A Hierarchic Task-Based Programming Model for Distributed Heterogeneous Computing

Jorge Ejarque[1], Marc Domínguez[1] and Rosa M. Badia[1,2]

Abstract

Distributed computing platforms are evolving to heterogeneous ecosystems with Clusters, Grids and Clouds introducing in its computing nodes, processors with different core architectures, accelerators (i.e GPUs, FPGAs), as well as different memories and storage devices in order to achieve better performance with lower energy consumption. As a consequence of this heterogeneity, programming applications for these distributed heterogeneous platforms becomes a complex task. Additionally to the complexity of developing an application for distributed platforms, developers must also deal now with the complexity of the different computing devices inside the node. In this paper, we present a programming model that aims to facilitate the development and execution of applications in current and future distributed heterogeneous parallel architectures. This programming model is based on the hierarchical composition of the COMP Superscalar (COMPSs) and Omp Superscalar (OmpSs) programming models, that allow developers to implement infrastructure-agnostic applications. The underlying runtime enables applications to adapt to the infrastructure without the need of maintaining different versions of the code. Our programming model proposal has been evaluated on real platforms, in terms of heterogeneous resource usage, performance, and adaptation.

Keywords

Distributed Computing, Heterogeneous Computing, Task-based Parallel Programming Models

Introduction

In recent years, the computing ecosystem is becoming more and more heterogeneous. On the one hand, trends in computer architecture focus on providing different computing devices (CPUs, GPUs and FPGAs) and memories in a single chip or computing node, with the aim of providing better computing devices for different types of algorithms and applications. On the other hand, distributed computing platforms such as Clusters, Grids and Clouds, which traditionally have been composed of homogeneous nodes, are starting to be built with heterogeneous nodes, which include processors with different core types, accelerators and memory capacities. This heterogeneity is required to achieve better computing performance with lower energy consumption. However, the development of applications for machines with powerful computing devices requires an extraordinary effort by developers. Learning how to better use these computing resources is not easy, since executing the same algorithm in one or another device can have different outcomes in terms of performance and energy consumption. Therefore, selecting the proper device for each application part is a key factor to achieve an efficient application execution.

Moreover, programming these heterogeneous platforms is not an easy task. For each accelerator, the developer has to add some specific c ode t o e nable t he application execution in the computing device (e.g. manage transfers between device memories, spawn processes on these devices and collect the results). For that reason, in the framework of the TANGO project (Djemame et al. 2019), we propose a programming model to facilitate the development and execution of applications for future distributed heterogeneous architectures. This programming model is based on a hierarchical combination of the COMP Superscalar (COMPSs) (Badia et al. 2015) and the OMP Superscalar (OmpSs) (Duran et al. 2011) task-based programming models, where COMPSs deals with the distributed platform heterogeneity and OmpSs deals with the intra-node heterogeneity following the same programming paradigm. This approach facilitates application development, since developers do not require to learn several programming models and resource provider APIs, and the resultant application is infrastructure-agnostic. During the application execution, the programming model runtime detects the inherent application parallelism by analysing data dependencies between application tasks, and transparently manages their execution in the different computing devices. Apart from the basic execution management, the runtime is also capable of adapting the application execution to the underlying infrastructure by selecting the most appropriate available device for each type of tasks and deciding the number of resources to use according to the number of parallel tasks. All these features are provided without having to maintain different versions of the code. The rest of the paper is organized as follows: Section presents the related work; The programming model

[1]Barcelona Supercomputing Center (BSC), Spain

[2]Artificial Intelligence Research Institute - Spanish National Research Council (CSIC), Spain

Email: {jorge.ejarque, rosa.m.badia}@bsc.es

is presented in Section ; Then, Section evaluates a prototype of this programming model; Finally, Section draws the conclusions and presents the guidelines for future work.

Related Work

As introduced in previous paragraphs, computing nodes are incorporating different types of devices in order to be more efficient w hen c omputing d ifferent t ypes o f applications, either by accelerating the computation or by reducing the energy consumed. However, this brings more complexity to the application development, since each of these devices has its own programming language or API used to spawn the computation to the different devices. For instance, FPGAs are traditionally programmed with the VHDL language; and for deploying and running the computation, developers have to use the tool chain provided by the FPGA vendor. A similar situation is given for General Purpose GPUs: NVIDIA offers the CUDA framework (NVIDIA Corp. 2019) for programming and running applications in their devices, and other vendors offer similar frameworks to do the same.

Current research is focusing on reducing the complexity of programming these heterogeneous nodes, as well as, providing portability between architectures allowing the reuse of code for similar devices. One example of this is OpenCL (Stone et al. 2010). It was developed with the intention of providing a common programming interface for heterogeneous devices (including not only GPUs, but also DSPs and FPGAs). With a syntax very similar to C, the same code can be used in several accelerators. However, similar to CUDA, it requires the programmer to write specific c ode f or d evice h andling, w hich reduces programmability. OpenACC (OpenACC Consortium 2011) is another example of a programming standard for parallel computing designed to simplify parallel programming of heterogeneous CPU/GPU systems. Based on directives, the programmer can annotate the code to indicate those parts that should be run in the heterogeneous device. The OpenMP standard (OpenMP Architecture Review Board 2018) tackles the programmability issues for heterogeneous devices in a similar way to OpenACC, however, it also considers many other aspects of parallelism which makes it a stronger option. Finally, OmpSs (Duran et al. 2011) is a task-based programming model proposed by BSC which promotes both programmability and portability of codes. It hides to the programmer the architecture details, that are managed by the runtime instead of by the developers. For instance, the parallelism of the architectures is inherently exploited by analyzing the data dependencies between tasks, and the allocation of memory in the device or data transfers are automatically managed by the runtime.

