CLOMP v1.5

Summary Version

1.5

Purpose of Benchmark

CLOMP is the C version of the Livermore OpenMP benchmark developed to measure OpenMP overheads and other performance impacts due to threading. For simplicity, it does not use MPI by default but it is expected to be run on the resources a threaded MPI task would use (e.g., a portion of a shared memory compute node). Compiling with -DWITH_MPI allows packing one or more nodes with CLOMP tasks and having CLOMP report OpenMP performance for the slowest MPI task. On current systems, the strong scaling performance results for 4, 8, or 16 threads are of the most interest. Suggested weak scaling inputs are provided for evaluating future systems. Since MPI is often used to place at least one MPI task per coherence or NUMA domain, it is recommended to focus OpenMP runtime measurements on a subset of node hardware where it is most possible to have low OpenMP overheads (e.g., within one coherence domain or NUMA domain).

Characteristics of Benchmark

CLOMP's target input approximates a typical scientific application inner loop workload under strong scaling conditions (that is, not a lot of work available to hide OpenMP overheads). The overall speedup and implied overhead of several OpenMP scheduling algorithms are then measured. Most current OpenMP benchmarks tolerate OpenMP overheads several orders of magnitude higher than is necessary in order to get reasonable performance out of threading loops with just a few hundred thousand cycles of work in them. In order to get good performance with CLOMP's target input, and with many of our scientific applications, it is critical for there to be hardware support for threading and for the OpenMP compilers and libraries to be implemented to effectively use this OpenMP-accelerating hardware. The CLOMP benchmark can be used to demonstrate the need for new techniques for reducing thread overheads and to evaluate the effectiveness of these new techniques. The CLOMP benchmark is highly configurable and can also be used to evaluate the handling of other well-known threading issues such as NUMA memory layouts, work-load imbalance, cache effects, and memory contention that also can significantly affect performance.

Examples of OpenMP Hardware Support

Examples of OpenMP hardware support in current systems include support for atomic operations and locking operations in L2 cache and mechanism for efficiently distributing work to a large number of threads. For today's systems that synchronize threads thru main memory, current best-in-class implementations of OpenMP have overheads at least ten times larger than is required by many of our applications for effective use of OpenMP. For these applications to most effectively use OpenMP with 8 threads per MPI task, they require thread barrier latencies on the order of 200 processor cycles and total OpenMP "parallel for" overheads on the order of 500 processor cycles on high performance systems. With a single read from main memory often

taking several hundred cycles, it is clearly impossible to achieve these overhead goals without specialized hardware support.

Parameters of Benchmark

The CLOMP benchmark is configured entirely at run time using the command-line parameters. The usage information for the CLOMP Version 1.5 benchmark that is output when it is run with no arguments is shown in Fig. 1. The `numThreads` argument specifies the number of OpenMP threads to use when running the benchmark, where the special value '-1' specifies that the default number of OpenMP threads (usually the number of processors or the number of threads set via OMP_NUM_THREADS). The `allocThreads` argument specifies how many threads will be used to allocate the memory. The typical values for `allocThreads` are 1 and -1. Setting `allocThreads` to 1 emulates what most of our codes do, which allocates all the memory touched in the main thread (which causes poor thread memory layout on systems that exhibit NUMA effects). Setting `allocThreads` to -1 (or the number of threads used) threads the allocation using OpenMP the same way the calculations are threaded so that the same threads will allocate the memory as use them, thereby improving thread memory layout. (This is not currently guaranteed to be true by OpenMP, but it appears to be true for some OpenMP implementations). Although ideally programs would allocate all the memory a thread touches in that same thread, in practice this is often very hard to do. Thus, we are interested in the performance difference caused by whether the allocations themselves were done serially or in parallel.

