CS 5220: Project 2
Smoothed Particle Hydrodynamics

Saugata Ghose, Shreesha Srinath, Jonathan Tse
{*sg532, ss2783, jdt76*}*@cornell.edu*

***Abstract*—We improve upon a baseline smoothed-particle hydrodynamics algorithm to bring it down from $O(n^2)$ time to $O(n)$ time for n particles. We do so through a number of optimizations, which allow us to achieve up to a 67x speedup for large particle sizes over the baseline serial implementation. From this, we construct a parallel version of the code using OpenMP. We can achieve a 3.8x speedup over our optimized serial version when executing 8 threads. Furthermore, we show linear speedups as both the number of threads and the number of particles increase.**

I. Introduction

Thread-level parallelism (TLP) is one of the most common approaches to harnessing the power of multiprocessor machines. In today's environment of pervasive multicore computing, the importance of TLP continues to grow as making significant improvements to single-thread performance has become more difficult over the last decade. Unfortunately, writing parallel programs to exploit TLP is not a trivial problem, and not all applications are well-suited to exploiting parallelism. Ideal–embarrassingly–parallel applications exhibit minimal communication between threads, and have large amounts of work that can be divided amongst a large number of processors.

Smoothed-particle hydrodynamics (SPH) is a technique that simulates the flow of liquids within a defined space. Specifically, it evaluates the near-field interactions of discrete particles as well as the local particle density, in order to calculate the movement of particles over time. The SPH algorithm lends itself relatively well to parallel computing, as the limited scope of particle interactions translates into reduced inter-thread communication. However, SPH does not appear to be embarrassingly parallel, and so a reasonable amount of effort must be put into developing a parallel SPH program.

OpenMP is a popular method for implementing thread-level parallelism using shared memory. In OpenMP, when requested through the use of pragmas, multiple threads are spawned, each sharing some portion of memory for communications (this may be done explicitly or implicitly). In our multicore multiprocessor setup, cache coherency is used to send updated values between the cores. Effectively, when a new value is written to a shared memory location by a core, all stale copies residing in the caches of other cores are invalidated. These cores must then re-request the data, which will now contain the updated value. As a result, it is ideal to keep communications to a minimum, especially to reduce the number of shared memory writes within a program.

II. Experimental Setup

The tunings described by this paper are targeted for an Intel Xeon E5504 quad-core processor, which is part of the Core i7 family. Our code is restricted to using two of these processors, which are connected together using the Intel QuickPath Interface (QPI). Table I shows the relevant parameters of the processor for this project. All binaries were compiled using GCC version 4.4.3 on an Intel Xeon E5405 quad-core processor–part of the Core 2 family. For cache behavior analysis, we used Cachegrind, a part of the Valgrind memory analysis toolkit (version 3.2.1) [2].

We must note that currently, the worker nodes appear to be exhibiting inconsistent behavior. While it is not clear, we believe that the Sun Grid Environment queuer is time-sharing nodes amongst several job submissions, and as a result, submitted jobs are being context-switched. As a result, several of our performance results, notably those for the parallelized program, appear to be significantly skewed, and may affect our conclusions. Furthermore, we were unable to complete some experiments as a result of the ensuing bottleneck for simulation time.

TABLE I
Intel Xeon E5504 Processor Specifications [1]

Frequency	2.0 GHz
Number of cores	4
Address width	64 b
Cache line size	64 B
IL1/DL1 size	32 KB / 32 KB
IL1/DL1 associativity	4-way / 8-way
L2 size	256 KB, private
L2 associativity	8-way
L3 size	4 MB, shared
L3 associativity	16-way

III. Naive SPH Algorithm

The SPH algorithm provided simulates the behavior of a fluid represented by a discrete number of particles n. Each of these particles moves according to the equations of Newtonian mechanics, including inter-particle interaction forces. The simulation state contains information about the density, position, velocity, and acceleration of each particle, which is updated in discrete time steps. During each time step, particles interact with neighboring particles within an interaction radius h. The number of particles is determined as a function of h:

$$n = (\lfloor 0.65h^{-1} \rfloor)^2 \tag{1}$$

The movement of each particle is updated based on the forces of interactions. To evaluate this, the algorithm provided needs to update the density of each particle, followed by the force computation to know the acceleration. First, a particle is identified and compared to every other particle in the simulation space to evaluate individual densities in an expensive brute-force manner. The updated density values are used in the force calculations, where the interacting forces are determined by comparing a particle to the entire particle space. Both the density and acceleration routines take advantage of the symmetry in the system, but one clearly sees that this approach would not scale well for large number of particles in the system.

IV. Binning the Search Space

Our first optimization was to implement binning, in an attempt to reduce the number of particle interaction computations. As described in Section III, the loops used to compute the particle density and acceleration are of order $O(n^2)$, as every particle is checked for interactions with every other particle.

We use a spatial partitioning method to approach an algorithm of order $O(n)$. Our partitioning is constructed by overlaying a grid of boxes onto the particle space. Since our SPH algorithm only examines near-field interactions, we compare a pair of particles only when they are at most a distance h away from each other. As we will discuss further in Section IV-A, if we can use this fact to definitively bound the number of boxes that need to be examined, we will approach an algorithm that is of order $O(n + m)$, where m is the number of boxes that we need to loop over.