PGI (Group 2010) and HMPP (Dolbeau et al. 2007) programming models are two other approaches quite related to OmpSs. PGI uses compiler technology to offload the execution of loops to the accelerators. HMPP also annotates as tasks functions to be offloaded t o t he a ccelerators. We think that OmpSs has higher potential in that it shifts part of the intelligence that HMPP and PGI delegate in the compiler to the OmpSs runtime system. Although these alternatives do support a fair amount of asynchronous computations expressed as futures or continuations, the level of look-ahead they support is limited in practice.

In any case, these solutions are just able to manage the heterogeneity inside a node. If the application requires to run in several nodes (e.g. big amount of data or large parallelism), the solutions mentioned before must be combined with other distributed computing frameworks that manage the processes' spawning and data movements between the different computing nodes. Developers can program this by hand using the TCP/IP and threading libraries, but this option requires a lot of programming effort and parallel programming skills. One of these distributed computing frameworks is MPI (MPI Forum 2015). It provides an API for interchanging data messages between the different processes for Single Program Multiple Data (SPMD) applications. Another option are PGAS programming models such as GASPI (Alrutz et al. 2013) or UPC (El-Ghazawi and Smith 2006), which allow to create a global address space and use shared memory programs in different nodes. Both options are working quite well running SPMD applications in homogeneous clusters interconnected with a very fast network. However, in heterogeneous environments distributed across different locations they are not reaching good performance due to the long latency of the Wide Area Network.

Other approaches to distributed computing can be found in data analytic frameworks such as MapReduce (Dean and Ghemawat 2008), Spark (Zaharia et al. 2016), Graph-processing frameworks such as Pregel (Malewicz et al. 2010) or Deep learning frameworks such as TensorFlow (Abadi et al. 2016).

Workflow management systems are also an alternative for executing applications in distributed computing platforms. These tools provide an interface to describe and execute a workflow (Graphical or by means of a Workflow Description Language). This workflow description is mainly used to combine the execution of different binaries in a static way. Examples of these tools are Galaxy (Goecks et al. 2010), Taverna (Oinn et al. 2004), Kepler (Altintas et al. 2004) or Fireworks (Jain et al. 2015). Finally, we can also find task-based frameworks where users can implement workflows in a dynamic way such as Dask (Rocklin 2015), which provides a Python API to describe task-based workflow as programs and Parsl (Babuji et al. 2018) which from a Python interface enables the execution of workflows in distributed environments.

Finally, COMPSs (Badia et al. 2015) is the sibling of OmpSs for distributing computing. It applies the same concepts as OmpSs but instead of distributing tasks to the different devices of a node, it distributes tasks across computing nodes taking node heterogeneity into account.

In this paper, we propose to combine the COMPSs and OmpSs features to define a hierarchical programming model that enables the development of infrastructure-agnostic parallel applications. This proposal supports a single application code, which at execution is adapted by the runtime to the available underlying infrastructure. One relevant difference of this approach compared to others is that programming is simplified by the use of a single programming paradigm, which does not require the use of APIs to manage the interaction with different computing

devices. In our proposal, the application execution is transparently distributed by the programming model runtime to the different computing nodes of the platform and the heterogeneous computing devices available in each node.

Programming Model

StarSs is a family of task-based programming models where developers define some parts of the application as tasks, indicating the direction of the data required by those tasks. Based on these annotations the programming model runtime analyzes data dependencies between the defined tasks, detecting the inherent parallelism and scheduling the tasks on the available computing resources, managing the required data transfers and performing the tasks' execution. Two frameworks currently compose the StarSs programming model family: COMP Superscalar (COMPSs), which provides the programming model and runtime implementation for distributed platforms such as Clusters, Grids and Clouds, and Omp Superscalar (OmpSs), which provides the programming model and runtime implementation for shared memory environments such as multicore architectures and accelerators (such as GPUs and FPGAs).

The work presented in this paper proposes to combine COMPSs and OmpSs in a hierarchical way, where an application is represented as a workflow of coarse-grain tasks developed with COMPSs. Each of these coarse-grain tasks implements as well a workflow of OmpSs finer-grain tasks. At runtime, coarse-grain tasks will be managed by COMPSs runtime optimizing the execution in a platform level by distributing tasks in the different compute nodes according to the task requirements and the cluster heterogeneity. On the other hand, fine-grain tasks will be managed by OmpSs which will optimize the execution of tasks at node level by scheduling them in the different devices available in the assigned node.

To program an application with the proposed programming model, developers have to identify the parts of the code which are candidates to be coarse-grain tasks. These are usually functions which are repeated several times in the code and with enough computation to compensate the overhead of spawning a remote process (around 10ms). To indicate that a method is a coarse-grain task, developers just need to annotate the code with a preprocessor directive and also indicate the directionality of the method parameters. The main code of the application can be implemented as a normal sequential C/C++ code. The same procedure is done for fine grain tasks. In this case, tasks can have finer granularity due to the shared memory environment. In addition, annotations for tasks that are accelerator kernels should also include a *target* clause in the directive to differentiate them from regular CPU tasks.

In the case of the coarse-grain tasks, in addition to the normal task definition, the programming model provides mechanisms to support different tasks' versions and allocation of resources based on tasks' constraints, in order to make application codes adaptable to the underlying infrastructure. By adding an *implements* clause in the directive, we are indicating that the task implements the same functionality as another task. So, the runtime can execute any implementation of the tasks according to the available resources. For instance, a part of an application can implement a feature programmed to run in a CPU. Moreover, a CUDA kernel which implements the same feature could be also available to run in a GPU. In this case, developers can define the CUDA kernel as a task which implements the CPU task behavior.

