Preliminary equilibrium optimization with `mango`

B.J. Faber

University of Wisconsin-Madison

Resource challenges in stellarator optimization

- The problem of 3D magnetic confinement:
  - Evaluating realistic physics objective functions can be (prohibitively) expensive
  - Vast space of possible configurations to explore, how is this best accomplished?
- Many routes to an optimized stellarator:
  - Surrogate modeling
  - Novel optimization schemes (stochastic optimization, etc.)
  - Automatic differentiation, adjoint methods
  - “Brute-force” methods
- Most likely a combination of all of these will lead to new breakthroughs in stellarator physics

Stellarator optimization is expensive

- Governments are heavily invested in pushing the computing envelope (Exascale Computing Project)
- The resources are there, provided they can be leveraged effectively
- Demonstrated need exists for an optimization framework that can operate extremely large scales
  - Exploring 10^8 quasisymmetric configurations with surrogate models still requires significant resources
- For realistic physics targets, the devil can be in the details:
  - Finite-build coils can alter quasisymmetry spectrum (secondary optimization problem)
  - Magnetic field ripple with finite-build coils can significantly degrade energetic particle confinement (requires expensive orbit-following)
  - Details of linear instability/turbulence spectrum can significantly impact turbulence saturation levels (requires knowledge of mode coupling)

New paradigm in high-performance computing

- Moore's Law has ended; performance gains in HPC focus on increasing concurrency
- Many physics simulation codes can no longer simply scale by adding more computing cores
- Next gen NERSC Perlmutter system: 64 CPU cores, 256 GB shared memory, 4 NVIDIA Tesla A100 GPUs per node, 5+ TB/s flash filesystem
- A scalable optimizer needs to leverage all of these resources

Leveraging resources

- Portable, efficient resource allocation is a difficult problem, but solutions exist
- Requires a shift from traditional (MPI-based) computational physics programming:
  - Lightweight computational **threads** operating on shared memory, not fenced processes
- Thread and process management libraries exist to unify access to CPU-GPU address spaces and optimize thread/execution scheduling:
  - Kokkos - Sandia: programming ecosystem for parallel execution, memory abstraction
  - HPX - LSU: parallel execution model focusing on hiding latency, fine-grained parallelism
- Portable physics algorithms: CPU or GPU execution policy only determined at compile time (no need for separate algorithms)
- Portability further ensured by strict adherence to ISO standards (C, C++, Fortran)
- Portability can also be ensured by using images: Docker, Singularity, Shifter

Natural problem splitting

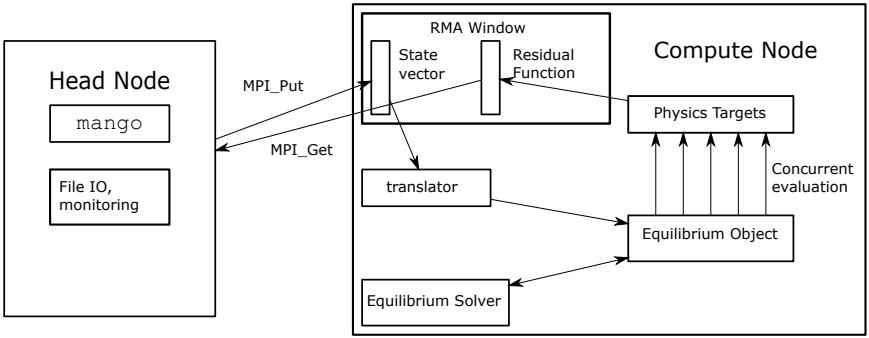
- Stellarator optimization naturally has 2 levels of parallelization: parallel equilibrium evaluation, parallel objective function evaluation
- Physics targets only require knowledge of the equilibrium
- Logical splitting: evaluate one equilibrium per shared memory node, evaluate physics targets concurrently on both CPU and GPU

```
targetResult = thread(targetObject::evaluate,&targetObject,
               &RuntimeOptions,&EqObject);
```

- `RuntimeOptions` and `EqObject` are provided through separate libraries
- Encourages standardization of physics targets
- Use `mango` to control the optimization context

