner malleable workloads achieve a greater throughput than their fixed counterparts.

Keywords

High-throughput Computing, Dynamic Resource Management, Adaptive Workload, MPI Malleability, Bioinformatics Productivity

Introduction

The expense in High-performance Computing (HPC) infrastructures is rapidly growing as more and more users are requiring its services, and the applications requirements are becoming more and more demanding. HPC systems are being used by a large number of users which execute a wide variety of complex applications. Whereas the users would like their applications to finish as soon as possible, HPC system administrators and managers are mostly concerned about the throughput of their premises, i.e., the number of jobs that are completed per unit of time.

Generally, a user of an HPC system asks for a given number of processes to be used for executing his/her application, and this request is stored in a pending jobs queue. The order in which the pending jobs will be executed is determined by a queue manager that will try to maximize the resources usage while keeping under an acceptable threshold the waiting time of the retained jobs. In this scenario, and depending on the users' requests and the system resources, both the resource usage and the waiting time may well be suboptimal.

On the other hand, the number of processes of a malleable job is dynamically determined by the system, not by the user, and may be subsequently modified during its execution Feitelson (1996). Therefore, by including malleable jobs in an HPC system workload, an adaptive workload, which can be tailored to the job queue status and the resource occupation level, is obtained. In this scenario, a fair balance between global throughput and individual performance can be more easily attained.

To tailor the resources assigned to an application to the available resources, dynamic reconfiguration tools are used. These tools are composed at least of the next two components: i) a resource management system (RMS); and ii) a parallel runtime. While the former controls which resources are available at a given time, the latter performs the job resizes required to accommodate the malleable job to the variations on the system load.

Current malleability frameworks are aimed to regular applications that either have a clearly defined iterative scheme or do not consider data transfers among processes

[1]Universitat Jaume I (UJI), Castelló de la Plana, Spain
[2]Barcelona Supercomputing Center (BSC)

Corresponding author:
Sergio Iserte, Department of Computer Science and Engineering, Universitat Jaume I, 12071, Castelló de la Plana, Spain.
Email: siserte@uji.es

(resorting to the disk for storing and loading the current state when a reconfiguration is performed). In fact, all the malleability efforts made so far have ignored *out-of-target* applications, i.e., non-iterative applications or applications with an irregular communication model (such as the *producer-consumer*) that requires data transfer management.

The main goal of this work is to demonstrate that by leveraging the DMR API Iserte et al. (2017), out-of-target applications can be considered for malleability. Unlike other malleability frameworks, the DMR API provides the coding flexibility and programming easiness that are required to reconfigure most of the current target and *out-of-target* applications.

To illustrate this capability, we have ported the *out-of-target* biological application HPG-aligner Medina et al. (2016). HPG-aligner is an MPI-based fast RNA sequencer that can be executed on large-scale clusters. HPG-aligner is used to obtain the RNA of an individual by aligning a large number of small fragments of his/her RNA to a reference genome. Our interest in porting this application is to increase the number of RNA fragments that can be sequenced per unit-of-time: i.e., to improve the productivity of a system when being used to map RNA sequences.

Since HPG-aligner is the first non-iterative application with a producer-consumer data communication model that has been converted into malleable and introduced in an adaptive workload, with this study we redefine the boundaries of malleability. Furthermore, the experimental results unveil an important reduction in the job completion time (queue waiting time plus execution time) —by minimizing the jobs waiting time—, and present a considerable increase in the throughput when processing adaptive workloads composed exclusively of HPG-aligner instances.

Related work

Next we present the most relevant efforts in providing frameworks to develop malleable applications, and the applications they have been evaluated on.

ReSHAPE, a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment, was presented in Sudarsan and Ribbens (2007). It uses performance data collected at runtime to support reconfiguring actions, and increases the number of processes of an application when there are available processors in the system and the iteration time has improved due to a previous increase or when the number of processes has never been expanded before. It has been evaluated on:

- Three kernels and one pseudo-application from the NAS Parallel Benchmarks[*]: Integer Sort (IS), Conjugate Gradient (CG), Fourier Transform (FT), and Lower-Upper Gauss-Seidel solver (LU) (see Sudarsan and Ribbens (2009).)
- LAMMPS[†], a molecular dynamics simulator. This is the most complex HPC application that has been turned into malleable so far. The malleability capabilities were added exploiting the LAMMPS support for file-based checkpoint and restart (C/R) (see Sudarsan et al. (2009).)
- A synthetic workload (see Sudarsan and Ribbens (2016).)

Flex-MPI, an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems, is described in Martín et al. (2013). It uses a computational prediction model to evaluate multiple potential reconfiguration scenarios and choose the one which is predicted to best satisfy a performance objective. It was evaluated on three iterative *single program multiple data* (SPMD) applications: Jacobi, CG, and EpiGraph[‡]. In Martín et al. (2015), a more comprehensive evaluation of its performance on these applications was conducted using a synthetic workload.

The DMR API, dynamic management of resources for OmpSs jobs using Slurm, was presented in Iserte et al. (2016). The DMR API reconfiguration policy considers queued jobs and available resources in order to perform a purely throughput-aware scheduling. It was evaluated on the Conjugate Gradient (CG), Jacobi, and the N-body problem Iserte et al. (2017).

In Comprés et al. (2016) the malleability framework Elastic MPI was presented. In this work, Slurm [§] and MPICH [¶] are extended with new functionalities in order to deal with jobs reconfiguration. Slurm has been extended to manage the creation and deletion of MPI processes while handling the resource allocation; whereas new functions were added to MPICH. When these functions are used, the application is initially defined as malleable and its processes will periodically check whether Slurm has initiated a reconfiguration. Although this framework is evaluated on a master-worker application, the application handles the transfers of its sparse grid via MPI IO, without using point-to-point communications and only broadcasting a few variables.

An extension of the Torque/Maui batch system with efficient adaptive scheduling for malleable and evolving applications, was introduced in Prabhakaran et al. (2015). Regarding the reconfiguration criteria, the authors resort to a *dependency-based expand/shrink* algorithm that considers the minimum job requirements and the resources of the later expanded job. It was evaluated modifying the ESP benchmark Wong et al. (2000) to contain various percentages of rigid, malleable, and evolving jobs. For the malleable part, one synthetic application was replaced by LeanMD[‖], a molecular dynamics mini-application which implements a simplified version of the force calculations of NAMD[**].