Also, in the case of coarse grain tasks, in addition to different implementations, developers can also add a *constraints* directive to the tasks to specify the minimal resource requirements required to run this task type, a certain software, etc. The runtime takes these constraints into account when scheduling these tasks in resources which must fulfill them. For instance, for a coarse-grain task which is composed of a set of fine-grain tasks targeting accelerators we have to add a task constraint to indicate that it requires a given accelerator (i.e. a GPU, a FPGA, etc). Or in the case of coarse-grain tasks implementing a workflow of fine-grain tasks, the constraints directive can be also used to indicate the number of cores required to run the fine-grain tasks in parallel. An example of how an application is programmed is depicted in Section .

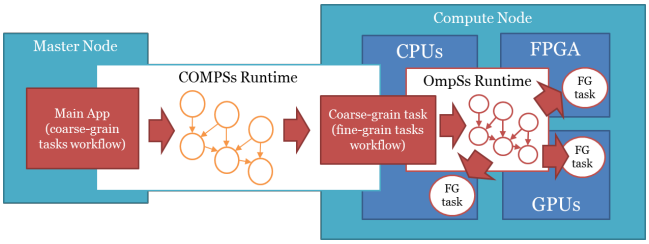

Figure 1. Application execution overview.

Figure 1 shows an overview of how applications are executed in the distributed environment. The main code of the application is linked to the platform-level runtime (COMPSs) which detects the data dependencies, builds a Direct-Acyclic-Graph (DAG) of the coarse-grain tasks. The COMPSs runtime also schedules dependency-free tasks taking into account data location and the coarse-grain constraints, deciding which tasks can run in parallel in each node and ensuring that the different tasks are not colliding in the use of resources. For instance, if there is a single GPU in a computing node, the runtime will not schedule two tasks requiring a GPU. Once the tasks have been assigned to a computing resource, the platform-level runtime will execute these tasks by initiating a node-level runtime (OmpSs) configured to use the assigned devices only. For each coarse-grain task, the node-level runtime builds a DAG of the fine grain tasks and schedules them in the resources assigned by the platform level scheduling.

The proposed programming model presents different advantages with respect to other approaches:

First, it provides a programming model which integrates distributed computing with heterogeneous systems, allowing developers to implement parallel applications in distributed heterogeneous environments without changing the programming model and paradigm. Programmers do not need to learn different programming model and APIs. They are only

required to decide which parts are tasks, the direction of its data, and its granularity level.

Second, developers do not have to deal with data transfers, like in MPI. The programming model runtime analyzes data dependencies at distributed and node levels, and keeps track of the data locations during the execution. So, it tries to schedule tasks close to the required data and when this is not possible, it transparently transfers the data to the available resource.

Third, we have extended the versioning and constraints capabilities of these programming models to make applications adaptable. With these extensions, developers are able to define different versions of tasks for different computing devices (CPU, GPUs, FPGA) or combinations of them. So, the same application will be able to adapt its execution to the different resource capabilities of the heterogeneous platforms without having to modify the application.

Finally, in task-based programming models, the runtime has a deep knowledge of the application parallelism by analysing the generated task-dependency graph. For instance, it can be aware of the parallel workload at each moment of the application execution using as estimation the equivalent load of the number of dependency-free tasks. Moreover, some distributed computing platforms such as Clouds, or Slurm managed clusters allow users to dynamically request or release resources to their allocations. The COMPSs runtime is able to combine this to capabilities in order to adapt the number of resources used by the application to its parallel workload.

Evaluation

We have validated the proposed programming model with a prototype integration of the COMPSs and OmpSs programming model and we have deployed it in the MinoTauro cluster Barcelona Supercomputing Center (2019) at the Barcelona Supercomputing Center. MinoTauro is a heterogeneous cluster where each node contains two Xeon processors with eight cores each and four NVIDIA K80 GPUs. To demonstrate the features presented in the paper, we have implemented two benchmark applications: the matrix multiplication as an example for Linear-algebra applications and the K-means clustering algorithm as an example of a Machine Learning application. With these applications, we have done three experiments. In the first experiment, we have executed the applications in different resource configurations, and we have measured the execution time to see how the programming model runtime is able to detect the available resources and select the appropriate tasks' version for each configuration. In the second experiment, we evaluate the scalability of the prototype in both applications. Finally, with the third experiment, we show how the runtime adapts the used resources to the application load. The rest of the section is organized as follows: First, we will show how the Matrix Multiplication and K-means applications are implemented with the proposed model and after this we will present the results of the experiments.

Application Implementation

Matrix Multiplication The first application used to validate the Programming Model is a two-levels block matrix multiplication. The first level splits matrices into blocks and computes the matrix multiplication at the block level. Each block multiplication is defined as a coarse-grain task with two implementations one for CPUs and another for GPUs. The matrices' blocks are also decomposed in smaller blocks, and the block multiplication task is is implemented as fine-grain tasks operating on sub-blocks. In the case of the GPU version, the fine-grain tasks are implemented by a CUDA kernel.

```
int main( int argc, char      argv ){
  Matrix A, B, C;
  int N = atoi(argv[1]);
  int M = atoi(argv[2]);
  compss_on();

  //Load Matrices
  Matrix A = Matrix::init(N, M);
  Matrix B = Matrix::init(N, M);
  Matrix C = Matrix::init(N, M);

  //Calculate Matrix Multiplication
  for( int i=0; i<N; i++) {
    for( int j=0; j<N; j++) {
      for( int k=0; k<N; k++) {
        (C.block[i][j])->multiplyBlocks(A.block[i][k],
                                        B.block[k][j]);
      }
    }
  }
  compss_off();
}
```

Figure 2. Matrix Multiplication main code. N is the number blocks and M is the number of elements per block

Figure 2 shows the main code of the benchmark application where a loop of the *multiplyBlock* coarse-grain task is implemented. As can be seen, nothing special must be included to call the coarse-grain tasks. They are called as regular functions.

