A Runtime Library for Platform-Independent Task Parallelism *

Panagiotis E. Hadjidoukas *Evaggelos Lappas* *Vassilios V. Dimakopoulos*
Department of Computer Science
University of Ioannina, Ioannina, Greece, GR-45110
{phadjido,elappas,dimako}@cs.uoi.gr

Abstract

With the increasing diversity of computing systems and the rapid performance improvement of commodity hardware, heterogeneous clusters become the dominant platform for low-cost, high-performance computing. Grid-enabled and heterogeneous implementations of MPI *establish it as the de facto programming model for these environments. On the other hand, task parallelism provides a natural way for exploiting their hierarchical architecture. This hierarchy has been further extended with the advent of general-purpose* GPU *devices. In this paper we present the implementation of an MPI-based task library for heterogeneous and* GPU *clusters. The library offers an intuitive programming interface for multilevel task parallelism with transparent data management and load balancing. We discuss design and implementation issues regarding heterogeneity support and report performance results on heterogeneous cluster computing environments.*

1. Introduction

With the advent of multi- and many-core technology, heterogeneous parallel and distributed computing systems emerge as the primary low-cost and powerful platform for high-performance applications. As commodity nodes improve their performance and expand their architecture diversity, not only the computing power but also the heterogeneity of these systems, in terms of hardware and software, increases. Meanwhile, new programming challenges for exploiting the hierarchical structure of these systems arise.

Task-based parallelism is very well suited to heterogeneous computing. In the (limited) form of the master-worker execution model, it has been applied on a wide class of applications and computing platforms. The latter range from multiprocessors and clusters of them to computational grids, as well as many-core architectures like (GP)GPUs. Moreover, MPI appears to be the de facto programming model for these platforms, considering the need for explicit communication between nodes and the availability of grid-enabled / heterogeneous MPI implementations.

There exists a number of programming models and runtime systems that provide support for task parallelism, but they have been traditionally platform-dependent. On shared-memory platforms, well known examples include Cilk [1] and OPENMP V.3.0 [2] which introduced the ‘task’ construct. On distributed-memory machines, a common approach combines data with task parallelism in the message-passing programming model (e.g. HPF/MPI [3]). Task Pool Teams [4] and Tlib [5] are runtime libraries that support the coordination of hierarchically structured tasks, extending the SPMD execution model to MPMD, where subsets of processors execute different data-parallel tasks. ADLB (Asynchronous Dynamic Load Balancing) is an MPI library for master-worker programming; worker processes put and get units of work into a shared queue without interacting with a master through explicit calls [6]. ADLB, does not currently exploit multithreading itself and requires hard partitioning of MPI processes, with the number of servers and workers explicitly set by the user.

The above systems assume mostly homogeneous systems where nodes are of similar architecture. Supporting heterogeneous hosts in distributed-memory settings is yet another important issue that has to be dealt with. Some MPI implementations have been presented for heterogeneous distributed-memory systems, such as MPICH-G2 [7] and OpenMPI [8]. These implementations adopt various strategies to address issues that concern network, processor and runtime environment heterogeneity [9]. Virtualization technology offers hardware independence and a straightforward solution for handling heterogeneous clusters. Setup of virtualization, however, requires root privileges, while recent studies have shown that virtualized multicore machines may suffer from significant performance degradation compared to native execution, both on distributed and shared-memory parallel computing environments [10, 11]. Similarly, Java solves the heterogeneity problem at the cost of

*This work was supported in part by the Greek Republic and the European Commission through the Artemisia SMECY project (grant 100230).

possible performance loss. Satin [12] is a Cilk-like extension to Java for divide-and-conquer applications targeting mostly wide-area grids. Satin requires a byte code rewrite and does not currently exploit the shared memory present in SMP/multi-core nodes.

Finally, GPUs present yet another architectural dimension. GPU-specific support for tasking has been provided by systems like StarPU [13] and GPUSs [14]. For GPU clusters, the most common programming approach is the hybrid MPI + GPU (CUDA or OPENCL) model. Research efforts that try to provide a unified programming platform include MGP [15], an implementation of OPENCL on clusters, Pleiad [16], a Java-based middleware that offers a shared-memory abstraction, and the extension of UPC for GPU clusters [17].

