CPU Optimization of Particle Deposition in PIC Simulation Code

by

David A. Rimel

B.S., University Of Colorado Boulder, 2016

Defense Date: 10/25/16

Advisor: Prof. John Cary, physics

Honors Committee Memeber: Prof. Paul Beale, physics

Other Committe Memebers:

Dr. Greg Werner, Physics

Dr. Rhonda Hoenigman, Computer Science

David A. Rimel, (B.S. , Engineering Physics)

CPU Optimization of Particle Deposition in PIC Simulation Code

Thesis directed by Prof. John Cary and Dr. Greg Werner

Abstract

This thesis discusses how to optimize computational physics software for speed through maximizing the use of novel architectural features of current CPUs. Specifically, the optimization of the Particle-In-Cell (PIC) algorithm is considered. The PIC algorithm is widely use in the study of plasmas, rarified gases, fluid dynamics, and gravitational dynamics. For this algorithm, the main performance bottleneck is the deposition of charge onto the grid. The goals of the optimizations described in this paper were to maximize cache reuse to overcome memory bandwidth limitations and improve data processing speeds by taking advantage of multithreading and vector instructions. The particular techniques, discussed in detail in the paper, were sorting particles into tiles, operating on tiles in a manner that prevents race conditions, separating out the vectorizable operations (also known as strip mining), and sorting particles based on cells. Performance analysis results are given at each step of the optimization. An overall improvement of a factor of 20 in computational speed was obtained.

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

Central Processing Units (CPUs) utilize novel architectures in order to increase their performance. CPU performance gains have been achieved through increasing the floating point operations per second (FLOPS) of a CPU by introducing multiple levels of parallelism, such as Single Instruction Multiple Data (SIMD) and multi-core architecture. CPUs are also able to increase FLOPS by mitigating bandwidth limitations through caching. In order to receive the full performance speed up offered by current CPU architectures, computational software needs to be modified. The goal of this paper is to discuss what sorts of modifications allow one to better utilize the performance possible with current CPUs.

There has already been much work in this area, for many different algorithms, including Particle-In-Cell (PIC) simulations [4] (the subject of this paper), as well as Monte Carlo simulations [7] and fluid simulations [4]. Computational software is additionally being ported to completely new computational devices, such as Graphics Processing Units (GPUs) [2]. However, this paper will be restricted to modifications needed for current CPU architectures; specifically, GPU coding will not be discussed.

The Particle-In-Cell algorithm provides a method solving for the motion of particles self-consistently in the fields that they generate. PIC has be extensively studied due to its ability to simulate a wide variety of physics, such as gravitational dynamics, fluid dynamics, and plasma phenomena [5]. There are four steps in PIC: (1) interpolation of the fields to the each particle's position (gather), (2) acceleration and move of the particles, (3) the deposition of the particle charge and/or current back onto the grid, and (4) the integration of the fields given the just computed charge and/or current densities [1] [10].

The most important and most difficult of these steps for obtaining good performance is the deposit step. It is the most important because this is generally where most of the computational time is spent. This is a result of there typically being many more particle degrees of freedom than field degrees of freedom, and because this is the most flop intensive part of the algorithm involving particles. It is the most difficult step to optimize because it naturally involves writing the data from multiple particles to the same memory locations, and it has the potential for particles close in memory to write to memory locations that are far apart. Consequently, this paper will concentrate on the deposit step, in particular the deposition of charge, which is all that is needed for an electrostatic simulation.

We begin with a straightforward implementation of the deposit step. We analyze this implementation to determine what is limiting it from using the full capability of the CPU, and then we modify the algorithm to mitigate the slowdown. Eventually we apply multiple techniques, **tiling** to improve cache usage and reduce race conditions, strip mining to separate the vectorizable part of the algorithm, particle sorting to improve cache usage and to improve usage of vector instructions. These different CPU performance optimization methods are described and performance gains are analyzed. A final method is presented that offers significant speed up due to fully utilizing current CPU architectures.

Chapter 1 introduces the motivation and the topic of this paper, while Chapter 2 introduces key concepts of PIC and current CPU architectures that are used throughout this paper. Chapter 3 describes different optimization techniques that applied to the deposit, as well as an analysis of the performance gained using these techniques. a novel method to vectorizing the deposit method on current CPU architectures, and the speed up gained due to this method is also presented at the end of Chapter 3. Chapter 4 is the conclusion that summarizes the final results, and discusses future work.

Chapter 2

Introduction to Key Concepts

The purpose of this chapter is to introduce both the PIC algorithm and current CPU architecture. For current CPU architecture caching, multithreading, and vectorization will be introduced. A basic overview of the PIC algorithm will be presented as well as the deposition method of the PIC algorithm. An explanation of what the deposition method is, and how the deposition method is implemented will also be given.

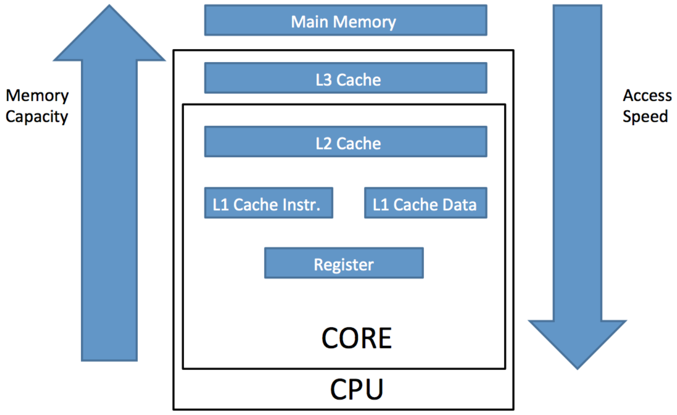
2.1 Modern CPU Architectures

CPU developers are always trying to increase the computational speed, floating point operations per second (FLOPS), of CPUs. Developers have introduced new performance features that take different approaches to increasing the FLOPS of a CPU. One approach is to decrease the of number times a CPU accesses memory that is external to the it, which requires a large amount of time. A feature called the *memory hierarchy* is able to achieve this by having smaller memory buffers that exist on the CPU. These buffers are called *cache* and have shorter access times compared to accessing memory external to the CPU. Caching allows a CPU to load memory once onto cache and only access memory from cache for future operations. Another approach to increasing the FLOPS of a CPU is to add features that allow a CPU to perform work in parallel. One feature that allows a CPU to achieve parallelism is called *multi-core architecture*. A multi-core processor is a type CPU that consist of smaller processing units called *cores*. These cores can perform work independently of each other and in parallel. Another feature that allows a CPU to achieve parallelism is called *SIMD*. SIMD is the existence of specialized instructions on a CPU that allows the CPU to simultaneously operate on every element of a contiguous chunk of memory in a single instruction call. The following sections will provide a more detailed explanation on what these features are, and how these features can increase CPU performance.

2.1.1 Caching

Accessing data from electronic memory (called *main memory*) that resides outside of the CPU takes a long time. Caching is a way to limit the number of times the CPU has to access main memory, by storing data in smaller capacity memory buffers (called *cache*) that exist on the CPU. Data that is stored on cache can be quickly accessed by the CPU. Current CPUs have multiple levels of cache that have increasingly smaller memory capacities and faster access speeds, as seen below in Figure 2.1. This paper will not be concerned with how data is handled after it is loaded onto cache.

Figure 2.1: Memory Hierarchy



One way cache limits the number of times the CPU accesses main memory is by reusing data elements that are already loaded onto cache; this is called *cache reuse*. If one needs to update a data element multiple times, then a CPU only has to access main memory once and load that data element onto cache. Once the data element is loaded onto cache the CPU can update it multiple times by quickly accessing cache rather than main memory.

2.1.2 Multithreading

Multi-Core architecture is an important performance feature present in current CPUs. A CPU is a multi-core processor if it consists of smaller processing units, called *cores*. Multithreading is one way to utilize multi-core processors and achieve parallelism by allowing the individual cores of a CPU to operate on memory that is visible to all the other cores. This type of parallelism is called *shared memory parallelism*. The ideal speed up due to multithreading is equal to the number of cores in the processor, and is achieved if all the computational work can be equally divided among all the cores of a CPU. A visualization of the architecture is presented in Figure 2.2.

Figure 2.2: Visualization of Multicore Architecture

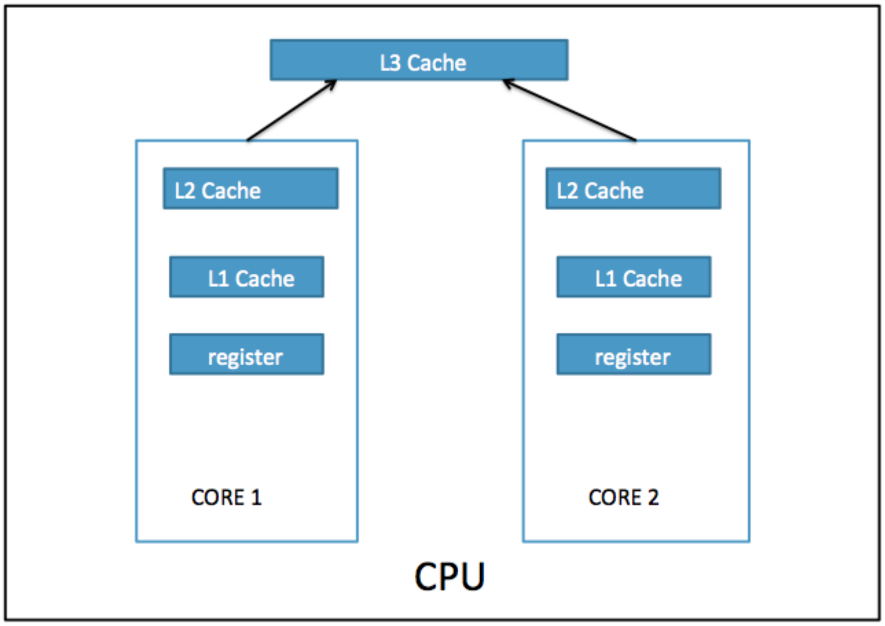
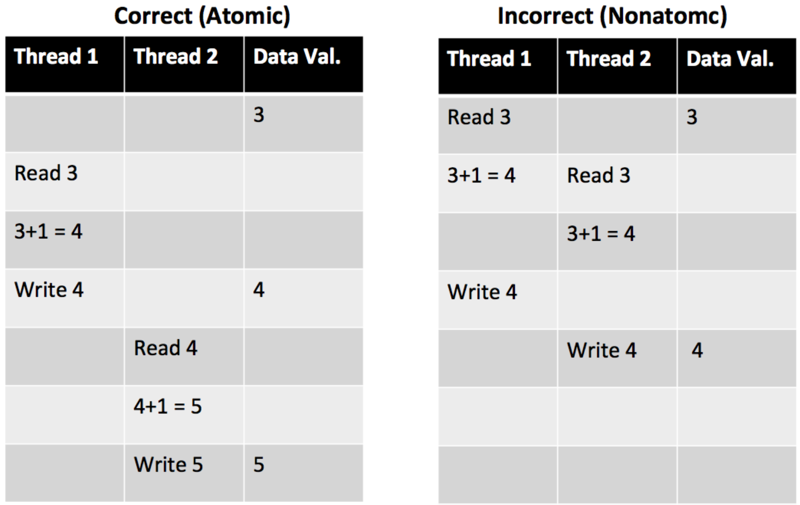


Figure 2.3: Race Condition



Correct (Atomic)

Thread 1	Thread 2	Data Val.
		3
Read 3		
3+1 = 4		
Write 4		4
	Read 4	
	4+1 = 5	
	Write 5	5

Incorrect (Nonatomc)

Thread 1	Thread 2	Data Val.
Read 3		3
3+1 = 4	Read 3	
	3+1 = 4	
Write 4		
	Write 4	4

A problem associated with shared memory parallelism is when multiple cores try to operate on a common memory location. This is called a *race condition.* A visualization of a race condition can be seen in Figure 2.3. One way to avoid race conditions when multithreading is to perform an atomic operation on the memory location. An atomic operation is an operation where the CPU enforces synchronization among all the cores trying to operate on a common memory location, ensuring the correct answer. This can be seen in the table labeled *correct* in Figure 2.3. Atomic operations add extra computational time that is associated with checking if multiple cores are attempting to operate on a common memory location. Throughout this paper OpenMP is used to enable multithreading and atomic operations [3].

