2009 International Conference on Parallel Processing

Speeding Up Distributed MapReduce Applications Using Hardware Accelerators

Yolanda Becerra, Vicenç Beltran, David Carrera, Marc González, Jordi Torres and Eduard Ayguadé
Technical University of Catalonia (UPC) - Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{yolanda.becerra, vbeltran, david.carrera, marc.gonzalez, jordi.torres, eduard.ayguade}@bsc.es

***Abstract*—In an attempt to increase the performance/cost ratio, large compute clusters are becoming heterogeneous at multiple levels: from asymmetric processors, to different system architectures, operating systems and networks. Exploiting the intrinsic multi-level parallelism present in such a complex execution environment has become a challenging task using traditional parallel and distributed programming models. As a result, an increasing need for novel approaches to exploiting parallelism has arisen in these environments. MapReduce is a data-driven programming model originally proposed by Google back in 2004 as a flexible alternative to the existing models, specially devoted to hiding the complexity of both developing and running massively distributed applications in large compute clusters. In some recent works, the MapReduce model has been also used to exploit parallelism in other non-distributed environments, such as multi-cores, heterogeneous processors and GPUs. In this paper we introduce a novel approach for exploiting the heterogeneity of a Cell BE cluster linking an existing MapReduce runtime implementation for distributed clusters and one runtime to exploit the parallelism of the Cell BE nodes. The novel contribution of this work is the design and evaluation of a MapReduce execution environment that effectively exploits the parallelism existing at both the Cell BE cluster level and the heterogeneous processors level.**

I. Introduction

In an attempt to increase the performance/cost ratio, large compute clusters are becoming heterogeneous at multiple levels: from asymmetric processors, to different system architectures, operating systems and networks. At the same time, latest research trends show that we are in a massively distributed computing era, with cluster, grid and cloud computing being some of the most relevant topics of research in computer science for the next years to come. Exploiting the intrinsic multi-level parallelism present in such a complex execution environment has become a challenging task using traditional parallel and distributed programming models. As a result, an increasing need for novel approaches to exploiting parallelism has arisen in these environments.

MapReduce is a framework originally designed by Google to exploit large clusters to compute parallel problems. It is based on an implicit parallel programming model that provides an easy and convenient way to express certain kinds of distributed/parallel problems. This framework is composed of an execution runtime and a distributed file system that take care of all the details involved in the distribution of tasks and data across nodes. The runtime and the distributed file system also handle all the issues related to fault tolerance and reliability, which are crucial in such a dynamic environment.

In the last years, an important increase in the internal heterogeneity of nodes has been observed in an attempt to improve its power efficiency and performance. Clear examples of this tendency are the IBM's Cell BE processor and the Nvidia's Tesla GPUs. This extreme intra-node heterogeneity, that starts becoming a usual scenario in current computing facilities (i.e. the MareIncognito cluster used in this paper, which is composed of a mix of Cell BE and Power6 blades, and the RoadRunner [4] cluster, composed of Opterons and Cell BE blades) and will be the norm in the future, hinders the efficient use of node resource in a convenient way. In this new scenario, the actual challenge is to exploit heterogeneous intra-node resources in a transparent way. To address this situation we need to extend current programming models and runtime frameworks to be able to exploit these specialized resources, but without breaking the easy and convenient way to design and run distributed problems that the MapReduce model provides.

In this paper we present and evaluate a prototype system that leverages the MapReduce programming model to exploit the multi-level parallelism existing in next-generation heterogeneous clusters. The prototype, based on the Hadoop implementation of MapReduce and tested on top of more than 60 Cell BE-enabled nodes, is able to distribute the load across nodes while taking advantage of the hardware accelerators. The system may be easily extended to take advantage of other existing accelerators in the system, such as GPUs or new developments to come.

Using the prototype, we present different experiments in which we evaluate the potential raw performance of hardware accelerators when used in a single node and without any of the overheads introduced by distributed computing runtimes, to later on evaluate its performance in a real distributed computing environment. We use two different workloads in our evaluation: one that is data-intensive though still CPU-demanding, and another that is CPU-intensive only. As the experiments demonstrate, the effectiveness of distributely using hardware accelerators can be clearly differentiated according to the nature of the workload.

The rest of the paper is structured as follows. In section II we introduce some technical background necessary to understand our working environment. In section III we describe the

0190-3918/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPP.2009.59

IEEE computer society

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

prototype architecture designed to run our experiments, that we present in section IV. In section V we show which are the next steps of our work. Finally, section VI shows the related work and section VII the conclusions of our work.

II. Technical Background

This section provides some technical background about both the MapReduce programming model and the Cell BE processor characteristics.

A. MapReduce

MapReduce [16] is a programming model proposed by Google to facilitate the implementation of massively parallel applications that process large data sets. This programming model hides from the programmer all the complexity involved in management of parallelization. The programmer only has to implement the computational function that will execute on each node (the *map()* function) and the function to combine the intermediate results to generate the final result of the application (the *reduce()* function).