We assign each box with a single box number, which is computed based on its column and row number as in Equation 2.

$$i = (\text{column} \times \text{numRows}) + \text{row} \tag{2}$$

Using a single sequential numbering allows us to use a single index to access the boxes. The C language can only support two-dimensional

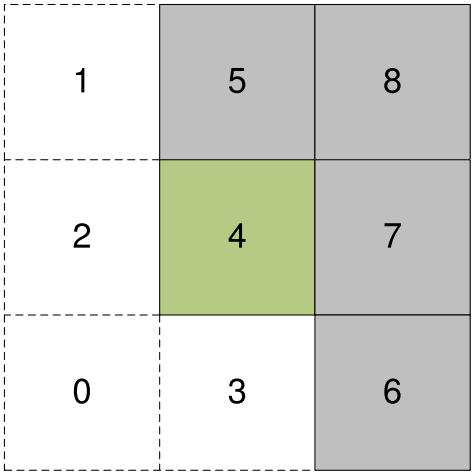

Fig. 1. Example of exploiting grid symmetry to reduce communication with neighbors. Box 4 is the box currently being operated on. Boxes 0-3 have already updated their interactions with box 4 previously. Boxes 5-8 will be visited by box 4, and updated with any interactions that occur.

access through the maintenance of two levels of pointers, which we believed would significantly slow down program performance due to the fact that it would require four instructions for a single access: add, access memory, add, access memory.

Assuming that each box B_i and its immediately neighboring boxes account for all the possible particle interactions the particles contained in B_i, a single box interacts with at most eight other boxes. We can also exploit symmetry as was done in the initial algorithm–described in Section III–to reduce the number of computations and loops during the interaction phase by half.

Figure 1 helps illustrate this technique. Assuming that symmetric calculations occur, then if we are on box B_i, all computations for particles in box B_i with particles in box $B_j, \forall j : j < i$ will have been performed already–when we were examining box $B - j$. The implication here is as follows: the neighboring boxes that must be checked for particles interacting with those in box B_i have indices are greater than i.

A. Selecting a Box Size

Ideally, the box size should ensure that the particles within a box will only have to interact with other particles inside the box. In practice, this is impossible without overlapping boxes, because any particle within the maximum interaction distance h of the box border could interact with particles in adjacent boxes. However, we can use the size of the box to limit the number of boxes beyond that which must be examined. We therefore set a lower bound: all of our boxes must be at least of size $h \times h$, such that in the worst case, a particle can only interact with particles inside boxes that its own box is adjacent to. In practice, as we will see in Section IV-B, floating-point rounding error can create edge cases that we must account for in the box dimension.

B. Floating-Point Behavior

Initially, upon implementing binning, we attempted to perform verification against the reference algorithm by comparing the text output of each frame. What we found was that, despite code correctness, there was an error in the way the particle positions were being calculated. Careful examination led us to the realization that floating-point error can alter the behavior of the SPH algorithm.

One caveat of the IEEE 754 floating-point standard is that, for two floating-point numbers A and B, $A + B \neq B + A$ in certain cases. This can happen as the result of a large shift of the mantissa to align numbers within the floating-point hardware when the exponents are significantly different. While error introduced by shifting can sometimes directly affect the current time step of a simulation, this error can be slowly integrated over many time steps, reducing calculation precision, causing instabilities, or other similar errors.

Inside the `compute_density()` function, we find one such case, seen inside Code Block 1. On Line 1, the value `C` ends up being a number of a very large magnitude–order 10^{10}. On Line 12, while calculating `rho_ij`, `C` is multiplied by the cube of `z`, which is a number between 0 and 1. For the default parameters, values of `z` were of order 10^{-6}. This results in the interesting phenomenon that, when the addition on Lines 13-14 are performed, the order in which the different values of `rho_ij` are added affects the final value of `rho` due to the lack of precision for `rho_ij`.

Code Block 1. Portion of the `compute_density()` function.

```
1  float C = 4 * s->mass / M_PI / h8;
2
3  memset(rho, 0, n*sizeof(float));
4  for (int i = 0; i < n; ++i) {
5      rho[i] += 4 * s->mass / M_PI / h2;
6      for (int j = i+1; j < n; ++j) {
7          float dx = x[2*i+0]-x[2*j+0];
8          float dy = x[2*i+1]-x[2*j+1];
9          float r2 = dx*dx + dy*dy;
10         float z = h2-r2;
11         if (z > 0) {
12             float rho_ij = C*z*z*z;
13             rho[i] += rho_ij;
14             rho[j] += rho_ij;
15         }
16     }
17 }
```

When we bin the particles together using our spatial grid of boxes, we no longer perform computation in the order that we were in the initial naive implementation of SPH, as the particles are now grouped by box. As a result, we saw that our results were slightly off from the reference implementation, though a visual comparison of simulation output shows that the behavior is functionally equivalent.