Figure 3 depicts the two implementations of the block multiplication coarse-grain task. In that figure, we can see how developers define coarse-grain tasks with the *#pragma compss task* directive and the fine-grain tasks with the *#pragma omp task* directive. We can also see in this figure how to include the implements clause to indicate that the *Block::multiplyBlocks_GPU* task implements the same behavior as the *Block::multiplyBlocks* task, and the *#pragma compss constraints* directive to define the task constraints for each implementation.

Regarding the fine-grain implementation, we can have different parallelization strategies depending on the computing device. In the CPU case, as they are executed in a shared-memory environment, a fine-grain level of tasks can be defined by annotating one of the inner loops of the block matrix multiplication. In the case of the figure, we have defined the second loop as a task, so each fine-grain task is in charge of computing a row of elements of the resultant block. In the GPU case, the big matrix blocks are decomposed in smaller blocks in order to fit in the GPU device memory and finer-grain tasks are defined as the multiplication of these small blocks. The fine-grain task in this case is the CUDA kernel defined by the *Muld* function which is annotated using

```
#pragma compss constraints(processors={
    processor(Type=CPU, ComputingUnits=6)})
#pragma compss task in(A, B)
void Block::multiplyBlocks(Block A, Block B) {
  for (int i=0; i<M; i++) {
    #pragma omp task in(A.data[i][0;M], \
            B.data[0;M][0;M]) out(data[i][0;M])
    for (int j=0; j<M; j++) {
      for (int k=0; k<M; k++) {
        data[i][j] += A.data[i][k]*B.data[k][j];
      }
    }
  }
  #pragma omp taskwait
}
#pragma compss constraints(processors={
            processor(Type=GPU, ComputingUnits=1)})
#pragma compss task in(A, B)
                implements(Block::multiplyBlocks)
void Block::multiplyBlocks_GPU(Block A, Block B){
  int NB = M/SB_SIZE; //GPU sub-block size
  for (int i=0; i<NB; i++) {
    for (int j=0; j<NB; j++) {
      for (int k=0; k<NB; k++) {
        Muld(A.data[i*NB+k], B.data[k*NB+j],
             data[i*NB+j], NB);
      }
    }
  }
  #pragma omp taskwait
}

#pragma omp target device(cuda)
                    ndrange(2, 64, 64, 32, 32)
#pragma omp task in(A[0:NB*NB], B[0:NB*NB])
                inout(C[0:NB*NB])
__global__ void Muld(double* A, double* B,
                     double* C, int NB);
```

Figure 3. Task definitions

the *#pragma omp task* to indicate it is a task and *#pragma omp target* to indicate it should run in a CUDA device.

K-Means Following the same approach, we have also implemented the K-means algorithm which is a widely used clustering technique to group data in clusters, where each cluster is identified by a centroid point, whose distance with the rest of the points of the cluster is the minimum. K-means is normally implemented with an iterative algorithm starting with a random centroids' set. The distance of all the points to the different centroids is calculated and each point is assigned to the closest centroid. According to these assignments, the new centroids are calculated. This is repeated until the difference of the centroids from one iteration to the other is close to 0.

The parallelization of the K-means algorithm is based on the observation that each of the distance calculations between a point and a given cluster is independent, and they can be computed in parallel. However, these computations are too small for being distributed across nodes. For this reason, we group several points in fragments and the computation of all these fragments is defined a s coarse-grain tasks while each distance computation can be defined as a fine-grain t ask. O nce t his p arallel r egion h as been finished, the distance results are used to recompute the cluster centroids, which has been implemented with two coarse-grain tasks: one that merges the distance calculation results of two fragments in order to reduce all results and another that updates the cluster. We have also defined as a coarse-grain task to load the different fragments in memory, to automatically parallelize the memory load and distribute the points in the different computing nodes.

The main code for the K-means algorithm with our proposed programming model is shown in Figure 4. The code is composed by two main parts. The first consists of a loop which initializes the fragments by reading the points from a file. Then, there is a do-while loop which implements the K-means iterations. Each iteration consists of two loops: one for computing the distances of all the points of a fragment to all the clusters and one to merge the results. Finally, there is a call to update the clusters from the merged distance results.

Figure 5 shows the definition and implementation of the coarse-grain tasks. The *compute_distances* task is parallelized with fine grain tasks. We have created two versions: *compute_distances* and *compute_distances_GPU*, one for executing the task in 4 CPU cores and another which is ready to run in a GPU. The same figure also shows the implementation of both *compute_distances* task versions, where we can see how to indicate that a code block is a fine-grain task, in the case of the *compute_distances* implementation. Also, we show how a CUDA kernel (*cuda_find_nearest_cluster*) is defined as fine-grain tasks in the *compute_distances_GPU* implementation. The other tasks called from the main code are defined as simple sequential coarse-grain tasks.