2.1.3 Vectorization

Another type of parallelism that exists in current CPU architecture is vectorization. Vectorization is the utilization of SIMD instructions. SIMD instructions allow the CPU to simultaneously operate on a contiguous chunk of data. An example of a vectorized algorithm can be shown by trying to multiply a scalar constant a to each element of a one dimensional array. A non-vectorized algorithm would iterate through each element of an array, calling an instruction to multiply each element by the constant a (Figure 2.4). A vectorized algorithm would iterate though contiguous chunks of the array and for each chunk call a single SIMD instruction that multiplies each element of a chunk by the constant a simultaneously (Figure 2.5). The number of elements that can be operated on in a single SIMD instruction call depends on the specific CPU (8 floating point numbers for the CPU used in this paper).

Figure 2.4: Non-Vectoized Method

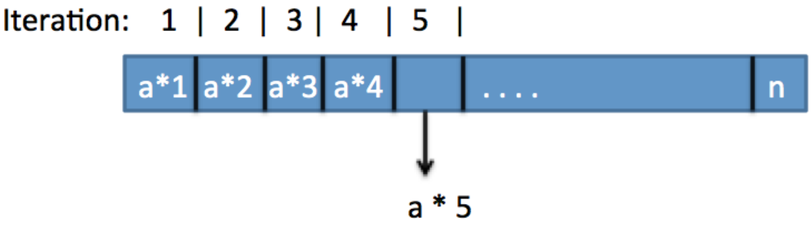
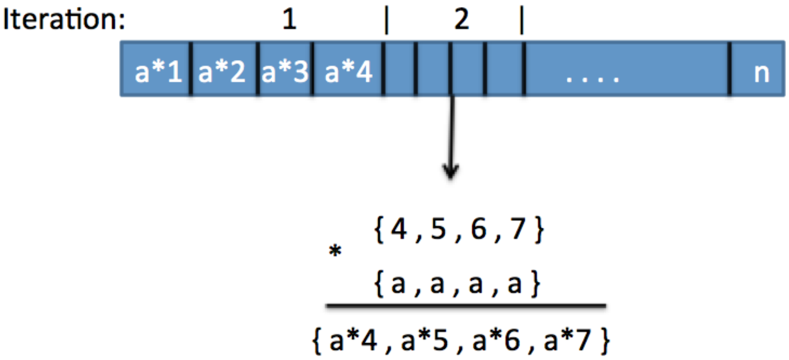


Figure 2.5: Vectorized Method



Compared to the non-vectorized example (Figure 2.4) the vectorized example (Figure 2.5) can yield an ideal speed up that is equal to the number of elements that can be operated on simultaneously in a single SIMD instruction. This is an ideal speed up because all of the algorithm's execution time occurs within the vectorized loop. This is not always true in algorithms, which is a large disadvantage of vectorization. Some algorithms require overhead complexity to transform them into a vectorizable form. When only part of the execution time is able to decrease due to vectorization Amdahl's law (Equation 2.1) can be used to calculate the total theoretical speed up, where P is the percentage of the task that is vectorized, and S is the speed up achieved in the vectorized region.

$$TheoreticalSpeedUp = \frac{1}{(1-p)+\frac{p}{s}} \tag{2.1}$$

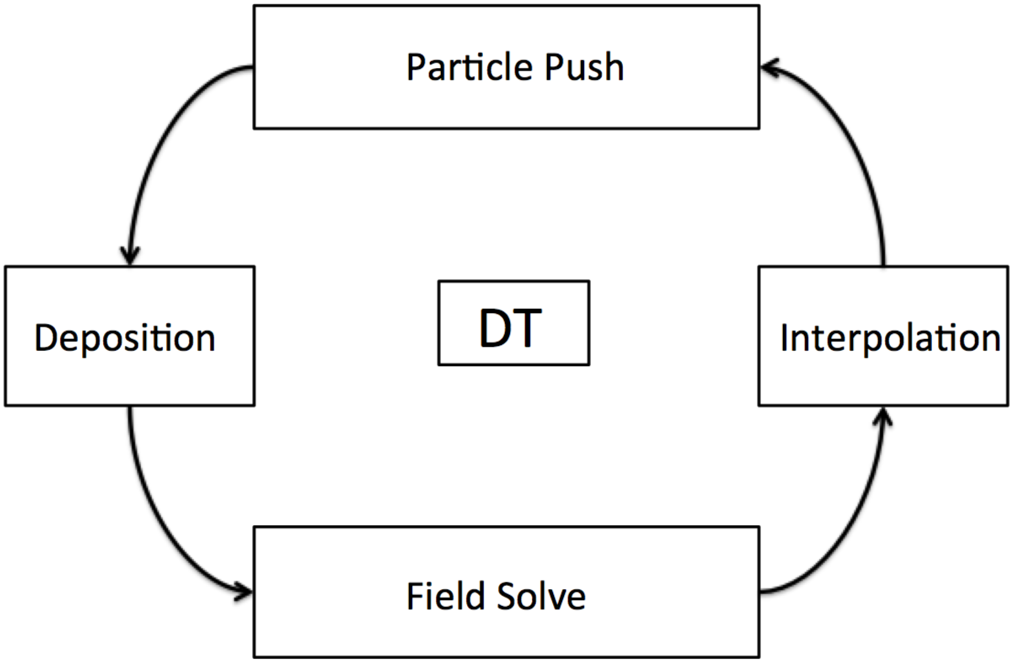
Modern day compilers are able to recognize vectorizable code and can vectorize loops for the user. This is known as auto-vectorization. Auto-vectorization is used throughout this paper to vectorize code. Optimization methods discussed in this paper, that utilize vectorization, will attempt to transform code into a form that the compiler can vectorize.

2.2 Particle-In-Cell Algorithm

Particle-In-Cell (PIC) is a general simulation technique that is widely used in many different fields of physics. PIC is able to simulate the motion of particles by solving for their fields using a grid. PIC can be broken up into four steps: (1) interpolation of the fields to the each particle's position (gather), (2) acceleration and move of the particles, (3) the deposition of the particle charge and/or current back onto the grid, and (4) the integration of the fields given the just computed charge and/or current densities [1] [10]. These steps can be seen below in Figure 2.6 .

This paper is concerned with the performance optimization of the deposition method, specifically charge deposition in two dimensions for current CPU architectures. Optimization techniques presented in this paper can be extended to include current deposition and higher dimensions.

Figure 2.6: Visualization of PIC Algorithm

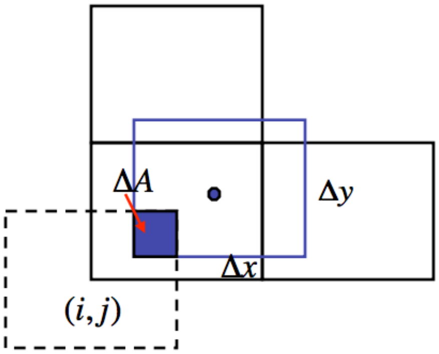


2.2.1 Particle Deposition

The particle deposition method in the PIC algorithm deposits each particle's charge to the nodes of the cell that the particle is contained within. The grid is a spatial charge distribution where each box in the grid is called a *cell*, and for each cell there are four nodes that represent the four corners of the cell. Charge is deposited and stored at the nodes of the grid. The deposition method iterates through each particle, and calculates the amount of charge to be deposited to the four nodes of the cell the particle is contained within. The amount of charge deposited to each node of a cell is based on the weighting scheme seen in Equation 2.2. In the PIC algorithm each particle is really a collection of particles that are close together in phase space. The distribution of these particles is defined by a shape function. For the weighting scheme used in this paper, the shape function is a rectangular step function with the same dimensions of an individual cell as seen in Figure 2.7 [1].

$$W_{i,j} = \frac{\triangle A}{\triangle x \triangle y} \tag{2.2}$$

Figure 2.7: Visualization of Zeroth Order Weighting Scheme [6]



Chapter 3

Optimization Methods

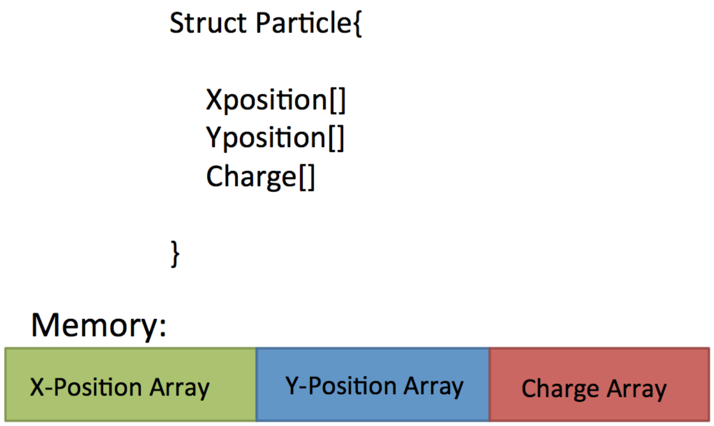
The goal of this chapter is to describe different optimization techniques that were applied to the deposition method. For each optimization technique this chapter will describe what problem it solves, how it was implemented, and an analysis of the results it gave. The first section of this chapter will give a more detailed explanation of the implementation of the PIC depositional method and the hardware specifications that the measurements were taken on.

3.1 Performance measurements

3.1.1 Deposit Method's Non-Optimized Implementation

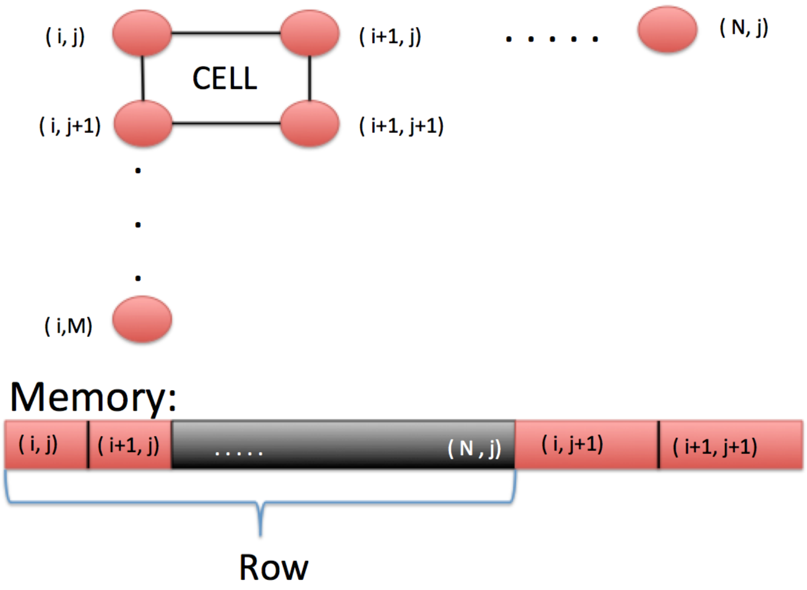
There are two fundamental data structures that the deposition method operates on. The first data structure holds all the particles in the simulation. The particle data structure is a collection of three one-dimensional arrays that are adjacent to each other in memory. The index i of each array represents a unique particle, and each array holds either the position or charge of each particle. The second data structure is a two dimensional grid. The grid is a discretized spatial charge distribution where charge is stored at the grid's nodes. Visualizations of the particle and grid data structures can be seen in Figure 3.1 and Figure 3.2 respectively.