In conclusion, it should be emphasized that none of the aforementioned frameworks, although providing different reconfiguration approaches, have ever been applied over non-iterative applications with irregular communications patterns with complex data structures.

This paper presents the first irregular malleable HPC application with a producer-consumer architecture that

[*] http://www.nas.nasa.gov/Software/NPB
[†] http://lammps.sandia.gov
[‡] http://epigraph.mpi-inf.mpg.de/WebGRAPH
[§] http://slurm.schedmd.com
[¶] http://www.mpich.org
[‖] http://charm.cs.illinois.edu/research/leanmd
[**] http://www.ks.uiuc.edu/Research/namd

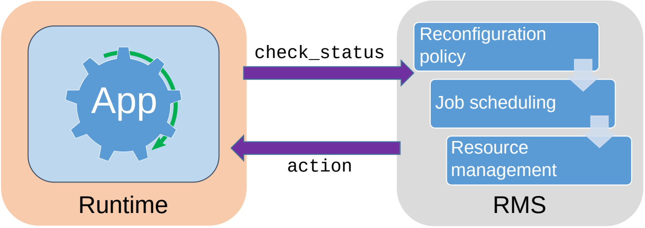

Figure 1. Communication scheme of the DMR API reconfiguration.

require data redistribution among processes. With this study, we prove that the DMR API can be leveraged to a wider range of applications, easing the path towards the deployment of full adaptive workloads in HPC facilities.

Background

In this section we present the DMR API malleability framework and the scientific application targeted in this work.

Overview of the DMR API

A dynamic reconfiguration system relies on two main components: an RMS and a parallel runtime. In particular, the DMR API, depends on the OmpSs[††] runtime (called Nanos++) and Slurm Yoo et al. (2003) workload manager. As shown in Figure 1, the parallel runtime supports the execution of the application and allows it to inform the RMS that it can be resized by issuing a check status request. On the other hand, the RMS is aware of the resource utilization and the queue of pending jobs. When the RMS receives a check status request, it inspects the global status of the system, decides whether any rescaling action should be done, and informs of this decision to the parallel runtime. If a rescale action is due, the RMS, the parallel runtime, and the application will collaborate to continue the application execution with a different number of processes.

A general schema of a malleable job using the DMR API is depicted in Listing 1. Once the data has been initialized (line 2), the main loop is entered (line 7). At the beginning of each main loop iteration, the application calls the DMR API `dmr_check_status()` function (line 8) to inform the resource manager that a resizing can be performed. If no action is returned, the execution will continue normally (line 13). Else, if the RMS responds with a resizing action, either *expand* or *shrink*, a *handler* of a new communicator and the number of processes created in that communicator (*handlerNProcs*) will be returned. Using this information, the application will set the data dependencies and the communication pattern in order to redistribute the data among the new processes (line 10).

Overview of HPG-aligner

HPG-aligner is a bioinformatics application for fast and accurate mapping of RNA sequences on a cluster of computers Martínez et al. (2013). It employs several MPI processes, which exploit node level parallelism through POSIX threads. One of these MPI multi threaded processes is configured as a *writer*, and the others as *workers* (see Figure 2). Each *worker* operates over a particular part of the input file, composed by short RNA fragments (*reads*) produced by a Next Generation Sequencing (NGS) sequencer. At the beginning of the execution, each *worker* calculates the indexes of its part of the input file. Then, each *worker* performs its computation over its self-assigned reads, and sends the alignments it obtains to the *writer*. Meanwhile, the *writer* saves the reported alignments to disk.

```
1  void main(void) {
2    init(data);
3    compute(data, 0);
4  }
5
6  void compute(data, step) {
7    for (t = step; t < TIMESTEPS; t++) {
8      action = dmr_check_status(&handlerNProcs,
     ↪ &handler);
9      if (action) {
10       #pragma omp task inout(data) onto(handler
     ↪ , myRank)
11       compute(data, t);
12     } else {
13       /* Computation */
14     }
15   }
16 }
```

Listing 1: General schema of a malleable application using the DMR API.

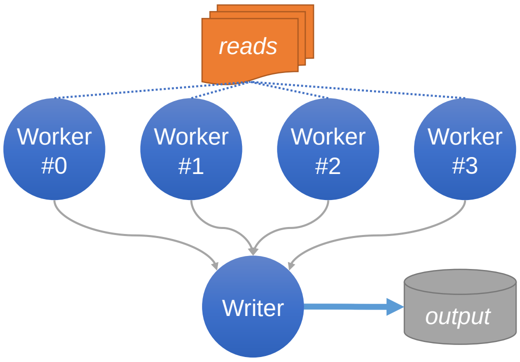

Figure 2. HPG-aligner original version workflow.

HPG-aligner malleable version

In this section we describe how the DMR API has been employed to convert HPG-aligner into a malleable application.

Adapting the HPG-aligner workflow

The original implementation of HPG-aligner performs a static distribution among all the processes of the whole dataset (i.e., all the RNA *reads* obtained from an NGS sequencer). This strategy is not desirable for a malleable application, since it would complicate both the redistribution of the yet-to-be-done work among the new processes, and

[††] http://pm.bsc.es/ompss

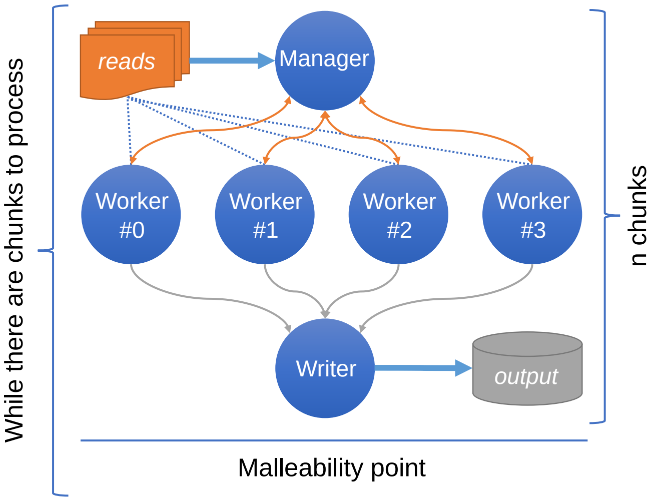

Figure 3. HPG-aligner malleable version workflow.

the determination of the synchronization point that all the processes must reach to evaluate whether a reconfiguration should be performed. Thus, we have redesigned the HPG-aligner workflow in order to split the input workload into a user-defined number of chunks, which will be dynamically assigned to the *worker* processes.

