Effective Use of Multicore-based Parallel Computers for Scientific Computing

Wenjie Wei

Thesis submitted for the degree of Philosophiae Doctor
Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
August 2012

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1265

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika publishing.
The thesis is produced by Akademika publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

This thesis studies how the multi-core hardware architecture can be efficiently used for real-world scientific applications that arise from computational cardiology and computational geoscience. The investigation has been carried out from different angles: numerical algorithms, parallel programming and performance modeling and prediction. It is shown that high-performance implementations and optimizations must match both the underlying computations and the target parallel platform. Several good practices are summarized for parallel programming and performance analysis on the multi-core architecture, which can be of help to many other scientists.

Keywords: multi-core, OpenMP, mixed programming, performance modeling

Acknowledgements

I started my PhD research in the autumn of 2008 at Simula Research Laboratory. Therefore, I would first like to thank Simula for funding my PhD project and offering an excellent work environment.

In the past few years, my PhD road has been busy, demanding, joyful and even a little bit painful. However, in any case, it has now become a very precious part of my life, permanently. Wherever I may be in future, this special experience will always bring my memory back to Simula, to Norway.

During my PhD study, I obtained great support from many people to finish this hard work. I am very glad to have this opportunity to express my sincere gratitude to them for what they have done for me.

Under the supervision of Professor Xing Cai, Dr. Ola Skavhaug and Professor Gerhard Zumbusch, I learned a lot not only on how to do research but also on attitude and methodologies of coping with hard situations. First, I am indebted to Professor Cai who spent a great amount of time on supervising my research. Without his help, I definitely could not reach the phase of writing this thesis. Professor Cai, thank you for those invaluable discussions, your patience and all other aspects even beyond my PhD work! I would like to thank Dr. Skavhaug for his encouragement and suggestions, which were invaluable when I experienced the hard time in winter 2011. His words strengthened my belief in those dark days. Many thanks also go to Professor Zumbusch. Although we could not talk frequently because he works in Germany, his ideas inspired my research very much during his visits to Simula.

As a member of the Center for Biomedical Computing, the Computational Geoscience department and the Simula School of Research and Innovation, I received a lot of help from my colleagues there. Particularly, many thanks go to Dr. Pan Li, Dr. Stuart R. Clark and Dr. Omar Al-Khayat for their kind and consistent support through collaborations. Their humor and expert knowledge gave me lots of fun and inspiration. Many thanks also go to Professor Are Magnus Bruaset for his support.

I would also like to thank Dr. Tao Yue, Dr. Yan Zhang and Dr. Mei Wen for giving fruitful suggestions. Specially, I thank Dr. Wen very much for her consistent encouragement and invaluable suggestions when I went through difficult times.

Finally, I would like to thank my parents, who always give me so much while demanding so little. Without their unconditional love, understanding and support, I could never ever be strong enough to face those difficulties and finish this hard work.

Wenjie Wei, Oslo, August 2012

List of Papers

- **Paper I**

  Evolution of Intracellular Ca^{2+} Waves from about 10,000 RyR Clusters: Towards Solving a Computationally Daunting Task
  P. Li, W. Wei, X. Cai, C. Soeller, M. Cannell, A. V. Holden
  Published in Proceedings of the Fifth International Conference on Functional Imaging and Modeling of the Heart, 2009, Pages 11-20.

- **Paper II**

  Computational Modeling of the Initiation and Development of Spontaneous Intracellular Ca^{2+} Waves in Ventricular Myocytes
  P. Li, W. Wei, X. Cai, C. Soeller, M. Cannell, A. V. Holden
  Published in Philosophical Transactions of the Royal Society A, Volume 368, 2010, Pages 3953-3965.

- **Paper III**

  An OpenMP-enabled Parallel Simulator for Particle Transport in Fluid Flows
  W. Wei, O. Al-Khayat, X. Cai
  Published in Procedia Computer Science, Volume 4, 2011, Pages 1475-1484.

- **Paper IV**

  Numerical Analysis of a Dual-sediment Transport Model Applied to Lake Okeechobee, Florida
  S. R. Clark, W. Wei, X. Cai
  Published in Proceedings of the Ninth International Symposium on Parallel and Distributed Computing, 2010, Pages 189-194.

- **Paper V**

  Balancing Efficiency and Accuracy for Sediment Transport Simulations
  W. Wei, S. R. Clark, H. Su, M. Wen, X. Cai
  Submitted to journal for publication.

