Productive Cluster Programming with OmpSs

Javier Bueno[1,2], Luis Martinell[1], Alejandro Duran[1], Montse Farreras[1,2], Xavier Martorell[1,2], Rosa M. Badia[1,3], Eduard Ayguade[1,2], and Jesús Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC-CNS)
[2] Universitat Politècnica de Catalunya (UPC)
[3] Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC)

Abstract. Clusters of SMPs are ubiquitous. They have been traditionally programmed by using MPI. But, the productivity of MPI programmers is low because of the complexity of expressing parallelism and communication, and the difficulty of debugging. To try to ease the burden on the programmer new programming models have tried to give the illusion of a global shared-address space (e.g., UPC, Co-array Fortran). Unfortunately, these models do not support, increasingly common, irregular forms of parallelism that require asynchronous task parallelism. Other models, such as X10 or Chapel, provide this asynchronous parallelism but the programmer is required to rewrite entirely his application.

We present the implementation of OmpSs for clusters, a variant of OpenMP extended to support asynchrony, heterogeneity and data movement for task parallelism. As OpenMP, it is based on decorating an existing serial version with compiler directives that are translated into calls to a runtime system that manages the parallelism extraction and data coherence and movement. Thus, the same program written in OmpSs can run in a regular SMP machine, in clusters of SMPs, or even can be used for debugging with the serial version. The runtime uses the information provided by the programmer to distribute the work across the cluster while optimizes communications using affinity scheduling and caching of data.

We have evaluated our proposal with a set of kernels and the OmpSs versions obtain a performance comparable, or even superior, to the one obtained by the same version of MPI.

1 Introduction

Parallel and distributed programming has always been a difficult endeavour. But the rise in the number of systems and their complexity have put a stress in the way parallel applications are programmed. In recent years, there has been a significant effort to improve programming models to yield more productive models which still are able to provide good performance.

Applications for clusters have traditionally been programmed with MPI. But, while MPI allows to achieve very good performance it comes at the cost of programming at a very low-level which is error-prone and difficult to debug. Even more, as clusters become even larger obtaining high-performance requires using asynchronous communication and overlapping of computation which exacerbates the previous problems.

Shared-memory models, like OpenMP, offer a more productive and easy to debug environment but all efforts to devise implementations that could scale on clusters have failed so far except for some applications. Other models built specifically for clusters, like UPC or Co-array Fortran, try give programmers the illusion of a global address-space where data can be explicitly placed on each node and communications happen implicitly. But these models do not support well emerging task parallelism where work is more dynamic and synchronization and communication follow irregular patterns.

Asynchronous Partitioned Global Address Spaces (APGAS) languages, like X10 or Chapel, were created to address these needs but require programmers to rewrite their applications completely. Furthermore, to implement communication overlapping, data prefetch or locality scheduling, the compiler needs to implement complex and costly analysis of the application.

We designed OmpSs, which combines ideas from OpenMP[11] and StarSs[12], to try to tackle these problems. It enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures. It incorporates the idea of disjoint address spaces that allows the compiler/runtime to automatically move data as necessary and perform different kinds of optimizations.

Our previous work has shown successful implementations of these ideas for multicore [12], the Cell B.E.[13] and GPUs[10]. In this work, we present the implementation of the model and evaluation with a set of applications for clusters of multicores. We show that using OmpSs the same application prepared to run in an SMP can be run in a cluster. The runtime takes care of moving the data around the different nodes as needed and of performing different optimizations to improve the overall performance.

Our results with different applications show that, compared with MPI, the speed-up obtained with OmpSs can be on par or even higher thanks to the asynchronous parallelism that can be expressed with it.

The paper is structured as follows: section 2 describes OmpSs, the programming model used to develop the presented work, section 3 presents the main contribution of this work, the design of Nanos++ for clusters, our implementation of OmpSs, section 4 shows the evaluation of Nanos++, section 5 describes related work to this project and section 6 concludes and discusses the future directions of this work.

