Argobots: Lightweight Threading/Tasking Framework

Cyril Bordage (UIUC) and Huiwei Lu (ANL)

ANL: Pavan Balaji, Pete Beckman, Sangmin Seo, Marc Snir
UIUC: Laxmikant V. Kale, Jonathan Lifflander, Yanhua Sun
UTK: Georges Bosilca, Damien Genet, Thomas Herault
PNNL: Sriram Krishnamoorthy

https://collab.cels.anl.gov/display/ARGOBOTS

Charm++ Workshop 2015
Motivation: extreme scale computing

• Massive on-node parallelism
  – Need for supporting fine-grained asynchronous work units
  – Lightweight threading and tasking methods

• Need for interoperability among multiple programming models in a single application
  – Different runtime strategies
  – Domain Specific Language
  – Better productivity and performance

• Part of the Argo project
Argobots

- Lightweight Low-level Threading/Tasking Framework
- Massive parallelism
  - Execution Streams guarantee progress
  - Work Units execute to completion
- Offers low-level abstraction of threads and tasks
Outline

• Execution model
• Interface
• Evaluation
• Programming models
Execution Model
Semantic

• Execution Stream (ES)
  - Responsible for the execution of work units
  - Corresponds to one hardware resource
  - No preemption between work units

• Work Unit
  - Associated with function call and executes to completion
  - Two types
    • User Level Thread (ULT): has its own stack
      - Can yield and synchronize
    • Tasklet: no stack
Design

• Pools: set of ready work units
  - Private or Shared

• Schedulers
  - Different strategies
  - Stackable
Interface
Basic Operations

• Explicit creation of execution streams
• Creation of ULTs and tasklets (needs the target pool)
• Yield and yield to
• Migration
• Synchronization
  – Termination (join)
  – Future, mutex, barrier, condition variable...
Hello world example

```c
int main(int argc, char *argv[]) {
    int num_xstreams = 4;
    ABT_xstream xstreams[num_xstreams];
    ABT_pool pools[num_xstreams];

    /* Initialization */
    ABT_init(argc, argv);
    ABT_xstream_self(&xstreams[0]);
    for (int i = 1; i < num_xstreams; i++)
        ABT_xstream_create(ABT_SCHED_NULL, &xstreams[i]);

    /* Get the first pool associated with each ES */
    for (int i = 0; i < num_xstreams; i++)
        ABT_xstream_get_main_pools(xstreams[i], 1, &pools[i]);

    /* Create tasks */
    for (int i = 0; i < num_xstreams; i++)
        ABT_task_create(pools[i], task_hello, NULL, NULL);

    /* Switch to other work units */
    ABT_thread_yield();

    /* Finalize */
    for (int i = 1; i < num_xstreams; i++) {
        ABT_xstream_join(xstreams[i]);
        ABT_xstream_free(&xstreams[i]);
    }
    ABT_finalize();
    return 0;
}

/* Task function */
void task_hello(void *arg) {
    printf("Hello, world!\n");
}
```
Evaluation
Microbenchmarks: create and join

Tests on 2 Intel Xeon E5-2699 (2.3GHz, 36 cores, 72 threads)
Microbenchmarks: yield

Tests on 2 Intel Xeon E5-2699 (2.3GHz, 36 cores, 72 threads)
Microbenchmarks: different work units

Tests on 2 Intel Xeon E5-2699 (2.3GHz, 36 cores, 72 threads)
LeanMD with Argobots: speedup

Speedup on Intel Xeon Phi

Argobots

Ideal

Number of Execution Streams

Speedup

1 2 4 8 16 32 64 128 240

1 10 100 1000
Programming models
**Charm++ with Argobots: integration**

**Goal:**
1. Test the completeness and performance of Argobots APIs with Charm++ programming model
2. For Charm++ programs, interoperate with programs written in other models (MPI, Cilk, etc.)