To allow HPG-aligner to dynamically distribute the dataset, a new process is created, the *manager*, which is in charge of the distribution of the input dataset chunks among the *worker* processes. In this new workflow, in Figure 3, when a *worker* has finished its previously assigned work, it asks the *manager* for more work. Then, the *manager* computes the index of the next dataset chunk to be processed, and assigns it to that *worker*.

Moreover, the HPG-aligner workflow has been also modified in order to allow the processes to periodically reach a malleability point, i.e., a synchronization point where the RMS can be asked if a reconfiguration should take place. This may be attained by instructing all the HPG-aligner processes to reach the malleability point after a given number, n, of chunks have been processed. Figure 3 illustrates this iterative schema: all the HPG-aligner processes cooperate to compute n chunks of reads, and then all of them reach the malleability point, where a reconfiguration could be triggered. These two steps, processing and reaching the malleability point, are repeated until all the chunks have been processed.

To implement this iterative schema and synchronize all the processes at a malleability point, the next strategy and communication schema have been employed. After dispatching *n* chunks, the *manager* stops distributing work. Instead, whenever a worker asks for more work, the *manager* signals to proceed to the malleability point. The *worker*, for its part, after receiving this signal, propagates it to the *writer*. This communication schema, see Figure 4, ensures that all processes will eventually reach the malleability point after n chunks are processed.

The algorithms used to implement the described strategy and communication schema are depicted next using an MPI pseudo-code. In this pseudo-code, the first argument of the MPI send calls is the data to be transferred, and the second, its destination. Likewise, the first argument of the MPI receive calls is a buffer for the data to be received, and the second argument, the senders from whom that data is received.

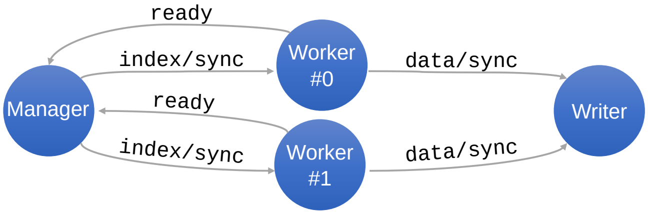

Figure 4. Communication schema of the processes in the HPG-aligner malleable version.

Algorithm 1 shows the pseudo-code corresponding to a *manager* iteration. While there are chunks to be processed (line 1), the *manager* waits for a work petition from any *worker* (line 2), and in answer to each of these, it provides the index of the next chunk to be processed (line 3). When all the chunks of the current iteration have been assigned, a special signal (a -1) is sent to all the *workers* (lines 4 to 6) in order to synchronize them at the malleability point (line 7).

Algorithm 1 Pseudo-code of a *manager* iteration

```
1: while chunks do
2:     MPI_Recv(&$worker\_id$, MPI_ANY_SOURCE)
3:     MPI_Send($index$, $worker\_id$)
4: for each $worker$ do
5:     MPI_Irecv(&$worker\_id$, $worker$)
6:     MPI_Isend(-1, $worker$)
7: /* Malleability point */
```

Algorithm 2 describes the pseudo-code of a *worker* iteration. First, the *worker* informs the *manager* that it is ready to process more work (line 2). Then, it waits until the *manager* sends back the index of the next chunk of reads assigned to it (line 3). If a -1 index is received (lines 4 to 6), it will propagate this signal to the *writer* and proceed to the malleability point (line 9). Otherwise, it will process the assigned chunk and send its results to the *writer*.

Algorithm 2 Pseudo-code of a *worker* iteration

```
1: while 1 do
2:     MPI_Send($worker$, $manager$)
3:     MPI_Recv(&$index$, $manager$)
4:     if $index$ == -1 then
5:         MPI_Send(-1, $writer$)
6:         break
7:     compute($index$, &$data$)
8:     MPI_Send($data$, $writer$)
9: /* Malleability point */
```

Finally, Algorithm 3 describes the pseudo-code of the *writer* iteration. The *writer* listens for data from any worker (line 2). If a -1 is received, a counter is increased, and when the value of this counter reaches the number of *workers*, it will proceed to the malleability point (lines 3 to 6). Otherwise, if an actual result is received, it will write it to disk (line 8).

Algorithm 3 Pseudo-code of the *writer* iteration

```
while 1 do
    MPI_Recv(&$data$, MPI_ANY_SOURCE)
    if $data == -1$ then
        $cnt = cnt + 1$
        if $cnt == n\_workers$ then
            break
    else
        write_to_disk($data$)
/* Malleability point */
```

HPG-aligner data redistribution patterns

On the previous section, a malleable workflow was proposed for HPG-aligner so that it could periodically reach a malleability point to ask the RMS whether it should be reconfigured or not before continuing processing the input data. In this section, we will discuss how the data of the HPG-aligner processes could be redistributed in the event of a reconfiguration.

HPG-aligner reconfigurations consider two data structures in their data redistributions. To begin with, all the HPG-aligner processes share some identical information (*common data*). Furthermore, HPG-aligner *workers*, during their execution, keep the mapping information to the reference genome (*worker data*). This data structure is used by the HPG-aligner *workers* to improve their performance and their mapping quality. It is worth notice that although the workers do not require their particular *worker data* to be initially populated, HPG-aligner uses the aggregated data from all the workers in a later stage in order to be able to correctly align those reads that could not be previously aligned.

Common data and *worker data*, hereafter referred as *cData* and *wData* respectively) must be distributed from the current processes to the new processes. For this reason, we have defined two different patterns depending on the reconfiguration type. If the reconfiguration is an expansion, i.e., the application will be executed on more processes after the reconfiguration, the next strategy is followed:

- Each initial w_i *worker* sends both its *cData* and *wData* to the new w'_i *worker*.
- The initial *manager* sends its *cData* to half of the remaining new processes (which will become new *workers*).
- The initial *writer* sends its *cData* to the rest of the remaining new processes (two of which will become the new *manager* and the new *writer*).

Figure 5 illustrates this redistribution pattern when expanding from 4 to 6 processes. The initial *workers* send its *cData* and *wData* to their peer MPI ranks in the new communicator (wide arrows in the figure). At the same time, the initial *manager* and *writer* send their *cData* to the remaining newly spawned processes (narrow arrows in the figure).