Initially, we used row-major ordering to address the bins. This significantly increased the number of discrepancies we saw in our visual verification. We found this to be a result of the particle initialization, which, in contrast, uses column-major ordering. Since this initialization order dictates the order in which the particles were accessed, the floating point errors were significant. After 40,000 time steps, the magnitude of the floating-point error was quite noticeable. We switched our bin access routines to column-major ordering as well, and found that the particle dynamics were much closer to the reference implementation. As a result, all subsequent optimizations contain bin accesses performed in column-major order.

Floating-point error also affects the computation of which box–bin–each particle belongs to. As described in Section III, the SPH algorithm defines a 1-by-1 space within which all particles interact. As we mentioned in Section IV-A, we would like to size each bin according to the interaction distance h, where $0 < h \leq 1$. This requires us to perform a floating-point division to determine the column and row that a particle belongs in. We found that, without bounds-checking, the division would take a valid particle and assign it to a bin outside of the simulation space, due to slight rounding errors.

This binning error also manifests itself in a more subtle manner. Even with bounds checking, it is possible that a particle on the boundary between two boxes could be nudged into either box,

dependent on the final rounding. While this doesn't particularly matter for the particle itself, it does make a difference when it comes to considering its interactions with other particles. Using Figure 1 as a reference, imagine two particles: particle A sits on the boundary between box 1 and box 4, and particle B other sits on the boundary between box 4 and box 7. If they are at the same height, and the boxes are of size h, particle nudging could cause us to miss an interaction. If particle A is nudged into box 1, and particle B is nudged into box 7, our theory of only inspecting immediate neighbors no longer holds. To ensure that this never happens, we must enforce a minimum box size of $h + \epsilon$, so that at most one particle of an interacting pair can be nudged. (Naturally, ϵ must be large enough to overcome rounding error.)

C. Performance

Code Block 2 shows our initial binning implementation. At each time step, we take the existing state, allocate a new state for each grid box using dynamic memory–Line 3, and copy over the state to a gridded version. We then compute the acceleration, copy the gridded state back to the original state vector–Lines 5-6, and then perform leapfrogging and verification. At the end of each loop, we free the grid from the heap–Line 9. As one can imagine, the constant re-allocation of memory is costly. We will examine this further in Section V.

Code Block 2. Outer loop of the binned SPH program.

```
1  for (int frame = 1; frame < nframes; ++frame) {
2     for (int i = 0; i < npframe; ++i) {
3        grid = buildGrid(state, gridDimension);
4        compute_accel(grid, &params, gridDimension);
5        readStateFromGrid(state, grid,
6              gridDimension);
7        leapfrog_step(state, dt);
8        check_state(state);
9        clearGrid(grid, gridDimension);
10    }
11    write_frame_data(fp, (int) n, state->x, NULL);
12 }
```

Upon executing this version, we find significant slowdowns from the original code. While this version certainly scales better for larger numbers of particles, we do quite poorly for small particle counts, performing 5x slower for the default parameters. For brevity, we omit these results from any figures.

V. Performance Optimizations

A. Dynamic Memory Usage

Our first optimization to the binned code is to eliminate the performance bottleneck from the multiple allocations (Section IV-C). We performed this in two steps to evaluate their individual contributions. The first version, *SingleAlloc*, only allocates memory for the boxes once during program execution. In order to do this, we allocate enough space for each box such that it can hold all of the particles in the simulation if they happened to be assigned to the box.

As one can see, this is a costly approach, and one that will scale extremely poorly. This is because the number of boxes is directly related to the interaction distance h, which also controls the number of particles created. We are creating a large amount of memory of which most of it will never be utilized. However, at smaller particle counts, it will allow us to observe the effect of removing allocations and deallocations.

One other significant optimization is the elimination of a double `for` loop within our code. Our initial code (*Binned*) contained an initial loop that counted the number of particles assigned to each box, and then a second loop, which was a nested `for` loop to assign

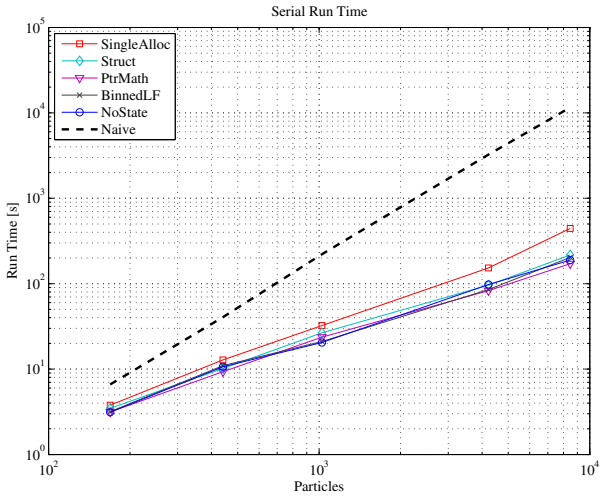

Fig. 2. Time spent executing several serial versions of the SPH code, sweeping over the number of particles simulated.

