**Microblocks**

**Serial Microblocks**

The microblocking algorithm divides the simulation space into nearly square blocks with lengths on the order of magnitude of cutoff. After particles creation, each particle is then placed inside a single microblock. Collisions are detected for a single particle by scanning with all other particles within its own and neighboring microblocks. Since the forces are repulsive and the particle density is ~1/2 particle for each cutoff square, the average number of particles within a microblock is constrained to a low value. Design space exploration revealed microblocks with length equal to twice cutoff (corresponding to 2-3 particles per microblock) to be optimal. Thus, during collision scanning, each particle must check for 21 collisions or ~21n = O(n) collision checks in the whole system. After force application, the particles must be scanned for rebinning into new microblocks. However, rebinning considers each particle in isolation and thus is still O(n).

Some small optimizations were used to speed up this microblocks implementation. Rather than actually store the particle data in the microblocks, the particle list simply contains addresses to particles in a global particle array. This allows rebinning to be done by simply moving single pointers around instead of moving an entire particle data structure. The microblock particle list is modeled after STL vectors and thus insertions and deletions are of O(1) complexity. Similar to STL vector, the particle list is capable of expanding to greater sizes as needed; however, for particle counts attempted, the length of a single list rarely rose above the default size of 4. To prevent possible errors due to a particle jumping across an entire microblock within a single time step, rebinning is done from a global perspective (instead of neighbor passing) by calculating the target bin using two floating point multiplications.

Figure ##[figures/serial\_mb\_n.png] verifies the serial microblocks implementation is of O(n). The drop in performance for very high particle counts is most likely due to memory cache issues rather than algorithmic failure. Figure ##[figures/serial\_mb\_flops.png] provides evidence to this theory as the floating point utilization (measured using CrayPAT) drops considerably as n increases. Even so, the latter figure demonstrates this problem is largely memory bound instead of compute bound, with most time spent setting up collision detection and comparing particle rather than actually calculating forces.

**OpenMP**

The OpenMP implementation uses the microblocks algorithm as it was found to be extremely easy to develop. In fact, the serial microblocks algorithm was chosen intentionally to serve as this a precursor to OpenMP. Only two major adjustments were required from the serial base. OpenMP pragmas were added to farm out particle collision detection/force calculation, particle movements, and particle migrations, all by microblock. Also, in the particle migration code, locks were added to the microblock particle lists to prevent possible race conditions when multiple threads add or remove particles from the same microblock simultaneously. Implicit barriers sit between the force calculation, movement, and migration stages due to features of the OpenMP for pragma feature.

Figure ##[figures/openmp\_mb\_n.png] demonstrates the OpenMP microblocks implementation also scales roughly with O(n). When p is low, performance scales very precisely as O(n) for low n whereas for higher p, performance scales with O(n) for higher n. The former phenomenon is explained by overhead and synchronization needed by OpenMP using many threads that is exposed at low n. The latter phenomenon can again be explained by memory effects: when p is higher, more memory caches are available thus allowing the handling of higher n. Figure ##[figures/openmp\_mb\_p.png] demonstrates that the OpenMP implementation scales by O(n/p) when n≥10000 and p≤12. The boundary at 12 corresponds with known features of the Hopper architecture: the system is composed of 12-core processors. When the number of cores used increases above 12, requests must now be serviced on a larger interconnect network, incurring higher overhead. Due to the leveling off with p in this figure, OpenMP was not even attempted for p>24.

Figure ##[figures/openmp\_mb\_flops.png] shows that the floating point utilization (measured using CrayPAT) drops as the number of threads increases. Again, this is likely due to memory effects. However, in contrast to the serial implementation where memory issues generally equated to cache misses, here the slowdown from memory operations is attributable to cache coherence overhead amongst the system cores. Figure ##[figures/openmp\_mb\_rtime.png] tracks the time spent in portions of particle simulation (for select n) versus p. Somewhat surprisingly, the portion of time spent handling particle migrations is constant across both problem size *and* number of threads used (even though this is the only portion of code using locks). The rise in relative time spent handling movements is most likely a symptom of memory overhead since the actual computation is very simple thus most time is spent dealing with memory. Also, the OpenMP worker threads are redistributed within each compute portion (to help maintain a fair workload as particles move around), thus taxing the cache coherence system.

