A Runtime Library for Platform-Independent Task Parallelism

Panagiotis E. Hadjidoukas
Evangelos Lappas
Vassilios V. Dimakopoulos

Department of Computer Science
UNIVERSITY OF IOANNINA
Ioannina, Greece
Programming Environment

- Execution model: an application consists of multiple MPI processes with private memory, running on cluster nodes
- Multithreading is used to exploit the multi-core configuration of nodes
  - Each MPI process consists of 1 or more worker threads
- Task: (remote) execution of a function on a set of data that are passed as arguments (RPC-like)
  - Tasks are executed asynchronously and in any order
  - No data dependencies or point-to-point communication
  - Parent-child relationship, arbitrarily nested
  - Instantiated with lightweight user-level threads
- Decoupling of tasks and execution vehicles
  - Nested parallelism in a seamless way
  - Any task can become a master and spawn new tasks
Primary API calls

- The programmer has to designate which program functions can be used as tasks and be executed on remote nodes
  - 2-3 API functions are enough (C, Fortran)

- Library initialization: `torc_init()`

- Task creation: `torc_task()`
  - Task function, number of arguments, argument list
  - An MPI-like description for each argument: size, data type and an intent attribute `CALL_BY_{VAL, RES, REF}` (IN, OUT, INOUT)
  - task + data movement performed transparently by TORC

- Task joining: `torc_waitall()`
  - The calling (master) task suspends itself until all children have finished
  - The underlying worker thread does not block
#include <math.h>
#include <torc.h>

void taskf(int x, double *y) {
    *y = sqrt((double)x);
}

int main(int argc, char *argv[]) {
    int i;
    double results[100];
    torc_init(argc, argv, MODE_MW);
    for (i=0; i<100; i++) {
        torc_task(-1, taskf, NULL, 2,
                   1, TORC_INT, CALL_BY_VAL,
                   1, TORC_DOUBLE, CALL_BY_RES,
                   i, &results[i]);
    }  
    torc_waitall();

    /* print results */
    return 0;
}
Built on top of MPI and POSIX

Architecture:

- One of more MPI processes
- POSIX kernel threads (workers) that execute tasks
- A two-level thread model is implemented (tasks are executed as user-level threads)
- Server thread (per-process) for task and data management
- Private and public worker-specific and node-specific queues
General Architecture

- **NODE #0**
  - Private & Public Local Queues
  - Server thread of node #0

- **NODE #N**
  - Worker threads (virtual processors)

**Communication Subsystem (MPI)**
Additional Programming Features (I)

- **Data broadcasting**: global data initialized by the master and then broadcast to the workers.
- **Callbacks**: allow for asynchronous execution of post-processing code on the process where the parent task runs on.
- **Reductions**: useful when each child task computes a partial result. Supported for both scalar variables and arrays.
- **Detached tasks**: the parent is not able to wait on them. Task management is left up to the programmer.
- **Tied tasks**: run on the process initially submitted for execution.
Additional Programming Features (II)

- **Mixed Mode Programming**
  - At runtime the application can dynamically switch between SPMD and master worker execution model
  - The programmer explicitly specifies in `torc_init()` the desired execution model (`MODE_MW` or `MODE_SPMD`)
  - Takes advantage of common scientific SPMD libraries
  - Native MPI applications enriched with tasking

- **Integration with OpenMP**
  - Expressing intra-node parallelism of task functions with OpenMP
  - TORC workers + OpenMP threads: possible oversubscription of processor cores
  - Solution: use TORC to implement an OpenMP runtime library, map OpenMP threads to TORC tasks
  - TORC-based threading layer into the OMPi OpenMP compiler
Task Scheduling

- Programmer is provided the flexibility to specify the target process/worker based on a decided distribution scheme
  - First argument of `torc_task()` is the target queue
- TORC inherently supports work stealing
- Scheduling loop of worker threads
  - Activated when the current task blocks or finishes
- Task selection and execution
  - First extract the task at the front of the local ready queue
  - **Intra-node stealing**: try to steal from the rest of the intra-node queues (shared memory)
  - **Inter-node stealing**: issue requests for work to remote nodes (message passing)
  - Stealing is always performed from the back of the ready queues
Supporting Heterogeneous Clusters