2 OmpSs: From Multicores to Clusters

2.1 Overview

Our proposal is to have a single programming model, OmpSs, covering the different homogeneous and heterogeneous architectures in use today and opened to future ones. OmpSs is based on the OpenMP programming model with some modifications to its execution and memory model. This changes and additions come from ideas from the Star SuperScalar (StarSs) programming model.

StarSs is a programming model focused on exploiting asynchronous parallelism expressed using annotations on a sequential code like OpenMP. StarSs also targets different architectures. StarSs is actually the conjuction of a collection of programming models that targeted different architectures: CellSs, SMPSs, ClusterSs and GridSs.

Execution model. The OmpSs execution model is a thread-pool model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all other threads cooperate executing the work it creates (whether it is from worksharing or task constructs). Therefore, there is no need for a `parallel` region. Nesting of constructs allows other threads to become work generators as well.

Memory model. OmpSs assumes a non-homogeneous disjoint memory address space. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and for shared data it must specify how it is going to be used (see below). This assumption is true even for SMP machines as the implementation may reallocate shared data taking into account memory effects (e.g., NUMA).

Function tasks. OmpSs allows to annotate function declarations or definitions, *a la Cilk*[3], with a `task` directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task will be captured from the function arguments.

Dependence synchronization. OmpSs integrates the StarSs dependence support[9]. It allows to annotate both task and worksharings constructs with three additional clauses:

input It specifies that the construct depends on some input data, and therefore, it is not eligible for execution until any previous construct with an `output` clause over the same data is completed.
output It specifies that the construct will generate some output data, and therefore, it is not elegible for execution until any previous construct with an `input` or `output` clause over the same data is completed.
inout It specifies a combination of `input` and `output` over the same data.

The target construct. To support heterogeneity and data motion between address spaces a new construct is introduced: the `target` construct[1]. The `target` construct can be applied to either `task`, worksharing constructs or functions. Its syntax is:

```
#pragma omp target [clauses]
  task construct | worksharing construct | function definition | function header
```

Where the possible clauses are:

device. It allows to specify on which devices should be targeting the construct (e.g., cell, gpu, smp, ...). If no `device` clause is specified then the target devices are decided by the implementation (by default SMP).
copy_in. It specifies that a set of shared data may be needed to be transferred to the device before the associated code is going to be executed.
copy_out. It specifies that a set of shared data may be needed to be transferred from the device after the associated code is executed.
copy_inout. This clause is a combination of `copy_in` and `copy_out`.
copy_deps. It specifies that if the attached construct has any dependence clauses then they will also have copy semantics (i.e., `input` will also be considered `copy_in`, `output` `copy_out` and `inout` `copy_inout`).

implements. It specifies that the code is an alternate implementation for the target devices of the function name specified in the clause. This alternate can be used instead of the original if the implementation considers it appropriately.

The different **`copy`** clauses are advisory and not mandatory. This allows the implementation to take advantage of devices with access to the shared memory or implement different caching and prefetch techniques. To make sure that data that could have moved to a device is valid again in the host, SMP code must also use the **`copy`** clauses or appear after an OpenMP **`flush`** (either explicit or implicit).

2.2 Example

Fig. 1 shows some of these features applied to a SparseLU code. The master thread traverses the *kk* loop and because a **`task`** construct prepends the functions *lu0*, *fwd*, *bdiv* and *bmod*, it will spawn tasks for each of the calls. When the tasks are created, the runtime will use the dependence information to build a dynamic task graph. The `[BS][BS] pointer` notation specifies that a block of size $BSxBS$ is being pointed by the pointer. Task with no predecessors will be elegible for execution and they will release new tasks as they finish. Note that because of the sparseness of the computation it is difficult to have a pre-computed task dependence graph.

```
#pragma omp target copy_deps
#pragma omp task input([BS][BS] diag) inout([BS][BS] col)
void fwd(float *diag, float *col);
#pragma omp target copy_deps
#pragma omp task input([BS][BS] row, [BS][BS] col) inout([BS][BS] inner)
void bmod(float *row, float *col, float *inner);
#pragma omp target copy_deps
#pragma omp task input([BS][BS] diag) inout([BS][BS] row)
void bdiv(float *diag, float *row);
#pragma omp target copy_deps
#pragma omp task inout([BS][BS] diag)
void lu0(float *diag);

for (kk=0; kk<NB; kk++) {
  lu0(A[kk][kk]);
  /* fwd phase */
  for (jj=kk+1; jj<NB; jj++) {
    if (A[kk][jj] != NULL)
      fwd(A[kk][kk], A[kk][jj]);
  /* bdiv phase */
  for (ii=kk+1; ii<NB; ii++)
    if (A[ii][kk] != NULL)
      bdiv (A[kk][kk], A[ii][kk]);
  /* bmod phase */
  for (ii=kk+1; ii<NB; ii++)
    for (jj=kk+1; jj<NB; jj++)
      if (A[kk][jj] != NULL) {
        if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
        bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
      }
}
#pragma omp taskwait
```