Figure 3.1: Particle Data Structure



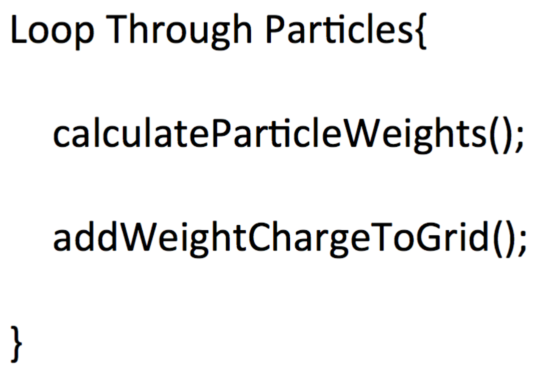
```
Struct Particle{

    Xposition[]
    Yposition[]
    Charge[]

}
```

Figure 3.2: Grid Data Structure



The pseudocode for the deposition method can be seen below in Figure 3.6. The overview of this algorithm is as follows. First, one iterates through the particle array. For each particle, its four weighted charges are added to the corresponding four nodes of the cell the particle resides in. Weighted charge is the amount of charge that will get deposited to a specific node based on the first order weighing scheme seen in Figure 2.7.

Figure 3.3: Pseudocode of Deposition Method Before Any Optimization Techniques Applied

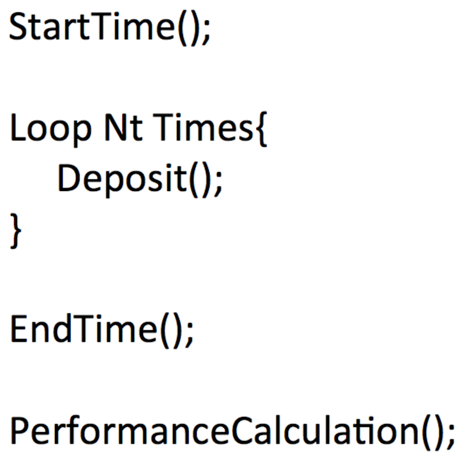


```
Loop Through Particles{

    calculateParticleWeights();

    addWeightChargeToGrid();

}
```

3.1.2 Measurement and Hardware Specification

All measurements throughout this paper are performed using a program that runs the deposition method of the PIC algorithm with an array of particles that are randomly distributed on the grid. The purpose of this program, seen in Figure 3.4, is to measure the time-per-particle of the deposition method. The time-per-particle is the average time (including program overhead) to deposit the charge from one particle to the charge distribution field (called *the grid*); that is, it is the total time divided by $NT \times PPC \times NC$. NT is the total number of times the deposition method ran in a measurement. PPC is the average number of particles in a cell, and NC is the total number of cells on the grid, such that $PPC \times NC$ is the total number of particles in the simulation. Time-per-particle should tend towards a constant as the total number of particles grow. By analyzing time-per-particle versus other general simulation parameters one can characterize the deposition methods performance on a CPU.

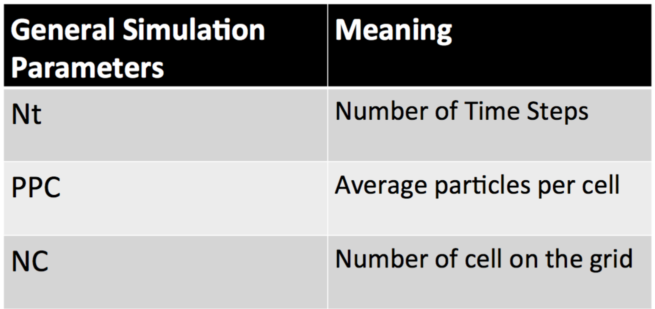
Figure 3.4: Pseudocode of the Program Used to Take Measurements



```
StartTime();

Loop Nt Times{
    Deposit();
}

EndTime();

PerformanceCalculation();
```

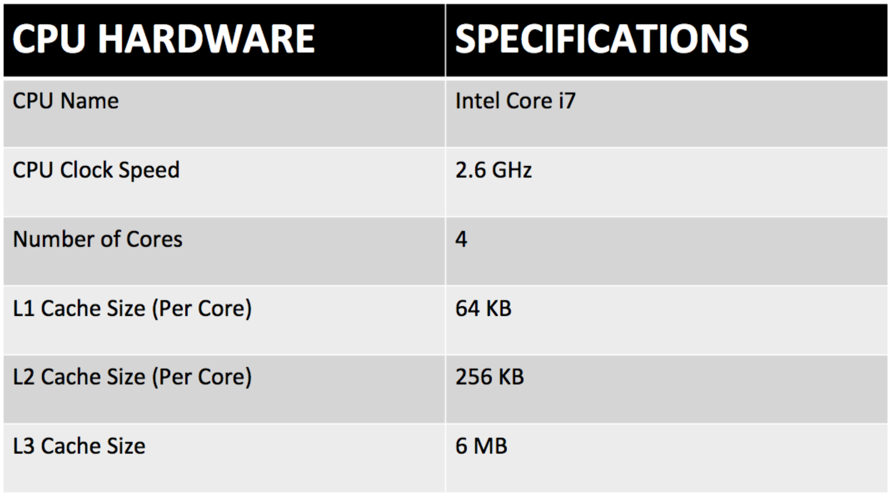
$$\frac{(EndTime - StartTime)}{(NT * PPC * NC)} \tag{3.1}$$

Figure 3.5: List of General Simulation Variables



General Simulation Parameters	Meaning
Nt	Number of Time Steps
PPC	Average particles per cell
NC	Number of cell on the grid

Figure 3.6: Table of Hardware Specifications



CPU HARDWARE	SPECIFICATIONS
CPU Name	Intel Core i7
CPU Clock Speed	2.6 GHz
Number of Cores	4
L1 Cache Size (Per Core)	64 KB
L2 Cache Size (Per Core)	256 KB
L3 Cache Size	6 MB

3.2 Tiling

The purpose of the memory hierarchy is to limit the number of times the CPU has to access data from main memory. One way the memory hierarchy achieves this is by loading memory onto cache. The CPU can then reuse memory that is already loaded onto cache, limiting the number of times the CPU accesses main memory. This is known as cache reuse and is the central idea behind the optimization technique of tiling.

The performance of the non-optimized implementation of the deposition method is seen in Figure 3.7. This implementation experiences a decrease in speed when the size of the grid is greater than L3 cache (Figure 3.7). When the grid size is greater than L3 cache, grid nodes will be accessed from main memory multiple times. For example, suppose in the particle array there are two particles that reside in a common cell, but one particle is at the beginning of the array and the other particle is at the end. Initially the four nodes that correspond to the cell containing both particles get loaded onto cache. The first particle adds it weighted charges to these nodes. Because all the nodes of the grid cannot fit onto L3 cache, by the time the deposition method reaches the last particle in the array these four nodes have been evicted from cache to make room for nodes corresponding to other particles in the array. In the worst-case scenario every time nodes are to be updated they have to be reloaded onto cache, and the deposition method will experience slower speeds due to the access time associated with main memory (Figure 3.7).

Figure 3.7: The Initial Performance of Deposit Method

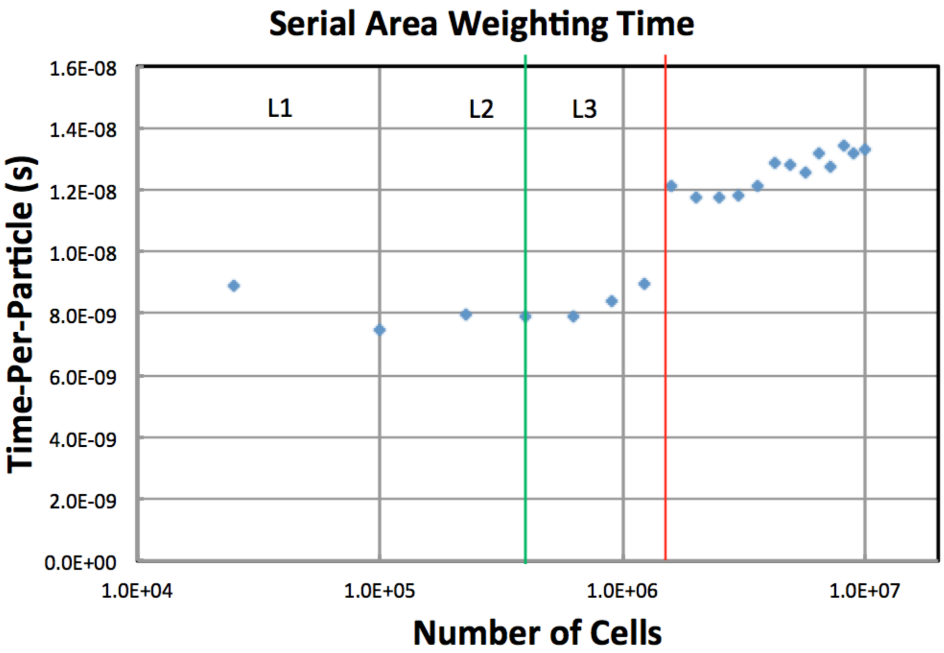
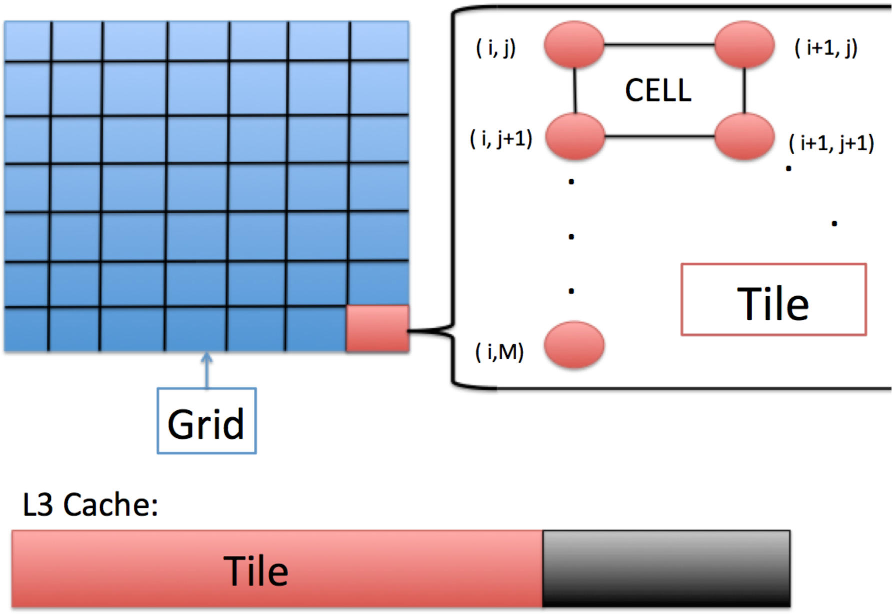


Figure 3.8: Visualization of Tile Sorting



As seen in Figure 3.7 when the size of the grid is less then L3 cache all of the nodes of the grid only have to be loaded onto cache once and can stay on cache for the remainder of the particle deposition, enabling the CPU to reuse cache. The idea behind tiling is to partition the grid into tiles, such that the number of nodes per tile is less than the number of nodes that can fit onto L3 cache. Then one needs to sort the particles according to the tiles they reside in. This ensures that when iterating through all the particles in a tile, the nodes associated with these particles only have to get loaded from main memory onto cache once, allowing the CPU to reuse cache for updating nodal values. A visualization of this optimization scheme can be seen in Figure 3.8.