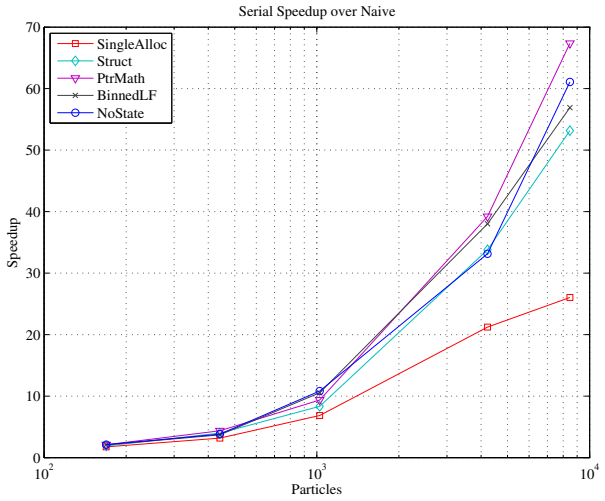

Fig. 3. Speedup of the serial code over the initial algorithm.

particles to bins. This effectively kept part of our code operating at $O(n^2)$, since the number of boxes m and the number of particles n are both directly related to h. With the statically-allocated memory, there is no longer a need to perform an initial count. Therefore, we can count and assign to boxes within a single loop. Furthermore, we can remove the loop nesting, because the integration of counting also allows us to determine how many particles are already in each box when we copy over our current particle.

Figures 2 and 3 show how *SingleAlloc* performs compared to the initial SPH algorithm, *Naive*. We see significant speedups from reducing the allocations and eliminating the $O(n^2)$ calculations, unlike for *Binned*.

As a next step, we improved our memory allocations. Our second version of optimized memory allocation, *Struct*, creates the `particle_t` data structure, shown in Code Block 3. This structure breaks apart the original vectors and creates 36-byte data elements for each particle. While we do lose some locality that we achieved by only traversing over a subset of the particle properties in each function, we still maintain locality within boxes, as each particle's entire data is now kept in a contiguous region. To exploit memory compaction, we only allocate a single array of n `particle_t`

blocks, and maintain an array of offsets to determine where each box's data begins.

Code Block 3. The `particle_t` data structure.

```
typedef struct particle_t {
    float rho;
    float x;
    float y;
    float vxh;
    float vyh;
    float vx;
    float vy;
    float ax;
    float ay;
} particle_t;
```

Figures 3 and 4 allow us to get a better picture of exactly how *Struct* improves over *SingleAlloc*. As we see, the data compaction provides a significant speedup boost for larger particle sizes, confirming that, as we expected, the data structures inside *SingleAlloc* were becoming excessively large for the caches.

B. Pointer Operations

We can also look to other sources of inefficiency within our program. We find that several of our functions contain inefficient nested array indexing. An example of this can be seen in Code Block 4. This block contains a portion of the code within a loop that reads each particle within the state and uses nested indexing to determine the write location.

Similar to the two-dimensional indexing discussed in Section IV, each of these accesses will take five operations: add, access memory, add, access memory, add. Since there are two arrays, and both the inner and outer index are variables, there is little hope for the compiler to optimize this to a static address.

Code Block 4. Portion of the `buildGrid()` function.

```
assignedBox = (assignedColumn * numGridRows)
      + assignedRow;

boxes[boxEnd[assignedBox]].rho =
      rho[particle];
boxes[boxEnd[assignedBox]].x =
      x[(2 * particle) + 0];
boxes[boxEnd[assignedBox]].y =
      x[(2 * particle) + 1];
boxes[boxEnd[assignedBox]].vxh =
      vh[(2 * particle) + 0];
boxes[boxEnd[assignedBox]].vyh =
      vh[(2 * particle) + 1];
boxes[boxEnd[assignedBox]].vx =
      v[(2 * particle) + 0];
boxes[boxEnd[assignedBox]].vy =
      v[(2 * particle) + 1];
boxes[boxEnd[assignedBox]].ax =
      a[(2 * particle) + 0];
boxes[boxEnd[assignedBox]].ay =
      a[(2 * particle) + 1];
boxEnd[assignedBox] += 1;
```

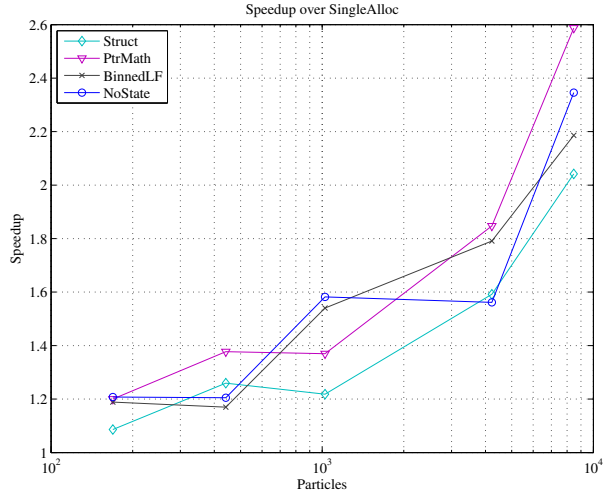

Fig. 4. Speedup of improved serial versions of SPH over the *SingleAlloc* version.