In this work we present TORC, a task-parallel library, for heterogeneous cluster computing platforms. TORC provides ease of programming and transparent load balancing to master-worker applications without requiring any interaction with the low-level message passing primitives. It also supports integration of the tasking model into traditional MPI programs, allowing one or more MPI processes to independently spawn tasks, and dynamic switching between SPMD and master-worker execution. To support heterogeneity on distributed memory systems we had to deal with design and implementation issues in TORC that cannot be directly addressed by a heterogeneous MPI library. Furthermore, we present an approach for dynamically offloading TORC tasks to the available GPUs of cluster nodes. Experimental results on several heterogeneous clusters demonstrate the ability of TORC to effectively exploit task parallelism and the seamless integration of MPI, OPENMP and OPENCL into a common programming and runtime environment.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the TORC tasking library. In Section 3 we discuss issues related to heterogeneity at cluster level, while in Section 4 we describe the support of GPU computing. Experimental evaluation is reported in Section 5. Finally, we conclude in Section 6.

2. Overview of TORC

TORC is a software library that provides C and Fortran interfaces for programming task-based parallel programs to be executed unaltered on both shared and distributed memory platforms. It takes advantage of and extends MPI: a single TORC application consists of multiple MPI processes running on cluster nodes, having one or more kernel-level POSIX threads that share the process memory. TORC implements a two-level thread architecture to support the tasking model. Each kernel thread is a worker that continuously dispatches and executes ready-to-run tasks; the latter are instantiated with user-level threads and submitted for execution to a set of ready queues. Due to the decoupling of tasks and execution vehicles, arbitrary nesting of tasks is inherently supported and any child task can become a master.

```
#include <math.h>
#include <torc.h>

void callback(int a, double *res) {
  printf("sqrt(%d)=%lf\n", a, *res);
}

void taskf(int a, double *res) {
  *res = sqrt(a);
}

void main(int argc, char *argv[]) {
  int i, double y[10];

  torc_init(argc, argv, MODE_MW);
  for (i=0; i<10; i++) {
    torc_task(-1, taskf, callback, 2,
      1, TORC_INT, CALL_BY_VAL,
      1, TORC_DOUBLE, CALL_BY_RES,
      i, &y[i]);
  }
  torc_waitall();
}
```

Figure 1. A TORC program with callback

TORC tasks have a parent-child relationship; they are associated with the process (parent node) they were created on and can be executed either locally or remotely. Internally, each task is represented with a data structure, called *task descriptor*, that encapsulates all the information of a task. In the task creation routine (torc_task()), the programmer specifies the task function and its arguments, providing the (MPI-like) type, size and intent attribute for each argument. Possible values of this attribute are CALL_BY_{VAL,RES,REF} used similarly to the IN, OUT and INOUT intent attributes of Fortran 90. For instance, CALL_BY_REF represents data that are sent with the task descriptor and returned as a result in the parent node's address space.

All data transfers in TORC are performed with explicit messaging: a kernel-level server thread in each MPI process is responsible for the remote queue management and the transparent to the user and asynchronous data movement. Besides MPI's send/recv interface, TORC is open to other forms of data management/fetching, such as MPI-2's one-side communications.

Fig. 1 presents a complete master-worker TORC application. We observe the complete absence of explicit messaging and the usage of three TORC routines (torc_{init,task,waitall}()), which initialize the library, create and join tasks respectively. After initialization, there is only one instance of the main routine that runs as the primary application task on the process with rank 0, while all other processes block waiting for work. When a task finishes, its result is transparently stored back to the appropriate place in the array and a user-supplied *callback* routine, which prints that result, is submitted for execution on the

parent process. The spawned tasks of the example code are distributed cyclically across the available workers. However, the programmer may specify the target process/worker in the task creation routine. As task stealing is inherently supported by TORC, the programmer has only to decide about the task distribution scheme.

An important feature of TORC is the support of dynamic switching between the master-worker and the SPMD execution model. MPI processes are usually launched and executed according to the SPMD execution model. The programmer explicitly specifies in torc_init() the desired execution model (`MODE_MW`, `MODE_SPMD`) that a parallel application will follow immediately after the library initialization. For the master-worker model, only one of the MPI processes executes the main routine and the rest of them become workers. At runtime, the application can switch its execution model to SPMD by spawning and distributing a dedicated task to each process. This approach enables the integration of existing SPMD MPI codes into a master-worker application. Furthermore, TORC allows the integration of the tasking model into legacy MPI applications [18].

On multi-core SMPs, TORC operates as a two-level threading library that seamlessly exploits intra-node task parallelism with multiple worker threads. Due to its design, TORC implements a tasking framework that integrates not only MPI but OPENMP as well. The intra-node parallelism of a task function can be expressed with OPENMP directives, in accordance to the hybrid MPI + OPENMP programming model. The OPENMP compiler, however, must support interoperability between OPENMP and independent POSIX threads. Moreover, caution is needed because the combination of TORC workers and OPENMP threads can easily oversubscribe the system.