Figure 3.9: Speed Up due to Tiling

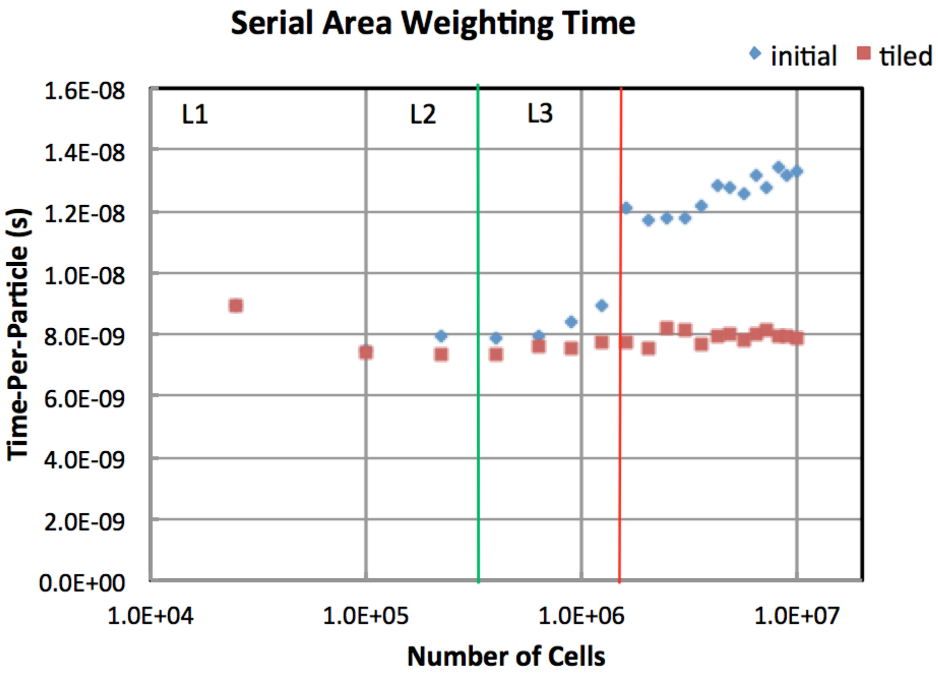
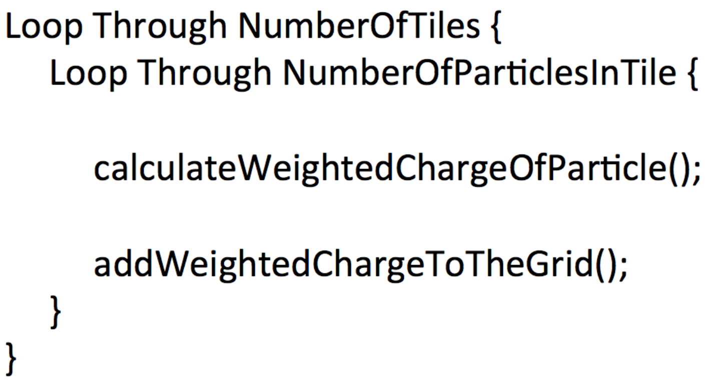


Figure 3.10: Tile is an array of Particle structures



```
Loop Through NumberOfTiles {
    Loop Through NumberOfParticlesInTile {

        calculateWeightedChargeOfParticle();

        addWeightedChargeToTheGrid();
    }
}
```

When tiling is implemented in the deposition method the CPU is able reuse cache, avoiding any performance lose when the grid becomes larger than L3 cache (Figure 3.9). This optimization scheme also introduces a performance tuning parameter to the deposition method called *tile size.* The tile size is defined as the number of cells per tile. When the size of a tile increases, the number of cells within a tile increases, and the total number of tiles on the grid decreases. The pseudocode for this algorithm can be seen in Figure 3.10, the deposition method now iterates through tiles and then iterates through the particles that are contained within a tile. While this optimization method allows effective use of the memory hierarchy through cache reuse, there is now extra computational time that is associated with sorting the particles into tiles. This sorting time can incur significant overhead, but it will not be discussed in this paper.

3.3 multithreading

Multithreading is a technique that allows the multiple cores of a multicore CPU to work in parallel. Multithreading is known as shared memory parallelism, because each core can operate on a chunk of memory that is visible to all the other cores of a CPU. A core can update a tile by adding the charges of all the particles that reside within the tile to the grid. The deposition method can assign all the cores of a CPU to update all the tiles of the grid in parallel. This method produces race conditions, because multiple cores can update neighboring tiles, thus updating common memory locations (Figure 3.11).

One solution to prevent race conditions in multithreading the deposition method is to perform an atomic add when adding the weighted charge of a particle to the grid. Using atomic operations avoids race conditions but at the cost of a large overhead in computational time (Figure 3.15).

Figure 3.11: Race Conditions of Neighboring Tiles

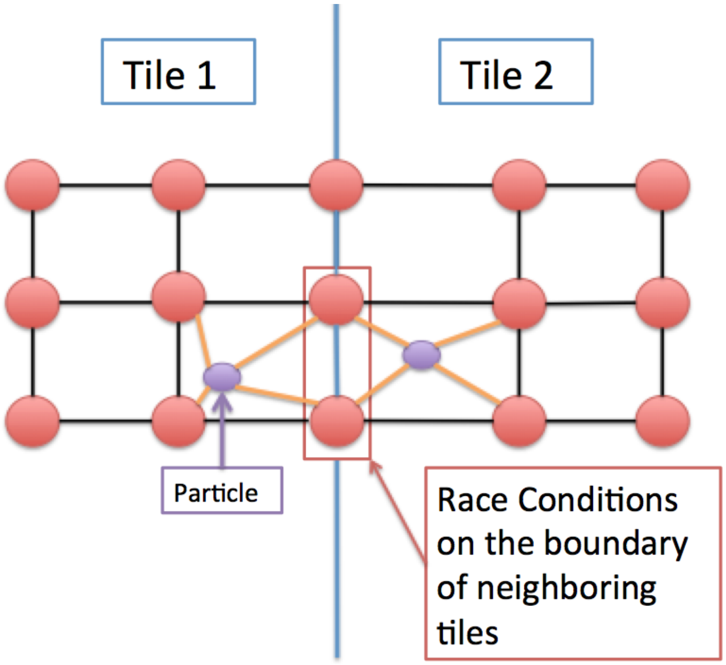
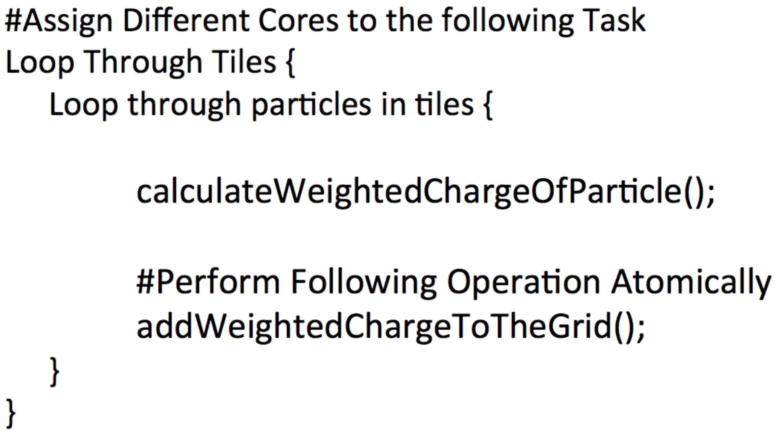


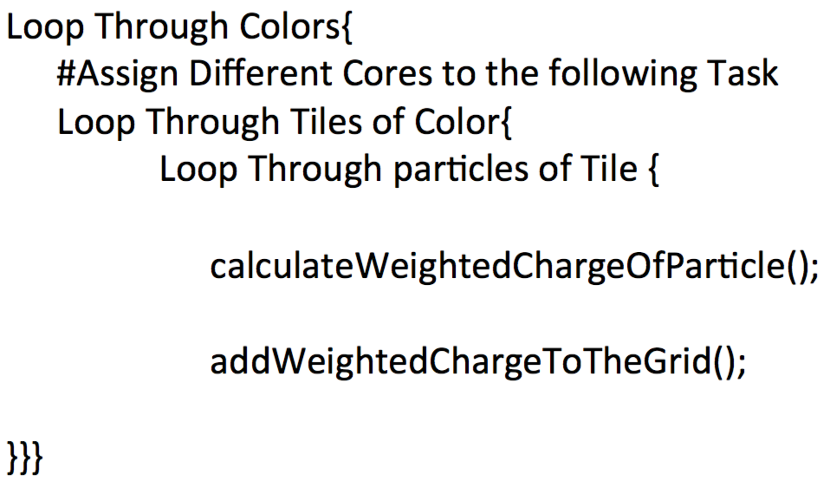
Figure 3.12: Pseudocode for OpenMP Using Atomics



```
#Assign Different Cores to the following Task
Loop Through Tiles {
    Loop through particles in tiles {

            calculateWeightedChargeOfParticle();

            #Perform Following Operation Atomically
            addWeightedChargeToTheGrid();
    }
}
```

Another way to avoid race conditions when applying shared memory parallelism to the deposition method is to ensure that each core only operates on a group of tiles that are not adjacent to each other on the grid. This method is called *tile coloring.* Tile coloring is an implementation of multithreading that allows the cores of a CPU to only update groups of tiles that do not neighbor each other on the grid. These groups can be imagined as a checkered pattern of four colors as seen in Figure 3.14. For example, in Figure 3.14 all the tiles belonging to either the red, blue, green, or orange group do not neighbor each other, and therefore can be updated in parallel without risking a race condition.