The `partsPerThread`, `zonesPerPart`, `zoneSize`, and `flopScale` command-line arguments in Fig. 1 will be described more fully below as we describe the mesh and computer kernels of the CLOMP benchmark. The last command line argument, `timeScale`, is provided as a convenience for those running the CLOMP benchmark; it scales the run-time of the benchmark. A `timeScale` of 100 was designed to run for between 5 and 30 seconds for a serial run of the kernel (depends on mesh size and machine speed) that should provide reasonably accurate timings given the resolution of the timers used. A `timeScale` of 1 runs the benchmark very quickly to identify scaling problems and correctness but probably inaccurate timings. Using a `timeScale` of 100 is probably a reasonable, yet still short, run on most current machines but may have to be adjusted for machine speed.

```
CLOMP Version 1.50 (CORAL2 RFP)
Usage: clomp numThreads allocThreads partsPerThread \
           zonesPerPart zoneSize flopScale timeScale

New in Version 1.2: Compile with -DWITH_MPI to generate clomp_mpi
New in Version 1.3: zonesPerPart can be range expression start-end
                    (i.e., 700-100)
                    zonesPerPart can be random over range by adding R[seed]
                    (i.e., 100-300R2)
New in Version 1.5: Args change: numParts = numThreads * partsPerThread

  numThreads: Number of OpenMP threads to use (-1 for system default)
  allocThreads: #threads when allocating data (-1 for numThreads)
  partsPerThread: Number of independent pieces of work per thread
  zonesPerPart: Number of zones in the first part (3 flops/zone/part)
  zoneSize: Bytes in zone, only first ~32 used (512 nominal, >= 32 valid)
  flopScale: Scales flops/zone to increase memory reuse (1 nominal, >=1 Valid)
  timeScale: Scales target time per test (10-100 nominal, 1-10000 Valid)

Some interesting testcases (last number controls run time):
           Target input:    clomp 16 1 1 400 32 1 100
   Target/NUMA friendly:    clomp 16 -1 1 400 32 1 100
UNBALANCED Target input:    clomp 16 1 4 700-100 32 1 100
RANDOM UNBALANCED input:    clomp 16 1 4 1-800R4 32 1 100
    Weak Scaling Target:    clomp N -1 1 400 32 1 100
      Weak Scaling Huge:    clomp N -1 1 6400 32 1 100
  Strong Scaling Target:    clomp -1 -1 1024 10 32 1 100
       Mem-bound input:    clomp N 1 1 640000 32 1 100
Mem-bound/NUMA friendly:    clomp N -1 1 640000 32 1 100
  MPI/OMP Hybrid Target:    (mpirun -np M) clomp_mpi 16 1 1 400 32 1 100
```

Figure 1 Usage information for CLOMP v1.5 benchmark.

The CLOMP benchmark creates a simple unstructured mesh (see Figures 2a, 2b, and 2c) that is configured via the command-line parameters `partsPerThread`, `zonesPerPart`, and `zoneSize` that were shown in Fig. 1. The mesh consists of "`numThreads * partsPerThread`" independent zone partitions where each zone partition contains "`zonesPerPart`" zones. The `zonesPerPart` parameter may be a constant (e.g., 4) as shown in Fig. 2a, a deterministic range expression (e.g., 4-1) as shown in Fig. 2b, or a random range expression (e.g., 1-4R2) as shown in Fig. 2c that pseudo-randomly picks zone counts from the specified range (1-4) with the random seed specified (2). The non-constant `zonesPerPart` expressions allows creating situations where OpenMP dynamic scheduling is expected (or desired) to outperform static scheduling.

The CLOMP benchmark explicitly allocates all the zones (e.g., Zone01, Zone02, Zone 03, and Zone04 in Fig. 2a) in each zone partition (e.g., Part0 in Fig. 2a) in a continuous block, so that all the partition's zones are immediately adjacent to each other (even though they are accessed via a linked list). This zone allocation strategy should allow prefetching to work well while traversing each partition's zones. The amount of memory allocated for each zone is set by the `zoneSize` parameter (shown in Fig. 1), although there is a system dependent minimum size that is usually 32 bytes. Only the first approximately 32 bytes of each zone is actually used, and the `zoneSize`

parameter is provided mainly as a way to increase the memory footprint of the mesh without creating more work.