Experiment 1: Execution in different resource configurations

To validate the adaptability features of the programming model, we have executed the Matrix Multiplication in different resource configurations: using only CPUs, only GPUs or both, and we have measured the time to complete the matrix multiplication with different matrices' sizes and resource configurations. This means that we have executed the application using only CPUs, assigning a different number of cores to each coarse-grain task to allow to run several fine grain tasks in parallel, different number of GPUs and different combinations of CPU and GPU together.

Figure 6 shows the measured execution times when running the matrix multiplication with different matrix sizes. The matrix size is indicated by the number of blocks and the size of the blocks. We have evaluated matrices with a range of 4×4 to 16×16 blocks and block sizes from 256×256 to 2048×2048 elements. For each configuration, we have plotted the best result achieved. We can see that the best configuration depends on the matrix size. For small number of blocks and sizes, the CPU only configuration is the one that is performing better. This is reasonable, since GPUs require an extra data and process management, and the speed up achieved with GPUs is not enough to overcome the overhead of the extra memory allocations and copies required. When increasing the matrix size (either by increasing the number of blocks or the block size, the version with both CPU and GPU is becoming the best option. So, the

```
int main(int argc, char **argv) {

  compss_on();
  for (i=0; i<nFrags; i++){
     init_Fragment(objsFrag, nCoords, filePath, fragments[i]);
  }
  do {
     old_clusters = clusters;
     for (i=0; i<nFrags; i++){
        compute_distances(objsFrag, nCoords, nClusters, fragments[i], clusters, newClusters[i],
                          newClusterSize[i]);
     }
     int neighbor = 1;
     while (neighbor < nFrags) {
        for (int f = 0; f < nFrags; f += 2 * neighbor) {
           if (f + neighbor < nFrags) {
              merge_distance_results(nCoords, nClusters, newClusters[f], newClusters[f+neighbor],
                                     newClusterSize[f], newClusterSize[f+neighbor]);
           }
        }
        neighbor *= 2;
     }
     update_Clusters(nCoords, nClusters, clusters, newClusters[0], newClusterSize[0]);
     compss_wait_on(clusters);
  } while (!hasConverged(old_clusters, clusters));
  compss_off();

}
```

Figure 4. K-means main code

programming model runtime is able to run tasks in all kind of resources which increases the parallelism and reduces the execution time. In the best case, we have achieved a gain of 75% with respect to CPU-only cases and 65% with respect to GPU-only when using both resources. In the worst case (very small matrices), we could have a 6% loss with respect of using a CPU-only configuration.

Similar results have been observed when running the K-means application. Figure 7 shows the execution time when we run the K-means application to find 5 0 c lusters i n 24 million point of 50 dimensions. In the figure, we can see an important acceleration when using the GPUs with respect to the case of using the CPU only, but we can have an extra 10% of improvement when we combine the CPU and GPUs.

Experiment 2: Scalability

We have also evaluated the scalability when running both applications (Matrix Multiplication and K-means) distributed on different computing nodes.

Figure 8 shows the execution time and speed-up of executing the K-means algorithm to find 1024 clusters in 320 fragments of 200K point of 128 dimensions.

Figure 9 shows the execution time and the efficiency when increasing the problem size in the same proportion of the computing resources keeping the computation per node constant. In this case, we can see the execution time has suffer a small increment.

Figure 10 shows the execution time and speed-up of executing the matrix multiplication for matrices of 64×64 blocks of 1024×1024 elements each, using from 1 to 8 computing nodes. We observe a reasonable performance, and the difference from the ideal speed-up is mainly caused by the overhead of the data transfers, the load and write of the blocks to disk, and the OmpSs runtime initialization at the beginning of each coarse-grain task.

Experiment 3: Resource Adaptation

Finally, the third experiment is designed to demonstrate how the programming model runtime is able to detect the inherent parallelism of the application and adapt the number of resources used to the computing load. In this experiment, we have run the same K-means application but configuring the programming model runtime to partially manage the execution in an elastic way. In the runtime execution configuration, we have set the runtime to use two static computing nodes and we have activated the elasticity option to let the runtime decide how many nodes it will use depending on the number of parallel tasks in the application, the task duration and the time to get a new resource from the resource manager.