In the MapReduce model all nodes in the cluster execute the same function but on a different chunk of input data. The MapReduce runtime is responsible for distributing and for balancing all the work across the nodes, dividing the input data into chunks, assigning a new chunk every time a node becomes idle and collecting the partial results.

There are many runtime implementations to support this model, depending on the type of environment they aim to support. For example, in the case of Google implementation [16] and in the case of Hadoop [2], they target the execution on a distributed execution environment that uses a distributed file system to support the data sharing (the *GFS* [20] in the case of Google, and the HDFS [3] in the case of Hadoop). There also exists implementations for shared-memory multiprocessors as, for example, the Phoenix implementation [24] and the implementation for Cell processor [15].

B. Cell BE

The Cell BE architecture [19], has nine processing cores on a single chip: one 64-bit Power Processing Element (PPE core) and eight Synergistic Processing Elements (SPE cores) that use 18-bit addresses to access a 256K Local Store. The PPE core accesses system memory using a traditional cache-coherent memory hierarchy. The SPE cores access system memory via a DMA engine connected to a high bandwidth bus, relying on software to explicitly initiate DMA requests for data transfer. The DMA engine can support up to 16 concurrent requests of up to 16K, and bandwidth between the DMA engine and the bus is 8 bytes per cycle in each direction. Each SPE uses its Local Store to buffer data transferred to and from system memory. The bus interface allows issuing asynchronous DMA transfer requests, and provides synchronization calls to check or wait for previously issued DMA requests to complete.

SIMD support in the Cell is one of the most important sources of computational power [21]. The Cell BE supports vector operations that operate on memory contiguous data sets of 16 bytes. Several restrictions apply, concerning to memory alignment conditions. In particular, the Cell architecture requires every vector operation to operate with aligned data to 16-byte memory boundaries.

The combination of a multi-core design and the available SIMD support, converts the Cell BE architecture into a very power efficient architecture, being a suitable accelerator unit to be considered in data centers.

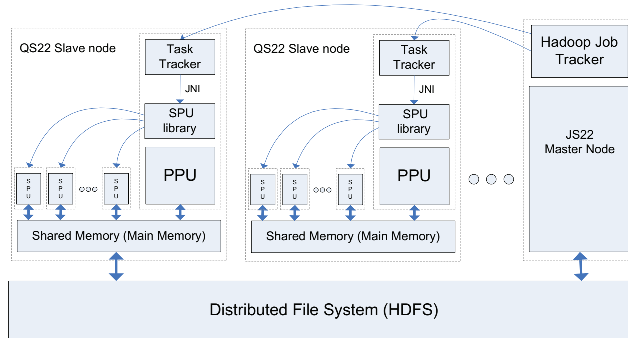

Fig. 1. Architecture of the system: 2 levels of parallelism

III. Prototype Architecture

We have designed a prototype system in order to explore the use of the MapReduce programming model to fully exploit the two levels of parallelism existing on a cluster of heterogeneous multi-core nodes. Our prototype aims to provide the programmers with automatic support to distribute the data and work across the cluster nodes and then across the intra-nodes processing units.

This prototype architecture has two main components, one of each devoted to manage one of the parallelism levels. The first component is based on Hadoop, a MapReduce implementation from Apache. The task of this component is to partition the data and to assign a piece of work to each node in the cluster. The processing routine executed by each node invokes the second component of our prototype. This second component has to implement a second level partition of the data, that is the intra-node distribution of the data, and then to execute the actual computing function. The resulting architecture is shown in Figure 1.

In the next subsections we describe in detail each one of the components of our prototype architecture.

A. MapReduce Implementation in the Hadoop project

Hadoop is the MapReduce runtime implementation that we use to exploit the cluster-level parallelism in our prototype system. Hadoop is open source and is provided by the Apache Software Foundation. This runtime is designed to manage applications on a distributed execution environment.

The applications initially targeted by Hadoop are massively parallel applications with a very large data set input. The main goals of Hadoop are to provide the applications with high scalability, with a scheduling of map tasks that balances the load of the nodes and that speeds-up the access to shared data and with a fault tolerance execution environment.

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

The mechanism to implement shared memory is based on a distributed file system also provided by the Hadoop project: Hadoop Distributed File System (HDFS). Thus, data sharing is implemented through HDFS files. File blocks are distributed across the local disks of the nodes and can be replicated, in order to implement fault tolerance. In addition, HDFS can decide to change the blocks location in order to favour local accesses. The architecture of this File System is a master/slave architecture. The master process (NameNode) manages the global name space and controls the operations on files, and each slave process (DataNode) implements the operations on those blocks stored in its local disk, following the NameNode indications.

Hadoop allows the programmer to have two different work partition levels: the first level defines the work assignment unit of a node (which is named *split*) and the second level defines the work unit of a map() function (which is named *record*). A *split* can later be divided into one or more *records*.

