Abstract:

**Introduction:**

The code to be profiled, optimized and parallelized is one currently used in my graduate research. The idea is fairly simple, we want a stochastic simulation of dust particles ‘flowing’ through a simple slit filter. The simulation is agent based, creating a dust\_grain object for each particle, a vector of which is used in the dust\_list object that processes them at each timestep. The location of each particle is held internally, but also externally in the world object which primarily consists of an integer matrix of particle id’s. The main object starts the simulation off, with various versions of the code existing for different operating systems and parameter input methods. Originally (at the start of the semester) the code was last edited some ten years ago by an undergraduate working for my current advisor. Some other students and post-docs have edited various portions, but with very little documentation or commenting. Further, the original version had a fixed simulation (world) size of 500 by 500, a visualization, only Windows support, and no methods for handling the merging of particles or their sticking to the filter. Prior to the HPC 1 mid-term exam the project was developed to allow for no visualization, an arbitrary simulation (world) size, robust output file system organization, and Linux support.

**Port to Unix:**

The changes required to allow the program to run on Linux we’re fairly small, simply changing swapping the relevant includes and file system functions. As well as creating a Makefile. Further editing was required to get the program to run on the CCR, as there was much difficulty in getting the program there to read in parameters from a file. The solution to this problem was to have parameters passed in command line through int argc and char \*argv[].

**Framework changes:**

An effort was made to allow for visualization of arbitrary world sizes, but OpenGL (GLUT) proved to be rather difficult to handle. During this attempt the co-ordinate system was normalized, as previously some functions called the vertical component the ‘y’ direction, and others would call it the ‘x’ direction. Current code uses the same standard as OpenGL, which is that ‘y’ is vertical on the screen and ‘x’ is horizontal.

An initial attempt at allowing for dust to impact and stick to the filter was pursued at the request of the advising professor. This original version of filter sticking was proven to be incredibly inefficient, and so when requested to add a ‘similar’ function to handle dust merging, the new more efficient logic for checking if two particles collided was also used to replace the function which checked if a particle impacted a filter. While some functions did originally exist which lent themselves to this effort (such as one called merge\_g1\_to\_g2) none of them we’re logically tied together or even called at all. Further analysis and explanation will be follow in the report.

**Initial Profiling:**

Profiling of serial code was performed primarily with gprof, as well as the timer.c methods provided by the course instructor. Prior to implementation of dust merging the dust\_list::filterImpact(int, int, int, int) routine took upwards of 25% of total runtime (1037.84s for 10k ptcl, 65.18s for 1k ptcl), seemingly regardless of number of particles. The next most time consuming function showed more variation with the number of particles present, specifically dust\_grain::getXatc(int) went from ~14% (600.61s 10k ptcl) to ~8% (9.37s 1k ptcl). The reason for this excessive time use was that the filterImpact function would check if each cell occupied by the particle in question was going to move into a location occupied by a filter. Initially the function would call getXatc thousands of times, as the filter is extended across the horizontal. The y position of the filter was found once, then incremented each time a full horizontal sweep completed, up to the depth of the filter.

**Initial Serial Optimization:**

Simply loading in the filter and dust vectors to the filterImpact function reduced the time in the function to 17% (1161.01s for 10k ptcl) and changed the second most time consuming function to that of the iterator over the dust list, and made the function dust\_grain::getYatc(int) take to about 9% (602.23s for 10k ptcl). We may note that getYatc became more heavily used than getXatc, since the function no longer had to call getXatc thousands of times.

An initial attempt to check for filter impacts by using a partially complete legacy function dust\_list::getCollidingGrain(int, int, int) reduced the time used to complete the analogous task to 0.05% but broke the codes functionality. While erroneous, the results still aided in revealing which functions were requiring the most computation time. Data from these trials showed dust\_list::checkPoreFilled() taking the most time at 15% (413.74s 10k ptcl) with dust\_list::isOpen(int, int) taking the next longest at 10% (269.86s for 10k ptcl).

**Primary Serial Optimization:**

Since it was clear that the implementation of filterImpact was incredibly inefficient and the research project would require the use of the getCollidingGrain to implement dust merging, a major effort was made to enable dust merging and proper use of getCollidingGrain to check for filter impacts. Upon completion of this effort the new highest time consuming function is dust\_grain::getID() at 28% (38.48s for 10k ptcl). The comparison in efficiency between the two versions is difficult, since the merging of dust grains fundamentally changes the problem and processing required at each time step. Thus, profile comparisons will only be drawn between versions in the same problem set.