Contents

Introduction

1 Background

In this section, I first introduce some main challenges of multi-core programming, which are relevant to my research work. Then some common strategies of parallel computing and a simple performance model for the multi-core architecture are extracted, which may be useful for more general-purpose uses.

1.1 Challenges with multi-core programming

The multi-core architecture has emerged as a main-stream design to boost computing power in a practical way. It brings promising improvement of computing power while posting many programming challenges. Since the first commercial dual-core processor was manufactured by IBM [24] in 2001, multi-core has become a widely used processor architecture with different variations in the last decade. As its name implies, a number of computing cores are integrated into one processor. Normally, cores in the same processor have their own L1 caches but share the last level cache. For a computer node in a cluster, multiple processors may be equipped, which may adopt either the Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) design. Therefore, the memory hierarchy of the multi-core architecture is more complex than that of the conventional single-core CPU.

Before the multi-core era, advances in the processor frequency automatically gave rise to significant increases in the execution speed of software, with little effort needed from software developers. The emergence of multi-core processors posts new challenges to software developers, who must now master programming techniques necessary to fully exploit the multi-core processing potential [9].

The first challenge to the programmers is to develop parallel programs. Compared with developing sequential programs, programmers now have to adapt themselves to carefully coordinate work in a parallel way by considering resource sharing, contention, task scheduling, etc. Such programs are more error-prone and harder to debug. Beyond these common issues, when performance is taken into account, the second challenge emerges: implementing a parallel program with a decent performance on the multi-core architecture is much more difficult than just implementing a correct one.

Since mathematical and numerical details are given in the subsequent chapters containing published papers, the focus of this chapter is on parallel programming for multi-cores. Only characteristics of multi-core systems related to my work are selected to depict why they make parallel programming for the multi-core architecture difficult.

Performance bottleneck due to the memory system is not a new problem, which has existed for decades since the uni-core processor era. Nowadays, multi-core systems aggravate this issue. Some researchers even think multi-core is bad news for supercomputers [23]. We may at least agree that while the disparity between memory access speed and processor speed continuously grows up, multi-core reduces the amount of cache per thread and introduces competition for the bandwidth to memory [11]. Because the memory bandwidth does not increase linearly when more processor cores are used, compute-bound codes may become memory bound when many processor cores are used. This memory competition can be more serious for UMA systems when exceeding a certain number of cores. In order to relieve this bottleneck while allowing many cores work together, a widely adopted hardware solution for recent multi-core systems is to divide the memory into a certain number of regions, each accessed by a certain processor with the largest bandwidth but by others with lower bandwidth. When using this NUMA memory configuration, mapping data to different memory regions can affect performance substantially. The work of [22] shows that a mismatch between the data access pattern and the physical memory layout incurs a high overhead, meanwhile it is hard to find a data distribution that matches all possible access patterns. Programmers now are taking more responsibilities of controlling data locality either directly or indirectly.

Improving the utilization of cache is not an old topic of optimization for both sequential and parallel code. As multi-core processors introduce a more complex cache hierarchy, which involves both private and shared caches, the performance of parallel code is becoming more and more sensitive to cache utilization. With the aid of tools like PAPI [3], programmers can observe cache behaviors using hardware counters, which may give clues of performance bottlenecks. Meanwhile, some traditional uni-core performance metrics, such as L1 and L2 cache miss rates, do not adequately capture multi-core memory issues. This means that effective measurement, analysis, and optimization of memory performance bottlenecks intrinsic to multicores require a different and more complex approach than memory bottleneck detection and alleviation in uni-cores [13]. False sharing is another historical cache issue in shared memory systems starting from multi-processors. Relevant research can be traced back to 80's and 90's [12, 14, 27]. In the multi-core era, it still impacts performance severely. Recent work [26] shows that false sharing can cause performance degradation by 100x and even more. In general, some techniques like padding can remedy the false sharing problem, but how the techniques are used varies from case to case.

In summary, multi-core design brings a brilliant improvement on processors' peak performance, meanwhile lots of programming effort is required for programmers to translate a good performance in theory to a good performance in practice. The work of my thesis tries to be helpful for the above aspects.

