Integrating Futures as Async Provider into PaRSEC Runtime System

Yu Pei
(ypei2@vols.utk.edu)
University of Tennessee
April 15\textsuperscript{th} 2019
Motivation: Reduce Redundant Synchronization

- Bulk Synchronous Parallel (BSP) is simpler to reason but might limit parallelism with superstep
- Build the dependencies on tasks or data, the tasking runtime can provide a finer grain synchronization
- Then we can have asynchronous calls (via callback or periodically checking status)
Motivation: Reduce Redundant Synchronization

• Hidden in most task based runtime system, task dependencies need to be managed, can use asynchronous operations for progress
• Future is here to help!
  • C++ 11 included this structure for multithreaded programming
  • Javascript uses async operations extensively, has adopted this feature
  • The list goes on...
• Join force between Futures and Runtime!
  • UPC++ (C++ PGAS model)
  • Legion (data centric task based programming model)
  • MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation)

```cpp
// c++ "global" variables become rank-local state.
std::unordered_map<int, int> my_dht_local;

// owner does the work, result is a future<int>
upcxx::future<int> dht_fetch_inc (int key) {
    return upcxx::rpc( // `rpc` sends lambda to rank
        key % upcxx::rank_n(), // owner rank in key-to-rank partition
        [=]() { return my_dht_local[key]++; } // [=] captures `key`, used remotely
    );
}
```

UPC++ use futures for async
Motivation: Reduce Redundant Synchronization

- PaRSEC (distributed task based runtime system)
- http://tiny.utk.edu/parsec
What is a Future?

- A structure that usually contains the following components (in C++ you will get the templated functions)

```c
typedef struct parsec_base_future_t {
    parsec_object_t super; /* a base future type is a PaRSEC object*/
    void * cb_data; /* a pointer to the data this future is tracking */
    parsec_future_fn_t * future_class; /* struct that holds all the common function pointers*/
    parsec_future_cb_func cb_func; /* callback function! */
    uint64_t ready; /* flag to indicate whether the future is in ready state */
    parsec_atomic_lock_t future_lock; /* lockable for multithread access */
} parsec_base_future_t;

typedef struct parsec_countable_future_t {
    parsec_base_future_t super;
    int32_t count; /* extension of basic future with a count before ready, manipulate atomically */
} parsec_countable_future_t;

typedef struct parsec_future_fn_t {
    parsec_future_construct_t future_construct;
    parsec_future_destruct_t future_destruct;
    parsec_is_ready_t is_ready;
    parsec_set_t set;
    parsec_get_t get;
    parsec_future_init_t future_init;
} parsec_future_fn_t;
```
Patterns of Future Usage

- As a barrier, since you can block on the get() method
- One trigger one, the base case
- Multiple trigger one
- One/Multiple trigger multiple

Diagram:

- T1
  - T2
    - T3
- T1
  - T2
  - T3
- T1
  - Multiple callbacks?
  - T2
    - T3
Experiment Setup in PaRSEC via DTD Insert Task Interface

• Implement the future structure in PaRSEC as a class
• DTD insert task interface has “don’t track” option for the data, leaving the tracking to user, and we will use Futures for tracking
• We use the following callback function to track at task level
• Here we show results from two algorithms
  • POTRF
  • 2-D stencil

Once a future is set, it checks it’s callback function, calls it and inside that we insert the next task

```c
static void cb_func(parsec_base_future_t* future){
    parsec_task_t* this_task = (parsec_task_t*)future->cb_data;
    OBJ_DESTRUCT(future);
    parsec_insert.dtd_task(this_task);
}
```
Experiment: POTRF

- Algorithm is right looking without look ahead

```c
int parsec_core_potrf(parsec_execution_stream_t *es, parsec_task_t *this_task);
int parsec_core_trsm(parsec_execution_stream_t *es, parsec_task_t *this_task);
int panel_trsm(parsec_execution_stream_t *es, parsec_task_t *this_task);
int trailing_syrk(parsec_execution_stream_t * es, parsec_task_t * this_task);

CORE_zpotrf(uplo, m, A, lda, info);
//after computation, based on current row, start panel TRSM
panel = ((rows/m)-1-cur);
if(panel>0){
    parsec_countable_future_t *c_fut = OBJ_NEW(parsec_countable_future_t);
    c_fut->super.future_class->future_init(c_fut, cb_func, panel+1); //set counter and cb_func
    parsec_task_t* task = parsec_dtd_taskpool_create_task(dtd_tp,
               parsec_countable_future_t*, &c_fut, VALUE,
               sym_two_dim_block_cyclic_t*, &__dcA, VALUE,
               int, &rows, VALUE,
               int, &lda, VALUE,
               int, &cur, VALUE,
               int, &panel, VALUE,
               PARSEC_DTD_ARG_END);
    //set the cb_data to the newly constructed task struct
    //for base future, this will trigger callback, which will in turn insert downstream task panel trsm
    fut->future_class->set(fut, task);
}
return PARSEC_HOOK_RETURN_DONE;
```
Experiment: POTRF

- 2 Haswell E5-2650 v3 @ 2.30GHz, practical peak of 666 GFLOPS for DGEMM

- Share memory run, lower performance at the lower end likely due to the algorithm
- Minimal dependency overhead, since user encoded the dependency via futures
Experiment: 2-D Stencil

- Our old friend 2-D Stencil, created two grid of Futures corresponding to the two working grids.
- Future will trigger its 4 neighbors.
- Profiling on a 4 by 4 mesh with 100 iterations, the time interval between consecutive tasks in an execution stream (idle time between tasks, lower is better).
Open Questions

- Similar to the future idea, async MPI completion with callback, speed up communication for PaRSEC? (my colleague’s proof of concept)

- Adopt future to overlap independent operations within the task based runtime system (not just for PaRSEC). Open for suggestions/collaborations