The runtime has two different processes involved. The process which distributes work among nodes is named JobTracker. This process uses the method configured by the programmer to partition the input data into splits and thus, to populate a local job-queue. If a node in the system becomes idle, the JobTracker picks a new job from its queue to feed it. Thus, the granularity of the splits have a high influence on the balancing capability of the scheduler. Another consideration of the map tasks scheduling is the location of the blocks, as it tries to minimize the number of remote blocks accesses.

The process that controls the execution of the map tasks inside a node is named TaskTracker. This process receives a split description, divides the split data into records (through the RecordReader component) and launches the processes that will execute the map tasks (Mappers). The programmer can also decide how many simultaneous map() functions wants to execute on a node, that is, how many Mappers the TaskTracker has to create. When a TaskTracker ends the processing of one split and is ready to receive a new split, it sends a notification to the JobTracker. In addition, in order to provide the environment with fault tolerance capability, during the process of a split the TaskTracker sends periodic *heartbeats* to the JobTracker. This way, the JobTracker can detect a node failure and reschedule the task to another TaskTracker. The JobTracker is also responsible for collecting and sorting the partial results produced by the Mappers in order to use them as the input for the reduce phase.

In our prototype architecture, the Hadoop runtime is used to implement the first level of the data and work distribution, that is, the work distribution across the nodes. In this level, the implementation of the map() function invokes the routine to execute the distribution of both work and data inside one node, and waits until the parallel computation inside the node is finished.

B. Accelerating the Mappers using the Cell BE

In order to make the most of the cluster of Cell BE processor from the Hadoop framework we have used the Java Native Interface [26] to directly access native libraries. We have implemented two shared native libraries that allow Hadoop to benefit from the SPUs acceleration. The first native library that we have implemented is a simple runtime that allows us to divide and execute task on the SPUs, resulting in the prototype architecture depicted in Figure 1. The second library is a proxy to an existing MapReduce framework for the Cell processor described in [15], which deals with the explicit partitioning of data across SPEs using the MapReduce model. The use of the second library, is limited to the single node experiment presented in section IV-A, and is only included to evaluate its performance. In both cases, the map() function run by the TaskTrackers on the Hadoop runtime invokes the Cell framework, using the shared library that corresponds with the prototype architecture used at each moment.

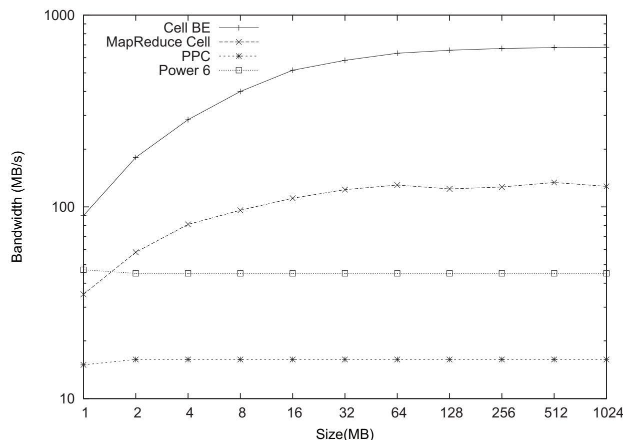

Fig. 2. Raw node encryption performance

IV. Experiments

In our evaluation we have conducted two different sets of experiments, the first set is focused on a data-intensive application and the second set is focused on a CPU-intensive application.

We choose the encryption of a very large working set as our data-intensive application, because this is a representative example of the kind of application MapReduce is designed for. The CPU-intensive application is a montecarlo program that estimates the value of Pi and can be easily run in parallel. The precision of Pi is proportional to the number of samples calculated, that is the input of this application.

For both workloads we run two versions of each application: one is a Java-pure application, which is the original code that would be run in a Hadoop cluster; the second one is a Cell accelerated code, which introduces the novelty of the experiments that we present. To run Cell-accelerated applications we needed to rewrite the kernel of these applications (encryption in one case, and montecarlo simulations in the other one) to be directly invoked by the Hadoop runtime.

In the following two subsections we present the results obtained for each kind of application. In both cases, we present first a single-node experiment in which we evaluate the potential of the Cell-accelerated version of the application

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

kernel. We do not use Hadoop nor any other kind of distributed computing middleware. Later on, we run the same application in a distributed environment, ranging from 4 nodes to more than 60 in our largest tests. The experiments included in each subsection will illustrate how the communication and coordination tasks introduced by the middleware affect the horizontal scalability of each workload.

The system used to run the experiments is a 66 IBM QS22 blades cluster, each one equipped with 2x 3.2Ghz Cell processors and 8GB of RAM. These nodes were used as Hadoop DataNodes and TaskTrackers. We also used one IBM's JS22 blade equipped with 4x4.0Ghz Power 6 processor and 8GB of memory to run the Hadoop JobTracker and Namenodes (See section III-A for more details on the Hadoop architecture). All the nodes were connected using a Gigabit ethernet. We used IBM's JDK 1.6.0 to compile and run the Java codes. For the implementation of the Cell-accelerated kernel we used IBM's SDK 3.0. To develop and run the MapReduce applications we used Hadoop 0.19.