The new most expensive function getID is called extremely frequently to actualize the new method of checking if a particle has impacted on a filter or another particle. A unique ID for each particle was required in the new framework since the process of merging would change their location in the dustList. This fact caused for an attempt to also add flags to the dust grains which would allow for parallel movement of grains within the dustList if they had no chance of intersecting. However, this parallel implementation was dropped in favor of a domain approach which is outlined later. One possible optimization method beyond parallelization could be to make sure that the dustList is always ordered from lowest to highest ID by a sorting function, and/or by use of a hash table.

The next most expensive function in the new serial regime is dust\_list::getVecLocByID(int) at 20% (27.81s for 10k ptcl), which requires the calling of getID() up to 10k times if the particle in question is the last in the dust list. Some other function may be preferred, for example, attempting to always use the vector location during movement unless the ID is explicitly required. Initial use of this function was due to time constraints in project preparation.

Running the simulation for ten times longer (20000 time steps) showed a decrease in the weight of the previously mentioned functions, changing getID to 21% (64.41s total) and getVecLocByID to 18% (55.63s). It is worth noting that the time spent in these functions had only doubled for an order of magnitude increase in simulation time. While this is partially due to a quirk (to be fixed later) where particles with a very large size would not move and therefore not require much processing. These particles would continuously absorb other smaller particles with hit it, much as the filter builds a layer of stuck particles. This is because a particles maximum velocity is inversely proportional to its size. In the future an implementation of grain splitting will be implemented to resolve these ‘dead’ particles.

The number of times getVecLocByID is called can be drastically reduce by saving the vector index to the order vector and iterating over that, but we would then need to call getID one additional time during each particle movement instead. While getID is already the most used function, each call should be significantly shorter in duration than getVecLocByID. An attempt to make this change was made and it indeed showed an increase in performance. Not only was time spent in getVecLocByID decreased to 12%, the fraction of time spent in getID also decreased to 15%.

However, the total time spent in these functions was very similar to that in the previous implementation, and yet the entire simulation completed close to one minute faster at an average of 240s. The variation in total simulation time for several runs with the same parameters was less than 10s. Thus the change of order containing id's to vector locations indeed yielded a significant speedup.

**MPI Configuration:**

A significant amount of time was spent in configuring the program to properly compile with Intel MPI, the final method required the following. The Makefile required CXX = mpiicpc, as specifying the compiler without defining CXX would be misunderstood and cause the code to be compiled by the default g++. Further, timer.c had to be changed to timer.cpp with the removal of the extern”C” prefix at each function and the addition of the explicit declaration of struct timespec at every instance of use. Finally, on the CCR the modules intel and intel-mpi must be loaded and the export of the MPI library defined. At this point much of the coding and debugging was done on a personal Linux computer, but with the amount of time spent on the CCR increasing an effort was made to streamline some of these configurations. The use of several alias', module loads and exports on login in .bashrc was added.

**MPI Debugging:**

Allinea DDT was to be used in debugging the MPI code, and took a fair amount of configuration. To interact with the DDT GUI the client first had to have X-win32 installed and PuTTY configured to add the -X flag. This enabled the running of ddt on a debug node, but the simulation was not yet properly running through ddt. At first the export DDT\_MPIRUN=srun was used, but ddt would warn that an integer p processes is required, which it also specifies should be input in the GUI box and not with srun arguments. While the CCR website suggests not using mpirun or mpiexec.hydra, an attempt to use mpirun with DDT was made successfully. The final solution was to leave out this export entirely and choose SLURM(generic) via the DDT GUI.

The next issue was that the ddt software was unable to recognize the codes .cpp files as sources, even when the project directory was added to the source directory list. Each .cpp was added to sources manually, but ddt would warn that breakpoints would not work within these files, as they were still not legitimately recognized sources. Several attempts were made to ensure that the -g compiler flag was correctly used. The clue which eventually led to the solution of the problem was that the only source file properly found was timer.cpp, which in the Makefile was the only .cpp explicitly listed in the build command for the final program. The issue was that the other files were first built as objects which were then used to build the executable and ddt was not able to recognize the .o files (or the .cpp files they depended) on as sources. The Makefile was edited to build directly from the .cpp files and ddt successfully recognized them and enabled the start of MPI implementation and debugging.