Fig. 1. SparseLU example with OmpSs

The **target** directive prepending each task specifies that the execution of the tasks requires that the specified data (in this case the same as the data dependences) is ready in the location where the task is executed. In case of a pure SMP run this information will be ignored, but for our cluster implementation it means that the runtime will need to move the data as necessary. After, the **taskwait** all tasks will be completed and the data will be available to the master thread (so it could print the result for example).

3 Implementation

Nanos++ overview. The OmpSs infrastructure is composed by the Mercurium compiler and the Nanos++ runtime library. The compiler gets as an input the program annotated with the OmpSs directives and generates a transformed version that invokes services of the Nanos++ runtime.

Nanos++ is an extensible runtime library that supports OmpSs. Its responsibility is to execute *task parallel* applications as specified by the compiler. Nanos++ offers mechanisms to schedule the execution of the tasks. The runtime schedules these tasks on the available resources making sure all constraints specified by the user (order, coherence, ...) are maintained. Nanos++ comes with a few scheduling policies (e.g. fifo, lifo, ...) but allows new ones by means of plug-in extensions.

Most of the runtime is independent from the actual target architectures supported (and various of these architectures can be active at the same time). Nanos++ currently supports several "conceptual" architectures: *smp*, *smp-numa*, *gpu*[10], *tasksim* (a simulated architecture)[14] and cluster.

In the following sections we describe the Nanos++ *cluster* architecture, the general ordering and coherence mechanisms that ensure the correctnes of the execution and the data-affinity scheduler that we have implemented to improve performance.

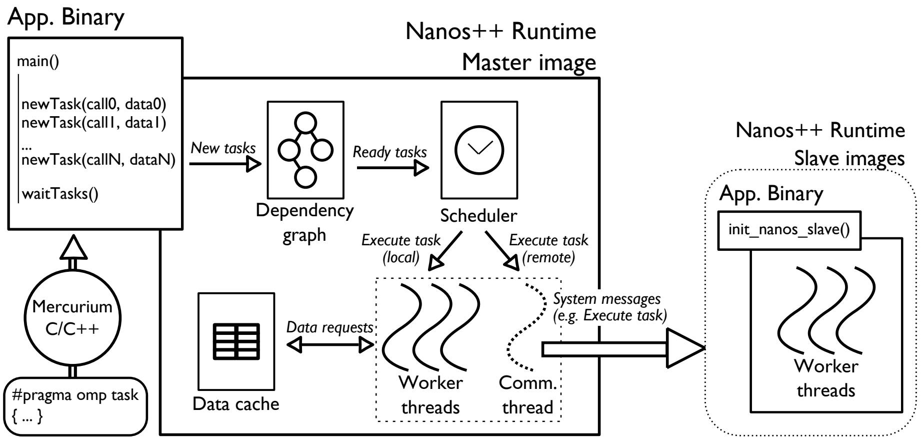


Fig. 2. Nanos++ cluster overview

Nanos++ cluster architecture. Fig. 2 shows the general design of Nanos++ for clusters. The main difference with respect to other supported architectures is that, when running in a cluster, there will be more than one image of the runtime running at the same time (i.e., one on each cluster node). When the execution starts, the first image will become the *master* image and the rest of the images will become the *slave* images. This structure creates an identical address space on each node, which gives the view of a single distributed address space. This eases the implementation of the proposed programming model(s) but it limits the total amount of memory used by the program (i.e., it can use only as much memory as is available in the master node).