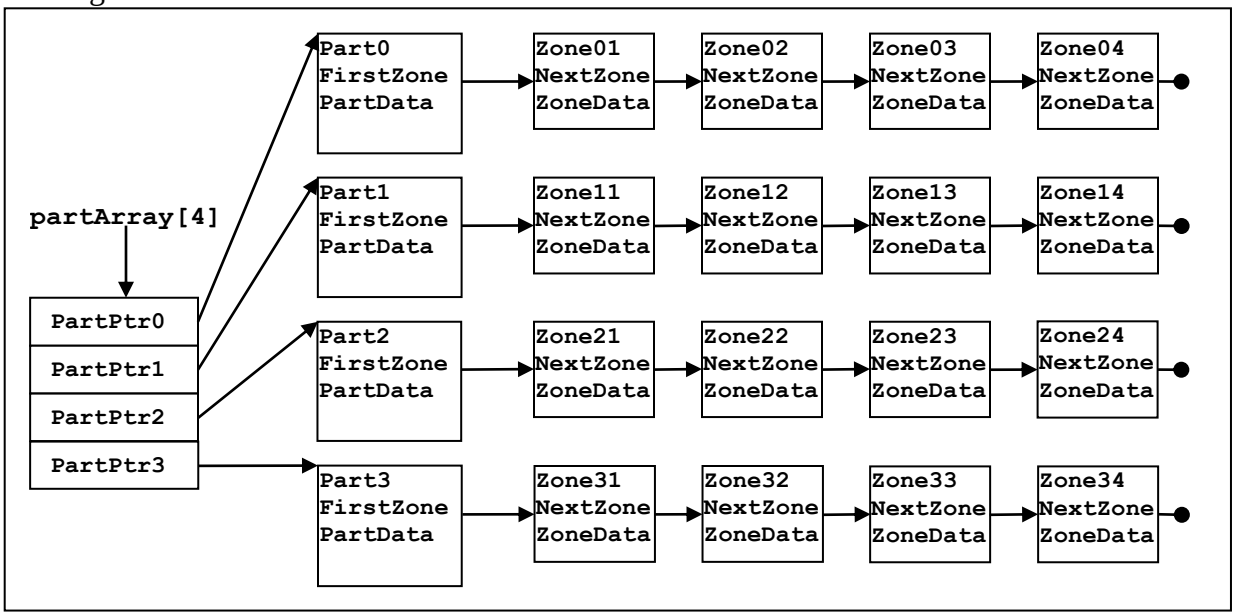

Figure 2a CLOMP balanced unstructured mesh data structures for numThreads * partsPerThread = 4 and zonesPerPart=4.