Figure 11 shows an image generated by the COMPSs runtime monitor, where we can see an estimation of the parallel workload (number of dependency-free tasks multiplied by its expected duration), represented with a blue shadow, and the number of resources used during the application execution, represented with a red line. We can see that the runtime starts with the initial resources (two nodes), and once the first set of tasks finishes, the runtime estimates the duration of executing the rest of the dependency-free tasks based on the duration of the previous executions. If this time is larger than the time to get a new resource in the resource manager, the runtime requests for a resource allocation to execute more tasks in parallel. In the actual execution shown in the figure, we can see that the runtime has decided to request two extra nodes, which are later discarded once the number of tasks decreases.

```
#pragma compss constraints(processors={processor(Type=CPU, ComputingUnits=4)})
#pragma compss task in(frag[0;nObjs*nCoords], clusters[0;nClusters*nCoords])
                    out(newClusters[0;nClusters*nCoords], newClustersSize[nClusters])
void compute_distances(int nObjs, int nCoords, int nClusters, float *frag, float *clusters,
                       float *newClusters, int *newClustersSize){
  int split = get_compss_task_cus();
  int block = (nObjs/split)+1;
  int *index = new int[nObjs];
  int k,i,j;
  for (i=0; k<nObjs; k++) {
     #pragma omp task firstprivate(k) in(frag[0;nCoords*nObjs], clusters[0;numClusters*nCoords])
                                      concurrent(index[0;nObjs])
     index[i] = find_nearest_cluster(nClusters, nCoords, &frag[i*nCoords], clusters);
     }
     #pragma omp taskwait in(index[0;nObjs])
     for (i=0; i<nObjs; i++) {
        newClustersSize[index[i]]++;
        for (j=0; j<nCoords; j++){
           if (newClustersSize[index[i]]==1){
              newClusters[index[i]*nCoords+j]= 0.0;
           }
           newClusters[index[i]*nCoords+j] += frag[i*nCoords + j];
        }
      }
}
#pragma compss constraints(processors={processor(Type=GPU, ComputingUnits=1)})
#pragma compss task in(frag[0;nObjs*nCoords], clusters[0;nClusters*nCoords])
                    out(newClusters[0;nClusters*nCoords], newClustersSize[0;nClusters])
                    implements(compute_distances)
void compute_distances_GPU(int nObjs, int nCoords, int nClusters, float *frag, float *clusters,
                           float *newClusters, int *newClustersSize){
  int *index = new int[nObjs];
  #pragma omp target device(cuda) copy_deps ndrange(1, nObjs, 64)
  #pragma omp task in(frag[0;(nObjs*nCoords)], clusters[0;(nClusters*nCoords)]) out(index[0;(nObjs)])
  cuda_find_nearest_cluster(nCoords, nObjs, nClusters, frag, clusters, index);
  #pragma omp taskwait in(index[0;nObjs])
   for (int i=0; i<nObjs; i++) {
      newClustersSize[index[i]]++;
      for (int j=0; j<nCoords; j++){
         if (newClustersSize[index[i]]==1){
             newClusters[index[i]*nCoords+j]= 0.0;
         }
         newClusters[index[i]*nCoords+j] += frag[i*nCoords + j];
      }
   }
}

#pragma compss task in(filename) out(frag[0;nObjs*nCoords])
void init_Fragment(int nCoords, int nObjs, File filename, float* frag){
    ...
}

#pragma compss task in(frag[0;nObjs*nCoords], clusters2[0;nClusters*nCoords], newClustersSize2[0;nClusters])
                    inout (clusters1[0;nClusters*nCoords], newClustersSize1[0;nClusters])
void merge_distance_results(int nCoords, int nClusters, float* clusters1, float* clusters2,
                            int* newClustersSize1, int* newClustersSize2){
    ...
}

#pragma compss task in(clusters2[0;nClusters*nCoords], newClustersSize2[0;nClusters])
                    inout (clusters1[0;nClusters*nCoords])
void update_Clusters(int nCoords, int nClusters, float* clusters1, float* clusters2, int* newClustersSize2){
    ...
}
```

Figure 5. K-means task definition

Conclusion and Future Work

In this paper, we have presented a proposal for a programming model that aims to facilitate the implementation of parallel applications in a platform of distributed heterogeneous resources. This proposal is based on two-levels of task-based programming models. The first level is in charge of

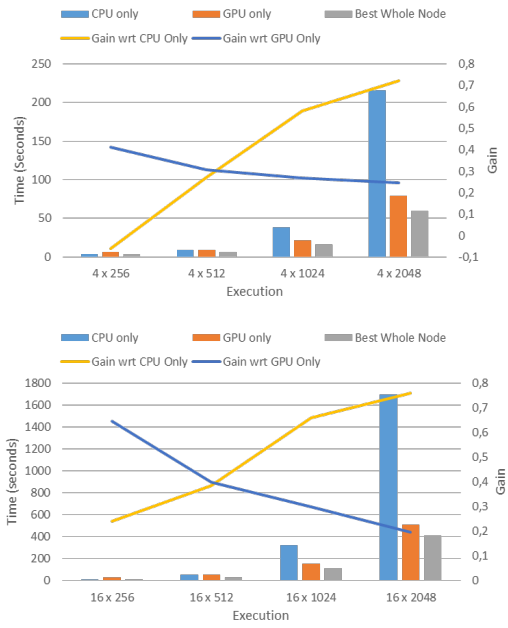

Figure 6. Matrix Multiplication benchmark with different resource configurations

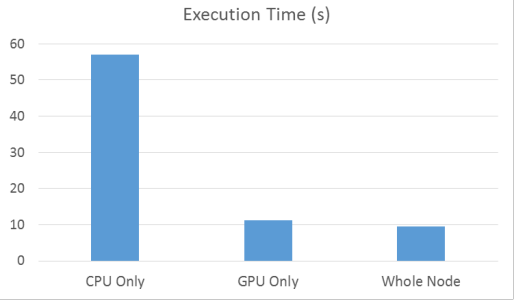

Figure 7. K-means benchmark executed with different resource configurations

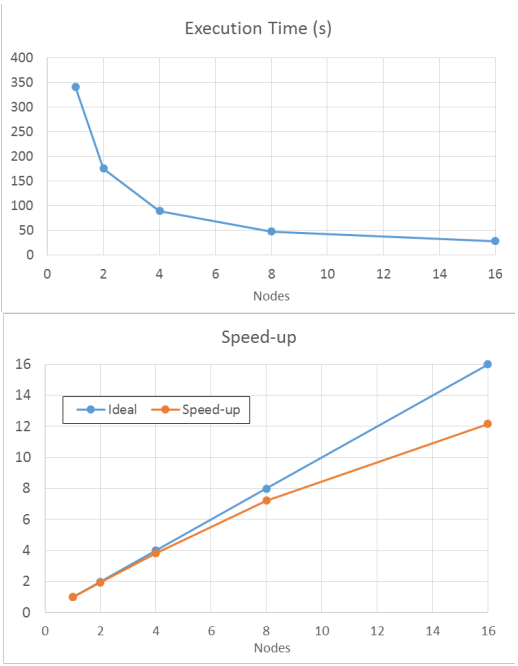

Figure 8. K-means strong scaling.