3. Supporting heterogeneous clusters

Built on top of the POSIX and MPI interfaces, TORC is highly portable across operating systems, processors and networks. Among MPI implementations, Open MPI is the most active one and provides internal and transparent support for processor and network heterogeneity, allows for different operating systems and has additional attractive features, such as thread-safety [8, 9]. Therefore, the extension of TORC on heterogeneous clusters is based on Open MPI.

Supporting heterogeneity in TORC is a multidimensional problem. The processors of a heterogeneous cluster can differ in data type representation, endianness and memory alignment rules. Because Open MPI does not currently support conversion between data types of different sizes, we consider systems with the same (32-bit) data model, regardless of the operating system and processor type. In the following paragraphs, we describe how we addressed issues in the design and implementation of TORC that concern heterogeneity support. These issues stem from (a) the different memory alignment between the members of a data structure, as imposed by the processor architecture and the compiler, (b) the different ordering in data representation, known as machine endianness, and (c) the different virtual address space of MPI processes due to the software configuration of cluster nodes.

Memory Alignment. Due to memory alignment rules, it is possible for the data structure of the task descriptor to differ in size and/or ordering of structure members in two different cluster nodes. As the descriptor is sent from one node to another, whenever a task is submitted remotely, it is important that all cluster nodes have the same representation of the descriptor. For this reason, we place the structure members of the descriptor appropriately so as to prevent any alignment enforced by the compiler. In cases where this is not possible, we use explicit padding.

Machine Endianness. If two communicating nodes do not have the same endianness, a byte reordering is required for the transferred data. For any data type other than MPI_BYTE, this procedure is performed transparently by the heterogeneous MPI library for explicitly sent data. This feature allows for a straightforward extension of TORC on heterogeneous clusters, without code modifications for managing task arguments. For the task descriptor, however, we do not rely on MPI. So, instead of creating a struct datatype, we transfer the descriptor as a sequence of bytes; for communicating nodes of different endianness, we apply explicit byte swapping for every member of the descriptor at the receiver node.

Virtual Addresses. In TORC, a task corresponds to a function submitted for execution on a cluster node. The function is specified in the task creation routine, issued on the parent node. The function, however, can be located at a different virtual address on the target node. This may occur for systems with different software configurations (e.g. operating system) where the running MPI processes do not have identical address and memory spaces. In order to solve this problem, TORC uses a mechanism that assigns a globally unique number to each task function. Each MPI process maintains a table that matches a function's local address to the assigned number. This number, instead of the address, is stored in the task descriptor and when a worker is about to start executing a task, it performs the reverse matching procedure. The above mechanism introduces an extension in the TORC API. In particular, it requires the programmer to call a registration routine for every task function that may be executed remotely. This must be done by all MPI processes before calling the torc_init() function.

A similar issue appears in data broadcasting. Several master-worker applications may have global data that is initialized by the master and then broadcast to the workers. The torc_bcast() routine allows any task to broadcast global

data to all MPI processes. Data broadcasting avoids unnecessary data transfers and benefits applications that otherwise would need to send the data with every task. Suppose that the master modifies a global array and then broadcasts it to slaves. To perform this, the master must specify the address of the array in torc_bcast(). As this address may differ between nodes, a registration procedure is required for global data too. This case slightly differs from that of task functions, because we need to provide matchings for any valid position within the global memory segment. Therefore, the programmer must register not only the address but also the size of the memory segment, allowing broadcast of any part of the registered memory.

4. Supporting GPU computing

General purpose graphics processing units offer a cost-effective many-core architecture for high-performance computing. OpenCL [19] is the programming framework for developing kernels (functions) that target GPUs. Typically, the programmer copies kernel arguments to the device memory explicitly, executes the kernel with a number of work-items and finally copies the results back to host memory. As only one kernel can be running on most GPU devices, task parallelism can be exploited with multiple GPUs and processors. Moreover, a common practice is to combine MPI with OPENCL for programming GPU clusters.

By integrating GPU computing into TORC, we implement a programming and runtime environment for adaptive task parallelism on GPU clusters. The programmer can use the TORC API to spawn and distribute tasks to cluster nodes; the task function can either target a processor core or issue a kernel to a GPU. To enable the concurrent utilization of all available processing units (CPU, GPU) TORC must be initialized with at least two worker threads, because the OPENCL kernel execution corresponds to a blocking call that keeps a worker busy.