All low level communications for control information and data transfers are implemented using *active messages*. We used GASNet [4] for this functionality since it offers a network-independent API with native support for various network technologies.

Initially there is only one task that is executed by the master. As this task starts creating new tasks, they will be scheduled to the local threads of the master node and to a communication thread that represents the remote nodes. When a task is scheduled to the communication thread it will be executed by a remote node. There is only one communication thread[1] that will be pooling the task pool for each node of the cluster in a round-robin fashion. This thread also keeps track of the execution of tasks on remote nodes, when a node has no task on execution, it will send a new one to it.

The remote execution of tasks is a straightforward process. First, the general coherence mechanisms of the runtime are invoked to ensure that all data that will be needed by a task is available in the remote node (and it is up-to-date). If not, data is gathered from its current location. If the data is available in the master node it is sent directly to the remote node. If it is only available in a remote node, a message is sent to that node so it sends the data to the new owner. This first step can be done concurrently with the execution of other remote tasks to overlap communication and computation but our current implementation does not apply this optimization yet.

After this, the master sends a control message with the task information to start the execution of the remote task. The slave images are constantly waiting for upcoming requests and they will start the execution of the task as soon as the request arrives. When the task finishes, another active message is sent back to the master to notify the completion of the task.

Tasks executed in a remote node can create new tasks that use the data transferred or created by their parent task. This allows scalable data decomposition to be coded in the application. These local tasks will be executed by any thread that becomes available in the node (and before going to fetch more work from the master node). Currently, we do not implement stealing between the local queues of the slave nodes.

Nanos++ coherence support. To run tasks in architectures with separated address spaces, is necessary a mechanism that copies data from the host (or where the task was created) to where the task will execute and to control the coherence between the different address spaces. With this mechanism Nanos++ is able to schedule tasks to run everywhere in the system.

A centralized directory keeps track of the physical location of the data and one software *cache* per cluster node stores data for tasks executed in that node. This *cache*

[1] Our design allows to have more than one if necessary.

manages the transfers from main memory to the remote nodes memory, avoiding unnecessary data movement and implementing different coherence policies (by default a writeback policy is used).

From the point of view of the runtime, copy operations can be of two types: synchronous and asynchronous. Having synchronous copies means that the cache will wait until all data is available when preparing a task for execution. Asynchronous copies allow the runtime to place all copy operations of a to-be-scheduled task and start doing other things while data is being transferred for that task. Although asynchronous operations are not implemented for clusters (as mentioned before), it is relatively easy to incorporate to the Nanos++ cluster architecture and would enable the runtime to overlap data transfers and computation.

It is important to notice that the coherence mechanisms assume program correctness. Applications where tasks write to the same data simultaneously without specifying dependencies result in an undefined behavior.

Nanos++ task scheduler. Minimizing the number of transfers through the network is critical in clusters to avoid network saturation and reduce the impact of latency in performance. A locality-aware scheduling policy has been implemented to favor scheduling of tasks in the remote nodes where most of their data reside. Directory entries store a map with the location of data in the system. When a new task is submitted, the scheduler computes, for each node, an affinity score based on the location and size of the data needed by the task. This score is used to place the task in the queue of the node with the highest affinity. A global queue is used when there is no node with the highest affinity.

The runtime looks for work for work on each node's queue first, if it is empty, the global queue is used, if this is empty too then another node's queue may be used to fetch a ready task, this aims to prevent application imbalance.

4 Evaluation

4.1 Methodology

In order to evaluate our runtime environment we measured the scalability of several applications on a cluster of SMPs. We implemented two versions of each application: one using OmpSs and one using MPI. With this, we compared the performance of the OmpSs version while running with Nanos++ with the performance obtained by the MPI version. We consider this a good measure of how good can OmpSs can be compared to a well known standard like MPI.