- Torc extension to heterogeneous clusters is based on Open MPI
- Open MPI provides internal and transparent support for processor and network heterogeneity
  - Different operating systems
  - Thread-safety
- Issues addressed in the design and implementation of TORC
  - Different memory alignment between the members of a data structure, as imposed by the processor architecture and the compiler
  - Different ordering in data representation (machine endianness)
  - Different virtual address space of mpi processes due to software configuration of cluster nodes
Supporting GPU computing (I)

- **OpenCL** is a programming framework for developing **kernels** (functions) that target GPUs.
  - The programmer copies kernel arguments to the device memory
  - Executes the kernel with a number of work-items
  - Finally copies the results back to host memory

- **Torc** integrates GPU computing

- Torc must be initialized with at least two worker threads
  - OpenCL kernel execution is a **blocking call**

- Two versions for every kernel: Cpu and GPU version

- Torc dynamically decide if the kernel will be offloaded to GPU or run on a CPU core

- Torc is able to handle all the required data transfers
  - Based on the description of the argument list
Supporting GPU computing (II)

- **gpu_init()**: called by cluster nodes preparing the GPU environment
- **gpu_getaccess()**: a worker request exclusive access to gpu
- **f_gpu**: gpu version of task
- **f_cpu**: cpu version of task

```c
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();
    ...}
```
Experiments

- Heterogeneous single-core platform
- Heterogeneous Linux Cluster
- Heterogeneous GPU cluster
Nodes

- **1<sup>st</sup> set**: 8 SUN Ultra20 AMD Opteron 1.8GHz, 1GB RAM, Linux 2.6.18, gcc 4.1.2
- **2<sup>nd</sup> set**: 4 SUN Ultra25 UltraSPARC-IIIi 1.34GHz, 1GB RAM, Solaris 9 gcc 3.4.6
- **3<sup>nd</sup> set**: 8 SUN Blade100 UltraSPARC-IIe 502MHz, 512MB RAM, Solaris 9 gcc 3.4.6
- Fast Ethernet
- OpenMPI 1.4.2

Three task-parallel applications:

- **NAS EP**: $2^{29}$ Random numbers, created 256 tasks
- **PMCMC**: 1024 tasks each $10^7$ iterations
- **Mandelbrot**: image size 8192x8192, 5000 iterations for each pixel, block size 128x128 (resulting 4096 tasks)
Heterogeneous single-core platform (II)

**NAS EP**

- Efficiency vs. number of processors for different platforms:
  - Ultra20
  - Ultra25
  - Blade100
  - Ultras
  - Coupled

**PMCMC**

- Efficiency vs. number of processors for different platforms:
  - Ultra20
  - Ultra25
  - Blade100
  - Ultras
  - Coupled

**Mandelbrot**

- Efficiency vs. number of processors for different platforms:
  - Ultra20
  - Ultra25
  - Blade100
  - Ultras
  - Coupled

**Coupled cluster**

- Efficiency vs. number of processors for different applications:
  - NAS EP
  - PMCMC
  - Mandelbrot

[Source: PDP 2012, Garching]
Nodes

- 13 HP6000 Pro PCs, Intel Core 2 Quad Q9400 2.6GHz, 2GB RAM (HP6000Q) running Linux 2.6.32, GNU gcc 4.4.5
- 14 HP6000 Pro PCs, Dual-core Intel Celeron E3300 2.5GHz, 3GB RAM (HP6000D) running Linux 2.6.32, GNU gcc 4.4.5
- 6 SUN Ultra20 workstations, Dual-Core AMD Opteron 2.6GHz, 1GB RAM (Ultra20D) running Linux 2.6.18, GNU gcc 4.1.2
- 8 SUN Ultra20 workstations, AMD Opteron 1.8GHz, 1GB RAM (Ultra20S) running Linux 2.6.18, GNU gcc 4.1.2

Three task-parallel applications:

- NAS EP: $2^{32}$ Random numbers, created 1024 tasks
- PMCMC: 1024 tasks each $10^7$ iterations
- Mandelbrot: image size 16384x16384, 5000 iterations for each pixel, block size 128x128 (resulting 16384 tasks)
Heterogeneous Linux Cluster (II)