To improve the programmability and adaptivity features of TORC, we can take advantage of the fact that a CPU version of OPENCL kernels can (and should) be always available. Taking also into account that OPENCL's language is based on C99, in many cases a few modifications are adequate for compiling a kernel code for CPU execution. In addition, both the CPU function and the OPENCL kernel may share the same arguments.

According to the above, a TORC task can dynamically decide if it will be offloaded to the GPU or run on a CPU core, depending on whether an OPENCL kernel is already running. Furthermore, as task arguments are mapped to the kernel arguments and their usage has been specified with the intent attributes during task creation, the programmer can optionally let TORC handle all the required data transfers between the host and the GPU memory.

```
void taskf(<arguments>) {
  if (gpu_getaccess()) {
    f_gpu(<arguments>);
  }
  else {
    f_cpu(<arguments>);
  }
}

main(int argc, char **argv) {
  gpu_init();
  torc_init(argc, argv, MODE_MW);
  ...
  torc_create(-1, taskf, <arguments>);
  torc_waitall();
  ...
}
```

Figure 2. GPU computing and TORC

An outline of the approach is depicted in Fig. 2. The gpu_init() routine, which is called by all cluster nodes, prepares the GPU environment, builds the OpenCL kernel and creates the execution context. With gpu_getaccess(), a worker thread requests exclusive access to the GPU device; in case of success, it proceeds with the execution of the OPENCL kernel, otherwise it calls the CPU version of the kernel.

5. Performance experiments

5.1. Heterogeneous single-core platform

We conducted our experiments on a portion of our departmental heterogeneous cluster, composed of 20 single-processor nodes. The configuration includes three sets of homogeneous nodes: the first set is composed of 8 SUN Ultra20 workstations with an AMD Opteron 1.8GHz CPU and 1GB RAM, running Linux 2.6.18 with GNU gcc 4.1.2. The second set includes 4 SUN Ultra25 nodes (UltraSPARC-IIIi 1.34GHz CPU, 1GB RAM), while the third one has 8 Sun Blade100 workstations (UltraSPARC-IIe 502MHz CPU, 512MB RAM). All nodes are connected to the same Fast Ethernet switch. Both Ultra25 and Blade100 nodes run Solaris 9 with GNU gcc 3.4.6. The Open MPI version we used was 1.4.2.

We provide experimental results for three task-parallel applications: NAS EP, PMCMC and Mandelbrot. The loop-level parallelism of the Embarrassingly Parallel (EP) benchmark is statically scheduled to a number of tasks. The task results are transparently accumulated through a reduction (+) operation provided by TORC. We run EP for 2^{29} random numbers and 256 tasks. PMCMC implements a Markov Chain Monte Carlo algorithm [6]. Each task is assigned a seed and performs a large number of Markov Chain computations. For PMCMC, 1024 tasks are spawned, each performing 10^7 iterations. Finally, Mandelbrot (adapted from LAM/MPI) creates a single task for each image block and uses a callback routine to copy the processed block to the image region. The application processes an image of

Table 1. Execution Times and Power Weights (PWs) for the cluster nodes

	NAS EP		PMCMC		MAND	
	Time (s)	PW	Time (s)	PW	Time (s)	PW
Ultra20	509	1.00	1226	1.00	723	1.00
Ultra25	858	0.59	8070	0.15	6150	0.12
Blade100	1340	0.38	18800	0.07	11800	0.06

8192x8192 pixels with maximum 5000 iterations for each pixel and block size 128x128, spawning thus 4096 tasks.

To evaluate application performance on the heterogeneous cluster, we use a performance model introduced in [20]. The model is based on the Power Weight (PW) metric [21], which is defined as the amount of work a machine (node) can complete in unit time. Table 1 depicts the execution time of the applications on a single node of each cluster and the corresponding PWs. We observe that the Ultra20 cluster provides the lowest execution time and therefore, it is taken as reference in the calculation of PW for the rest of the clusters.

Figs. 3 to 5 depict the efficiency obtained for the three applications. We provide results for the three homogeneous sub-clusters (Ultra20, Ultra25 and Blade100), a heterogeneous cluster (Ultras) consisting of equal numbers of Ultra20 and Ultra25 nodes and, finally, the heterogeneous cluster (Coupled) of all systems. In the latter case, we have set the number of Ultra20 nodes to be twice the number of Ultra25 and Blade100 nodes. A summary of the obtained efficiency on the Coupled cluster is depicted in Fig. 6.