1.2 Mixed programming and parallelization of FDM

As one of the parallel programming methods, mixed programming means combining shared and distributed-memory programming models in one code. Multi-core based clusters are distributed-memory systems in term of nodes, meanwhile each node itself is a shared-memory system. Such a mixed-memory configuration endows such clusters with a hardware foundation for mixed programing. Let us take the Finite Difference Method (FDM) as a particular example, which is widely used in scientific research and engineering fields due to its mathematical simplicity and ease of implementation. This method normally relies on uniform meshes described via multi-dimensional arrays. When the actual solution domain is irregular, a typical way for FDM to cope with is to adopt a uniform mesh large enough to completely include the irregular solution domain. A 2D example is provided in Figure 1, where the red line describes an irregular solution domain included by an uniform mesh. Therefore, not all the mesh points belong to the irregular solution domain. The partitioning of such a domain will be the focus of parallelizing FDM codes in the remaining part of Section 1.2.

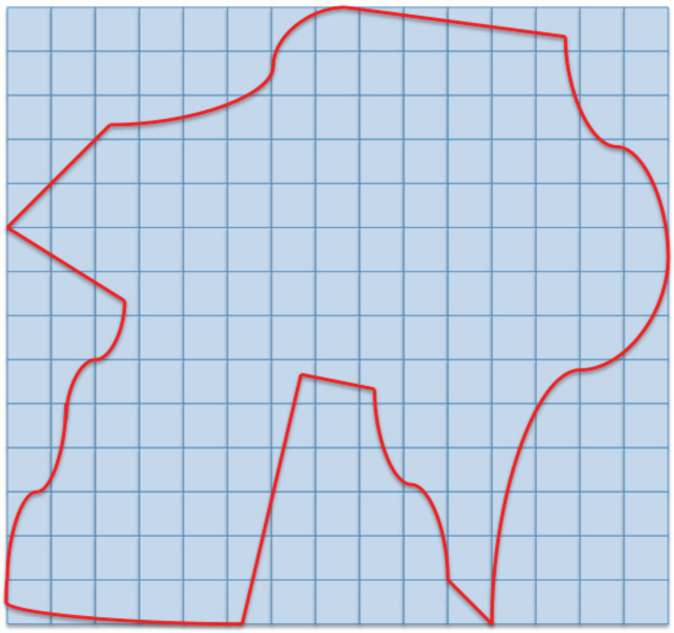
Figure 1: Irregular solution domain included by an uniform mesh.

In order to check if a mesh point belongs to the solution domain, a data structure for saving this information for all the mesh points is necessary. For this goal, a `boolean` array called `inSolutionDomain` is introduced, which corresponds exactly to the multi-dimensional uniform mesh. In `inSolutionDomain`, each array element will be marked as `true` or `false` according to whether its corresponding mesh point lies inside or outside the solution domain.

When such a mesh needs to be parallelized by the distributed-memory programming model, partitioning the mesh with cutting planes orthogonal to the dimension axes will be the most straightforward way. Although a general curved partitioning of the irregular solution domain can give the best partitioning balance, more complex data structures have to be introduced for each mesh point to save the partitioning information of its neighboring points. A parallel implementation of FDM using curved partitioning will thus become more complex than using the cutting-plane partitioning. Therefore, we focus on the domain decomposition by partitioning the whole mesh with cutting planes into a certain number of uniform sub-meshes as evenly as possible. Fewer sub-meshes means easier and better partitioning balance. This fits very well with the mixed programming due to partitioning the mesh with the number of computer nodes other than the total number of processor cores.

In details, when using m computer nodes, each equipped with n CPU cores, a parallel implementation solely adopting MPI, termed "flat MPI" in this thesis, will have $m \times n$ MPI processes generating $m \times n$ sub-meshes. When mixed programming using the least MPI processes is adopted, one MPI process is assigned to one computer node, where MPI process spawns a OpenMP thread for each core of the node. The latter implementation has only m sub-meshes, which naturally improves the partitioning balance.

In a more general situation, the computation load of each mesh point belonging to the solution domain may vary. If the problem is time dependent, the computation load may vary over time. Thus, a static work partitioning cannot give an ideal load balance for parallel computing in such a dynamic situation. In each sub-mesh, the parallelism comes from n OpenMP threads sharing the work load. All threads can be scheduled by a "dynamic" scheduler of OpenMP.