- Two linux sub-clusters
  - HP 27 nodes 80 cores
    - 13 HP6000Q
    - 14 HP6000D
  - HP-sun 30 nodes, 68 cores
    - 8 HP6000Q
    - 8 HP6000D
    - 6 Ultra20D
    - 8 Ultra20S

```
<table>
<thead>
<tr>
<th>Application</th>
<th>Cluster</th>
<th>Cores</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>HP</td>
<td>80</td>
<td>22.48</td>
<td>69.28</td>
<td>88.77%</td>
</tr>
<tr>
<td>EP</td>
<td>HP-Sun</td>
<td>68</td>
<td>33.40</td>
<td>46.62</td>
<td>83.50%</td>
</tr>
<tr>
<td>PMCMC</td>
<td>HP</td>
<td>80</td>
<td>21.69</td>
<td>56.52</td>
<td>81.73%</td>
</tr>
<tr>
<td>PMCMC</td>
<td>HP-Sun</td>
<td>68</td>
<td>25.68</td>
<td>47.74</td>
<td>79.14%</td>
</tr>
<tr>
<td>MAND</td>
<td>HP</td>
<td>80</td>
<td>62.84</td>
<td>63.72</td>
<td>83.93%</td>
</tr>
<tr>
<td>MAND</td>
<td>HP-Sun</td>
<td>68</td>
<td>80.85</td>
<td>49.52</td>
<td>80.03%</td>
</tr>
</tbody>
</table>
```
Heterogeneous GPU cluster (I)

- Non-linear Global Optimization application
  - Hooke and Jeeves algorithm
  - Modified to floats instead of doubles
  - Multistart: randomly chosen multidimensional point, searching for the minimum of the Rastrigin function

- 256K 8-d points, in chunks of 1024 points (256 tasks)
  - Initialized by the master
  - Distributed cyclically

- Nodes
  - Intel Core i7 930 Quad-Core 2.67GHz, 4GB RAM and an NVIDIA GeForce GT 220 GPU (6x8 cores, 1GB memory). (i7-930)
  - Intel Core i7 930 Quad-Core 2.8GHz, 4GB RAM and an NVIDIA GeForce 9400 GT GPU (2x8 cores, 1GB memory). (i7-920)
  - Both running Linux 2.6.32, GNU gcc 4.3.3, NVIDIA CUDA toolkit 4.0.1
Heterogeneous GPU cluster (II)

Multicore performance

Time (s)

Workers

CPU cores

GPU cores

GPU + Multicore performance

Time (s)

Workers

CPU cores

GPU cores

Cluster performance

Time (s)

Workers

CPU cores

GPU cores

<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>PW</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930 core</td>
<td>112.76</td>
<td>0.15</td>
</tr>
<tr>
<td>GT 220 GPU</td>
<td>16.36</td>
<td>1.00</td>
</tr>
<tr>
<td>i7-920 core</td>
<td>118.25</td>
<td>0.14</td>
</tr>
<tr>
<td>9400 GT GPU</td>
<td>21.77</td>
<td>0.75</td>
</tr>
</tbody>
</table>
Summary

- TORC: runtime infrastructure for task-based MPI parallelism on clusters of heterogeneous nodes and GPUs
- Several issues concerning heterogeneity support were discussed
- An adaptive scheme for CPU/GPU computing was introduced
- Experimental evaluation demonstrates the efficient support of task parallelism on a variety of computing platforms
  - homogeneous/heterogeneous clusters
  - with multicore processors and GPU devices
- Ongoing work:
  - Study of load balancing policies
  - Fault-tolerance
Thank you for your attention!
Programming Example - Callback

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

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

void taskf(int x, double *y) {
    *y = sqrt((double)x);
}

int main(int argc, char *argv[]) {
    int i; double results[100];
    torc_init(argc, argv, MODE_MW);
    for (i=0; i<100; i++) {
        torc_task(-1, taskf, callback, 2,
                    1, TORC_INT, CALL_BY_VAL,
                    1, TORC_DOUBLE, CALL_BY_RES,
                    i, &results[i]);
    }
    torc_waitall();
    return 0;
}
```
Supporting Heterogeneous Clusters
How Issues Addressed

- **Memory Alignment**: Data structure of the task descriptor may differ in size and/or ordering of structure members in two different cluster nodes

- **Machine Endianess**: Data reordering is required
  - For any datatype other than MPI_BYTE the procedure is performed transparently by the heterogeneous MPI library
  - Task descriptor reordering does not rely on MPI. Torc applies explicit byte swapping for every member of data structure

- **Virtual Addresses**: A task corresponds to a function that can be located on a different virtual address space on target node
  - Torc assigns a globally unique number to each task function
  - Each node maintains a table that provides a mapping
  - A registration function has to be called before `torc_init()` called

- **Data broadcasting**: uses global memory segments.
  - Matching is provided for every valid position within the global memory segment
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();
    ...
}