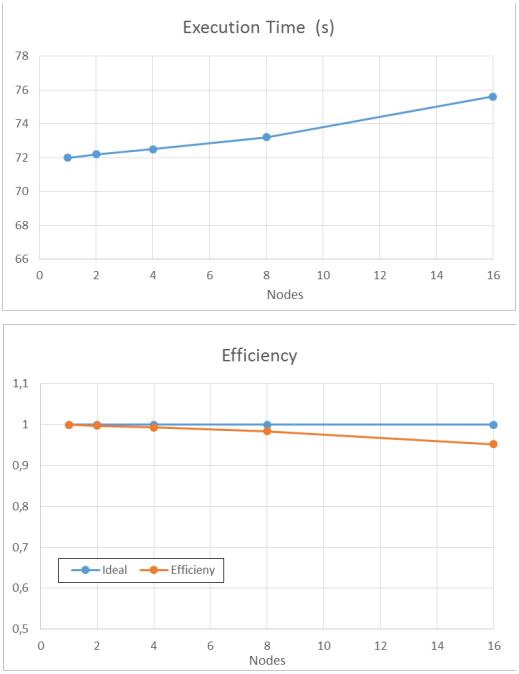

Figure 9. K-means weak scaling (40 Fragments/Node).

defining coarse-grain tasks of the application which will be spawned in the different nodes of the distributed computing environment. This coarse-grain level is managed by the COMPSs programming model and runtime. Each coarse-grain task implements a workflow of fine grain tasks which will be spawned on the different computing devices of a compute node. This fine-grain level is managed by the OmpSs programming model which spawns the processes on the computing node devices according to the constraints provided by the coarse-grain level runtime which coordinates the overall application execution.

The benefit of this programming model combination relies on the easiness of programming, which avoids the need of changing the programming model paradigm or using different APIs, providing means to implement a distributed application which takes profit of the different heterogeneous devices available in the computing nodes. Besides, the programming runtime transparently performs the required actions to efficiently execute the application in the computing infrastructure such as: analyze data dependencies between tasks, keep track of the data locations during the execution as well as schedule tasks close to data or transparently doing the required data transfer, and exploit the maximum parallelism. Moreover, due to the versioning capabilities of the programming model, the runtime can select the implementation which fits better to the available resources selecting the one which provides better performance. So, the same application will be able to adapt to the different capabilities of the heterogeneous platform without having to modify the application.

The results obtained in the evaluation demonstrate that the runtime transparently adapts the application execution to the underlying infrastructure with reasonable performance

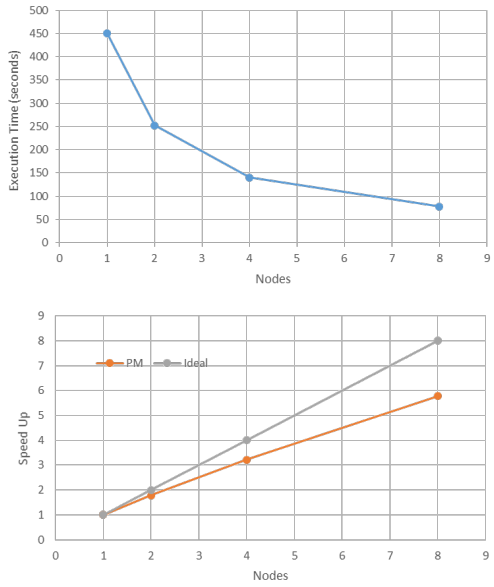

Figure 10. Matrix Multiplication scalability with the Programming Model

results. With the presented programming model developers do not need to maintain different codes as happens in alternative approaches.

However, during the prototype evaluation, we have also detected some sources of overhead. For instance, each time the runtime executes a coarse-grain task, it starts an OmpSs runtime which produces an initialization overhead, specially when a task uses GPUs which requires a costly Host-GPU memory allocation. This overhead will be reduced by introducing the solution proposed in the Intertwine project to achieve interoperability between different programming model runtimes (Intertwine Consortium 2018). This solution allows COMPSs to keep runtimes persistently loaded in the computing nodes during the whole application execution reusing the same runtime for the different coarse-grain task executions.

In addition to these performance improvements, we will extend the work performed in multi-objective scheduling for coarse-grain tasks in Clouds (Juarez et al. 2016) to support distributed heterogeneous environments. With this work, we aim at finding t he b est t ask s cheduling taking into account different factors such as execution time, energy consumption or power limitations. Finally, we will also study how the programming model can interact with the computing infrastructure in order to have more efficient executions, by requesting more resources in highly parallel regions as well as releasing or partially powering off the devices which are not currently used by the application.

Acknowledgment

This work has been supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 687584 (TANGO project) by the Spanish Government under contract TIN2015-65316 and grant SEV-2015-0493 (Severo Ochoa Program) and by Generalitat de Catalunya under contracts 2014-SGR-1051 and 2014-SGR-1272.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al. (2016) Tensorflow: a system for large-scale machine learning. In: *OSDI*, volume 16. pp. 265–283.

Alrutz T, Backhaus J, Brandes T, End V, Gerhold T, Geiger A, Grünewald D, Heuveline V, Jägersküpper J, Knüpfer A et al. (2013) Gaspi–a partitioned global address space programming interface. In: *Facing the Multicore-Challenge III*. Springer, pp. 135–136.

Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B and Mock S (2004) Kepler: an extensible system for design and execution of scientific workflows. In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, pp. 423–424.