**Parallel Pseudo Code:**

After outlining several options for MPI parallelization in pseudo code, the decision was made to begin implementing the originally proposed domain parallelization. A rough outline of the implementation is as follows:

1. Initially each non-root process spins up a world as in the serial version.
2. Root process will create a new superstructure, universe.
3. Time step taken until defined limit is reached.
   1. Universe/root tells worlds to move next particle until no world has unhandled particles.
      1. If a particle would leave current worldS, worldS messages particle info to root.
      2. Root calculates worldR that particle would move into and messages worldR.
         1. Handling of particles partially in two worlds is handled by universe.
      3. All potential moves are actually enacted.

**Parallel Implementation:**

Time constraints made implementation of the entire parallelization as outlined impossible, so an effort was made to at least demonstrate some use of MPI in the code which would lend itself to the proposed method when it is fully implemented in the future. The simple problem chosen would not yet create universe as intended, and would have root handle its own world. At each time step all worlds communicate to root the number of particles which would leave its boundary. Root then prints this information to an output file.

The first major error encountered when enacting this parallelization is that while each thread does create a world, they are not unique. This error was discovered by the fact that the number of particles moving out of the world is always equal across all threads. A solution to this problem might be to create the vector of worlds (myUniverse) and have different threads explicitly handle their given world. For now an attempt was made to simply create a vector of worlds as opposed to a well-structured object which would allow for efficient passing of dust particles in a contiguous manor. Indeed, by making a vector of pointers to different world objects, the running of logic on each world can be handled in the main function exactly as it had been. The variable myWorld is set to the rank element of myUniverse, instead of each thread owning the same pointer to the same world. Thus the identical results of particles exiting each world is resolved.

The previous solution created a new problem, now only the root world is having any particles exit it. This appears to be occurring even though the world controlled by a worker thread has elements in its exitedWorld vector. An attempt to resolve it will be to explicitly use myUniverse[iam] instead of saving that pointer to myWorld, but was unsuccessful. Further evidence that this is not the cause of the issue can be seen by setting the value of myXMax to the input parameter\*(iam+1), which was correctly reflected in each processes myWorld. By setting numDustTranser = iam it was verified that the issue was in MPI\_Gather itself, as in this case the output was all zeros, the initialized value. Finally, it was noted that the fifth parameter in MPI\_Gather was set to the number of processes (the size of the Gathered vector), when it should have been the size of the vector it received from each process. With this value rectified the code is able to properly output the number of particles which left each threads world at every time step.

**Load Balancing Considerations:**

Within the general pseudocode that was previously outlined there are many areas which should be studied in detail once the core MPI version of the code is complete. The first among these is load balancing, particularly with respect to the presence of filters. When scaling data is collected it will be important to investigate the impact of processing time required by worlds which contain a portion of the filter and those which do not. On this subject, it is worth noting that the periodic boundary conditions and existence of a filter in each world for the current code may produce results very similar to those of an extremely wide serial simulation and/or the proposed final parallel version. The deviation in results between these three scenarios is worth investigation.

**Conclusion:**

Various resources such as IBM Knowledge Center can be used to supplement the course lectures to provide the full parallel MPI capability as previously outlined. Specifically, these resources provide information to handle passing of advanced structures such as an entire dust\_grain object. Unfortunately, a poor choice in starting/focal points on the researcher’s part prevented the proposal from being implemented in its entirety. Further, without a parallel code which fully mirrors the functionality of the serial version, a scaling study could not be performed. While this report lacks significant portions of the requested content, it may still serve to demonstrate the authors understanding of course material. Logic programming is seen in the routines added to the serial version, concepts and consideration to efficiency profiling were also applied in the serial version, with the current MPI implementation indicating at least some knowledge of the interface and its nuances. Finally, the establishment of a connection to the CCR’s debug node and use of DDT were absolutely paramount in performing this work, accelerating coding time significantly. Work will continue on this effort until the outlined pseudo code is implemented with full functionality.