The experimental results demonstrate the effectiveness of TORC for heterogeneous computing: the obtained efficiency is higher than 90% for almost all configurations. For NAS EP (Fig. 3), the lower performance values for the Coupled cluster are attributed to load imbalance, imposed by the varying computational power of nodes and the relatively small number of tasks (256). In Fig. 4, we observe that PMCMC exhibits higher efficiency; this is mostly attributed to its larger number of tasks. Mandelbrot scales well too, despite the inherent load imbalance of the application and the inclusion of all callbacks completion in the measured execution time. Other factors that may affect parallel performance and limit efficiency are the communication overheads because of the low-cost communication channel and the contention between the worker and the server thread within each process. The task stealing mechanism of TORC is utilized in all the experiments. If a homogeneous cluster is used, tasks are distributed cyclically across the nodes. For the heterogeneous configurations, we adopt a hybrid task distribution scheme: the primary application task always runs on a node of the most powerful cluster and spawned tasks are submitted only to nodes of that cluster too. Therefore, low-performance nodes only issue and never serve task stealing requests. To reduce net-

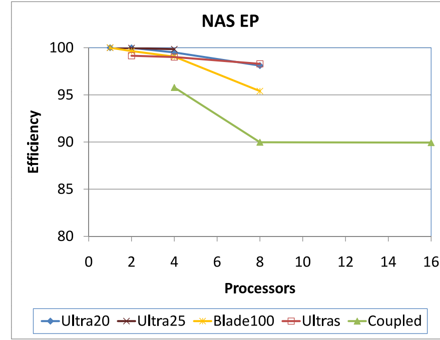

Figure 3. Efficiency of NAS EP

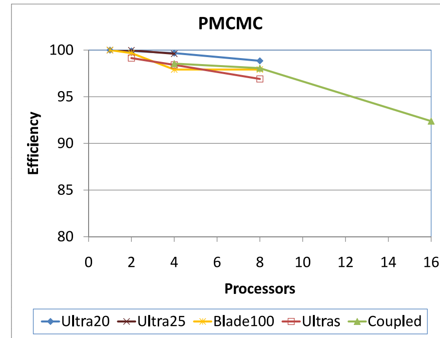

Figure 4. Efficiency of PMCMC

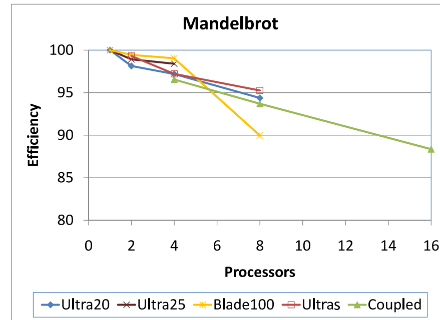

Figure 5. Efficiency of Mandelbrot

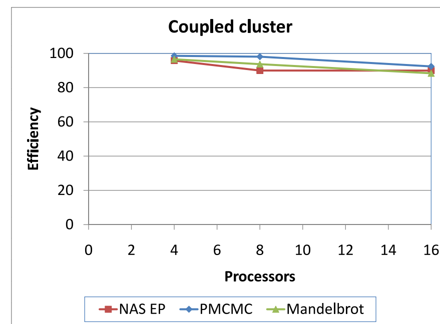

Figure 6. Efficiency on the Coupled cluster

Table 2. Execution Times and PWs for the processing cores of the Linux cluster nodes

	NAS EP		PMCMC		MAND	
	Time (s)	PW	Time (s)	PW	Time (s)	PW
HP6000Q	1557	1.00	1347	0.91	4113	0.97
HP6000D	1666	0.93	1567	0.78	4402	0.91
Ultra20D	2839	0.55	1456	0.84	4004	1.00
Ultra20S	4096	0.38	1226	1.00	5818	0.69

work traffic, an idle worker first searches for task at the node where it had previously found one. An in-depth study and evaluation of task scheduling is left for future work.

5.2. Heterogeneous Linux cluster

In order to demonstrate the scalability of TORC on a larger number of cores, we utilize a heterogeneous Linux cluster that consists of the following systems (in parenthesis the codename used for convenience, the last letter denoting the number of cores):

- 13 HP6000 Pro PCs with an Intel Core 2 Quad Q9400 2.6GHz CPU and 2GB RAM (HP6000Q).
- 14 HP6000 Pro PCs with a Dual-core Intel Celeron E3300 2.5GHz CPU and 3GB RAM (HP6000D).
- 6 SUN Ultra20 workstations with an Dual-Core AMD Opteron 2.6GHz CPU and 1GB RAM (Ultra20D).
- 8 SUN Ultra20 workstations with an AMD Opteron 1.8GHz CPU and 1GB RAM (Ultra20S).

