Plans for fine-grained multi-threading in Cactus to improve efficiency and scalability

Erik Schnetter, Perimeter Institute
Einstein Toolkit Workshop 2015
Stockholm, August 12, 2015
State of the Toolkit

• Current state:
  • Cactus parallelizes via MPI+OpenMP
  • Functions (“compute kernels”) are explicitly scheduled
  • Driver performs domain decomposition

• Problems:
  • MPI programming is cumbersome, OpenMP is not efficient
  • Writing a schedule is difficult (i.e. near impossible)
  • Domain decomposition doesn’t scale well

• Also:
  • Horizon finding / I/O are not really parallel
  • Prolongation leads to load imbalance

• No easy way to fix all this
The Plan

• Parallelism:
  • Execute flesh, scheduler, high-level functions only on one process
  • Treat compute nodes similar to accelerators

• Scheduling:
  • Determine dependencies dynamically (before/after), allow concurrent execution
  • Determine many actions automatically (sync, prolongation, boundary conditions)
  • Execute functions only when their results are needed
  • Manage time levels automatically

• Domain decomposition:
  • Decompose domain into small, equal-sized blocks (e.g. $8^3$)
  • Assign blocks to caches, reassign to balance load
Background

• Ideas from other codes:
  • Uintah, HPX, Madness, Charm++

• Theory:
  • Discussions with MPI developers
  • Disappointing open source OpenMP implementations
  • “MPI+MPI” programming model

• Other tools / languages:
  • Grand Central Dispatch (Apple), Qthreads (Sandia, Chapel)
  • HPX (LSU)
  • mpi4py, Boost.Serialization, Cereal
  • C++11
  • Haskell
Existing Ingredients

• Cactus scheduling:
  • Brief, conceptual work on “requirements”
  • Chemora (with J. Tao, S. Brandt): scheduled functions declare their inputs and outputs (“reads” and “writes”), used for OpenCL/CUDA programming

• FunHPC:
  • C++ library combining MPI, Cereal, Qthreads etc., for HPC programming in a functional style
  • Proof of concept: Standalone 1d WaveToy implemented via FunHPC
    • Easy to read (even the “schedule”)
    • Scales to 16k cores
Chemora

• See Steve Brandt’s presentation earlier

• In brief:
  • Schedule annotated via “reads”, “writes” statements describing inputs and outputs
  • Also describing affected regions (interior, boundary, everywhere)
  • Sufficient to detect most user-level errors
  • Used to *automatically* run calculations with CUDA, where data need to be copied between host and device

• Plans:
  • Automate many more things, e.g. syncs, boundary conditions
FunHPC

• Example: 1d WaveToy
  • Distributed via MPI, multi-threaded via Qthreads

• Simple code, easy to read, easy to get “right”

• Memory management:
  • Handled by C++11 (shared_ptr and friends)

• Multi-threading:
  • Conflicts (deadlocks, undefined behaviour) provably avoided by functional style

• “Cactus” structure (parameters, grid functions, schedule, routines, driver tasks) easily visible in code
Example: Fibonacci Numbers

- int fib(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  auto f1 = qthread::async(fib, n - 1);
  auto f2 = qthread::async(fib, n - 2);
  return f1.get() + f2.get();
}
Example: 1d WaveToy

• Uniform grid:
  • A distributed, lazy array, implemented via a tree where each element is a (small) vector

• template <typename T>
  using storage_t = adt::tree<funhpc::proxy, std::vector, T>;

struct grid_t {
  real_t time;
  storage_t<cell_t> cells;
};
Example: 1d WaveToy

- State vector (i.e. all relevant grid functions):

```c
struct state_t {
    int_t iter;
    grid_t state;
    grid_t error;
    qthread::shared_future<norm_t> fnorm;
    qthread::shared_future<real_t> fenergy;
    grid_t rhs;
};
```
Example: 1d WaveToy

State vector constructor (i.e. schedule):

```cpp
state_t(int_t iter, const grid_t &grid):
    iter(iter),
    state(grid),
    error(grid_error(grid)),
    fnorm(qthread::async(norm, grid)),
    fenergy(qthread::async(energy, grid)),
    rhs(rhs(grid))
{
}
```
Example: 1d WaveToy

- RK2 integrator:
  ```cpp
  grid_t rk2(const state_t &s) {
    const grid_t &s0 = s.state;
    const grid_t &r0 = s.rhs;
    auto s1 = axpy(s0, r0, 0.5 * parameters.dt);
    auto r1 = rhs(s1);
    return axpy(s0, r1, parameters.dt);
  }
  ```
Example: 1d WaveToy

• Main loop (driver)
  • There is an I/O token, so that we can wait until I/O is finished

• `qthread::shared_future<int> file_token = 
  qthread::make_ready_future(0);
state_t s(0, grid_init(parameters.tmin));
file_token = fun::fmap(file_output, file_token, s);
while (s.iter < parameters.nsteps) {
  s = state_t(s.iter + 1, rk2(s));
  file_token = fun::fmap(file_output, file_token, s);
}
file_token.wait();
std::cout << "Done.\n";
The Plan

• Put this into Cactus “as-is” as proof of concept
• Store futures/proxies instead of pointers to grid functions
• Use Carpet to produce respective domain decompositions (already implemented, used both for AMR and DGFE)
• Rewrite Cactus scheduler to use threads, futures

• See Chemora (scheduler rewriting)
• See DGFE (fewer ghost zones)
• See SpEC code (being redesigned with Charm++)