A. *Data-intensive workload*

The data-intensive application we have used to evaluate the prototype is data encryption. With security being an issue for many companies that host large data sets, this application may well represent an example of enterprise application that could benefit of the MapReduce model for both distributely hosting such a large amount of data, and distributely computing over the large working set.

We have implemented a 128 bits key AES encryption algorithm using both a Java code and a Cell-accelerated code. The Java version is based on the methods offered in the standard javax.crypto.Cipher package. The Cell accelerated AES encryption code is based on the work presented in [25].

In our first experiment using the AES encryption code we use one single Cell blade to evaluate the raw potential of the Cell acceleration when the workload is no subject to the communication and synchronization requirements that are present in distributed systems. For this purpose we run both the Java and the Cell application kernels to encrypt several working sets with different sizes. All the working set are cached in memory so that the encryption speed may be the maximum delivered by the hardware architecture. Notice that Hadoop is not involved in this experiment.

For comparison purposes we used 4 different configurations to test the encryption application: the first and second is the Java application encryption kernel on top of the PPE of a Cell processor and one a Power 6 processor, respectively; the third and forth configuration run on the Cell SPUs using the MapReduce runtime mentioned in section III-B and using a direct pthread implementation, respectively.

Figure 2 shows the encryption performance for the four evaluated configurations. The X axis are the size of the encrypted file in Mbytes, while the Y axis represents the encryption bandwidth in MBytes/s. Notice that both axis are in logarithmic scale. The results presented on this experiment are the average of ten executions for each configuration.

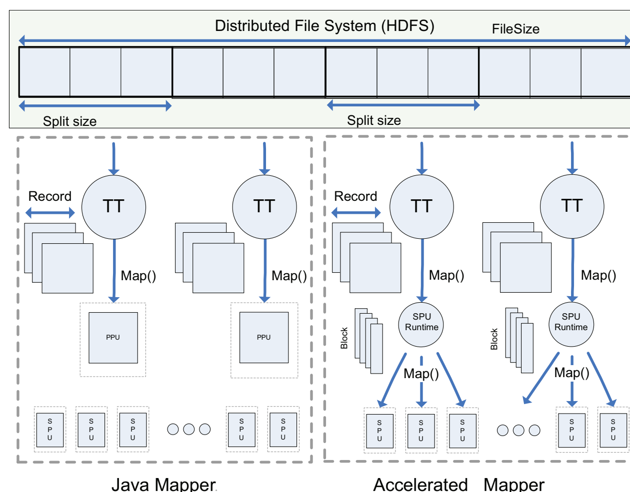

Fig. 3. Data distribution model

As it can be observed in this figure, the most performing implementation of the encryption kernel is the direct Cell-accelerated code. The second one is the version based on the MapReduce framework for the Cell that incurs in a considerable overhead because the way the PPEs are used to initialize the input data (basically the original input data must be copied again to internal buffers managed by the framework) Finally, the two Java configurations are the slowest. As it was expected, the results on the Power6 are much better than in the Cell PPE, since the PPE unit in the Cell is a limited implementation of the PowerPC family of processors (see section II-B for more details). As it is shown in the figures, the maximum data rate at which one Cell processor can encrypt data is near 700MB/s, while one Power6 core is around 45MB/s. Notice also that a QS22 blade such as the ones used in our experiments is equipped with 2 Cell processors, so the data encryption rate can be easily doubled. On a JS22 blade equipped with 4 Power6 cores the data encryption rate could be easily scale up to 4 times the value shown in Figure 2.

Once it has been demonstrated that the Cell processor can be used to accelerate data encryption tasks, it was time to run distributed experiments using MapReduce to see how the raw performance discussed above was affected by the overheads introduced by the Hadoop runtime.

For these experiments we used a Hadoop cluster composed of 1 JobTracker and 2 Namenodes running on top of a Power6 JS22 blade, and a variable number of DataNodes and TaskTrackers running on top of Cell-based QS22 blades. In particular, each Cell blade ran one DataNode process and two Mappers were run in parallel. The HDFS was configured to use 64MB blocks, and the files used on the experiments were set to have a replication level of 1 (so one single copy of each block was present in the cluster). The Hadoop log files were checked to verify that there were no errors in the executions.