The HP6000 PCs are running Linux 2.6.32 with GNU gcc 4.4.5, while the Ultra20 workstations Linux 2.6.18 with GNU gcc 4.1.2. We conducted experiments with the same set of applications as before but for larger problem size for EP and Mandelbrot. In particular, we run EP for 2^{32} random numbers and 1024 tasks and Mandelbrot for 16384x16384 image size, block size 128x128 and 16384 total tasks. The execution times of the three applications on a single processing core of each node and the corresponding Power Weights are depicted in Table 2.

In Table 3 we show representative experimental results on two Linux sub-clusters: the first cluster (HP) consists of 13 HP6000Q and 14 HP6000D nodes resulting in 27 nodes and 80 processing cores in total, while the second one (HP-Sun) uses 8 HP6000Q, 8 HP6000D, 6 Ultra20D and 8 Ultra20S nodes (30 nodes, 68 processing cores). A single MPI process is deployed on each cluster node, with the number of worker thread set equal to the number of processing cores of that node. We observe that the results accord with those attained on the heterogeneous single-core cluster, with the efficiency keeping above 79%, despite the low-performance interconnection network, a small serial fraction in the application for spawning and joining parallelism and the load imbalance that occurs when the number of tasks is not evenly divided by the number of processors.

Table 3. Performance on the Linux cluster

Application	Cluster	Cores	Time (s)	Speedup	Efficiency
EP	HP	80	22.48	69.28	88.77%
EP	HP-Sun	68	33.40	46.62	83.50%
PMCMC	HP	80	21.69	56.52	81.73%
PMCMC	HP-Sun	68	25.68	47.74	79.14%
MAND	HP	80	62.84	63.72	83.93%
MAND	HP-Sun	68	80.85	49.52	80.03%

5.3. Heterogeneous GPU cluster

We demonstrate the functionality of TORC on a GPU cluster with a nonlinear global optimization application that uses the algorithm of Hooke and Jeeves. The code was adapted from Netlib [22], modified to use floats instead of doubles in order to run as an OPENCL kernel on the NVIDIA GPUs we have access to. The Hooke application applies the algorithm to a number of randomly chosen multidimensional points, searching for the minimum of the Rastrigin function.

We performed our experiments on a dual-node multicore GPU cluster. The first node (called i7-930) is equipped with an Intel Core i7 930 Quad-Core processor at 2.67GHz, 4GB RAM and an NVIDIA Geforce GT 220 GPU (6x8 cores, 1GB memory). The second node (i7-920) has a similar processor (Intel Core i7 930 at 2.8GHz), 4GB RAM and an NVIDIA GeForce 9400 GT GPU (2x8 cores, 1GB memory). Both nodes are running Linux 2.6.32 with GNU gcc 4.4.3 and the NVIDIA CUDA toolkit 4.0.1 installed. We apply the Hooke algorithm on 256K 8-dimensional points, processing them in chunks of 1024 points. Our goal is to demonstrate our support of task parallelism on GPU clusters rather than to provide a highly-optimized GPU version of the application.

An outline of the parallelization strategy is depicted in Fig. 7. The 256 tasks are distributed cyclically to the TORC workers of the two MPI processes, with task stealing enabled. If a task succeeds to get access to the GPU it runs the OPENCL kernel, otherwise it proceeds with the CPU version of the Hooke algorithm. The execution of the kernel is performed with the gpu_runkernel() function, which copies the input task arguments to the GPU memory, issues the kernel with global work-items equal to the number of points (1024) and work-group size equal to 32 and finally copies back the results.

Table 4 shows the execution time of the application on a single core of the i7 processors and the two NVIDIA GPUs and the corresponding PWs. For the particular problem parameters (number of points and chunk size), the GT 220 GPU delivers the best performance and therefore PWs are computed against the execution time on that device.

Figures 8–10 illustrate the execution time of the Hooke application on three different hardware configurations of the

```
#define POINTS (256*1024)
#define VARS   (8)

void hooke_drv(float data[], float res[], int npts) {
  if (gpu_getaccess()) {
    gpu_runkernel(npts);
  }
  else {
    hooke_cpu(data, res, npts);
  }
}

main(int argc, char **argv) {
  gpu_init();
  torc_init(argc, argv, MODE_MS);
  ...
  stride = 1024;
  for (ipoint = 0; ipoint < POINTS; ipoint+=stride) {
    torc_create(-1, hooke_drv, 3,
        stride*VARS, TORC_FLOAT, CALL_BY_VAL,
        stride*VARS, TORC_FLOAT, CALL_BY_RES,
        1, TORC_INT, CALL_BY_VAL,
        &data[ipoint*VARS], &res[ipoint*VARS], stride);
    }
    torc_waitall();
    ...
}
```

Figure 7. Exploitation of CPU/GPU task parallelism in the global optimization application

GPU cluster. In the first experiment (Fig. 8), we evaluate the parallel performance of the application on the multicore systems. We observe that the application scales linearly when 4 TORC workers are used, while the maximum speedup is 5.48, obtained for 8 workers that fully utilize the 4 cores/8 threads of the Quad-core i7 processors.