Instead, we choose to use pointer arithmetic to replace this code—this new version of the program will be called *PtrMath*. As seen in Code Block 5, we can see that we only need to perform an increment to determine the destination pointer–Lines 3-4. For each variable being written, we only need to do a single addition on top of the initial assignment of `assignedBoxPtr`. Notice that we can also use pointer arithmetic to eliminate the need to perform multiplies every time the particle variable changes, replacing them with potentially faster integer additions. Even if the latency of the multiply is identical to that of the addition instruction, processors typically contain more addition units.

Code Block 5. Portion of the `buildGrid()` function implemented using pointer arithmetic.

```
assignedBox = (assignedColumn * numGridRows)
      + assignedRow;
particle_t *assignedBoxPtr =
      boxEnd[assignedBox]++;

assignedBoxPtr->vxh = *(vh++);
assignedBoxPtr->vyh = *(vh++);
assignedBoxPtr->vx = *(v++);
assignedBoxPtr->vy = *(v++);
assignedBoxPtr->ax = *(a++);
assignedBoxPtr->ay = *(a++);
assignedBoxPtr->x = *(x++);
assignedBoxPtr->y = *(x++);
assignedBoxPtr->rho = *(rho++);
```

We repeat this in all places where we find such operations. What we find is that while this is quite effective for nested indexing, the compiler does indeed optimize single indexing of arrays when accessing data structures; as such, we only implement pointer arithmetic on an as-needed basis.

Figures 3 and 4 show that the *PtrMath* algorithm is able to provide a consistent performance improvement over the *Struct* version. This is especially clear in Figure 4.

VI. Eliminating the State Vector

While our previous optimizations have provided us with significant performance improvements, we have not modified the outer loop of the program seen in Code Block 2. This means that the leapfrog code is still un-binned, and that we revert to the original `sim_state_t` structure at every frame. While the leapfrogging portion of the code looks at each particle independently, and therefore doesn't require any modifications for performance, it will be advantageous for parallelization to bin the code as well (*BinnedLF*). This is because (a) we can take advantage of locality by assigning bins to cores that they've already executed on when computing the particle acceleration, and (b) we can also exploit any load-balancing mechanisms on the leapfrog code.

What we find is that if we do bin leapfrogging, we effectively eliminate any reason to use the state vector, aside from performing the particle check and writing the data. If we also rewrite those functions, we can do away with the state vector entirely after the initial grid allocation.

There is one issue that must be resolved. Previously, as part of our having to copy to the state vector and rebuilding the grid, we were re-partitioning the particles every cycle, ensuring that any particles that migrated due to their interactions were regrouped into the appropriate bin. To maintain the correctness of our algorithm, in searching only immediately neighboring boxes, we must still re-grid every cycle. To reduce the copying overhead required, we simply allocate two grid arrays initially. At the end of each time step, we read from one of the arrays and reshuffle the particles into the second array, eliminating the need for any in-place sort. We then exchange the pointers to the two grids, such that our copy is now the one used for the next time step. We will refer to this final version of our serial code as *NoState*.

This two-grid arrangement provides us with the opportunity for a number of enhancements. First, as we have eliminated one of two copies per time step, we can use the performance delta to determine whether it will be wise to re-grid less frequently. If performance improves significantly, the copies are therefore taking a large chunk of time, and reducing the number of copies per frame will alleviate that bottleneck significantly. Another advantage is that we can potentially parallelize the I/O along with calculations for the next time step, since the data that needs to be printed will remain intact in the old grid. We may also find usefulness in parallelizing the state checking function, and may be able to do further thread-based parallelization.

Figures 3 and 4 illustrate the benefits of both *BinnedLF* and *NoState*. While we expected *BinnedLF* to have no tangible performance impact, we see that for larger particle counts, it actually outdoes *PtrMath*. This might be the result of improved loop unrolling on the part of the compiler, as a result of our binning structure. We also see in Figure 4 that *NoState* underperforms the other methods slightly, much to our surprise. We expect this to be the result of simulation environment noise, as we do not see a clear reason why it should perform worse than *BinnedLF*. As such, we still use *NoState* as our baseline for parallelization. However, we conclude that reducing the number of re-grid operations is not a worthwhile target for optimization at this time, as the reduced copy seems to make a negligible difference in performance.

VII. Parallelizing Interactions

We take the *NoState* version of the SPH algorithm and add in OpenMP pragmas to parallelize the code across multiple cores. Initially, in the *Static* version of the parallel program, we parallelized the outermost `for` loops within the acceleration and density computation functions, as well as within all of the leapfrog functions. Aside from pragma insertion, we also made significant modifications to prevent race conditions during memory writes, and to balance out the amount of work between the various threads. These optimizations will be discussed in the subsequent sections.

A. Thread Synchronization

As seen in Figure 1, using our algorithm, a box needs to communicate with its neighbors to the right of it, as well as the box directly above it. However, if we parallelize this work, we find that there may exist rare conditions in which two threads attempt to update the same box concurrently. For example, in Figure 1, if thread A is working on box 4, and thread B is working on box 7, we may see a conflict, as thread A, when accessing its neighbors, will update the `rho` or the `a` variables for density and acceleration, respectively, of box 7. At the same time, thread B will also be updating those values as part of the calculations for box 7.