Furthermore, in each sub-mesh, there will be N nested for-loops standing for the N dimensional sub-mesh to function as a computing kernel. Parallelizing this kernel using the OpenMP directive `#pragma omp parallel for` on the outmost for-loop cannot give the sufficiently fine parallel granularity for a better load balance. To solve the issue, these nested for-loops can be collapsed to a long single for-loop that can be parallelized by OpenMP for a better load balance. In order to achieve this goal, a new clause `collapse` has been introduced since OpenMP 3.0 specification [2]. In Figure 2, a 3D sub-mesh is adopted as an example to show the parallelization with OpenMP, where three nested for-loops are parallelized via the clause `collapse`.

Meanwhile, we still prefer to explain the essence of the `collapse` clause because

```
#pragma omp parallel for default(shared), private(i, j, k), \
schedule(dynamic, chunkSize), collapse(3)
for (k = 0; k < z_len; k++)
  for (j = 0; j < y_len; j++)
    for (i = 0; i < x_len; i++)
      if (inSolutionDomain[k][j][i])
        computing(i, j, k);
```

Figure 2: OpenMP parallelization for a nested 3-dimensional for-loop.

the strategy can be applied on other multi-threaded programming models other than OpenMP and our further optimization in the remaining part cannot be implemented when using the `collapse` clause directly. Instead of parallelizing nested 3-dimensional for-loops, a single for-loop is parallelized in Figure 3, which does the similar work as the `collapse` clause.

```
total_length = z_len * y_len * x_len;

#pragma omp parallel for default(shared), private(idx, i, j, k), \
schedule(dynamic, chunkSize)
for (idx = 0; idx < total_length; idx++) {
  mapping1Dto3D(idx, &i, &j, &k);
  if (inSolutionDomain[k][j][i])
    computing(i, j, k);
}
```

Figure 3: One for-loop for OpenMP parallelization.

A further optimization can be adopted to remove mesh points outside the solution domain. This optimization reduces OpenMP scheduling overhead and improves the load balance of the OpenMP parallelization. We call this approach "compacted indices". In the implementation, an integer array `compactedIndices` is introduced for each MPI process to save the indices of those mesh points belonging to the solution domain in each sub-mesh. The optimized computing procedure is shown in Figure 4.

In sum, the mixed programming may deliver a better load balance of the whole parallel computing than flat MPI due to more balanced mesh partitioning and more flexible work scheduling in each sub-mesh. An optimization of compacting indices of computational mesh points continually is also introduced, which can further improve load balance.

1.3 OpenMP implementation of particle movement

Compared with mixing programming and parallelization of FDM (Section 1.2), the work in this sub-section also concerns OpenMP-based parallel programming but is more complex. For example, for an irregular solution domain described in Section 1.2, although the access to mesh points in the solution domain is not continuous, the accessing order is still regular. However, the accessing order may be random in many other application scenarios. Furthermore, the data structure of each mesh point can be variable because

```
len = 0;
for (k = 0; k < z_len; k++)
  for (j = 0; j < y_len; j++)
    for (i = 0; i < x_len; i++)
      if (inSolutionDomain[k][j][i])
        compactedIndices[len++] = idx;

#pragma omp parallel for default(shared), private(idx, i, j, k), \
schedule(dynamic, chunkSize)
for (idx = 0; idx < len; idx++) {
  mapping1Dto3D(compactedIndices[idx], &i, &j, &k);
  computing(i, j, k);
}
```

Figure 4: OpenMP parallelization on compacted indices.

the property of being computational or non-computational may vary over time, which is however fixed in the case described in Section 1.2. Therefore, the parallelization of application scenarios having such dynamic features will not be straightforward for example due to race condition. Simulating moving particles is a good example, where all the above issues are involved. The OpenMP parallelization strategy, multi-core based optimization and data structure design developed for the example is also applicable to more general-purpose uses.

Let us first introduce the basic details of the movement of particles as a preliminary background. The physical domain containing the particles will be discretized by a uniform mesh to form a certain number of same sized cells, each of which is either a rectangle or a box, respectively when the physical domain is 2D or 3D. Particles contained in the physical domain is organized cell by cell. The movement of the particles in one time step can be described as three sequential steps. (1) Inside each non-empty cell, the particles are first randomly split into a certain number of groups. (2) Then each group of particles independently moves from its host cell to a target cell following certain physical laws. (3) At each target cell, the incoming particle groups are inserted and merged together. From the description of the three steps, three main characteristics of such a movement scenario are concluded as follows.