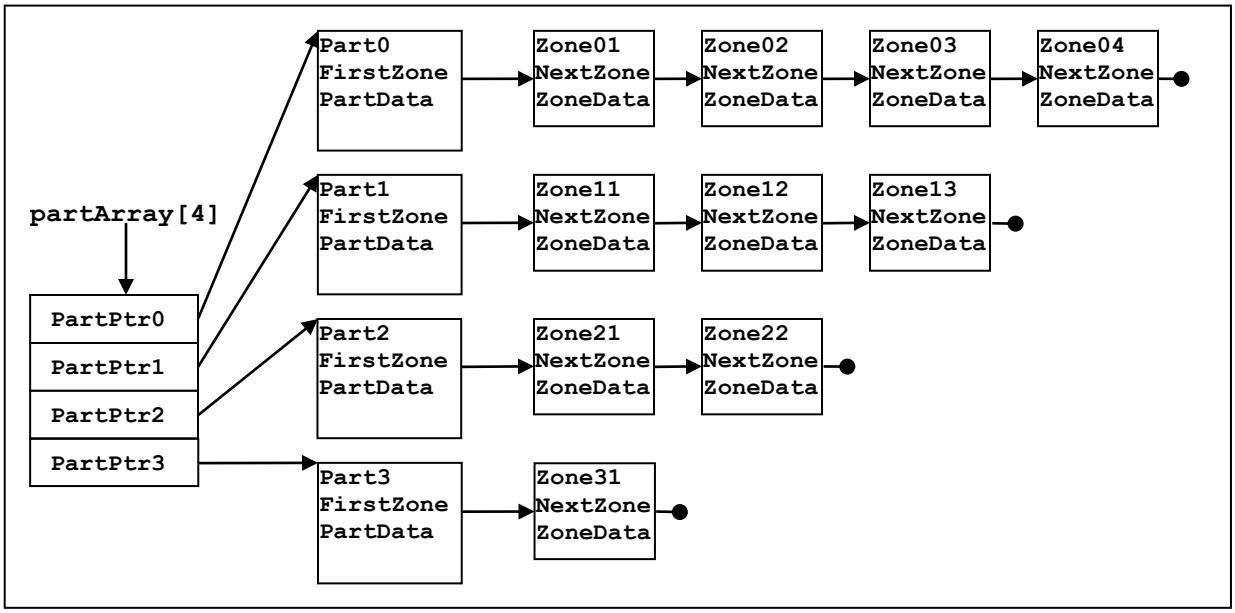

Figure 2b CLOMP workload-sorted unbalanced unstructured mesh data structures for numThreads * partsPerThread = 4 and zonesPerPart=4-1 (workload-sorted dynamic scheduling test configuration).

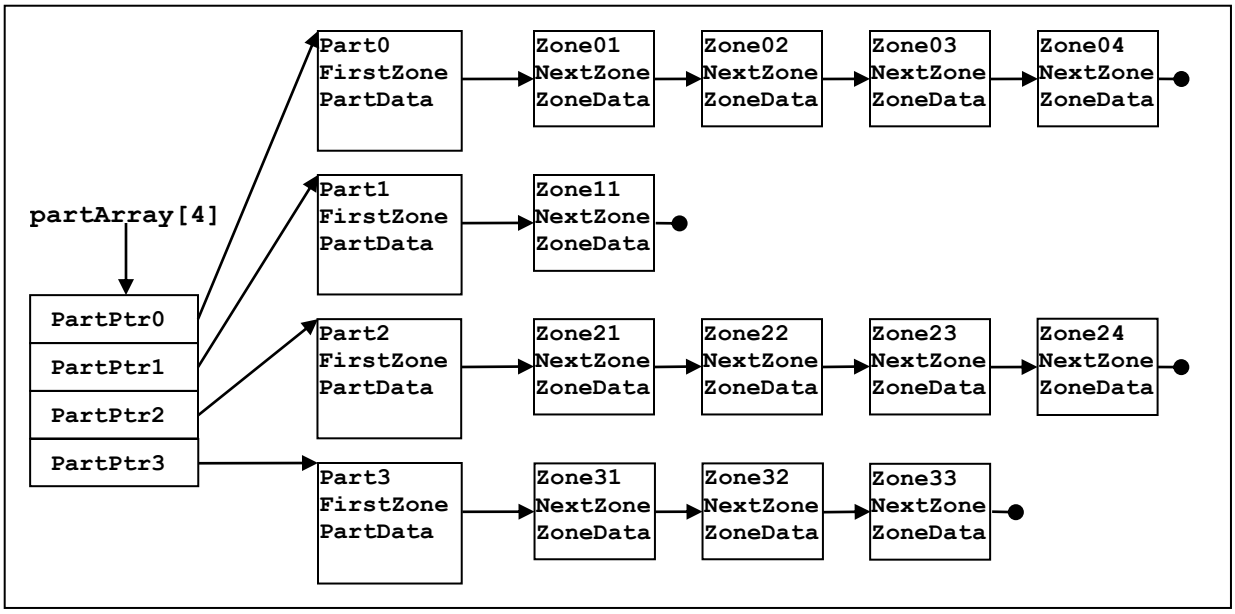

Figure 2c CLOMP unsorted randomized unbalanced unstructured mesh data structures for numThreads * partsPerThread = 4 and zonesPerPart=1-4R2 (unsorted workload dynamic scheduling test configuration).

No real or useful physics is done by the CLOMP benchmark, but a configurable amount of physics-like work is done in each zone during each "physics cycle," and the benchmark is designed to produce bit-for-bit reproducible (and predictable) answers no matter how many threads are used to calculate the results. The CLOMP benchmark uses this property (and other techniques) to detect many common threading errors. The thread-parallel kernel for the CLOMP benchmark is shown in Fig. 3 with OpenMP directives for static scheduling. The `calc_deposit()` call in Fig. 3 represents an MPI exchange of data (although currently no MPI is done in the timing sections CLOMP, even with -DWITH_MPI) and must be called from a single-threaded region and must be called after the previous thread parallel work is done. All the computational work is done in the `update_part()`, and it is the for loop around `update_part()` that is the target for threading.

```
deposit = calc_deposit (); /* Sync, non-threadable */
#pragma omp parallel for private (pidx) schedule(static)
for (pidx = 0; pidx < num_parts; pidx++)
   update_part (partArray[pidx], deposit);
```

Figure 3 CLOMP thread-parallel kernel with OpenMP directives and static scheduling.