We eliminate this by using column striping. The OpenMP parallel `for` construct provides an implicit barrier at the end of the loop. We take the density and acceleration loops, and partition them into two

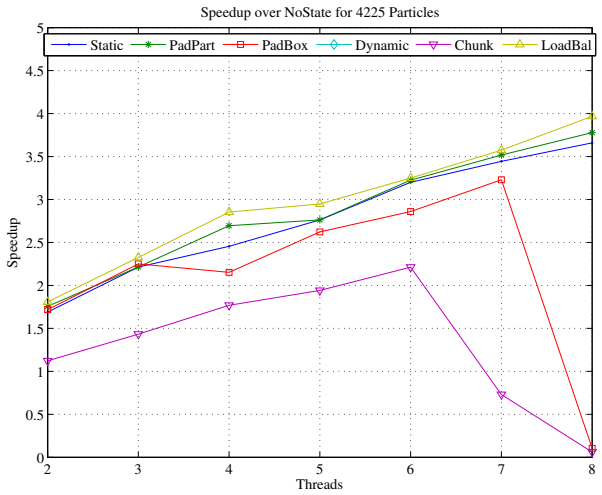

Fig. 5. Parallel speedup over the *NoState* serial program for increasing numbers of threads, where $h = 1 \times 10^{-2}$.

`for` loops. We partition the space by columns amongst the different cores, as opposed to at a finer granularity. The first loop calculates all of the even columns. Since a box can only update its neighbors, either they will be in the same column, which will be owned by the same thread (and therefore done in sequential order), or they will be in a column to the right, which will not be allowed to update its densities or accelerations at this time. Once the first loop completes all even columns, a second `for` loop calculates all odd columns. As a barrier exists between these two phases, there can be no race conditions when writing to the particle values.

B. Basic Performance

Figure 5 shows how *Static* scales over *NoState* as we increase the number of threads. We see that speedup scales linearly as we increase the number of threads, which is a good sign considering that this is only a moderate number of particles. When dealing with only 169 particles, we saw relatively flat speedups (approximately 1.2x), so it is encouraging to see such linear growth. However, it is important

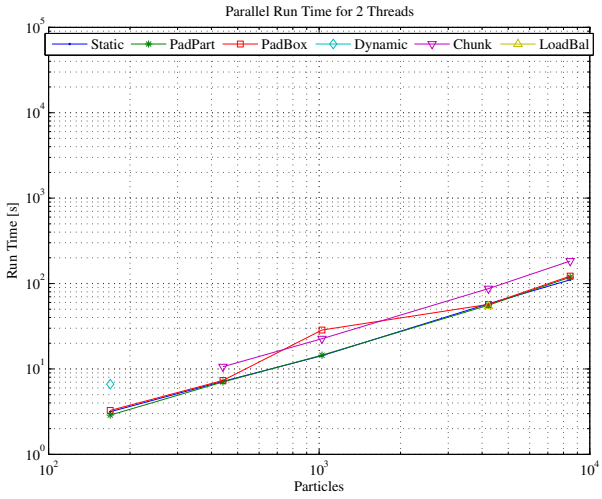

Fig. 6. Parallel runtime for various versions of the program, invoking 2 threads. (While 4 and 8 threads seem to show similar trends, irregularities in the data, likely due to simulation environment issues, prevented their inclusion.)

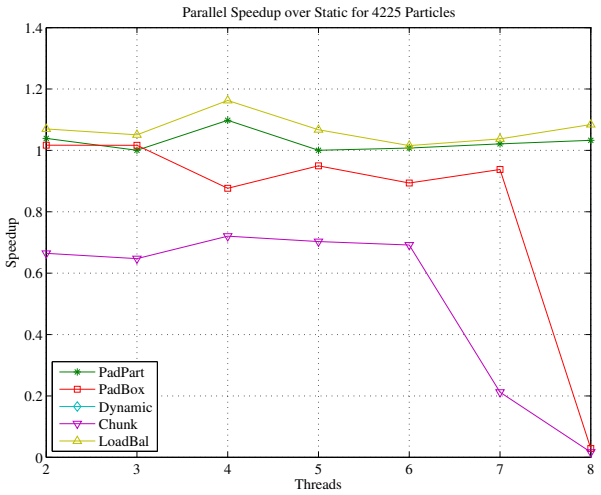

Fig. 7. Speedup over the baseline parallel code, *Static*, for increasing numbers of threads, where $h = 1 \times 10^{-2}$.

to note that the speedup still falls quite short of the ideal speedup. While it is difficult to pinpoint where this is occurring, much of this is likely due to the continual barrier synchronization that implicitly occurs at the end of each parallel `for` loop. We also probably incur penalties for cache coherency activity, as well as for sub-optimal work distribution. We will examine these optimizations in Sections VII-C and VIII, respectively.

We can also see from Figure 6 that, for 2 threads, run time increases linearly as the number of particles increases. This confirms that our algorithm successfully performs in $O(n)$ time. While we see similar trends in the data for 4 and 8 threads, we do not show these figures due to some anomalies in the data, the likely result of simulation environment issues.