We used two implementations of the encryption application in the data-intensive experiments. The first one uses Hadoop to distributely run the Java encryption kernel over the QS22 nodes; the second one uses Hadoop to distributely run the

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

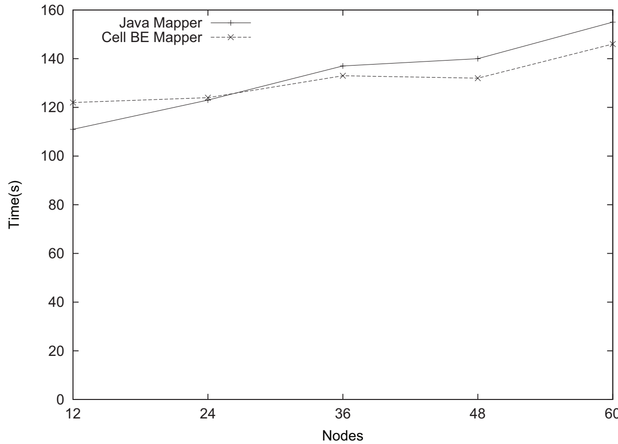

Fig. 4. Distributed encryption performance: proportional data set

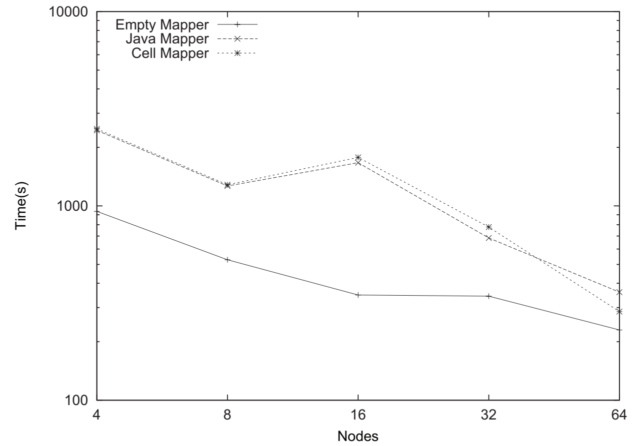

Fig. 5. Distributed encryption performance: 120GB data set

direct Cell encryption kernel described above. For these experiments, the data was partitioned as it is shown in Figure 3, using an split size of $FileSize/NumMappers$ and a record size of 64MB. In the case of the Cell-accelerated experiments, each record was split into 4KB data blocks that were sent to the SPUs in order to be encrypted.

Figure 4 shows the results for the first experiment in which the input data size is set to be proportional to the number of Mappers used in the test, using a fixed proportion of 1GB per mapper (i.e. the system is encrypting 120GB of data in the configuration that used 120 Mappers and 60 Cell blades, and 24GB in the configuration with 24 Mappers and 12 Cell blades). Notice that in these figures, the X axis represents the number of nodes used in the experiment (1 Mapper running in each of the two Cell processors of the QS22 blade). As it can be observed the Cell-accelerated mapper and the Java mapper offer a very similar performance for this application although it was demonstrated that the Cell mapper could clearly outperform the Java mapper in the single node experiment. This is due to the fact that most of the application time is spent on the Hadoop communication processes, including data being sent from DataNodes to the TaskTrackers using the loopback interface and the other way around. Thus, the runtime is the main limiting factor for this application.

In the second experiment involving the encryption application and Hadoop, we used a fixed data set size of 120GB and ran different executions of the application varying the number of nodes from 4 up to 64. We also included another implementation of the application (EmptyMapper) that did not perform any encryption, what resulted in an application that did read the data but did not perform any kind of processing over the data and did not collect any output, but remained using the Hadoop runtime to distribute the data across the TaskTrackers. The purpose of the last implementation is to estimate the overhead of the Hadoop runtime in a distributed system.

Figure 5 shows the results obtained for this second distributed experiment. As it can be observed, while the Hadoop runtime scales well with the number of nodes (reducing the executing time of the application when more nodes are used), the effect of hardware acceleration can be hardly noticed. Again, the cost of communications and synchronizations hide the effect of accelerating the execution time of each mapper. Notice also that the difference in the execution time between the Empty mapper (that is not performing any calculation over the data nor producing any output through the data collectors) and the other mappers is really small. We could measure that the *next* method in the application RecordReader class, what is used by the Hadoop runtime to send data to the mappers, was spending several seconds to send the data from the DataNode to the TaskTracker through the loopback interface, at a much slower rate than the actual maximum rate that can be delivered by such a virtual network interface, even in the case that all the data was resident in the OS buffer cache. This fact corroborates the assumption that communication is the limiting factor for data-intensive applications.

The above mentioned problem regarding the effective bandwidth at which data can be provided to the Mappers seems to be a problem not only for the encryption application but for most MapReduce applications using very large datasets. Looking at the results for the Terasort [6] contest, it can be observed that the most performing implementation of the sorting application (a Hadoop implementation made by Yahoo at the moment of writing this paper) is delivering an impressive overall sorting rate of 5017MB/s (1TB sorted in 209 seconds using 910 8-way nodes). Looking in more detail at the per node and per core rates, it can be observed that the testbed is sorting 5.5MB/s and each core does it at 0.6MB/s, what seems to point out that the effective data bandwidth at which data can be sent to the mappers was also the limiting factor, since the sorting capacity of a high-end processor may be well above that value.