Each initial *worker* preserves its *wData* by sending it to another *worker* in the new communicator. Since, after a reconfiguration, the number of *workers* changes, not all the new *workers* will receive *wData*. We proceed in this manner as we have experimentally determined that there was no appreciable differences in HPG-aligner performance or accuracy if some new *workers* did not have previous *wData*. In fact, distributing the same *wData* to more than one new process incurred a performance penalty because of the communication overhead.

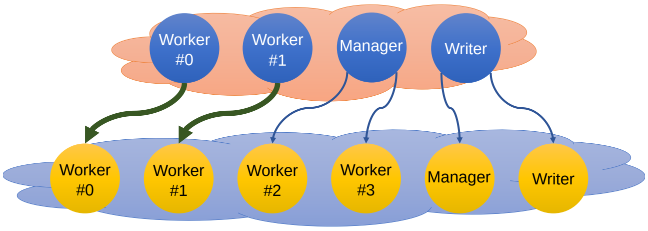

Figure 5. Data redistribution when expanding from 4 to 6 processes.

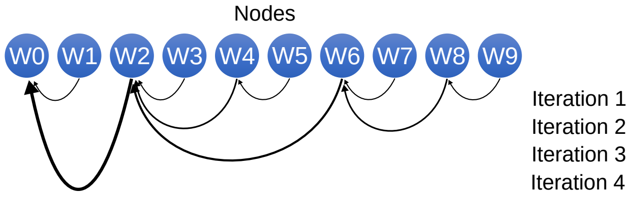

Figure 6. HPG-aligner parallel merge of the worker data structures.

When the action is a shrinking, the application will be executed on fewer processes after the reconfiguration. In this case, the already populated *wData* of those *workers* that will be removed should be preserved (as this information is paramount to a later stage of HPG-aligner). HPG-aligner implements an efficient parallel merge Medina et al. (2016), which uses a minimum spanning tree pattern to merge all the *wData* into the first *worker* (see Figure 6). Therefore, over the different options that could be employed to preserve and distribute this information, we have chosen to first call HPG-aligner parallel merge, and then follow the next strategy:

- The first *worker*, w_0, will send its *cData* and *wData* to the new first *worker*, w'_0. (The aggregated *wData* will be preserved on this *worker*.)
- The remaining *workers* will send their *cData* to the remaining new processes (the last two of them becoming the new *manager* and the new *writer*).

Figure 7 illustrates the redistribution pattern when shrinking from 6 to 4 processes. The first *worker* sends its *cData* and *wData* to the MPI rank 0 in the new communicator (wide arrow in the figure), and the remaining processes send their *cData* to their peer MPI ranks (narrow arrows in the figure).

HPG-aligner malleable version outline using the DMR API

Once the HPG-aligner modified workflow has been presented and the data redistribution patterns have been defined, we outline the HPG-aligner malleable main loop, which proceeds as follows (see Listing 2). When a process finishes its part of processing the first n chunks (line 5), it will reach the malleability point and call the DMR API

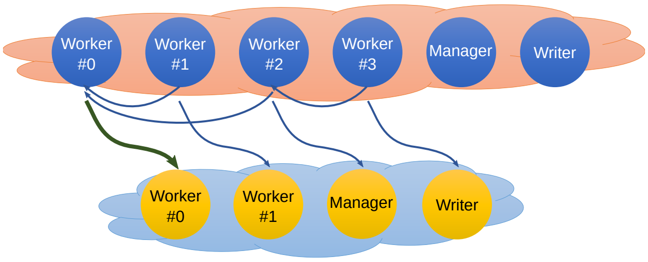

Figure 7. Data redistribution when shrinking from 6 to 4 processes.

`dmr_check_status()` function. If no resize action is planned, the current processes will proceed with the next n chunks. Else, if an expanding or shrinking action is scheduled, the corresponding data redistribution pattern, described in the previous section, is applied. Since the actual data redistribution is performed by the DMR API runtime, all that is required is to indicate in the source code which data has to be be distributed and to whom using the DMR API OmpSs-like `#pragma` directive. After the data redistribution, the new processes will proceed with the next n chunks.

Notice that the `dmr_check_status()` function allows defining some malleability conditions that the RMS will consider. These are: i) the minimum number of processes (`MIN`), ii) the maximum number of processes (`MAX`), and iii) the preferred number of processes (`PREF`) that should be assigned to this application. If a resize is due, the function will return: the action, the new number of processes (in `handlerNProcs`) and the new MPI communicator (in `handler`).

The `#pragma omp task` directives, as previously stated, are used to indicate which data has to be redistributed and to whom. For example, the directive:

```
#pragma omp task in(cData) onto(handler, myRank)
```

indicates that the *cData* data structure has to be conveyed from the current process to the new `myRank` MPI process in the `handler` communicator.

Validation of the proposed HPG-aligner malleable version

To compare the original and the malleable versions of HPG-aligner outputs, we have executed both versions 10 times with the next parameters and collected their execution times and results.

The input to both versions consisted of a dataset with 20 million RNA reads of 100 nucleotides (generated with BEERS Grant et al. (2011)), and the reference genome GRCh37.p73 human genome[‡‡].

For the malleable version, we configured the number of chunks to be equal to the number of initial *workers* in order to mimic a static dataset distribution and to prevent any reconfiguration.

The execution times were virtually the same for both versions, with close-to-negligible time differences of 0.28%, which were due to the malleability code overhead.

Since the HPG-aligner output depends on the order in which the reads are processed Martinez et al. (2015), it is not possible to obtain the same results on different parallel executions. However, we can consider that the obtained results are comparable if they present a similar accuracy. Moreover, the accuracy deviation of the results obtained by the malleable version with respect to the ones of the original version was a negligible ±0.2%, we can conclude that both outputs are comparable.

Experimental results

In this section we describe the setup for the experiments, the results obtained with a current production-size dataset, and, the results achieved when a larger dataset is employed.

Experimental setup

The experiments were performed using the *Marenostrum IV* supercomputer at the Barcelona Supercomputing Center (BSC). Each cluster node integrates 2 Intel Xeon Platinum 8160 sockets (24 cores at 2.10 GHz each) for a total of 48 cores with 96 GiB of RAM. The nodes are interconnected through a 100 Gb/s Intel Omni-Path network.

Of the 50 nodes of a standard queue in *Marenostrum IV*, 1 node was used to run the Slurm manager daemon and the other 49 nodes were used to run the jobs.