Environment. The benchmarks were run in the MareNostrum cluster of PowerPC 970MP @ 2.3GHz processors. Each node has 2 CPUs with 2 cores each and 4 Gb of physical memory and runs the SLES 10 operating system. The interconnection network of the cluster is based on Myrinet hardware along with the Myrinet Express driver. All benchmarks were compiled using the Mercurium C++ compiler version 1.3.5.7 using the GCC compiler as the backend compiler, -O3 optimization level was always used.

OmpSs was run using the MPI conduit for GASNet. Since there is no specific conduit available for the Myrinet Express driver, the MPI conduit allowed us to indirectly use Myrinet Express through the MPICH library that was installed on the system.

Applications

Matrix Multiply. It performs a dense matrix multiplication of two square matrices. Each matrix is divided in blocks; in the MPI version this is used to tile the execution of the algorithm, in the OmpSs version tiling is also applied, however the algorithm is structured slightly different. In the OmpSs version there are two different types of tasks; the mission of the first type tasks is to create the other type of tasks, which are the ones that perform the computation, and also distribute the data implicitly during the process. Using this schema the second tasks benefit from better data locality, since the parent task already requested their needed data. The MPI version and the OmpSs version used a simple kernel in order to perform the matrix multiplication. The matrices had 32x32 blocks of 400x400 doubles on both versions.

NAS EP. The EP benchmark generates pairs of Gaussian random deviates according to a specific scheme. The main loop keeps all its data private until the end of the execution, where a reduction is done. The implementation for OmpSs divides the main loop of the benchmark in tasks, and implements the reduction manually. We implemented the OmpSs version porting it from the C implementation of the 2.3 NAS Parallel Benchmarks. The MPI version comes from the original NPB v2.3 for Fortran. We used the class C problem size.

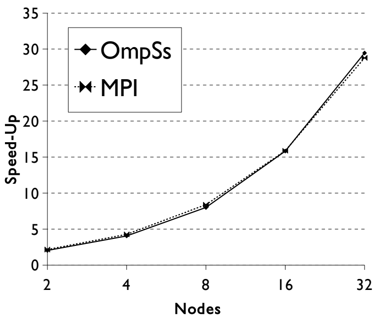
STREAM. STREAM is a benchmark that measures memory bandwith for simple kernels, intended for use with large data sets. It performs 4 simple operations on a three one dimensional arrays. It does not share any data between nodes so, as EP, we expected to be able to run STREAM without any problems with OmpSs. We used 500 Mb arrays in order to use the maximum memory that the GASNet configuration used could handle.

Sparse LU. The Sparse LU computes a LU decomposition on a sparse matrix and can have empty blocks. Due to the sparseness, the total number of tasks generated is less than in a regular LU. Task parallelism with dependencies can benefit from this situation as it can overlap multiple iterations at the same time whereas MPI needs barriers across different iterations. The matrix size used was the same as for the Matrix Multiply, 32x32 blocks of 400x400 doubles.

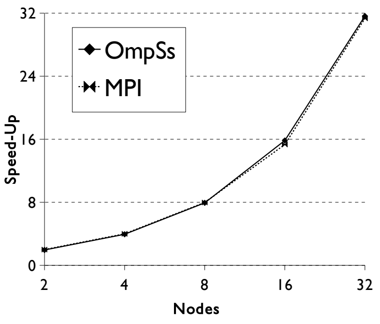
Experiments. We run the selected applications with different configurations of numbers of nodes to obtain the speed-up of each application for both OmpSs and MPI. We selected the biggest data set possible, since we did not wanted that the performance obtained was limited due to using a small problem size. As a baseline, for the speed-up we use the execution time of the serial version.

4.2 Results

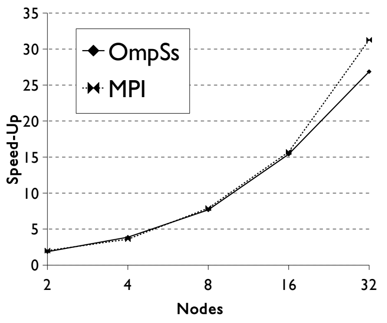
Matrix Multiply. The benchmark achieves a good performance on OmpSs, almost identical to the MPI version. This is somewhat expected since Matrix Multiply has a lot of data parallelism that can be exploited efficiently using either MPI or task parallelism. Figure 3(a) shows the results obtained for both OmpSs and MPI, perfect scalability is not achieved since communication and computation are not overlapped, but the scalability of the OmpSs code is on par with the MPI code.