Figure 3.13: Pseudocode for Tile Coloring



```
Loop Through Colors{
    #Assign Different Cores to the following Task
    Loop Through Tiles of Color{
            Loop Through particles of Tile {

                calculateWeightedChargeOfParticle();

                addWeightedChargeToTheGrid();

}}}
```

Figure 3.14: Visualization Tile Coloring

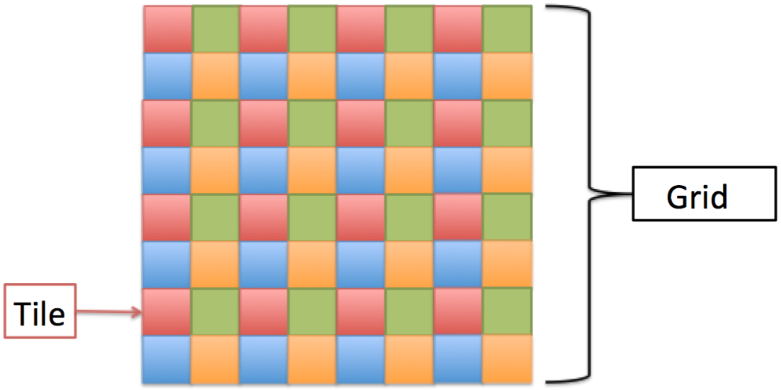
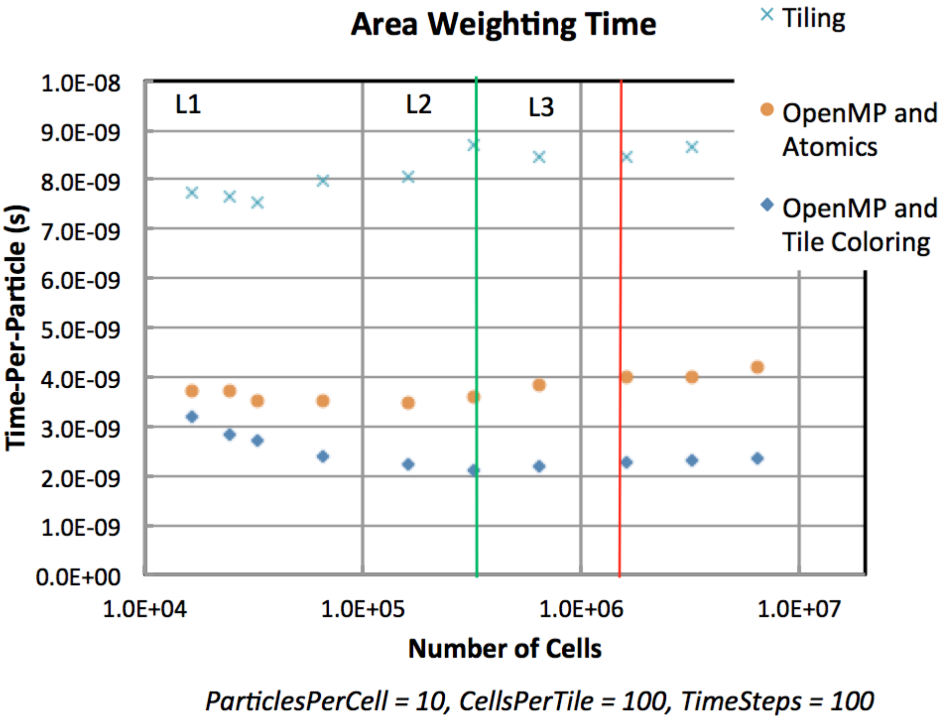
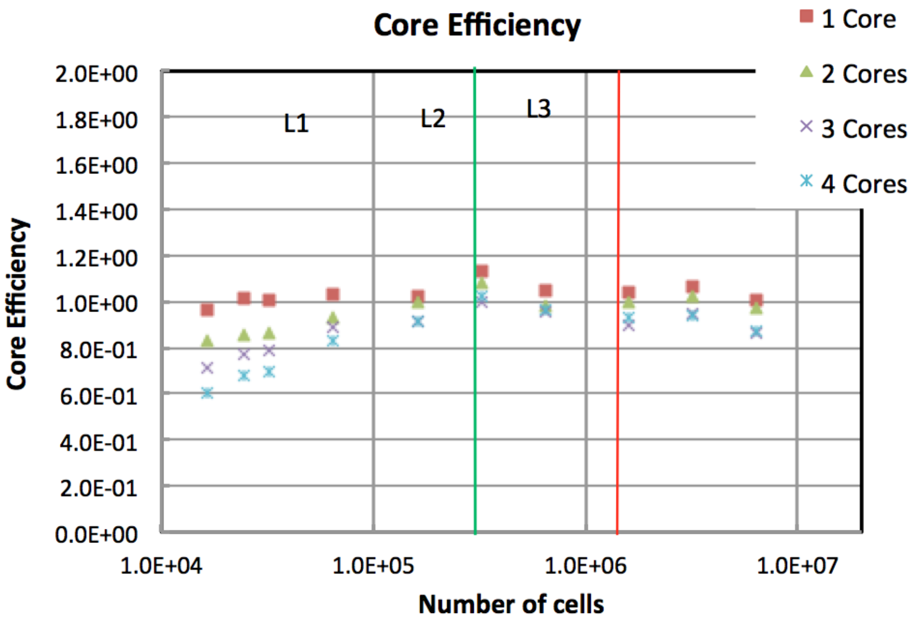


Figure 3.15: Speed Up Due to Tile Coloring



Tile coloring is able to avoid race conditions without the use of atomic operations, which incur a decrease in performance, as seen in Figure 3.15. Tile coloring is able utilize multi-core architecture, and achieve a speed up of 4 when four cores are used (Figure 3.15). Figure 3.16 is a graph of core efficiency, which is the speed up achieved divided by the number of cores used. As the number of cells grows the core efficiency tends towards 1 which means the tile coloring method is able to fully utilize every core of the CPU.

Figure 3.16: Core Efficiency



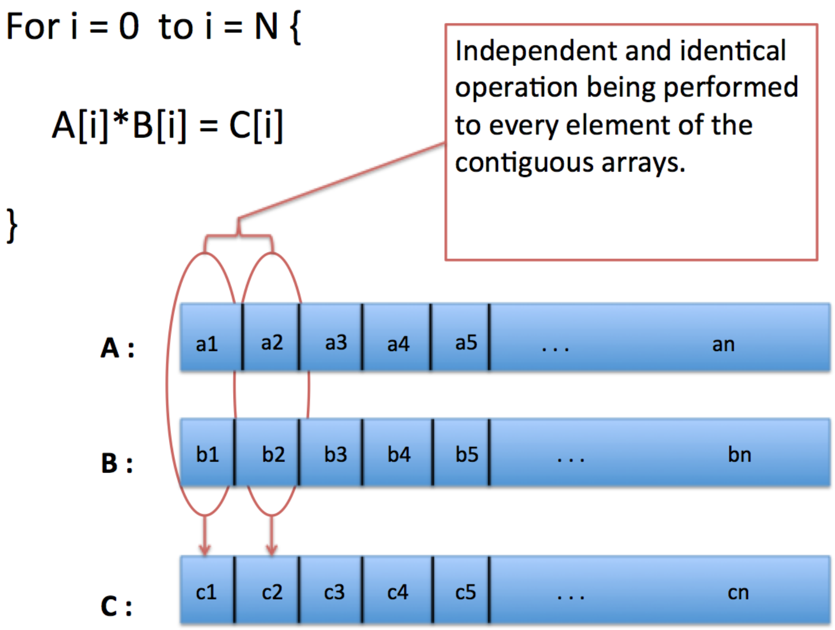
ParticlesPerCell = 10, CellsPerTile = 100, TimeSteps = 100

3.4 Vectorization

Vectorization is the process of transforming a loop into a form that can simultaneously operate on chunks of data rather than operate on each individual data element. A trivially vectorizable loop is a loop that consecutively performs an independent and identical operation on an array or arrays of contiguous data that are aligned. Trivially vectorizable loops can obtain the ideal speed up of vectorization. The goal of the vectorization methods discussed in this section are to transform the deposition method (as much as possible) into a trivially vectorizable form. An example of a trivially vectorizable loop can be seen in Figure 3.17.

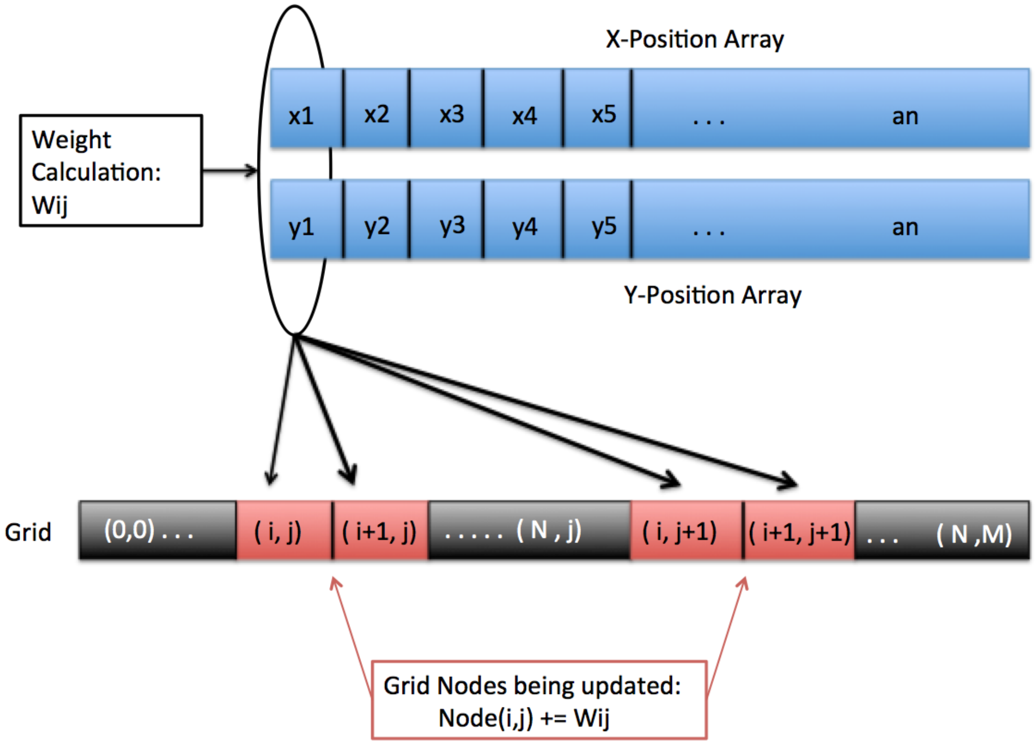
SIMD instructions operate on vector registers. Copying data into these registers is fastest when the first element of data is located at specific memory locations. Data that begins at these specific memory locations are called *aligned*. Data that is *misaligned* does not begin on these specific memory locations. Performing vector operations on misaligned data can prevent code from achieving the ideal speed of vectorization due to the extra computational time required to copy misaligned data into vector registers.

Figure 3.17: Visualization of Trivially Vectorizable Program



The deposition method is not trivially vectorizable because the nodes of a cell are not contiguous in memory as seen in Figure 3.2. The deposition method iterates through all of the particles within a tile and (for each particle) calculates the four weighted charges that will be added to the four nodes of the cell the particle resides in. The addition of weighted charges to the nodes of a cell are not trivially vectorizable because the four nodes of a cell are far apart from each other in memory (not contiguous in memory). A visualization of this problem can be seen in Figure 3.18.

Figure 3.18: Deposit Method Not Vectorizable Due to Nodes Not Contiguous in Memory

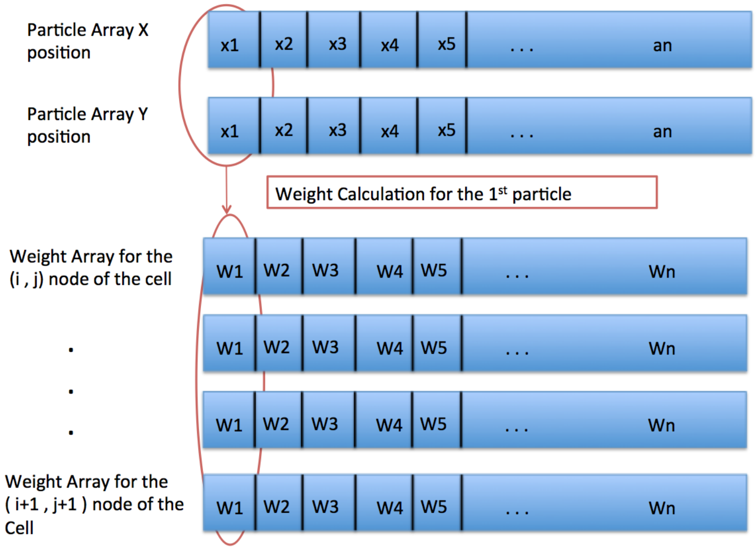


3.4.1 Strip Mining

Strip mining is a vectorization method that separates the vectorizable operations of an algorithm from the non-vectorizable operations by performing a loop transformation. The loop transformation splits a single loop into two loops. The goal of this loop transformation is that one of these loops become trivially vectorizable.