The following software versions were used: MPICH 3.2, OmpSs 15.06, and Slurm 15.08.

Workloads setup We have generated fixed and malleable workloads of 100, 250, 500, 1,000, and 2,000 jobs, where all the jobs in a fixed workload are not malleable; and all the jobs in a malleable workload are malleable.

The workloads were generated using the statistical model proposed in Feitelson (1996). This model characterizes fixed jobs based on observations from 6 logs of actual cluster workloads. It includes the distribution of job sizes in terms of number of processors, the correlation of runtime with parallelism, and repeated runs of the same job. For our experiments, we have customized the following parameters of this model: i) the number of jobs to be launched; and ii) the jobs inter-arrival time, modeled using a Poisson distribution with factor 10, in order to prevent receiving bursts of jobs while preserving a realistic job arrival pattern.

Jobs setup Each job in a workload will execute an instance of HPG-aligner with a simulated dataset of RNA reads, and the human genome, used in the previous section, as the reference genome.

As for the malleable jobs, the number of chunks have been set to be 4 times the maximum number of *workers*. Also, the number of chunks to be processed in each iteration has been selected to be variable and equal to the number of *workers* available at that moment.

Experiments with a production-size dataset

The input dataset used in these experiments is a production-size one consisting of 40 million RNA reads of 100 nucleotides, 8 GiB in total. It was generated using BEERS Grant et al. (2011).

Since the malleable version of HPG-aligner has to inform the RMS of its minimum, maximum, and preferred number of processes, we have first experimentally determined these

[‡‡] http://www.ensembl.org/index.html

```
void hpga_malleable(void *cData, void *wData, int chunkIndex) {
 for (ind = chunkIndex; ind < TOTAL_CHUNKS; ind += N) {

  /* Computation */
  do_my_part();

  /* Malleability point */
  action = dmr_check_status(MIN, MAX, PREF, &handlerNProcs, &handler);
  if (action == EXPAND) {
    if (am_i_a_worker()) {
     #pragma omp task in(cData) in(wData) onto(handler, myRank)
     hpga_malleable(cData, wData, ind);
    } else { //Manager or Writer process
      for (dst = firstDst(); dst < getDsts(); dst++) {
       #pragma omp task in(cData) onto(handler, dst)
       hpga_malleable(cData, NULL, ind);
      }
    }
  } else if (action == SHRINK) {
   merge_in_rank0(wData);
   if (myRank == 0) {
    #pragma omp task in(cData) in(wData) onto(handler, myRank)
    hpga_malleable(cData, wData, ind);
   } else if (myRank < handlerNProcs) {
    #pragma omp task in(cData) onto(handler, myRank)
    hpga_malleable(cData, NULL, ind);
   }
  }
 }
}
```

Listing 2: HPG-aligner malleable version outline using the DMR API.

for the already described experimental setup. For this purpose, we have launched the HPG-aligner malleable version with different numbers of processes and obtained their execution times, which are shown on Table 1. In order to evaluate the fitness of each configuration, we use a heuristic metric, referred as *gain slope*, which, unless there is a relevant gain between one configuration and the next one, decreases rapidly as the number of required nodes is increased. The *gain slope*, which is reported in the fourth column of Table 1, is computed as:

$$\frac{t_{i-1} - t_i}{t_0},$$

where t_i is the execution time of row i of the table. Using this metric, the minimum, preferred and maximum number of processes are determined as follows:

- The minimum number of processes is selected as the first number of processes for which the *gain slope* is greater than 75%. In this configuration, since there is no such case, the strictly minimum number of processes has been chosen: 3 processes (a *manager*, a *worker*, and a *writer*).
- The preferred number of processes, i.e., the number of processes for which a fair trade-off between performance and the resources allocated is attained, is selected as the last number of processes for which the *gain slope* is greater than 25%. In this case: 6 processes.
- Finally, the maximum number of processes is selected as the last number of processes where the *gain slope* is positive. In this case: 12 processes.

Table 1. Execution time and gain difference of the HPG-aligner malleable version when executed with an input dataset of 40 million reads of 100 nucleotides for different numbers of MPI multithreaded processes.

# Processes	# Cores	Execution Time	Gain
3	144	238 s	—
6	288	68 s	71.43%
12	576	40 s	11.76%
24	1152	44 s	-1.68%
48	2304	60 s	-6.72%

Regarding the job submission, jobs of the fixed workload have been submitted according to the next two premises: users want to have their results as fast as possible, and they can not know in advance how the cluster workload will evolve. Therefore, the fixed jobs have been submitted with 12 MPI multithreaded processes each, because with this configuration they should finish faster than if submitted with any other number of processes (as shown in Table 1). It should be noted that although on this experimental setup

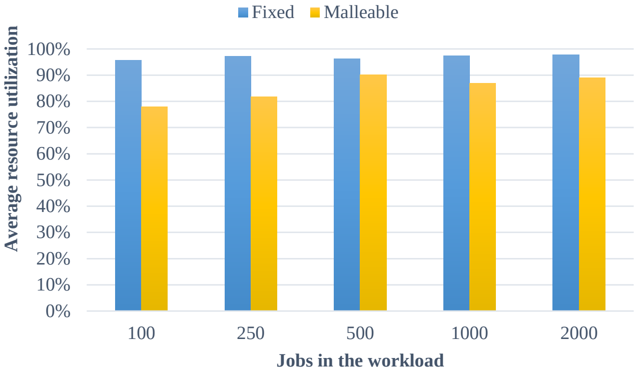

Figure 8. Average resource utilization on different fixed and malleable workloads with an increasing number of jobs.

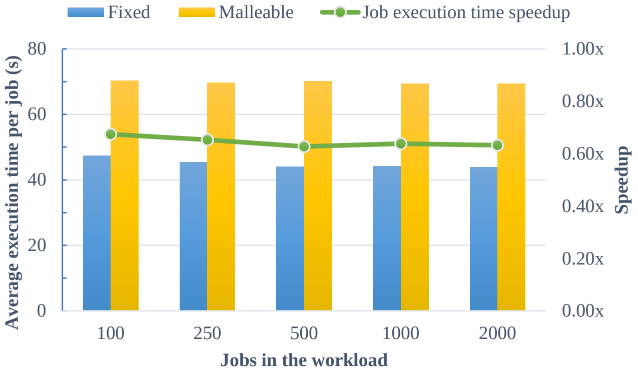

Figure 9. Average job execution time on different fixed and malleable workloads with an increasing number of jobs.

using 6 processes will lead to better completion times (similar to the achieved by their malleable counterparts), in a production environment, where the other running jobs will exhibit completely different characteristics, this could be not the case.