Babuji Y, Chard K, Foster I, Katz DS, Wilde M, Woodard A and Wozniak J (2018) Parsl: Scalable parallel scripting in python. In: *10th International Workshop on Science Gateways (IWSG 2018)*.

Badia RM, Conejero J, Diaz C, Ejarque J, Lezzi D, Lordan F, Ramon-Cortes C and Sirvent R (2015) Comp superscalar, an interoperable programming framework. *SoftwareX* 3: 32–36.

Barcelona Supercomputing Center (2019) Minotauro Supercomputer. Available at `https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/minotauro` (accessed 1 April 2019).

Dean J and Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1): 107–113.

Djemame K, Kavanagh R, Kelefouras V, Aguilà A, Ejarque J, Badia RM, García-Pérez D, Pezuela C, Deprez JC, Guedria L et al. (2019) Towards an energy-aware framework for application development and execution in heterogeneous parallel architectures. In: *Hardware Accelerators in Data Centers*. Springer, pp. 129–148.

Dolbeau R, Bihan S and Bodin F (2007) HMPP: A hybrid multi-core parallel programming environment. In: *First Workshop on General Purpose Processing on Graphics Processing Units*.

Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X and Planas J (2011) Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21(02): 173–193.

El-Ghazawi T and Smith L (2006) Upc: unified parallel c. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, p. 27.

Goecks J, Nekrutenko A and Taylor J (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology* 11(8): R86.

Group TP (2010) *PGI Accelerator Programming Model for Fortran & C*.

Intertwine Consortium (2018) Intertwine EU project homepage. Available at `http://www.intertwine-project.eu/`(accessed: 1 April 2019).

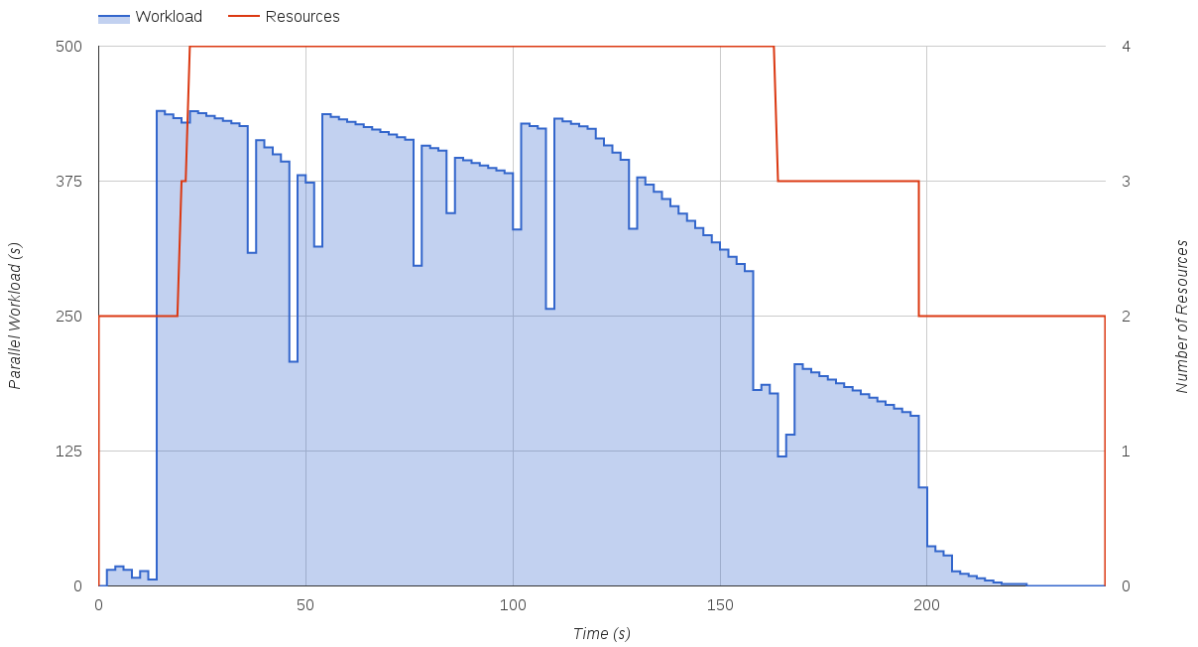

Figure 11. Resource adaptation managed by the Programming Model runtime.

Jain A, Ong SP, Chen W, Medasani B, Qu X, Kocher M, Brafman M, Petretto G, Rignanese GM, Hautier G et al. (2015) Fireworks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27(17): 5037–5059.

Juarez F, Ejarque J and Badia RM (2016) Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems* .

Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N and Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, pp. 135–146.

MPI Forum (2015) Message Passing Interface Specification. Available at `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf` (accessed 1 April 2019).

NVIDIA Corp (2019) CUDA Toolkit. Available at `https://developer.nvidia.com/cuda-toolkit` (accessed 1 April 2019).

Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock MR, Wipat A et al. (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20(17): 3045–3054.

OpenACC Consortium (2011) The openacc application programming interfaceversion 1.0. Available at `http://www.openacc.org/specification` (accessed 1 April 2019).

OpenMP Architecture Review Board (2018) OpenMP Application Programming Interface Specification. Available at `http://www.openmp.org/specifications/` (accessed 1 April 2019).

Rocklin M (2015) Dask: Parallel computation with blocked algorithms and task scheduling. In: *Proceedings of the 14th Python in Science Conference*. Citeseer, pp. 130–136.

Stone JE, Gohara D and Shi G (2010) Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12(3): 66–73.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ et al. (2016) Apache spark: a unified engine for big data processing. *Communications of the ACM* 59(11): 56–65.