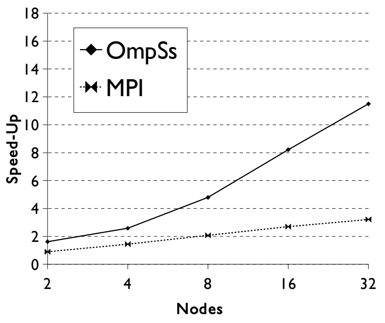
(a) Matrix Multiply (32x32 blocks of 400x400 doubles)



(b) NAS EP (Class C)



(c) STREAM (500Mb arrays)



(d) Sparse LU (32x32 blocks of 400x400 doubles)

Fig. 3. Results comparing OmpSs and MPI

STREAM. The results obtained by the STREAM benchmark are almost on par with MPI. As seen in figure 3(c) We achive almost the same scalability, only losing a little bit of performance due to the centralized initialization of the tasks. MPI outperforms OmpSs because of the SPMD model, since, besides the first synchronization done when initializing the MPI runtime, there is nothing to setup. On the other hand the creation of tasks in OmpSs takes some time, and it is proportional to the number of tasks, in addition, the distribution of these tasks also takes time proportional to the number of nodes of the execution.

Sparse LU. Sparse LU performs better than MPI since it exploits the advantages of having fine-grained tasks with dependences. This is specially important in sparse matrices since data parallelism is lower than in non-sparse ones. The MPI code can also be optimized in order to be conscious of this sparseness, however the changes we did to the benchmark only optimized the amount of data transferred between nodes, keeping the same application structure. On the other hand, the changes to the OmpSs code were minimal, and the benefits measured were greater than with the MPI version. Figure 3(d) shows this effect, where the OmpSs application achieves higher scalability than MPI on any number of nodes.

NAS EP. The EP benchmark has almost no data sharing among tasks, so it fits well on the set of applications that can achieve a good performance on distributed environments. With OmpSs there is no exception and the results showed a perfect lineal speed-up, on par with the original MPI implementation. Figure 3(b) shows the results we obtained executing the class C of the benchmark.

5 Related Work

Probably the best well known examples of parallel programming models are OpenMP and MPI. However, each of them has its own disadvantages and there has always been a good number of projects trying to address them or proposing new features in order to make them more suitable for the HPC systems we can find nowadays.

OpenMP was designed to provide a high productivity environment to produce parallel programs. Originally focused on dealing with loop-based parallelism, it was recently updated to the version 3.0, which includes new ways of expressing parallelism in the form of *tasks* [11].

OpenMP has influenced many projects due to its ease of use and simplicity. Cilk[3] is an example of a programming model which also provides task-based parallelism that can be expressed with simple keywords in a sequential code. Distributed Shared Memory (DSM) systems have also tried to offer this simple vision of a distributed environment by adding an extra software and/or hardware layer to the memory hierarchy that virtualizes the address space of the applications, allowing OpenMP, or other applications conceived for shared memory, to run on distributed memory architectures. The big disadvantage of these systems is that the memory access time increases dramatically, difficulting the task of achieving a reasonable performance. Techniques like data pre-send and pre-fetch along with relaxed memory consistency [8] have tried to overcome this, however, only a limited number of applications have benefited from such techniques.

Basumallik et al.[2] presented another approach that aimed to translate OpenMP to MPI, focusing on parallel loops. While it has achieved a good performance when running several OpenMP benchmarks, it does not offer asynchronous parallelism like we provide with OmpSs.

MPI has been a de facto standard in parallel programming for distributed environments. It offers explicit communication calls to transfer data among a set of processes running on different nodes of a cluster. The main disadvantage is that this approach can be complex to apply to some applications, and it requires a lot of effort from the programmer.