Jobs of the malleable workload have been submitted using a Slurm range submission of 3–12 MPI multithreaded processes, and a preferred number of processes of 6. This way, the resource manager will decide the number of nodes that will be initially assigned to the job, favoring the upper limit whenever possible. In practice, when a malleable job is being processed, although it could be started with 12 processes, it will usually be shrunk to 6 processes (the preferred number of processes) as long as there are other jobs waiting in the queue.

These differences on how the resources are (dynamically) allocated pose an impact on the cluster average resource utilization, as can be seen in Figure 8, where the malleable workloads present a lower average resource utilization than that of their fixed counterparts. As stated previously, the jobs of a workload are submitted at different times to the queue, simulating a real scenario. Besides, nothing prevents the malleable workload to make use of all the resources available at a given moment. Therefore, the lower average resource utilization on the malleable workloads indicates that these workloads achieve a better job completion throughput than their fixed counterparts.

If the job execution time is to be compared, since the jobs on the malleable workloads can be executed with 6 or even 3 processes, the average execution time of these should be lower than that of their fixed counterparts. This can be seen in Figure 9, where the slowdown in the average execution time of the malleable jobs reaches 60%.

But, when a job is submitted to a cluster, not only its execution time is taken into account; the time elapsed from the submission to the execution (waiting time) has to be also considered. Jobs are expected to be initiated earlier on a malleable workload because currently running jobs may release part of their resources in favor of the pending jobs. As expected, Figure 10 shows that the average waiting time in a malleable workload can be up to 77% lower than that in a fixed workload.

Considering everything, the balance is positive for the malleability workloads, since the average job completion time (i.e. the average job waiting plus execution time) can be up to 20 times faster than that for its equivalent fixed workload, as can be seen in Figure 11.

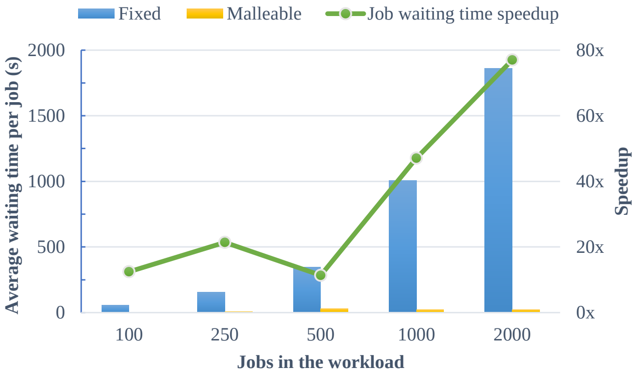

Figure 10. Average job waiting time on different fixed and malleable workloads with an increasing number of jobs.

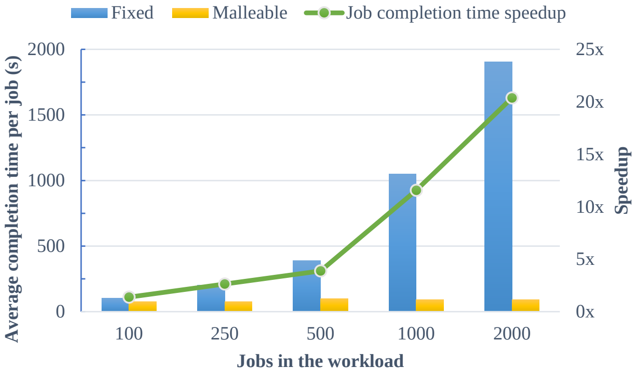

Figure 11. Average job completion time (waiting plus execution time) on different fixed and malleable workloads with an increasing number of jobs.

Figure 12 shows the completion time (the elapsed time between the first job was submitted and the last job is ended) of the different fixed and malleable workloads. Although execution time is increased, the workload completion time is mainly affected by the jobs waiting time (as it is shown in Figure 11. The workload completion time of the malleable version is up to 16% faster than their fixed counterparts. This result is especially significant from a cluster administrator/manager point of view, as it shows how the throughput of the system is increased when malleable workloads are used.

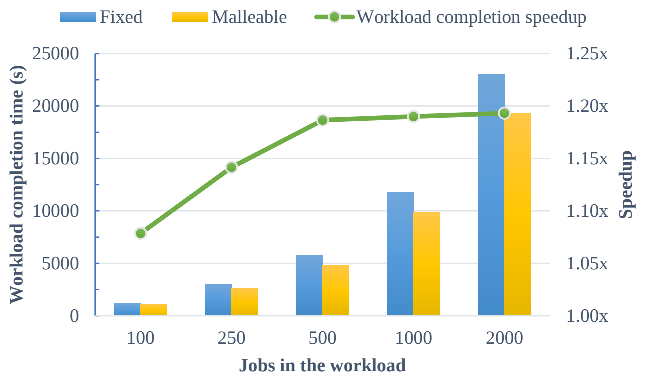

Figure 12. Completion time of different fixed and malleable workloads with an increasing number of jobs.

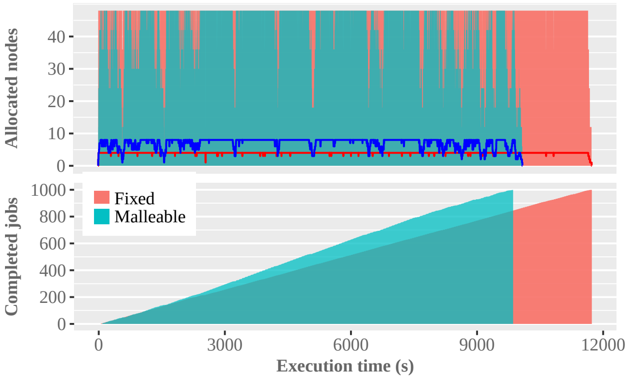

Figure 13. Time evolution of the allocated nodes (top chart), the concurrent jobs being executed (blue and red lines on top chart), and the completed jobs (bottom chart) on the 1,000 jobs fixed and malleable workloads.

Figure 13 details the time evolution of: i) the allocated nodes (top chart), ii) the concurrent jobs being executed (red and blue lines on top chart), and iii) the completed jobs (bottom chart) of the 1,000-job fixed and malleable workloads.

