HACC Code Architecture

Hal Finkel
ALCF

ATPESC 2013

HACC Code Architecture

- Build system
- Type neutrality
- Memory management
- Unit Conversions
- Code layout and layering
- Classic OO vs. template specialization
- Some comments on testing

HACC: Build System

- The repository has an 'environment' shell script for each supercomputer
- The variables set by the environment script are used by the make files
- Each subdirectory builds one or more static libraries
- Built object files, libraries, and executables are placed in machine-specific subdirectories
- Compiler flags, svn repository information, 'svn diff', etc. are recorded into a generated source file; this information appears in the run logs

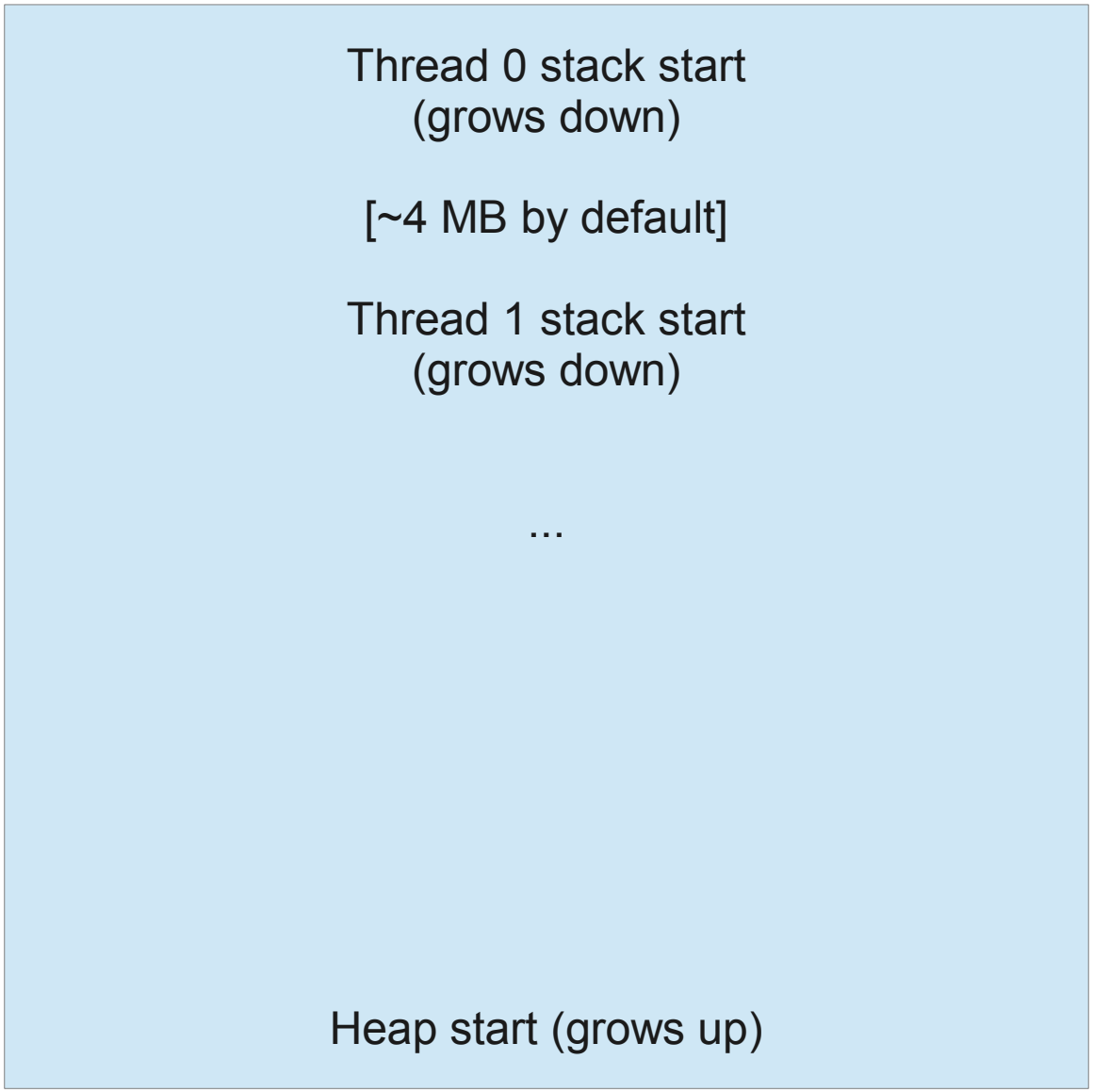
HACC: Type Neutrality

- Use typedef and preprocessor macros to name different types: ID_T, POSVEL_T, etc.
- The user can select single precision vs. double precision, 32-bit or 64-bit particle IDs, etc. at compile time.
- Important for characterizing round-off error, reducing memory overhead, and improving readability.

HACC: Memory Management

- Memory use minimization drives many design decisions in HACC.
- Both overheads and fragmentation must be reduced.
- Interfaces are designed so unnecessary copies are not required (use pointers or C++ iterators, but not std::vector references, for example). Many interfaces take allocator objects to customize internal allocations.
- Memory needed only for particular parts of each time step comes from a custom pool allocator (which is 'reset' after each top-level operator: the grid for the FFT and the tree for the short-range solver use the same memory)
- All particle arrays (x, y, z, etc.) are allocated together to avoid fragmentation.

Stack vs. heap



Stack vs. Heap (cont.)

```
void foo(int n) {
  int a; // on the stack
  double x[30000]; // on the stack
  double *y = new double[3000]; // on heap
  double q[n]; // in C99, C++11, on stack!
  double *z = (double *) malloc(3000*sizeof(double));
  free(z);
  delete [] y; // remember the [] here!
}
```

HACC: Unit Conversions

- Particle data needs to exist in different units: physical vs. program units; rank-local vs. global coordinates, etc.
- The 'Particles' class tracks the current units of the particle data. Any routine which depends on the units of the particles calls a coordinate conversion function of the 'Particles' class before accessing the particle data. If the required units don't match the current units, a conversion is performed.