Program flow

Schematic of possible program flow



Proposed equilibrium interface

- A robust equilibrium interface is extremely useful for stellarator optimization
- Physics targets would link against the library, rather than individually defining a geometry type
- Plasma Equilibrium Toolkit (PET, `https://gitlab.com/wistell/PET`)
  - Flexible, C++17: Parallel STL, `std::future`, `std::any`,...
  - Interface between equilibrium solvers, magnetic geometry data structures
  - Goal: multithreaded, asynchronous fetch and compute operations
    - Multithreaded: Useful for shared memory concurrency, no need to recopy data
    - Asynchronous: Different physics targets might require different coordinate representations (e.g. PEST vs. Boozer), only compute when necessary
  - Initial implementation: focus on VMEC, transformation to PEST coordinates for physics targets
  - Next steps: include SPEC interface, implement Boozer coordinate transformation

Testing with mango

- PET contains utilities to read VMEC Fortran namelist file, pass to Fortran
- Input parameters contained in `std::map<std::string,std::string>`, easy to pass through MPI (all have type `MPI_CHAR`)
- Easily construct a translator between state vector entries and VMEC parameters
- Example: boundary coefficients labeled by `"rbc(N,M)"` and `"zbs(N,M)"`

```
std::map<size_t,std::string> translator;
...
translator[n] = std::string("rbc(3,2)");
...
```

- Translator and VMEC input parameters can then be passed through `mango` using the `set_user_data(void*)` function

Testing with `mango`

- Simple test case set up: target rotational transform on select surfaces
- Use `mango::Least_squares_problem`:
  - $res(x) = \sum_i \left(\iota\left(s_i\right) - \iota_{\text{target}}\left(s_i\right)\right)^2 / \sigma_i^2$
  - s_i : surfaces in normalized toroidal flux, target $i = 5$ with $1.05 \leq \iota(s) < 1.25$, $s \in (0, 1)$
  - Target decreasing $\iota(s)$ profile
  - PET computes $\iota(s)$ using `boost::math::cubic_b_spline` $\Rightarrow$ portable and accurate
- Parameterize problem with first 13 lowest boundary coefficients
  - `rbc(n,m)`, `zbs(n,m)` – $\texttt{n} \in [-2, 2]$, $\texttt{m} \in [0, 2]$
- Use PETSc-TAO `brgn`, `POUNDERS` algorithms for parallel objective function evaluations
- Works. . . to some extent: eventually VMEC errors and calculation cannot recover (still working on this)

mango objective function implementation: simple!

- Head node:

```
vmecRuntimeData* vmec_data = (vmecRuntimeData*) userData;
std::map<std::string,std::string>* vmecOptions = vmec_data->parameters;
std::map<int,std::string>* translator = vmec_data->translator;
translate(stateVector,translator,vmecOptions); //Update the VMEC parameters
passStringMap(prob,vmecRuntimePars); //Pass updated parameters to workers
PET::VMEC vmec(vmecOptions);
vmec.initializeVMECData();
vmec.run_VMEC(comm_worker_groups);
vmec.retrieveVMECData(comm_worker_groups);
```

- Worker node:

```
while(prob->mpi_partition.continue_worker_loop()){
  vmecRuntimeData* vmec_data = (vmecRuntimeData*) userData;
  std::map<std::string,std::string>* vmecOptions = vmec_data->parameters;
  passStringMap(prob,vmecOptions); // Receive the updated parameters
  PET::VMEC vmec(vmecOptions);
  vmec.initializeVMECData();
  vmec.run_VMEC(comm_worker_groups);
  vmec.retrieveVMECData(comm_worker_groups);
}
```

Early lessons

- The largest impediment currently is VMEC:
  - Inexplicable memory leaks in PARVMEC code base: array entries observed to change without any traceable cause
  - Memory leaks invalidate any results obtained by VMEC, how can we trust?
  - Debugging VMEC is extremely difficult: limited debugger functionality, `valgrind` is slow and expensive
- Simple to connect with `mango`, only knowledge of API necessary
  - Comparatively little time spent interfacing with `mango` (hours) vs. VMEC (days/weeks)
- Largest majority of time spent debugging and validating VMEC results, transitioning to SPEC may be the better solution
- Optimizer comparison:
  - `POUNDERS` (derivative-free) able to reverse $\iota(s)$ profile
  - `brgn` (derivative-based) effective at moving $\iota(s)$ around $\iota = 1$, not in reversing slope