The flat shape in the top chart of Figure 13 depicts how the nodes have been allocated during the execution of the fixed workload. It reveals that almost all the resources are allocated all the time. Conversely, the malleable workload presents a saw-tooth pattern that finishes earlier: more resources are generally available, which allows a greater flexibility on the scheduling of new jobs.

The red and blue lines on the top chart of Figure 13 represent the concurrent running jobs on the fixed and malleable workloads, respectively. On the one hand, it can be seen, as was expected, that for the current experimental setup, the fixed workload is almost always running 4 12-node jobs at the same time. On the other hand, the malleable workload presents a greater variability, where the most seen configuration is 8 3-node jobs.

Finally, the bottom chart of Figure 13 represents the time evolution of the completed jobs on the fixed and malleable workloads. The evolution of both are overlapped until second 1,000, where more jobs per second begin to be completed on the malleable version. This behavior is consistent with the observed differences on the average job completion time of both workloads types with different number of jobs (see Figure 11 again): the differences were minimum for the 100 job workloads and became greater and greater as the number of jobs in the workloads were increased. Furthermore, due to its higher throughput, the malleable workload ends before the fixed workload.

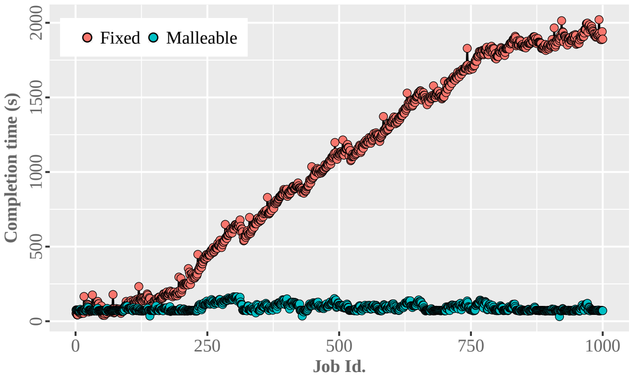

Figure 14. Completion time of each job of the 1,000 jobs on the fixed and malleable workloads. The greater the job id, the later the job was queued.

Table 2. Average job reconfiguration time of the 1,000-job workload.

Stage	Average Time
Scheduling	0.015 s
Resize	4.372 s
Total	4.387 s

The results presented so far can be summarized in a remarkable improvement of the individual job completion time when a malleable workload is processed. Figure 14 illustrates this fact, showing how all the malleable jobs are completed in almost a constant time, while the completion time of the fixed jobs becomes more variable: the later they are queued, the larger completion time they experience.

As for the reconfiguration time cost, it is due to the execution of the next two stages: *scheduling* and *resizing*. The *scheduling* time depends on the number of jobs in the queue, and should be negligible when compared to the *resizing* stage or the tasks being executed. The *resizing* time is the sum of the times of the next operations: spawn of processes, data redistribution, and termination of processes. Of these operations, the most costly one is data redistribution, which is determined by the amount of data to be transferred and the bandwidth of the underlying network. While data redistribution is usually in the order of seconds, the other two operations usually are in the order of milliseconds. Table 2 displays the average reconfiguration time in the 1,000-job workload. For this purpose, we have calculated the average time of each stage during the execution of the 1,000-job malleable workload. The results confirm that the reconfiguration time is mainly dominated by the *resize* stage, being the *scheduling* stage almost negligible.

Experiments with a larger dataset

The input dataset used in these experiments is larger than the currently usual production sizes. We have considered this larger size as it could become the usual production

Table 3. Execution time and gain difference of HPG-aligner malleable version when executed with an input dataset of 80 million reads of 400 nucleotides for different numbers of MPI multithreaded processes.

# Processes	# Cores	Execution Time	Gain
3	144	1382 s	—
6	288	345 s	75.04%
12	576	155 s	13.75%
24	1152	152 s	0.22%
48	2304	171 s	-1.37%

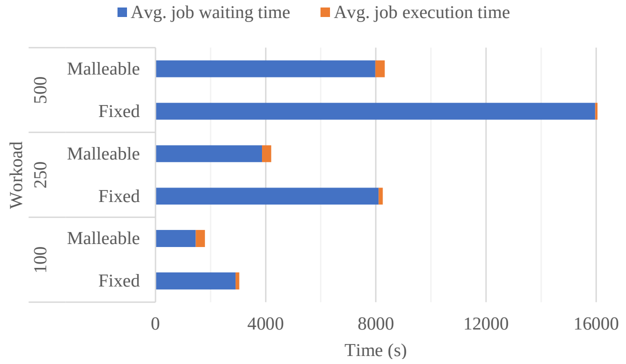

Figure 15. Average job completion time (waiting plus execution time) on different fixed and malleable workloads with an increasing number of jobs.

size in a few years. It consists of 80 million RNA reads of 400 nucleotides, 61 GiB in total (the production-size dataset described in the previous section was 8 GiB). This dataset was generated using BEERS Grant et al. (2011).

Again, since the malleable version of HPG-aligner has to inform the RMS of its minimum, maximum, and preferred number of jobs, we have first experimentally determined these for this particular experimental setup. In order to do this, we have launched the HPG-aligner malleable version with different number of processes and obtained their execution times, which are shown on Table 3, alongside with their *gain slope*. Using the *gain slope* metric, the minimum, preferred, and maximum number of processes are determined as detailed on the previous section, as:

- Minimum number of processes: 6.
- Preferred number of processes: 6.
- Maximum number of processes: 24.

Our experiments reveal again that the malleable workloads outperform their fixed counterparts. Figure 15 shows the average job completion time as the sum of the average waiting and execution times on different fixed and malleable workloads. The waiting time (blue part of the bars) is the predominant addend in all the cases, while the execution time (orange part of the bars) is far smaller in the malleable cases and almost imperceptible in the fixed workloads. In this figure it can also be seen that the average job completion time for the malleable workloads is around half of that of their fixed counterparts. In the interest of clarity, we have only shown the workloads up to 500 jobs, which let us appreciate the difference between both averaged times.