Strip mining can be applied to the deposition method by splitting the loop that iterates through the particle array into two loops. the first loop, called the *accumulation region*, iterates through all the particles within a tile and for each particle calculates its weighted charges and stores these charges into temporary arrays. This loop is now trivially vectorizable, because the process of calculating the weighted charges and writing them to temporary arrays can be performed consecutively and independently on each element of the particle array (Figure 3.19).

Figure 3.19: Visualization of Vectorized Accumulation


Particle Array X position
x1
x2
x3
x4
x5
. . .
an
Particle Array Y position
x1
x2
x3
x4
x5
. . .
an
Weight Calculation for the 1st particle
Weight Array for the (i , j) node of the cell
W1
W2
W3
W4
W5
. . .
Wn
.
.
.
Weight Array for the (i+1 , j+1) node of the Cell

The second loop, called the *nodal update region*, iterates through each element of the temporary arrays (weighted charges), adding each weighted charge to its respective node on the grid. This loop is not vectorizable because the weighted charges are being added to nodes that are not contiguous in memory (Figure 3.20). The pseudocode for the strip mining technique can be seen in Figure 3.21.

Figure 3.20: Visualization of Nodal Update

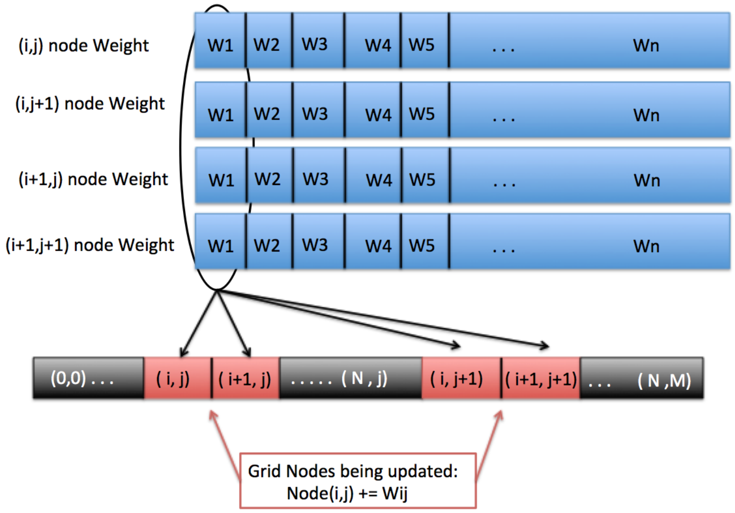
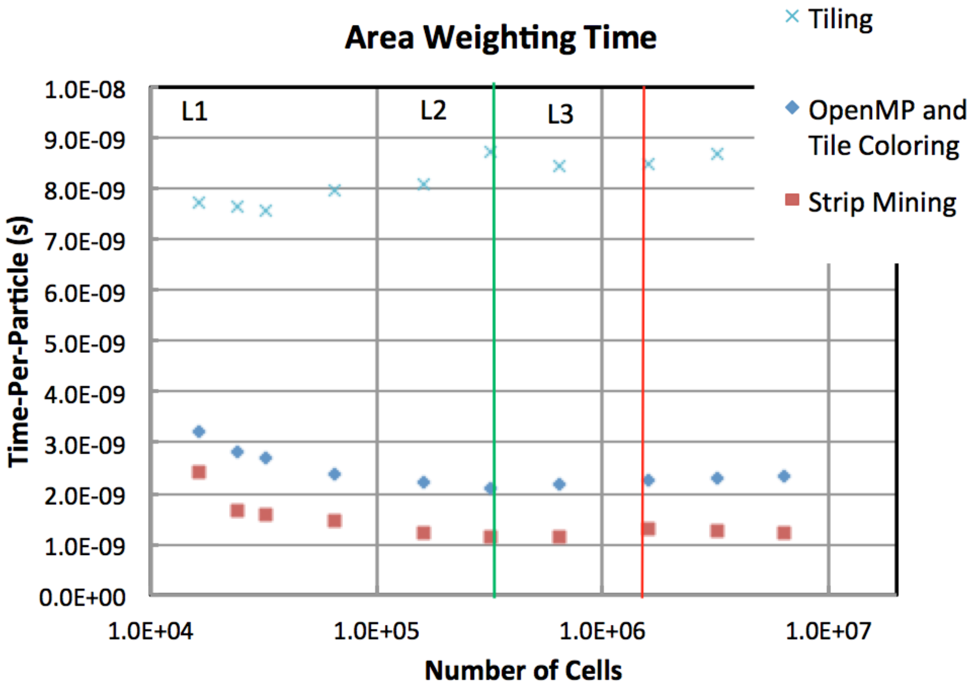


Figure 3.21: Pseudocode of Strip Mining

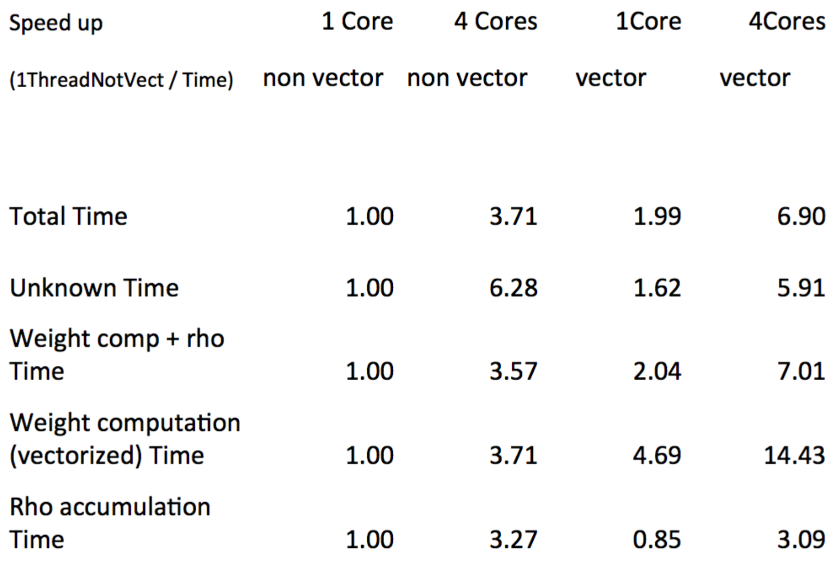
```
Loop Through Colors{
    #Assign Different Cores to the following Task
    Loop Through Tiles of Color{
            Loop Through particles of Tile {

                Wij[n] =  calculateParticleWeights();
                . . .
                index[n] = StoreGridIndexOfParticle();
            }
            Loop Through particles of Tile {

                Nodeij[index[n]] += Wij[n]
            }
}}}
```

Figure 3.22: Speed Up Due to Strip Mining



ParticlesPerCell = 10, CellsPerTile = 100, TimeSteps = 100

Figure 3.23: Performance Table of Strip Mining



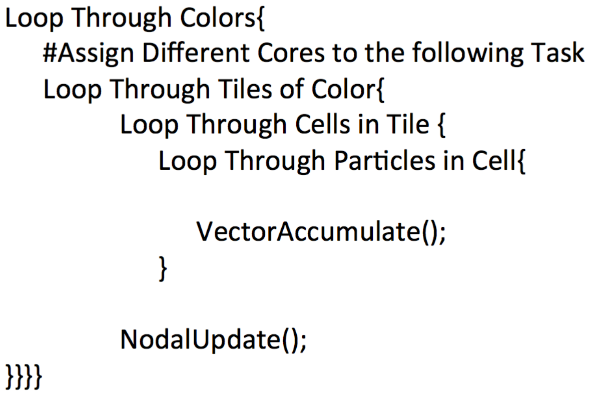
Speed up (1ThreadNotVect / Time)	1 Core non vector	4 Cores non vector	1Core vector	4Cores vector
Total Time	1.00	3.71	1.99	6.90
Unknown Time	1.00	6.28	1.62	5.91
Weight comp + rho Time	1.00	3.57	2.04	7.01
Weight computation (vectorized) Time	1.00	3.71	4.69	14.43
Rho accumulation Time	1.00	3.27	0.85	3.09

The ideal speed up of the vectorized region is a factor of 8 per core, because the CPU used can perform a vector operation on 8 floating point numbers. The speed up received in the vectorized region is only 4.69 per core, only 58 percent of the ideal speed up (Figure 3.23). With 4 cores the speed up expected would be a factor of 32 in the vectorizable region, which only a speed up of 14.43 is experienced, 45 percent of the ideal speed up (Figure 3.23). A reason for not achieving the ideal speed up could be due to the misalignment of data structures in memory, specifically the particle array or the temporary arrays. The total speed up achieved using strip mining when compared to the tile coloring method is about 2, and a total speed up of 8 is achieved when compared to the tiling method (Figure 3.22). The total speed up is limited by Amdahl's law (Equation 2.1), because only a portion of the computational time is receiving a speed up due to vectorization.

3.4.2 Cell Sort

Strip mining adds weighted charges to the appropriate nodes of the grid once per particle. By sorting particles based on the cells they reside in, cell sorting is able to add weighted charges to the appropriate nodes of the grid only once per cell. Cell sorting is able to achieve this by iterating through all the cells of a tile. For each cell, the cell sorting method first iterates through all the particles of the cell, accumulating the weighted charges due to each particle into four temporary arrays (one array for each node of the cell). Then, using the temporary arrays, the cell sorting method calculates and adds the four weighted charges due to all the particles in the cell to the cell's four nodes (Figure 3.24).

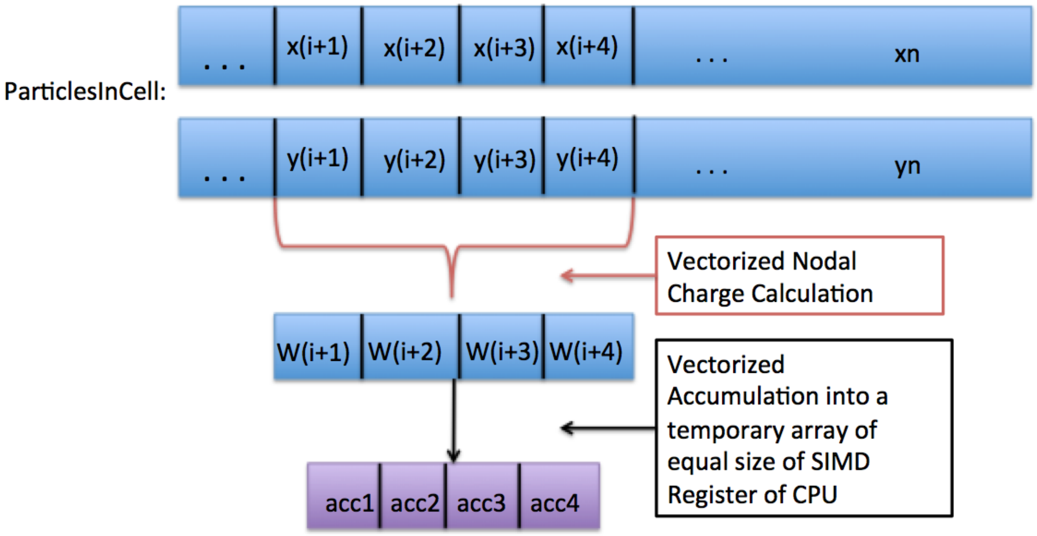
Figure 3.24: Pseudocode For Cell Sort Scheme



```
Loop Through Colors{
    #Assign Different Cores to the following Task
    Loop Through Tiles of Color{
            Loop Through Cells in Tile {
                Loop Through Particles in Cell{

                    VectorAccumulate();
                }

            NodalUpdate();
}}}}
```

The trivially vectorizable region of the cell sorting method accumulates the total amount of charge that is to be added to the nodes of a cell into four temporary arrays called *accumulation arrays*. Each accumulation array corresponds to one of the four nodes of a cell, and holds the same number of elements that a SIMD instruction can simultaneously operate on (8 floating point numbers for the CPU used). The trivially vectorizable region iterates through all the particles in a cell by contiguous chunks of 8 particles. For each particle in a contiguous chunk the region performs a weighted charge calculation and adds these weighted charges to corresponding accumulation arrays (Figure 3.25).