**MPI**

AN MPI implementation using the microblocks algorithm as the intra-cell kernel was also developed. Each MPI cell is a nearly square subdivision of simulation space in order to minimize needed communication. Naively, just this blocking could be used to resolve the problem in a manner similar to microblocks: cells simply send particles near the border to neighbors as ghosts, collisions are checked between all owned particles and ghosts, and then particles are migrated across cell boundaries as needed. However, the relatively large sizes of MPI cells versus microblocks make this naïve implementation inefficient. Thus, the MPI cells are further subdivided into microblocks inside of the single core and processed as in the serial implementation.

Besides directly speeding up collision detection as in the serial implementation, the microblocks algorithm also speeds up the handling of ghost zone particles and migration of particles between MPI cells. In each cell calculating possible ghosts to send, since microblocks are slightly larger than cutoff, only particles in the edge microblocks must be considered as candidates. Also, the microblocks algorithm already must iterate through each particle to check for migration between microblocks. This operation can be amended to simultaneously handle checking for particles migrating between MPI cells. These two optimizations provided nice performance gains in the MPI microblocks implementation. Other small optimizations were applied to handling the microblocks: they were placed in a pile that can be ‘cleared’ in O(1) time and the ‘ghost microblocks’ were linked to simulation microblocks on simulation setup to prevent the needing complicated neighbor lookups during main computation.

Figure ##[figures/mpi\_mb\_n.png] shows the MPI microblocks implementation follows O(n) for various p. Similar to OpenMP, the same phenomena for higher n with low p and lower n with high p manifest for the same reasons (although here amplified in magnitude due to the larger problem space explored). Figure ##[figures/mpi\_mb\_p.png] demonstrates the MPI microblocks implementation scales by O(n/p) as desired. The scaling for one million particles versus p most visibly demonstrates the power of the MPI implementation to exploit parallelism in this problem. In fact, since the MPI implementation strictly partitions the memory (compared to OpenMP), superscaling better than 1/p is exhibited as increasing p reduces the impact of memory issues. Even so, overhead costs from communication and synchronization limit the number of cores usable for sub-million particle problems to 48.

Figure ##[figures/mpi\_mb\_flops.png] explores how computational intensity (measured using IPM) changes as a function of the n and p. For very small n, the overhead in MPI communication causes lower p to be favored. However, for a million particles, the overhead is effectively hidden by the computation size thus allowing for good performance scaling with p. For ‘medium’ n, 24-48 cores form a sweet spot balancing increasing MPI overhead with decreasing memory footprint and thus deliver the best performance. Figure ##[figures/mpi\_mb\_rtime.png], which graph the relative time spent synchronizing, computing, and communication, sheds some light on these results. The optimal performance is gained when slightly more than half the time is spent computing. In general, the time spent synchronizing tracks the time spent communication (likely since synchronization waits are only incurred immediately following communication steps to ensure buffers are not overwritten).

**MPI vs. OpenMP**

The MPI implementation summarily outperforms the OpenMP implementation. The MPI libraries for Hopper are able to handle intra-node messages fairly efficiently. The MPI programming model also enforces strict memory partitioning between nodes. The OpenMP model relies on coherence protocols to move microblock and particle data between cores for processing as necessary. (Also, recall microblocks are redistributed fairly often in order to keep workload stabilized.) Thus, compared to MPI, OpenMP must transmit far more data is being transferred between core for the same n and p>1. This result is unsurprising and results principally by design. The OpenMP implementation was developed to be a simple and easy extension of the serial code, being implemented with only about 20 more lines of code. (This was an experiment to see if serial programs can be made parallel quickly and still gain reasonable performance benefits.) The MPI microblocks implementation required significant amounts of design, programming, and debugging time. Thus, from a programming time standpoint, the OpenMP implementation is still competitive since it provided reasonable gains versus serial for small p. However, MPI is still required to scale to large p efficiently. The MPI and OpenMP results, considered together, motivate a future solution using MPI for major blocking, but handling the intra-cell microblocks algorithm using OpenMP with 4-12 threads. This would permit the use of more processing resources without driving up MPI synchronization and communication overhead.