The second experiment demonstrates the support of TORC for concurrent GPU/CPU computing: one of the TORC workers always offloads tasks to the GPU while the rest of them run the CPU version of the Hooke algorithm. In Fig. 9, we observe that the execution time of the Hooke application is significantly lower compared to that of pure multicore execution and, moreover, decreases with the number of workers. On 4 cores (workers), the PW-based efficiency for both systems is approximately 95%.

Fig. 10 shows the performance of TORC on the dual-node cluster, using an equal number of TORC workers at each node. We provide results running Hooke only on the CPU cores and utilizing the available GPU and textsccpu cores at the same time. The latter case demonstrates the full functionality of TORC: a single application binary is able to dynamically adapt its execution to the underlying parallel hardware. For example, on 8 cores (4 workers per node), the parallel efficiency is 91% and the 256 tasks are executed as follows: 141 on the i7-930 node (41 on the 3 cores and 100 on the GT 220 GPU) and 115 on the i7-920 node (40 on the 3 cores and 75 on the 9400 GT GPU).

The smallest application execution time for each experiment is depicted in Table 5, verifying the effective utilization of all the available processing units of the multi-core GPU cluster.

Table 4. Execution Times and PWs for the processing units of the GPU cluster

	Time (s)	PW
i7-930 core	112.76	0.15
GT 220 GPU	16.36	1.00
i7-920 core	118.25	0.14
9400 GT GPU	21.77	0.75

Table 5. Minimum execution times of Hooke

System	Processing Units	Time (s)
i7-930	8 threads	20.57
i7-930 + GT 220	8 threads / 1 GPU	9.67
i7-920	8 threads	21.71
i7-920 + 9400 GT	8 threads / 1 GPU	11.92
i7-930 + i7-920	16 threads	10.89
i7-930 + GT 220 + i7-920 + 9400 GT	16 threads / 2 GPUs	5.62

Lastly, we experiment with the execution of intra-node parallelism with OPENMP, as the GNU gcc compiler supports the required interoperability with POSIX threads. Specifically, the 1024 loop iterations of a CPU-based Hooke algorithm, included in a single TORC task, are distributed to a team of OPENMP threads with a `parallel for` pragma. To avoid oversubscription, the application runs with 2 TORC workers; one interacts with the GPU while the other becomes the master of the team that executes the OPENMP parallel version of the Hooke algorithm. For the particular experiment, the difference in the performance between the two approaches (exclusive TORC workers vs OPENMP threads) is negligible. For instance, the execution time of the application with 2 TORC workers and 7 OPENMP threads on the i7-930 + GT 220 configuration is 9.72 seconds, which is very close to the 9.67 seconds when OPENMP is disabled and 8 TORC workers are used (second entry in Table 5).

6. Discussion

We presented the implementation of TORC, an infrastructure for programming and executing task-based MPI parallelism, on clusters of heterogeneous nodes and GPUs. Several issues concerning heterogeneity support in the runtime library were discussed. An adaptive scheme for CPU/GPU computing was also introduced. We evaluated TORC on a variety of parallel computing platforms, including homogeneous and heterogeneous clusters, equipped with multicore processors and GPU devices. The experimental results demonstrate the successful and efficient support of adaptive task parallelism on such platforms, within a common programming and runtime infrastructure. Our future plans include the study of load balancing policies and the support of fault-tolerance.