C. Inter-Thread Communications

A critical factor in improving performance in parallel applications is the reduction of inter-thread communications. In the OpenMP model, these communications are usually done implicitly, through the use of shared memory. An example of this is the scenario described in Section VII-A, where two threads can both update the same value, using synchronization, so that one of the threads can use it to perform subsequent computation (in this case, the new particle position and velocity vectors).

One caveat of the shared memory model is the problem of false sharing. If multiple variables are saved within the same cache line, and two threads, running on two separate cores, each access a different variable on this cache line, coherence messages will cause the line to ping-pong between the two cores on a write. This is because coherence protocols work on a line granularity, despite the possibility of using independent data which will not cause any memory inconsistencies if concurrently written to.

To avoid the false sharing pitfall, we make two improvements to our parallel program. For both of these improvements, we align our array of particles to a cache line boundary. For the first, *PadPart*, we then pad the size of the `particle_t` data structure, which is 36 bytes, to a 64-byte size, so that each particle takes up a full cache line. This means that, in the event of writing to a neighboring box's particle on another core, only that particle's data will be migrated over to the writing core's local cache. This reduces the possibility of ping-ponging data, as the barrier synchronization will ensure that the other thread will not be requesting the line at the same time.

One issue with the *PadPart* approach is that, while we are better protected against false sharing across threads when writing to neighboring cells, this likely isn't much of an issue in practice, due to the inherent synchronization. This means that we are being overly cautious with our implementation. What makes more sense is to pad each box, and to keep the particles within a box contiguous (*PadBox*). This still ensures that adjoining boxes assigned to different cores will not experience false sharing (which is an important issue in leapfrogging, where there is no synchronization). However, the computation for pointer offsets becomes significantly more complex. Notably, we must now maintain a vector of pointers to the heads of each box. This also reduces the effectiveness of pointer arithmetic, which could previously increment across boxes.

Figure 7 best shows the results of both *PadPart* and *PadBox*. We see that the improvements are relatively minor for *PadPart*, though they never perform worse than *Static*. Interestingly, the math required to compute the box start locations for *PadBox* is not compensated for by any performance improvements for 4225 particles, and in fact brings down performance when at least 4 threads are invoked. Due to simulation issues and time restrictions, it remains to be seen whether *PadBox* will do better for large particle sizes. However, based on these results, we make subsequent improvements off of the *PadPart* version of the algorithm.

VIII. Core Work Distribution

Another critical factor in the parallelization of the SPH program is the way in which we divide the work amongst the various threads. Figure 8 shows four ways in which we attempted data division amongst the threads.

A. OpenMP Methods

Our initial code, *Static*, uses static partitioning of the data amongst threads. Since we consider a column to be an atomic unit for the purposes of work distribution, static partitioning, as seen in Figure 8a, will take alternating columns and distribute them amongst the cores. In our case, since we look at odd and even columns separately, we will stripe all even columns across threads first, and then all odd columns. This will be true of all of the OpenMP pragma-based distribution methods.

The second version, *Dynamic*, uses dynamic partitioning. OpenMP tries to generate a dynamic job queue, which assigns columns to threads as they complete (see Figure 8b). This means that the assignment of columns has the potential to change from execution to execution, in an attempt to even out the workload on each core. In practice, this does not perform very well, and can be much better achieved through manual implementation of the dynamic queue.

The third form of OpenMP distribution that we explored is chunk-based partitioning (*Chunk*). Here, we partition the space of c columns into c/p contiguous segments, where p is the number of threads, as seen in Figure 8c. The advantage of chunk-based segmentation is that we only have one boundary of inter-core communication, which is much better than the previous implementations. However, this comes at the cost of inflexibility in the work distribution. Furthermore, for an imbalanced distribution of particles within the space, this form of allocation will likely do the worst in terms of distributing an equal number of particles to each core, since this partitioning does not break up particle clusters well.

Unfortunately, due to issues with the simulation environment, we were unable to get final results for the *Dynamic* configuration. Figure 7 does, however, show the improvement for the *Chunk* configuration. We see a notable dip in performance when compared to the performance of the baseline *PadPart* version. This suggests

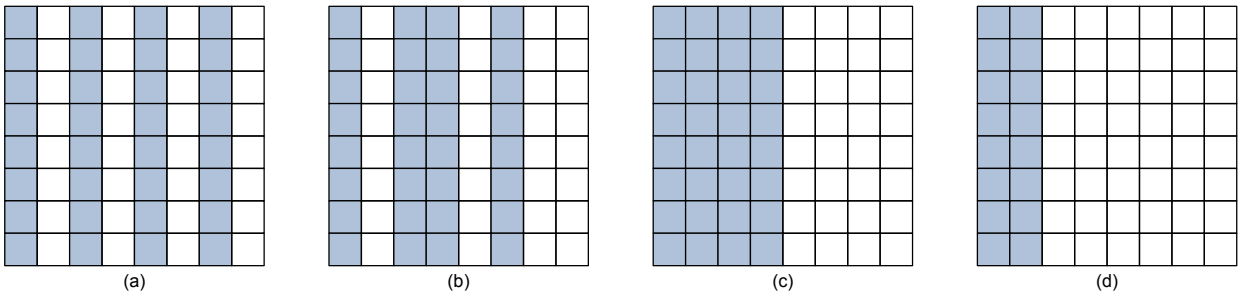

Fig. 8. Various work distribution algorithms for dividing the grid columns amongst the various threads. The blue boxes are assigned to thread A, while the white boxes are assigned to thread B. The assignment methods are: (a) *Static*, (b) *Dynamic*, (c) *Chunk*, and (d) *LoadBal*.

that, while communication may indeed be reduced, the imbalance in the number of particles prevents *Chunk* from improving on overall performance.