Although the focus of the CLOMP benchmark is the OpenMP kernel shown in Fig. 3, a simplified version of the compute kernel of `update_part()` is shown in Fig. 4 in order to explain the `flopScale` command-line parameter from Fig. 1. This `update_part()` kernel follows the linked list of zones in each zone partition and does a little bit of math on the zone's value. Its purpose is to consume cycles in a configurable way that produces verifiable output, not to actually do anything useful. When `flopScale` is 1 (the desired target setting), each iteration of the outside zone traversal loop does (with a reasonable optimizer) two loads from the zone (zone->value, a double, and zone->nextZone, a pointer), does a double multiple, a double add,

and a double subtract, and a double store to the zone (zone->value, a double). This memory access to flop ratio is representative of several interesting scientific calculations and can put a load on the memory system, especially prefetching logic of the memory system. By setting the `flopScale` parameter to 100 and reducing the mesh size by 100, one can get an input of approximately the same run time that is much less affected by the memory system. Using a value greater than 1 for `flopScale` is interesting only for explaining performance anomalies in the benchmark run when `flopScale` is set to 1. Note: The actual update_part() source code explicitly loop versions the zone loop in Figure 4 for flopScale == 1 to remove the scale_count loop overhead even for compilers than don't support that optimization.

```
for (zone = part->firstZone; zone != NULL; zone = zone->nextZone)
{
      for (scale_count = 0; scale_count < flopScale; scale_count++)
      {
         deposit = remaining_deposit * deposit_ratio;
         zone->value += deposit;
         remaining_deposit -= deposit;
      }
}
```

Figure 4 The non-threadable compute kernel of `update_part()`.

The CLOMP benchmark measures the total overhead for static, dynamic, and manual OpenMP "parallel for" loop scheduling and the speedup achieved when performing a "physics cycle" on the specified unstructured mesh. The potential "best-case" static-scheduled speedup is also determined in order to provide an approximate upper bound on static-scheduled threaded performance and in order to be able to calculate an efficiency rating for the OpenMP implementation. By running the CLOMP benchmark with several different mesh sizes and thread configurations, the performance effect of OpenMP overheads, NUMA effects, cache and memory bandwidth and latency effects, and prefetching effectiveness can be clearly seen. The amount of work each benchmark OpenMP test performs is run-time configurable and is, by design, independent of mesh size, so that a wide range of CLOMP benchmark runs can be done quickly.

Below we suggest some run configurations for CLOMP that have shown interesting results on the systems tested. There are probably many other useful configurations to try.

Mechanics of Building Benchmark

The CLOMP benchmark consists of one C file, clomp.c, and a Makefile that contains the compile line for a few compilers, and two similar example run script run_clomp.bgq and run clomp.cts1 of 22 interesting run configurations. These run_clomp* scripts generate run_clomp.summary and run_clomp.CORAL2_RFP which is summarizes the run data for the CORAL2 RFP. You can either compile clomp.c directly with the desired compiler arguments to get good OpenMP performance or you can put the compile line in the Makefile. Running the Makefile with no arguments shows the compiler lines available. For example, `make icc` builds CLOMP with icc and mpicc (assumed to be in your path) (on Linux) generating clomp, clomp_hwloc and clomp_mpi. If you are reporting results to us, please specify the compiler

options uses to build CLOMP. You must optimize clomp at least at the equivalent of -O3 unless you have a strong reason not to.

You may optionally compile with the -DHWLOC option which causes clomp to bind OpenMP threads to cores using the hwloc library. This binding yielded significant speedup on various x86 clusters tested. You may change the straightforward binding logic to better match your hardware.