Partitioned Global Address Space (PGAS) programming models have tried to simplify all this burden, and offer a more friendly environment to develop distributed applications. They try to accomplish this by providing a global address space, which is distributed among the memory of each node of the execution. With this, they offer the programmer a simplified vision of the distributed environment, easing the development and porting of sequential applications to the PGAS. UPC [7] is one of these models, it takes also ideas from OpenMP in the form of compiler annotations but also has explicit communication calls.

Chapel [5] and X10 [6] implement an Asynchronous PGAS (APGAS), which offer asynchronous parallelism and mechanisms to synchronize it. In both environments is the responsibility of the programmer to deal with the data distribution and coherence.

An alternative way to provide asynchronous parallelism on clusters is the one explored by Marjanovic et al.[15], a hybrid programming model that composes SMPSs, a programming model that inspired OmpSs, with MPI. The main idea is to encapsulate the communications in tasks so they are executed when the data is ready. This technique achieves an asynchronous dataflow execution of both communication and computation.

6 Conclusions and Future Work

We have presented an implementation of OmpSs for clusters, a programming model that aims to be a high productivity environment without loss of performance when compared to other solutions. Coming from StarSs and OpenMP, OmpSs parallelization comes in the form of compiler directives that can be used to annotate sequential code. With this annotated code, the Mercurium C++ compiler can generate parallel code to run on top of the Nanos++ runtime. Applications built this way can be run on several architectures including GPUs and clusters of SMPs. In this work, we have evaluated the performance of different applications when running on a cluster of SMPs, and we have compared the performance obtained against the same applications developed with MPI. The performance achieved by OmpSs is on par with the performance obtained by MPI and even in some cases it can outperform MPI thanks to the asynchronous parallelism implemented in the form of task-based parallelism with data dependencies.

Since Nanos++ is a young project there is still much work to be done. We plan to implement techniques that allow us to overlap computation and communication, in the form of pre-sending or pre-fetching data before a tasks starts the execution, this will be done in collaboration with a more conscious scheduling. Also, one of our goals is to being able to scale further than the number of nodes that we have presented on this work, and also to offer a better handling of multiple threads on slave images. To achieve this, we will have to implement techniques to improve data distribution and allow local memory allocation on the remote nodes. This will allow us to fully use the physical memory of the cluster. Another direction we would like to explore is to include other devices into the cluster architecture, making Nanos++ capable of managing the execution of a single application on a cluster composed by SMPs and GPU devices.

Acknowledgments. We thankfully acknowledge the support of the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT 217068) and the ENCORE project (FP7-248647), the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980) and the TEXT project (IST-2007-261580).

References

1. Ayguade, E., Badia, R., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Orti, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: IWOMP: Evolving OpenMP in an Age of Extreme Parallelism, Dresden, Germany, pp. 154–167 (June 2009)
2. Basumallik, A., Eigenmann, R.: Towards automatic translation of openmp to mpi. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 189–198. ACM, New York (2005)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
4. Bonachea, D.: GASNet Specification, v1.8. Technical report, U.C. Berkeley (2006)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21, 291–312 (2007)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, New York, NY, USA (2005)
7. UPC Consortium. UPC Language Specifications v1.2 (May 2005)
8. Costa, J.J., Cortes, T., Martorell, X., Ayguade, E., Labarta, J.: Running OpenMP applications efficiently on an everything-shared SDSM. J. Parallel Distrib. Comput. (May 2006)
9. Duran, A., Pérez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: OpenMP in a New Era of Parallelism, pp. 111–122. Springer, Heidelberg (2008)
10. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia., R., Ayguade, E., Labarta, J.: Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL. In: Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2010) (October 2010)
11. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
12. Josep, M., Perez, R.M.: Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In: IEEE Int. Conference on Cluster Computing, pp. 142–151 (September 2008)
13. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
14. Rico, A., Duran, A., Cabarcas, F., Ramirez, A., Etsion, Y., Valero, M.: Trace-driven Simulation of Multithreaded Applications. In: Proceedings of the 2011 ISPASS (to appear, 2011)
15. Ayguadé, E., Marjanovic, V., Labarta, J., Valero, M.: Effective communication and computation overlap with hybrid mpi/smpss. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 337–338. ACM, New York (2010)