Figure 3.25: Visualization of Vectorized Accumulation For Cell Sort



The non-vectorizable region in the cell sorting method adds each element of an accumulation array together (Figure 3.26), producing four weighted charges due to all the particles in the cell. These weighted charges are then added to the four nodes of the cell. After the nodal update is performed the cell sort method then continues to the next cell.

Figure 3.26: Visualization of Nodal Update in Cell Sort

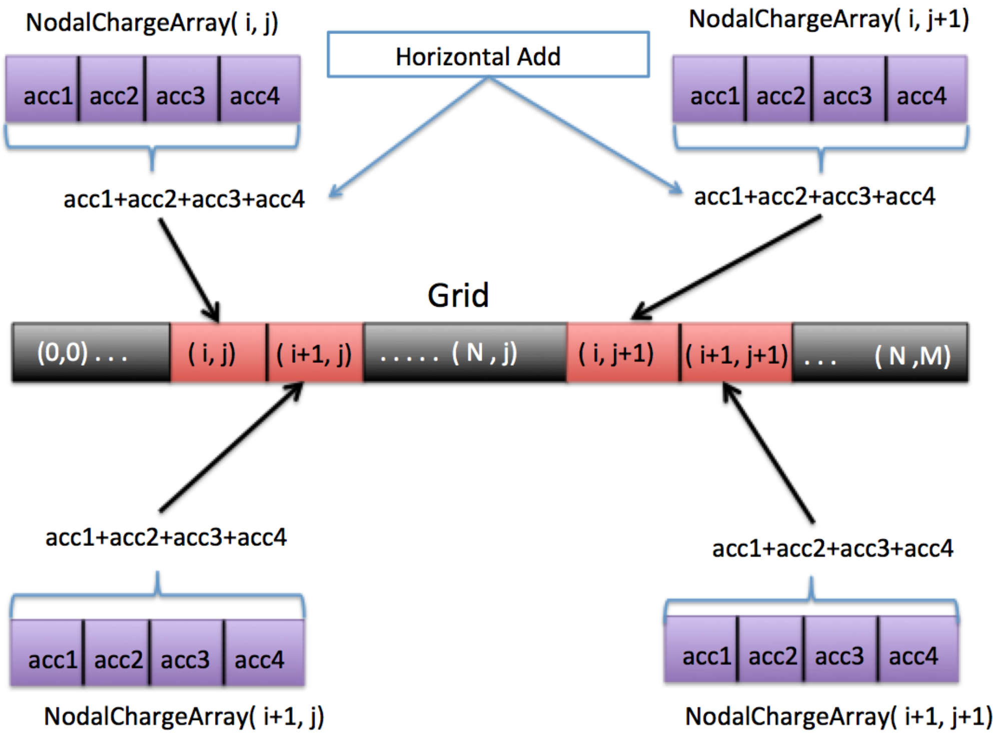
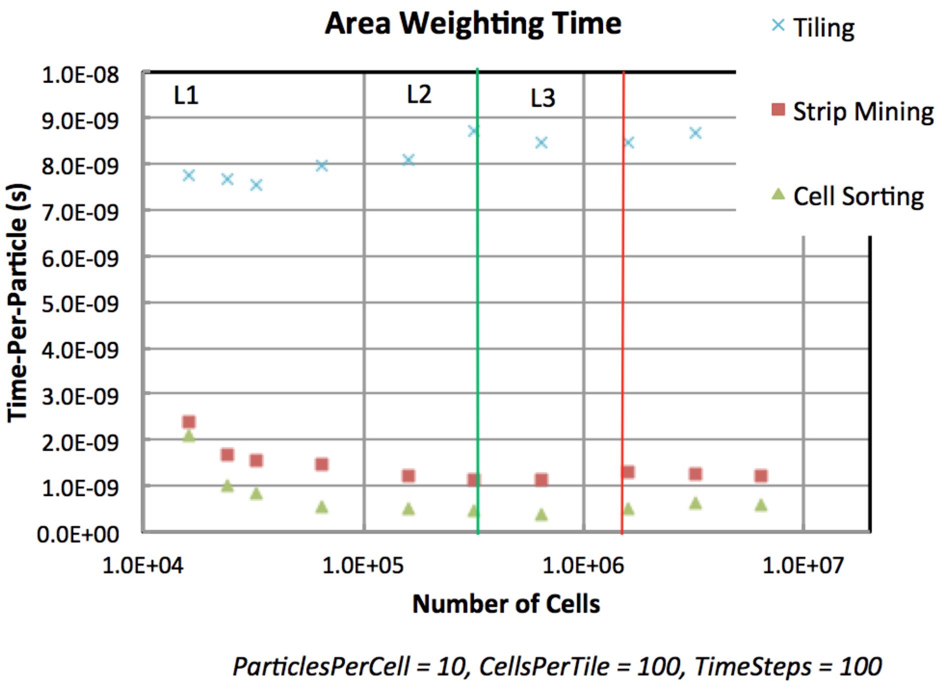


Figure 3.27: Total Speed Up Due to Cell Sorting



The total speed up due to cell sorting can be seen in Figure 3.27. Compared to strip mining, cell sorting produces a speed up of 2. The cell sorting method achieves a total speed up of 20 when compared to tiling (Figure 3.27).

Figure 3.28: Speed Up Due to Non-Vectorized Cell Sorting

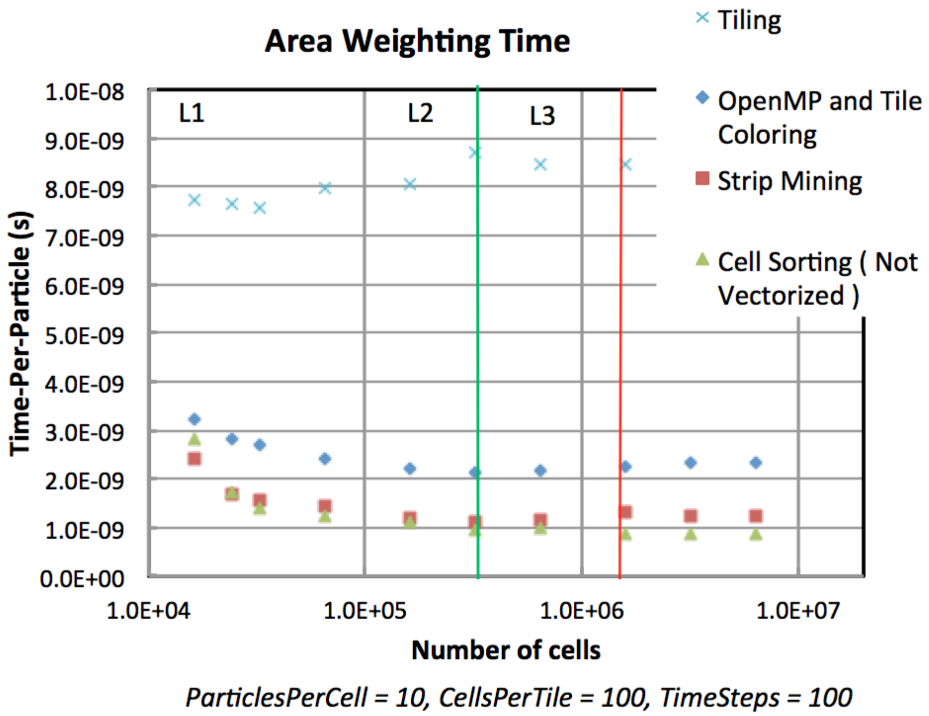
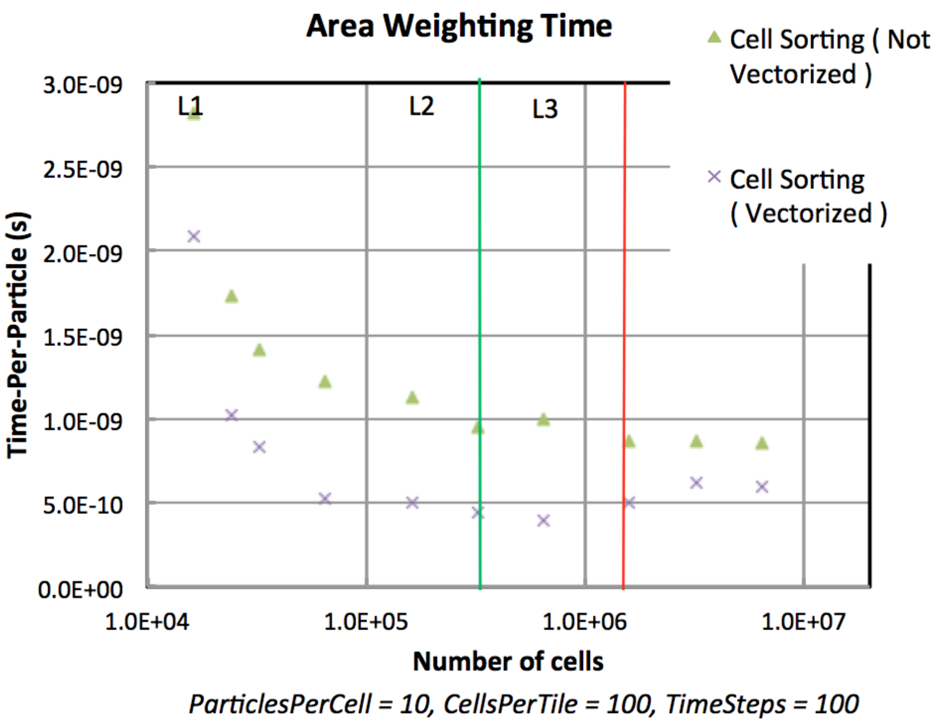


Figure 3.29: Speed Up Due to Vectorization of the Cell Sorting method



The speed up due to vectorization (Figure 3.29) is 2, which is equal to the speed up achieved using strip mining (Figure 3.23). As seen in Figure 3.28, when the compiler does not vectorize the cell sorting method a speed up of 2 is achieved when compared to tile coloring (a non-vectorized method). Cell sorting may achieve this speed due to cache reuse of its accumulation arrays or cache reuse of nodal values, but further investigation is needed to confirm.