B. CPU-intensive workload

For the CPU-intensive experiments we used a Pi number estimator based on a montecarlo probabilistic method. We took one of the Hadoop sample applications (PiEstimator) as the base for our experiment, and ported the code to the Cell

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

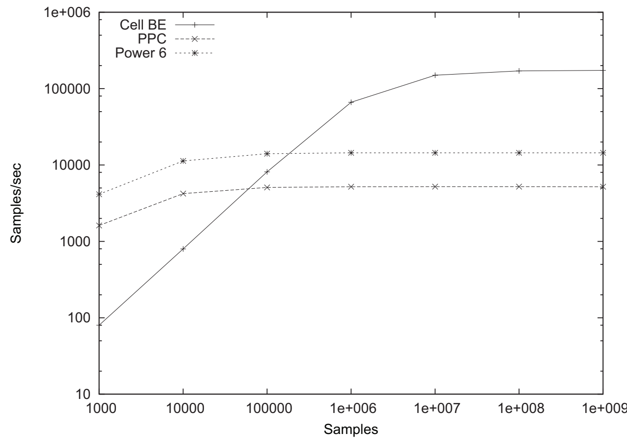

Fig. 6. Raw node Pi estimation performance

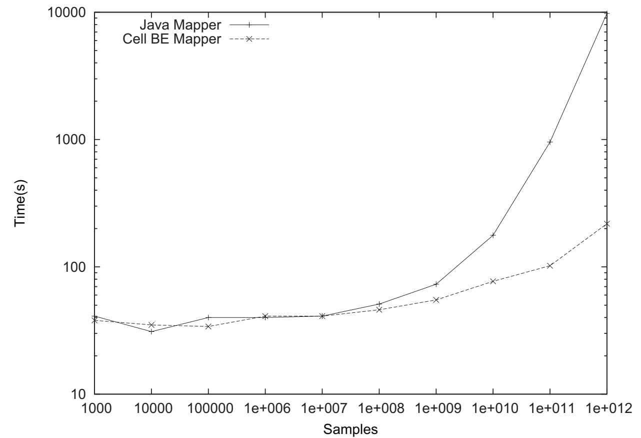

Fig. 7. Distributed Pi estimation performance: 50 nodes

platform to get a Cell-accelerated version of the Pi estimator. The Pi estimator code used produces an expected error of O(1/sqrt(N)) in the Pi estimation, where N is the number of samples done. For example, estimating Pi with 100,000,000 samples produces an actual accuracy of approximately 4 digits.

As with the data-intensive section, we first investigate the raw performance of a Cell processor isolated from the communication and synchronization overheads that are present in a distributed systems. We ran the Pi estimator application in a single Cell blade, using both the Java kernel and the Cell-accelerated kernel.

For comparison purposes we used three different configurations to test the Pi estimator application: For the first and second configuration we ran the Java application kernel on top of the Cell PPE processor and on a Power 6 core respectively; for the third configuration we ran the Pi estimator kernel on top of the Cell SPUs accelerators.

Figure 6 shows the performance of the three configurations for this single node experiment. The Y axis shows the number of samples per second that each application kernel implementation can calculate, while the X axis shows the total number of samples done, i.e. the size of the problem. Notice that both axis uses a logarithmic scale. As it can be seen, the Cell-accelerated kernel is the most performing implementation, although the overhead of work distribution about SPUs is only worth when the work to be done has some entity and is above the overhead of SPUs initialization. When the size of the problem is big enough, running more than 10 million samples, the Cell-accelerated kernel is one order of magnitude faster than the Java kernel running on top of the Power6, and even more when compared to the Cell PPE.

Once it has been demonstrated that the Cell processor can be used to accelerate the Pi estimator task, it was time to run distributed experiments using MapReduce to see how the raw performance discussed above was affected by the overheads introduced by the Hadoop runtime.

For these experiments we used a Hadoop cluster composed of 1 JobTracker and 2 NameNodes running on top of a Power6 JS22 blade, and a variable number of DataNodes and TaskTrackers running on top of Cell-based QS22 blades. In particular, each Cell blade ran 1 DataNode process and two Mappers were run in parallel. In this experiment there is no input working set since it is a CPU-intensive only task.

In the first experiment we run the Pi estimator application on top of 50 Cell BE nodes, and vary the number of calculated samples from 3e+10^3 to 3e+10^{12} (resulting in a range of samples per node that goes from 2e+10^3 to 2e+10^{10}). Figure 7 shows the results for this experiment. As it can be observed the Cell-accelerated mapper clearly outperforms the Java mapper when the number of samples calculated per node becomes high enough to overcome the overheads introduced by the Hadoop runtime.

In the second experiment, we set the number of calculated samples to be 1e+10^{11} and vary the number of nodes from 4 to 64. We also ran a second execution for the Cell accelerated application that calculates 10 times more samples than the original test, resulting in 3e+10^{12} samples calculated. Figure 8 shows the results for this experiment. As it can be observed, the Cell-accelerated mapper is clearly quicker than the Java mapper, and the difference in performance varies from one to two orders of magnitude, depending on the configuration. Notice that for the Cell-accelerated Mapper and configurations with 8 or more nodes, what is limiting the performance of the application is the Hadoop runtime. To evaluate the real performance of the accelerated code, we ran a second test with the Cell-accelerated mapper, but modified to calculate 10 more times samples than the previous time. This second run, in which 3e+10^{12} samples were calculated, is labeled in Figure 8 as *10x samples*. As it can be observed, when the load is higher, the limiting factor imposed by the Hadoop runtime is overcome, and the application shows the same linear reduction in the computing time than was observed for the Java mapper in the previous execution. The slope remains constant until the Hadoop runtime starts limiting the overall performance of the application again, in the 32 nodes configuration, where the application stops scaling its performance when increasing the number of TaskTrackers.