**Approach:**
1. Create an Execution Stream for each Charm++ Instance
2. All Charm++ messages are enqueued as tasks into the Argobots pool
3. Argobots schedules the messages in pool
Charm++ with Argobots: preliminary test

Test of LeanMD
MPI+Argobots: Data Movement in Distributed Memory Systems with Lightweight Threads

- Hybrid runtime of MPI and Argobots
  - Lightweight and dynamically adapt to the hardware resources
- Two level of threads provide an explicit semantic for concurrency
  - Execution Stream (ES) provides concurrent execution
  - User Level Thread (ULT) provides fast context switch
- Overlap communication with computation using ULT
  - Helps turn a MPI blocking call to a nonblocking one
  - ULT is lightweight because no lock is needed between two ULTs in the same kernel thread
MPI+Argobots: HPCG App

• High Performance Conjugate Gradient (HPCG)
  – Solves $Ax=b$, large and sparse matrix.

• Hiding Global Collective Communication
  – Overlap communication and computation between iterations
  – Fork a ULT to do ult_ddot and join in the next iteration

• Hiding Neighborhood Communication
  – For each neighbor, fork a ULT to do halo exchange and a small part of SpMV (communication)
  – Main ULT computes local spmv (computation)

HPCG

for k = [1: max_iter]:
  MG(A, r, z);
  if k > 1:
    ulti_join (thread);
    if (normr <= tolerance) break;
  ....
  ulti_fork(ulti_ddot, &param, &thread)

SpMV

for each neighbor:
  ulti_fork(es, ulti_spmv, &t[i]);
  for i in [0: nRows]:
    ulti_yield();
    for each j in row i:
      y[i] += val[j] * x[idx[j]];
    for each neighbor i:
      ulti_join(t[i]);
On 2,048 cores, HPCG using MPI+Argobots shows performance improvement of 12.6% over MPI-only version, or 26.9% over MPI+Pthreads version.

- As core number increases, the benefit of communication hiding begins to reveal.
Cilk with Argobots: Dynamic Task Splicing

- Cilk built on Argobots
  - Each Cilk worker (previously pthread) is now an Argobots Execution Stream
  - The Cilk work stealing scheduler runs in an Argobots ULT

- Fuse together multiple spawn trees to improve locality
  - Distinct spawn trees require their own stack
    - Create a new Argobots ULT for each spawn tree root to fuse
    - Modify Cilk compiler to generate Argobots ULT function wrapper
  - A steal may require stealing from multiple ULTs (or spawn trees)

- Motivation
  - Code may be in different libraries, manual fusion not possible
  - Dependencies between phases may inhibit manual fusion

- Implemented simple application
  - Matrix Vector Product and Transpose (PolyBench: \texttt{mtv})
    - Two concurrent phases that read entire matrix A
    - Cache locality can be improved if they are fused
PTGE (Pluggable Task Graph Engine) with Argobots

• Motivation
  – A decent scale application encompasses thousands of tasks
  – It is critical to minimize the number of tasks

• PTGE
  – Data centric task definition
  – Minimize the number of tasks by defining symbolic dependencies between sets of tasks
    • for(i = 0; i < SIZE; i++) task(A[i])
      become: task([A[i], i in [0 .. SIZE-1]])
  – Integrate with the communication engine to allow for dynamic creation of incoming data, matched with expected input for task(s)

• Hierarchical task scheduling
  – With the PTGE approach the first stage of scheduling is creating the task
    • Many possible strategies: first data available, all local data available, I/O prediction cost ...
  – Once the task is created, the graph of tasks is decorated with cost information for the upper level scheduling
  – Enable task stealing between hierarchies (both ways)
Conclusion
Conclusion

• Argobots is a lightweight threading/task Infrastructure
• Argobots is highly optimized and has good scalability on many-core processors
• Argobots can be easily integrated with different programming models (MPI, Charm++, Cilk, PTGE)

• Ongoing works
  – More applications with different programming models
  – Interoperability between different models

Download: https://collab.cels.anl.gov/display/ARGOBOTS
Thanks!
Questions?