Compiling with -DWITH_MPI enables MPI in the code. Our hwloc implementation is not MPI aware and generally generated very poor results with MPI as implemented. Our MPI launchers and bind helpers (e.g., mpibind) do a reasonable thread-binding by default. If your MPI launcher does not, please feel free to set up the appropriate binding when running clomp_mpi (either externally or by enhancing clomp.c). Since MPI is often used to place at least one MPI task per coherence or NUMA domain, focus on binding to a subset of node hardware where it is most possible to have low OpenMP overheads (e.g., within one coherence domain or NUMA domain).

Setting Up the OpenMP Environment

On the few systems we have tested so far, additional system-specific and/or compiler-specific environment variables need to be set in order to get good thread performance while not penalizing serial performance. We found OMP_WAIT_POLICY ACTIVE was critical on BG/Q but slowed down serial the serial sections of CLOMP by 25% or more on some other clusters, making the threaded portions appear artificially 'faster'. Clomp v1.5 now measures this serial impact and essentially scales down threaded speedups by the measured serial impact. Be sure to check this line of the clomp output to make sure the serial section is not being severely penalized (and thus the OpenMP speedup scores being penalized):

OMP Serial Impact: 1.03X Slower serial (2.694s / 2.619s, After OMP/Before OMP)

Ideally, the serial impact would be a 1.00X slowdown (no slowdown) or close to this.

Platform-specific thread settings sometimes give threads strong processor affinity, bind threads to separate processors, or direct the operating system to use all the available processors to run threads instead of just a few. In the past, we found these settings by looking at the SPEC OpenMP benchmark results where all environment variables set have to be specified and/or by asking the vendors for suggestions.

For example, on BGQ with the xlc compiler, these two environment variables had to be set for the best performance:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv BG_SMP_FAST_WAKEUP YES
```

On BG/Q, `OMP_WAIT_POLICY` set to `ACTIVE` did not have a negative impact on serial performance.

On Linux with recent icc compiler versions, setting `OMP_WAIT_POLICY` to `PASSIVE` was very important for good serial, as well as, OpenMP performance:

```
setenv OMP_WAIT_POLICY PASSIVE
```

Setting `OMP_WAIT_POLICY` to `ACTIVE` did slightly improve OpenMP runtimes but also significantly slowed the serial sections. With CLOMP v1.5 serial impact adjustment, this yield approximately the same net speedups. You will need to check your environment for the proper setting of `OMP_WAIT_POLICY`.

For both BGQ/xlc and Linux/icc, there were other thread environment variables available that we didn't explore with CLOMP v1.5. With some settings, OMP_NUM_THREADS had to be set to the number of threads used in order to get good performance (OMP_NUM_THREADS was used by the thread binding system) so we set OMP_NUM_THREADS in the run_clomp* scripts.

If you are reporting benchmark results to us, please describe what environment variables were set and why.

Mechanics of Running Benchmark

The CLOMP benchmark should be run on a dedicated (idle) node with the appropriate thread performance environment variables set (described above). All the runs should be done on the same machine around the same time (if possible) because we have found memory layout to sometimes be different on different nodes of the same cluster. The benchmark can be run directly (as described in Fig. 1), but when generating CORAL2 RFP results, it can be useful to modify the provided example script "run_clomp.bgq" or "run_clomp.cts1" to run a suite of CLOMP runs and automatically create comma-delimited result summary files that can be loaded into the CORAL2 results spreadsheet. The run_clomp.bgq script was designed to run 22 variations we have found most useful during our CLOMP benchmarking runs on our current machines. The script comments indicate which runs are for strong scaling performance, weak scaling performance, measuring dynamic scheduling benefits, detecting bandwidth limitations, detecting NUMA bandwidth limitations, and detecting hybrid OpenMP/MPI performance issues. The comma-delimited value file run_clomp.CORAL2_RFP is designed to be easily loaded into a spreadsheet and then pasted into the CORAL2 RFP results spreadsheet.

Interpreting the Output

The output of the CLOMP benchmark is fairly descriptive (pseudo code of what is being measured in printed in the full CLOMP output) and will not be described in detail here. The most important performance result of interest is the speedup for the OpenMP static schedule case over the serial run for ~400 zones per thread with the number of threads equal to the number of processor cores/threads dedicated to a single MPI task. Ideally, that speedup should be close to the number of threads used but is typically much lower due to current OpenMP overheads.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-745697