As more cores are applied to the cell sorting method, core efficiency decreases due to bandwidth limitations of the CPU, that is the algorithm operates on more data elements per unit time than can be transmitted to the CPU (Figure 3.30). By decreasing the bandwidth of the cell sorting method, core efficiency is to expected increase due to no longer being limited by the maximum bandwidth of the CPU. Decreasing bandwidth of the cell sorting method increases core efficiency (Figure 3.31), which confirms that the cell sorting method is limited by the bandwidth of the CPU. The cell sorting method incurs a large overhead associated with sorting particles by cells. Although the cell sorting method is bandwidth limited by the CPU and would have to incur large overhead due to sorting the particles, a total speed up of 20 is achieved when compared to tiling (Figure 3.28).

Figure 3.30: Thread Efficiency of Cell Sort Scheme

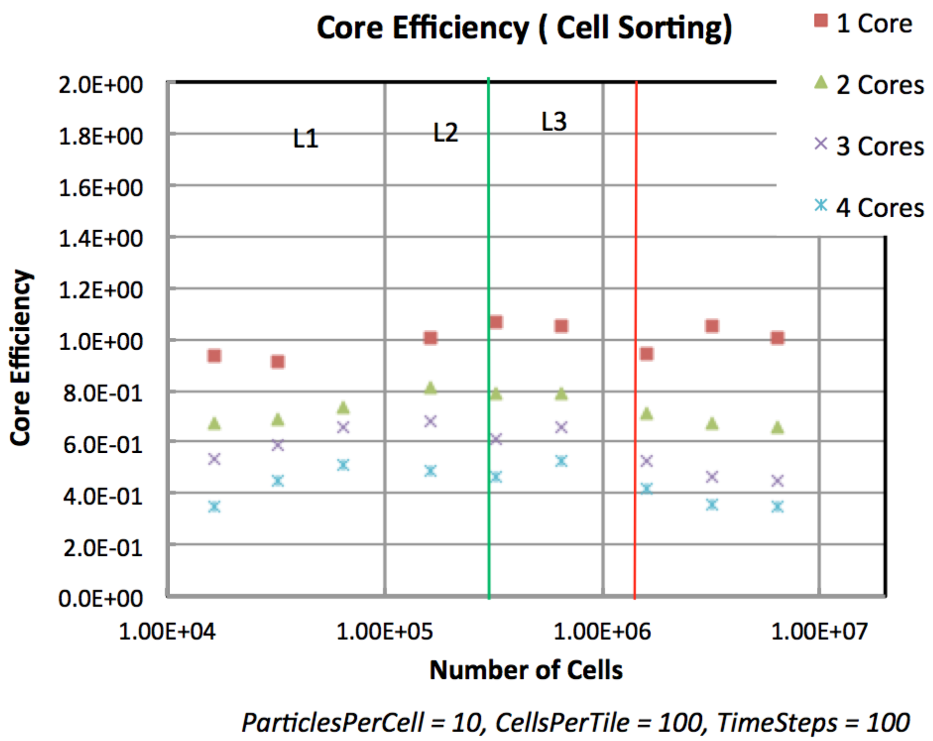
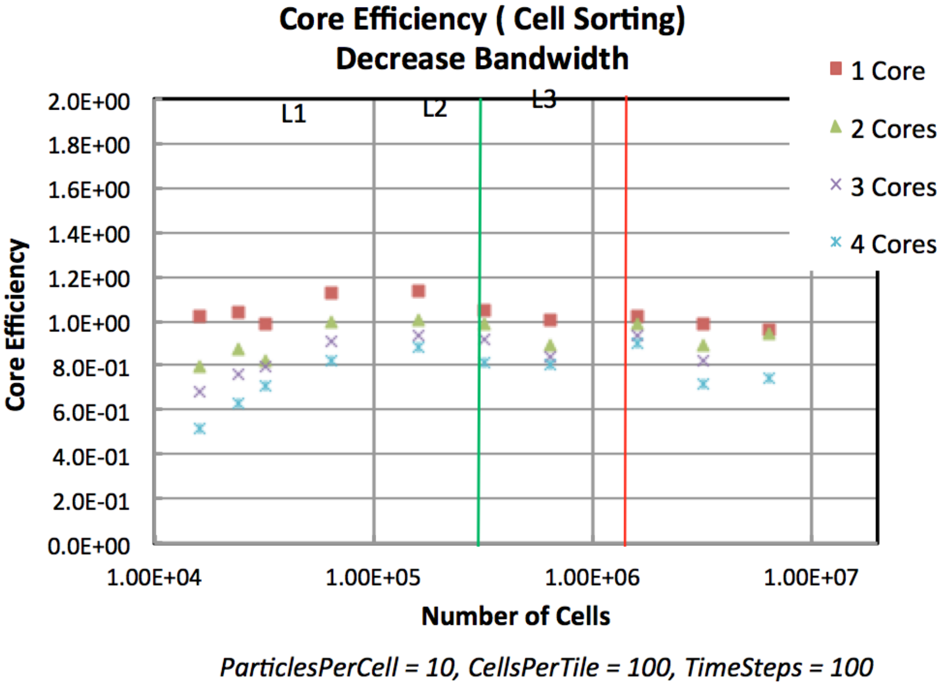


Figure 3.31: Thread Efficiency of Cell Sort Scheme with Decreased Bandwidth



Chapter 4

Conclusion

4.1 Future Work

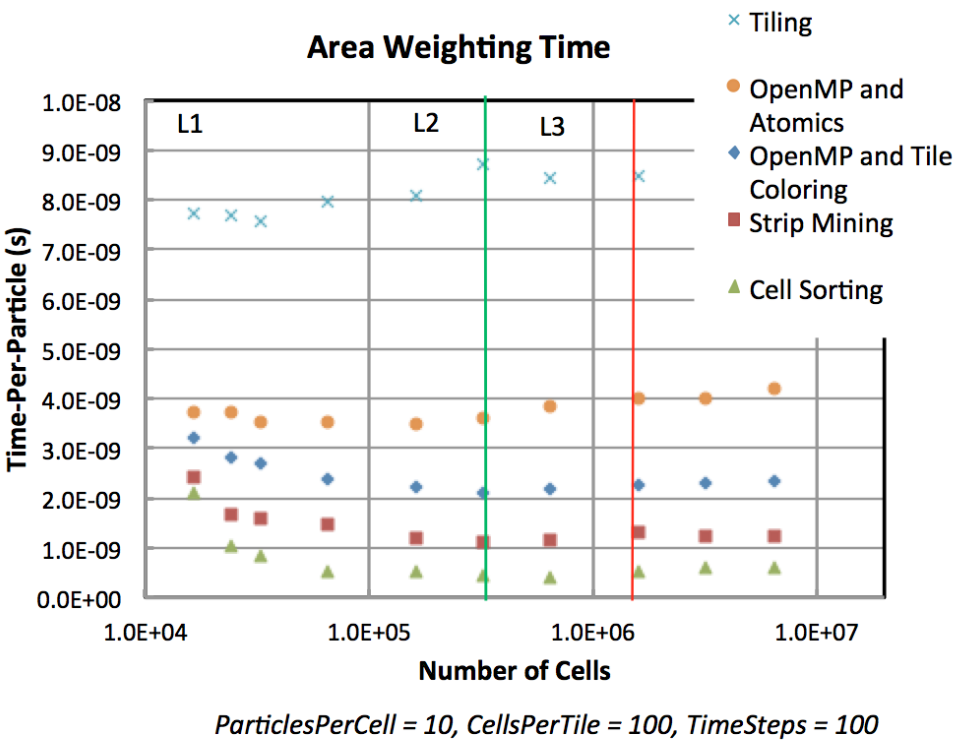
When the compiler does not vectorize the cell sorting method a speed up of 2 is achieved when compared to tile coloring. One reason for this speed up could be due to cache reuse of the weighted charge arrays. Sorting the particles based on cells, the cell sorting method can accumulate all the charge to be deposited to a cell instead of a tile. This requires temporary accumulation arrays that could be stored in cache, and take advantage of cache reuse. Another reason for this speed up would be L1 or L2 cache reuse. The cell sorting method iterates through cells and only adds charge to a cell once per cell. This could allow the nodes of a cell to be reused in L2 or L1 cache. Testing these hypotheses would provide a better understanding of the speed up due to cell sorting.

A study into different types of sorting algorithms could be of interest to reduce the overhead of the cell sorting method. An issue with the cell sorting method is the significant overhead associated with performing a particle sort every time step. A high performance particle sorting algorithm could reduce this overhead, allowing the PIC algorithm to obtain a larger total speed up due to cell sorting.

4.2 Summary

The deposition method deposits charge from the particles to the grid and is a major performance bottleneck of the PIC algorithm. Throughout this paper there were various optimization techniques that increased the performance of the deposition method of the PIC algorithm on current CPU architectures. Below one can see a comparison to the various methods applied to the PIC algorithm (Figure 4.1).

Figure 4.1: Performance Comparison Between All Optimization Techniques Used



Each optimization technique is applied to the deposition method of the PIC algorithm to better utilize various performance features present in current CPU architectures. Tile sorting allows the deposition method to utilize the memory hierarchy by encouraging the reuse of cache through sorting particles into groups called tiles. Another technique that is applied to the deposition method is called tile coloring, which utilizes shared memory parallelism and avoids race conditions. Tile coloring ensures that each core of a CPU only operates on a group of tiles that are not adjacent to each other on the grid. The last two optimization techniques utilize SIMD architecture in current CPUs by transforming the deposition method, as much as possible, into a trivially vectorizable form. The method known as strip mining achieves this by separating the vectorizable region of the deposition method from the non-vectorizable region. Because only a portion of the computational time can utilize vectorization, strip mining is limited by Amdahl's law. Cell sorting is another technique that attempts to transform the deposition method into a trivially vectorizable form, by sorting the particles according to the cells they reside in. While cell sorting incurs overhead associated with sorting the particles, and is limited by the bandwidth of the CPU, it produces a speed up of 20 when compared to tile sorting.

Bibliography

[1] Charles K Birdsall and A Bruce Langdon. Plasma physics via computer simulation. CRC Press, 2004.

[2] Heiko Burau, Renée Widera, Wolfgang Honig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tomas E Cowan, Roland Sauerbrey, and Michael Bussmann. Picongpu: A fully relativistic particle-in-cell code for a gpu cluster. IEEE Transactions on Plasma Science, 38(10):2831–2839, 2010.

[3] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE, 5(1):46–55, 1998.

[4] Eric J Hallman, Kristian C Beckwith, and Peter Stoltz. Performance and scalability of parallel pic and fluid codes on xeon phi based supercomputers. In 2014 IEEE 41st International Conference on Plasma Sciences (ICOPS) held with 2014 IEEE International Conference on High-Power Particle Beams (BEAMS), pages 1–1. IEEE, 2014.

[5] RL Morse and CW Nielson. Numerical simulation of the weibel instability in one and two dimensions. Physics of Fluids (1958-1988), 14(4):830–840, 1971.

[6] Chuang Ren. Introduction to particle-in-cell method in plasma simulations, 2005.

[7] P Schweitzer, S Cipière, A Dufaure, H Payno, Y Perrot, DRC Hill, and L Maigne. Performance evaluation of multithreaded geant4 simulations using an intel xeon phi cluster. Scientific Programming, 2015, 2015.

[8] Takayuki Umeda, Yoshiharu Omura, T Tominaga, and Hiroshi Matsumoto. A new charge conservation method in electromagnetic particle-in-cell simulations. Computer Physics Communications, 156(1):73–85, 2003.

[9] John Villasenor and Oscar Buneman. Rigorous charge conservation for local electromagnetic field solvers. Computer Physics Communications, 69(2-3):306–316, 1992.

[10] Hockney R. W. and Eastwood J. W. Computer simulation using particles. Bristol: A. Hilger, 1981.