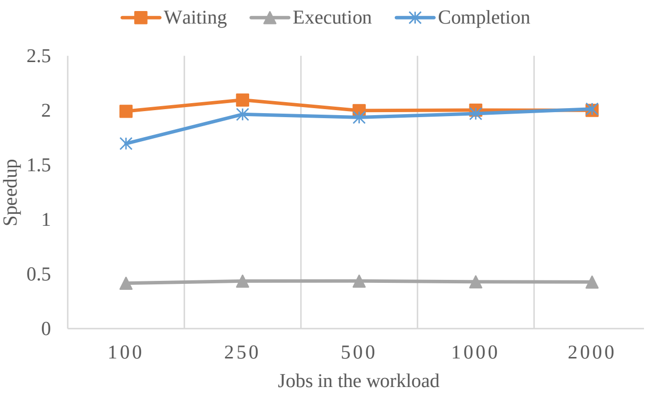

Figure 16. Speedups of the job completion time, of the job execution time, and of the job waiting time on malleable workloads with an increasing number of jobs over their fixed counterparts.

The speedups of the job completion time, of the job execution time, and of the job waiting time on malleable workloads with an increasing number of jobs over their fixed counterparts are shown in Figure 16. The speedups of the job execution time are under 1, which indicates that jobs in the malleable workloads are running slower than jobs in the fixed cases. As already discussed, this is due to the malleable jobs being usually shrunk in order to accommodate more running jobs. Nevertheless, the job waiting time speedup widely compensates the slower execution time, leading to a job completion time speedup around 2 in all the workloads.

Conclusions

Adapting a regular application to accommodate malleability can be a hard task, since the applications are usually developed without ever considering the possibility of being asked to reconfigurate its processes. Nevertheless, in this work we have shown that the DMR API can be used to ease this task, and we expect to have started to pave the road towards the dynamic job reconfiguration and the standardization of adaptive workloads through the inclusion of malleable jobs on them.

In particular, we have presented a malleable version of HPG-aligner using the DMR API. This is the first case, to our knowledge, where a non-iterative producer-consumer application with irregular communication patterns of complex data structures has been turned into malleable. This work, together with the different malleable applications presented in Iserte et al. (2017), proves that the DMR API can handle a wide variety of applications, including those with an irregular design.

As for the experimental results, we have shown that by adding malleability to HPG-aligner, the throughput of malleable workloads can be doubled. At the same time, as the waiting time is greatly reduced, the jobs in a malleable workload are completed faster and in an almost constant time. These results confirm how useful the adoption of this strategy may be in a production system.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable, insightful comments that improve the quality of this paper.

This work was supported by the Project TIN2014-53495-R and TIN2015-65316-P from MINECO and FEDER. Antonio J. Peña was cofinanced by MINECO under Juan de la Cierva fellowship IJCI-2015-23266.

References

Comprés I, Mo-Hellenbrand A, Gerndt M and Bungartz HJ (2016) Infrastructure and API Extensions for Elastic Execution of MPI Applications. In: *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*. ACM Press. ISBN 9781450342346, pp. 82–97. DOI:10.1145/2966884.2966917.

Feitelson DG (1996) Packing schemes for gang scheduling. In: *Lecture Notes in Computer Science book series (LNCS, volume 1162)*. Springer, Berlin, Heidelberg, pp. 89–110.

Grant GR, Farkas MH, Pizarro AD, Lahens NF, Schug J, Brunk BP, Stoeckert CJ, Hogenesch JB and Pierce EA (2011) Comparative Analysis of RNA-Seq Alignment Algorithms and the RNA-Seq Unified Mapper (RUM). *Bioinformatics* 27(18): 2518–2528.

Iserte S, Mayo R, Quintana-Ortí ES, Beltran V and Peña AJ (2017) Efficient Scalable Computing through Flexible Applications and Adaptive Workloads. In: *10th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*. Bristol.

Iserte S, Peña AJ, Mayo R, Quintana-Ortí ES and Beltran V (2016) Dynamic Management of Resource Allocation for OmpSs Jobs. In: *1st PhD Symposium on Sustainable Ultrascale Computing Systems (NESUS PhD)*. Timisoara, Romania. ISBN 978-84-608-6309-0, pp. 55–58.

Martín G, Marinescu MC, Singh DE and Carretero J (2013) FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In: *Euro-Par Parallel Processing*. ISBN 978-3-642-40046-9, pp. 138–149.

Martín G, Singh DE, Marinescu MC and Carretero J (2015) Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. *Parallel Computing* 46: 60–77.

Martinez H, Barrachina S, Castillo M, Tarraga J, Medina I, Dopazo J and Quintana-Orti ES (2015) Scalable RNA Sequencing on Clusters of Multicore Processors. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE. ISBN 978-1-4673-7952-6, pp. 190–195. DOI:10.1109/Trustcom.2015.631.

Martínez H, Tárraga J, Medina I, Barrachina S, Castillo M, Dopazo J and Quintana-Ortí ES (2013) A Dynamic Pipeline for RNA Sequencing on Multicore Processors. In: *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*. New York, New York, USA: ACM Press. ISBN 9781450319034, p. 235. DOI:10.1145/2488551.2488581.

Medina I, Tárraga J, Martínez H, Barrachina S, Castillo MI, Paschall J, Salavert-Torres J, Blanquer-Espert I, Hernández-García V, Quintana-Ortí ES and Dopazo J (2016) Highly Sensitive and Ultrafast Read Mapping for RNA-seq Analysis. *DNA Research* 23(2): 93–100.

Prabhakaran S, Neumann M, Rinke S, Wolf F, Gupta A and Kale LV (2015) A batch system with efficient adaptive scheduling for malleable and evolving applications. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. ISBN 978-1-4799-8649-1, pp. 429–438. DOI:10.1109/IPDPS.2015.34.

Sudarsan R and Ribbens C (2009) Scheduling Resizable Parallel Applications. In: *International Symposium on Parallel & Distributed Processing*. IEEE.

Sudarsan R and Ribbens CJ (2007) ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In: *International Conference on Parallel Processing*.

Sudarsan R and Ribbens CJ (2016) Combining Performance and Priority for Scheduling Resizable Parallel Applications. *Journal of Parallel and Distributed Computing* 87: 55–66.

Sudarsan R, Ribbens CJ and Farkas D (2009) Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS. In: *International Conference on Computational Science (ICCS)*. pp. 175–184.

Wong AT, Oliker L, Kramer WT, Kaltz TL and Bailey DH (2000) Esp: A system utilization benchmark. In: *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, pp. 1–12.

Yoo AB, Jette MA and Grondona M (2003) SLURM: Simple Linux Utility for Resource Management. In: *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. ISBN 978-3-540-39727-4, pp. 44–60.