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

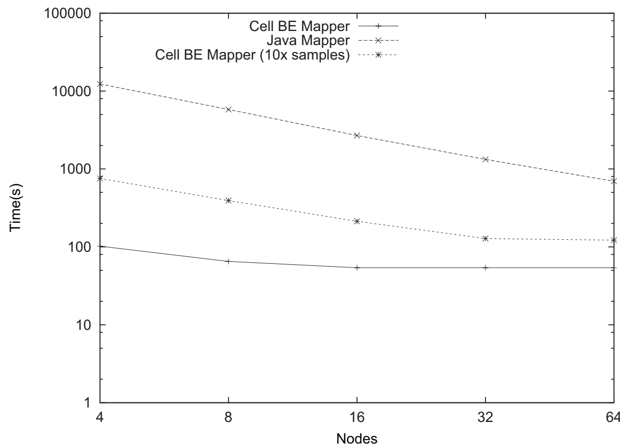

Fig. 8. Distributed Pi estimation performance: 10e+11 samples

V. Open Issues

From our experiences in the use of MapReduce to exploit the multi-level parallelism of heterogeneous clusters we have identified several upcoming problems to be addressed in our future work. In this section we summarize the most relevant experiments we want to carry on in the future.

The first one of the open issues we have identified is the automatic generation of accelerated code. While MapReduce is a high level programming model, the use of hardware accelerators requires some hardware conscious coding. Developing techniques that allow programmers to write high-level code using their favorite programming languages to be later transformed into hardware accelerated versions of the same code is maybe the most challenging issue yet to address.

The second aspect that requires more research is the study of energy issues involved in the use of hardware accelerators for data-intensive applications. Although experiments show that they can not benefit of hardware acceleration in the same degree that CPU-intensive only applications do, data-intensive applications can still benefit of energy savings resulting of using specialized cores to do accelerated tasks. Notice that while in data-intensive tasks the work done by the accelerators is not in the applications' critical path, doing that work in shorter time, more efficiently and with specially designed hardware can save energy, very specially in distributed environments composed of thousands of nodes.

We also plan to carry on research on clusters with an increasing level of heterogeneity, involving a dynamically variable number of both nodes enabled with hardware accelerators and general purpose nodes. The glimpse of a future made of compute clouds anticipates very dynamic and variable environments, in which the effects of dynamically changing the machines in which Mappers and Reducers run will have to be understood.

Finally, virtualization is a technology that is causing big impact in enterprise data centers in their process to keep growing. Large and virtualized enterprise data centers [11] are candidates to exploit hardware accelerators as we have presented in this paper, and virtualization technologies will need to deal with the management of hardware accelerators, exposing them to the virtual machines that run on top the physical machines.

VI. Related work

Combining two programming models under a hybrid approach it is not a new proposal [22], [23], [17]. Industry has taken this approach as a method to ensure scalability, specially in high performance computing systems. For instance, a distributed memory paradigm like the Message Passing Interface (MPI) [27] is used for node communication, while a shared memory paradigm like OpenMP [5] is used for the parallel execution within the node. Other similar approaches directly use a threading paradigm (e.g: pthreads) for the intra-node execution [4]. This hybrid solutions have been extensively used for scientific computing, but have not become generally used in other fields basically for two key issues. On one hand, these hybrid implementations require huge programming efforts to obtain a reasonable scalability of the system. This leads to very poor productivity. Secondly, the system is not tolerant to node faults, leading to an unacceptable solution when quality of service is important and have to be maintained above specific levels. Recently, the OpenMP programming model has included the definition of asynchronous tasks in its specification [8]. This has motivated the study of porting OpenMP to heterogeneous architectures [7], mapping asynchronous tasks to accelerator units. Although this clearly improves the programmability within a node, it is not yet clear if this will make the OpenMP programming model a candidate for the programming of high productivity computing systems.

PGAS languages have become an alternative to traditional MPI+OpenMP paradigm, as they mainly target the system scalability and overall productivity. Languages like Unified Parallel C (UPC) [10], CoArray Fortran [13] or Titanium [14], recover previous ideas on data distribution from distributed memory paradigms (e.g: High Performance Fortran). Under this paradigm data is distributed among the nodes under the programmer control. PGAS implementations deploy good scalability, but although productivity is highly improved respect a hybrid paradigm based on MPI and OpenMP, still important programming efforts are required to tune the application. Besides, current proposals have been proved to be reliable only in the scientific domain.