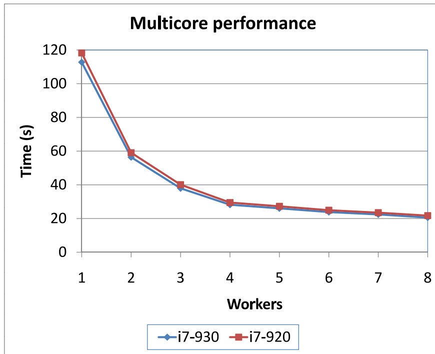

Figure 8. CPU performance of Hooke

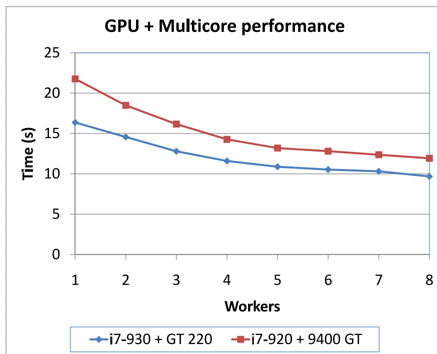

Figure 9. CPU/GPU performance of Hooke

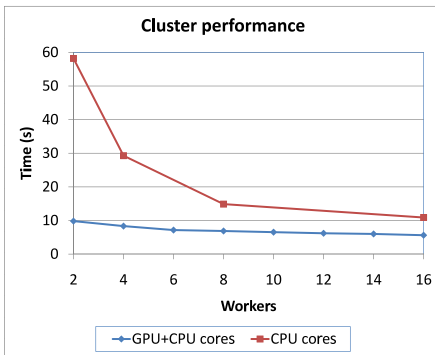

Figure 10. Cluster performance of Hooke

References

[1] R.D. Blumofe et al. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1), 1996.

[2] OpenMP Architecture Review Board, *OpenMP Application Program Interface, V.3.0*, May 2008

[3] I. Foster et al. A library-based approach to task parallelism in a data-parallel language. *J. Parallel Distrib. Comput.*, 45(2), pp. 148-158, 1997.

[4] J. Hippold, G. Runger. Task pool teams: A hybrid programming environment for irregular algorithms on SMP clusters. *Concurr. Comput.: Pract. Exp.*, 18(12) pp. 1575-1594, 2006.

[5] T. Rauber and G. Runger. Tlib - A library to support programming with hierarchical multi-processor tasks. *J. Parallel Distrib. Comput.*, 65(3), pp. 347-360, 2005.

[6] ADLB library, URL: http://www.cs.mtsu.edu/∼rbutler/adlb/

[7] N. Karonis, B. Tonnen, and I. Foster. MPICH-G2: a grid-enabled implementation of the message passing interface. *J. Parallel Distr. Comput.*, 63(5), pp. 551-563, 2003.

[8] R.L. Graham et al. Open MPI: A High-Performance, Heterogeneous MPI. In *5th Intl. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, 2006.

[9] Y-H. Jiang, Q.L. Zhao, Y.T. Lu, and X.J. Yang. Heterogeneity issues and supports in MPI implementations: An overview. In *8th Intl. Conf. on Grid and Cooperative Computing*, Lanzhou, Gansu, China, 2009.

[10] J. Han et al. The effect of multi-core on HPC applications in virtualized systems. In *5th Workshop on Virtualization and High-Performance Cloud Comput.*, Naples, Italy, 2010.

[11] J. Tao, K. Furlinger, and H. Marten. Performance evaluation of OpenMP applications on virtualized multicore machines. In *7th Intl. Workshop on OpenMP (IWOMP 2011)*, Chicago, IL, USA, 2011.

[12] R. van Nieupoort, J. Massen, T. Kielmann, H.E. Bal. Satin: simple and efficient java-based Grid programming. In *Workshop on Adaptive Grid Midlleware (AGridM 2003)*, Antibes Juan-les-Pins, France, 2003.

[13] C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2), pp. 187-198, 2011.

[14] E. Ayguade et al. An extension of the StarSs programming model for platforms with multiple GPUs. In *15th Intl. Euro-Par Conference*, Delft, The Netherlands, 2009.

[15] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE International Conf. on Cluster Computing Workshops and Posters*, Heraklion, Crete, Greece, 2010.

[16] K. I. Karantasis and E. D. Polychronoulos. Programming GPU clusters with shared memory abstraction in software. In *19th Intl. Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, Ayia Napa, Cyprus, 2011.

[17] L. Chen et al., Unified Parallel C for GPU clusters: Language extensions and compiler implementation. In *23rd Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2010)*, Houston, TX, USA, 2010.

[18] P.E. Hadjidoukas, C. Voglis, V.V. Dimakopoulos, I.E. Lagaris, and D.G. Papageorgiou. High-performance numerical optimization on multicore clusters. In *17th Intl. Euro-Par Conf.*, Bordeaux, France, 2011.

[19] Khronos Group, The OpenCL specification, 2011.

[20] J. Al-Jaroodi et al. Modeling parallel applications performance on heterogeneous systems. In *Intl. Parallel and Distr. Processing Symp. (IPDPS'03)*, Nice, France, 2003.

[21] X. Zhang, and Y. Yan. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *7th IEEE Symposium on Parallel and Distr. Processing (SPDPS'95)*, San Antonio, TX, USA, 1995.

[22] NetLib Repository, URL: http://www.netlib.org/opt/