- This kind of 'on demand' conversion tends to be both less error-prone and more efficient than using an explicit interface protocol between components.

HACC: Code Layout and Layering

- HACC is divided into a number of components (initializer, FFT and poison solver, particle handling, halo finding, etc.).
- The components sit in separate subdirectories, are built into separate static libraries, and satisfy layering requirements:
- If A depends on B, then B cannot depend on A!
- How do you decide what is a component: lack of cyclic dependencies, and independent testability.
- Remember: const and restrict are part of the interface!
  -- foo(const double * restrict a, double * restrict b) {}

HACC: OO and Templates

- HACC uses both classic object oriented design (including use of virtual functions), and C++ templates.
- Classes with virtual functions are used for mixing in large components (types of solvers, for example).
- C++ templates are used to customize performance-critical (or sometimes memory-overhead critical) components. No virtual function calls in inner loops!
- To maintain user flexibility we often instantiate several variants of important templates (such as the short-range force solver). Each variant is given a name and the user can select the variant using the configuration file.

HACC: Testing

- Many components have unit tests (and they all should).
- Run tests under valgrind and/or address/memory sanitizer to check for hidden problems.
- We have a set of standard test problems, and a good statistical way (P(k) primarily) to compare the results.
- Always run a known problem before doing production science: bugs in your code (or in the toolchain, MPI implementation, etc.), such as use of an uninitialized variable, can make seemingly-unrelated changes affect your answers.
- Be careful when using shared libraries: understand the difference between -L/some/path and -Wl,-rpath,/some/path – and make sure to note what can change underneath you! 'ldd' is your friend.

rpath?

- -rpath and LD_RUN_PATH go into the executable.
- When the dynamic loader (/lib*/ld*.so*) uses the rpath, LD_LIBRARY_PATH and some system-specific set of search paths.
- The linker also needs to find the libraries when linking: uses -L, LIBRARY_PATH and some system-specific set of search paths.