More recently, the PGAS model has evolved to the Asynchronous PGAS (APGAS) model. Examples of languages based on this model are X10 [18] or Chapel [12], both in current development under the DARPAs High Productivity Computing Systems (HPCS) program [1]. The term asynchronous refers to the fact that parallelism is expressed under the form of tasks that are to be executed asynchronously one from another. Data distribution and work balance are explicitly considered within the language definition. The use of accelerators to improve general system performance has been studied in [9], but this work is focused on a single node, while our work is aimed to exploit this accelerators in a distributed environment.

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.

VII. Conclusions

In this paper we have evaluated the use of the MapReduce programming model to exploit the computing capacity present in a cluster of nodes equipped with hardware accelerators. MapReduce is an emerging programming model for massively distributed environments and was originally developed to run computations over extremely large working sets. Hardware accelerators are more and more present in system configurations, being the Cell BE processor from IBM and Nvidia's Tesla GPU the more extended solutions. Hardware accelerators can really boost certain computational tasks while reducing the energy consumed in the process. Exploiting the computing capacity of such a heterogeneous environment is a challenging task since it comprises two potential levels of parallelism at both the cluster level and at the node level. In our experiments we leverage the MapReduce programming model to easily code applications that exploit both levels of parallelism with a minimal effort for the programmer. We use two different applications in our evaluation: a data encryption workloads as a case of data-intensive application; and a probabilistic Pi number estimator as a case of CPU-intensive application. Results show that while the use of MapReduce in such a distributed environment can take advantage of the hardware accelerators, the communication and synchronization overheads can hide the performance benefits of hardware accelerators in data-intensive applications. On the other hand, benefits on CPU-intensive only applications are remarkable. While from the point of view of raw performance data-intensive workloads may not seem to be the best candidates for using hardware accelerated versions of their applications, further research must be conducted in this area since energy efficiency of such a configuration may well be lower than the equivalent distributed application not using hardware accelerators.

Acknowledgements

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, the European Commission in the context of the FP7 HiPEAC Network of Excellence (contract no. IST-004408) and the FP7 PRACE Partnership for Advanced Computing in Europe (contract no. RI-211528), and the MareIncognito project under the BSC-IBM collaboration agreement.

References

[1] *DARPA project: High Productivity Computing Systems. http://www.highproductivity.org/*.

[2] *Hadoop Map/Reduce Tutorial http://hadoop.apache.org/core/docs/current/mapred_tutorial.html*.

[3] *HDFS Architecture http://hadoop.apache.org/core/docs/current/hdfs_design.html*.

[4] *Los Alamos Lab: High-Performance Computing: Roadrunner. http://www.lanl.gov/roadrunner/*.

[5] *The OpenMP Programming Model (OpenMP). http://www.openmp.org*.

[6] *Terabyte Sort Benchmark. http://www.hpl.hp.com/hosted/sortbenchmark/*.

[7] E. Ayguadé, R. M. Badia, D. Cabrera, F. I. Alejandro Duran, M. Gonzalez, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ort. A proposal to extend the openmp tasking model for heterogeneous multicores. In *5th International Workshop of OpenMP (IWOMP 2009)*, 2009.

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[9] V. Beltran, J. Torres, and E. Ayguadé. Improving web server performance through main memory compression. In *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 303–310, 2008.

[10] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi. Productivity analysis of the upc language. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 254, 2004.

[11] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. Enabling resource sharing between transactional and batch workloads using dynamic application placement. In *9th ACM/IFIP/USENIX International Conference on Middleware*, pages 203–222, 2008.

[12] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[13] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, D. Chavarria-Miranda, F. Cantonnet, T. El-Ghazawi, A. Mohanti, and Y. Yao. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2005)*, 2005.

[14] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an npb experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[15] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the Sixth Symposium on Operating System Design and Implementation, San Francisco*, pages 137–150, December 2004.

[17] N. Drosinos and N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04)*, 2004.

[18] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of the 2005 P-PHEC workshop, co-located with HPCA 2005*, 2005.

[19] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[21] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. 2005.

[22] G. Jost, H. Jin, D. an Mey, and F. F. Hatay. Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster. In *NAS Technical Report NAS-03-019*, 2003.

[23] G. Mahinthakumar and F. Saied. A hybrid mpi-openmp implementation of an implicit finite-element code on parallel architectures. In *Proceedings of the International Journal of High Performance Computing Applications*, pages Vol 16. 371–393, 2002.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[25] S. Siewior. *Diploma thesis: Acceleration of encrypted communication using co-processors http://diploma-thesis.siewior.net/html/*.

[26] Sun Microsystems, Inc. *Java Native Interface. http://java.sun.com/docs/books/jni/*.

[27] The Message Passing Interface. *The Message Passing Interface (MPI). http://www-unix.mcs.anl.gov/mpi/*.

Authorized licensed use limited to: UNIVERSITAT POLIT?CNICA DE CATALUNYA. Downloaded on June 10,2010 at 09:10:15 UTC from IEEE Xplore. Restrictions apply.