B. Explicit Load Balancing

In an attempt to combine the low-communication design of *Chunk* and merge it with an even particle distribution, we develop our own load balancing implementation, *LoadBal*. When we build the grid at each time step, we are already maintaining a count of particles within each box. We extend this to also maintain a count of particles that belongs in each column of boxes.

Code Block 6. The `partitionJobs()` function.

```
1  particlesPerThread = numParticles / numThreads;
2  jobList[0] = 0;
3
4  for(unsigned int currentThread = 1; currentThread
5        < numThreads; ++currentThread)
6  {
7     while((threadSum < particlesPerThread) &&
8           (columnIndex < numGridColumns))
9     {
10       threadSum += columnCount[columnIndex++];
11    }
12
13    if((threadSum - particlesPerThread) >
14          (columnCount[columnIndex - 1] / 2))
15    {
16       threadSum -= columnCount[--columnIndex];
17    }
18
19    jobList[currentThread] = columnIndex *
20       numGridRows;
21    numParticlesLeft -= threadSum;
22    particlesPerThread = numParticlesLeft /
23       (numThreads - currentThread);
24    threadSum = 0;
25 }
```

We then use a new function, `partitionJobs()`, to examine the column counts and partition the jobs amongst the threads (see Code Block 6). We calculate the ideal number of particles x to be assigned per thread (Line 1), and then keep assigning columns to the first thread until we assign at least x particles (Lines 7-11). We then determine whether to keep the last column, based on whether the number of particles was closer to x before or after it was added to the thread's queue (Lines 13-17).

At this point, we decide to recalculate x, using the number of remaining particles and yet-to-be-assigned threads (Lines 21-23). The reason we do this is because removing a number of particles per thread x', where $x' \approx x$, results in a small difference ϵ. If we do not recalculate the averages, then the last thread $p-1$ will be assigned the following particles:

$$x_{p-1} = n - \frac{n \times (p-1)}{p} - \sum_{i=0}^{p-2} \epsilon_i \tag{3}$$

Depending on the magnitude of the ϵ values, this could add up to a large load imbalance for the last thread, especially since our column-based assignment of particles is relatively coarse-grained. Our hope is that by recalculating the average after assigning each thread, the values of ϵ will be diminished somewhat.

The overall goal of such partitioning is to achieve the distribution illustrated in Figure 8d, where we maintain reduced communication while being cognizant of the particle distribution.

Figure 7 shows the results of using our custom load balancing algorithm. We find that the *LoadBal* algorithm slightly outperforms *PadPart* across the board. As a result, we use this as the final version of our code.

IX. Future Work

Unfortunately, we were unable to get to several optimizations which we had hoped to integrate into our code. We believe this will contribute significantly to improved parallel performance, though it remains to be seen if these would be enough to approach ideal speedup.

The first idea is the implementation of an *eviction queue*. Instead of re-gridding every time step, we wait several time steps before doing so. In the meantime, particles that leave the box that they are currently in must be accommodated in some way. If we size our box dimensions to be large enough, the number of evicted particles is expected to be low. A linked list could be used to hold these particles. A box, when updating, would now not only check its own particles and its neighbors, but would also check all particles in the eviction queue.

Another implementation is the use of *quadtrees* or *Hilbert curves* to bring neighboring data into contiguous memory locations. We believe that this will provide a significant improvement to access locality. Currently, it is disadvantageous for us to use large box sizes, as we do not want to compare a large number of particles and incur cache misses due to a lack of spatial locality with neighboring boxes. However, these algorithms help achieve that locality, making larger boxes a more viable option.

Several other optimizations may also help our performance. We would like to investigate *flexible partitioning* of the simulation space, so that boxes of varying sizes can be used to store a constant number of particles. It would also be quite beneficial to explore several advanced methods of *load balancing*. Finally, we would like to sweep over the interaction radius h, to find the ideal box size.

X. Conclusion

We improve upon a baseline smoothed-particle hydrodynamics algorithm to bring it down from $O(n^2)$ time to $O(n)$ time for n particles. We do so through a number of optimizations, which allow us to achieve up to a 67x speedup for large particle sizes over the baseline serial implementation. From this, we construct a parallel version of the code using OpenMP. We can achieve a 3.8x speedup over our optimized serial version when executing 8 threads. Furthermore, we show linear speedups as both the number of threads and the number of particles increase.

References

[1] Intel Corp. "Intel Xeon Processor 5500 Series Specification Update." April 2011.

[2] Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. "Building Workload Characterization Tools with Valgrind." In *Proc. of the IEEE Int'l. Symp. on Workload Characterization*, October 2006.