(i) Most of cells in the physical domain are empty in the whole simulation time.

(ii) The number and location of non-empty cells vary over time, which is full of dynamic.

(iii) For any non-empty cell, it needs a complex data structure to save values of velocity, acceleration, mass, etc., for particle groups, which can be substantial on memory use.

1.3.1 Data structure design

For each non-empty cell, to study all its resident particles as one unity, a complex data structure is needed per cell to store information such as the cell position in the physical domain, physical quantities of particles, etc. A naive implementation is to create a multi-dimensional array, which is of the same dimension as the background mesh, such that each array entry is a data structure mentioned above. Each array index naturally corresponds to the cell position in the physical domain while each array element is a complex data structure to save physical quantities. This design has deficiencies in the sense that the memory utilization is low because most of cells, i.e. array elements, are empty in the whole simulation time. Meanwhile, with the progress of the simulation, the physical distance between a source cell and a target cell may be reflected as large memory jumps when moving a group of particles.

In order to improve the memory utilization, a more practical design is to use compressed data structures solely storing those non-empty cells. Thus, a short 1D array called `compressedData`, of which each entry corresponds to a non-empty cell, is adopted for this goal. Because the indices of `compressedData` cannot tell the physical positions of the non-empty cells in this design, the position information will be stored in a separate 1D array of integers, named `Nidx`, of same length of the `compressedData` array. The value of `Nidx[i]` records the physical position of the i'th non-empty cell inside array `compressedData`.

In connection with implementing a particle group's movement from its host cell to a target cell, there is the need for quickly checking whether the target cell is empty or not. Moreover, if the target cell is non-empty, we need to know where its associated data structure can be found inside `compressedData`, so that the incoming particle group can be added. An assistant array named `lookup` is therefore introduced. The length of the 1D `lookup` array equals the total number of cells in the background mesh, each entry of `lookup` is an integer, and we have chosen a convention such that `lookup[i]<0` means cell number i of the background uniform mesh is empty. Otherwise, `compressedData[lookup[i]]` returns the associated data structure of the non-empty cell i.

For simplicity, a 2D scenario is selected for an example without loss of generality. In Figure 5, the 2D domain is divided into 10×10 cells while `Nidx` is of the length 10. As illustrated in Figure 5, the mapping between `Nidx` and `lookup` can be expressed as `lookup[Nidx[i]]=i`.

Although searching the `Nidx` array can also tell us whether a specific cell is occupied, which is the main function of `lookup`, we have chosen to use `lookup` for the performance concern.

Because time dependent simulations normally are conducted via updating relevant variables from the current time step to the next one, two copies of `Nidx` and `compressedData` are needed. We denote the status of the current time step by `compressedDataCurrent` and `NidxCurrent` while using `compressedDataNext` and `NidxNext` for the next time step. To save memory, only one copy of `lookup` is used. At the start of each time step, the old content of `lookup` is typically erased so `lookup` is refilled to correctly record

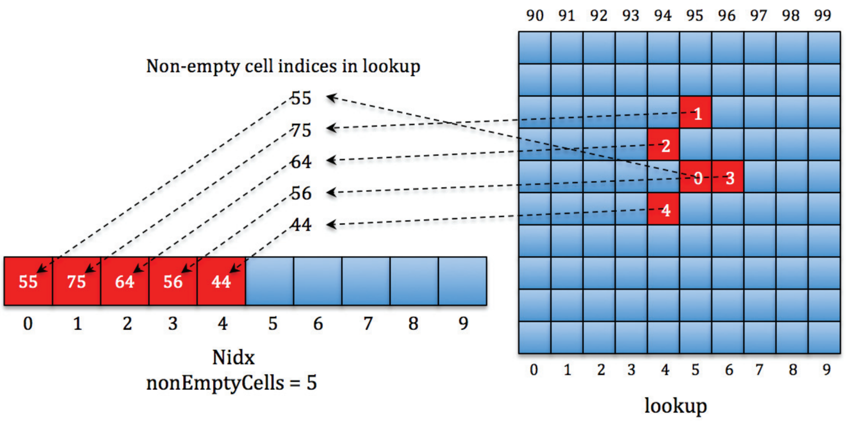

Figure 5: An illustration of the mapping between `Nidx` and `lookup`.

the current occupancy of cells.

1.3.2 Parallelizing particle movement

As mentioned earlier, at the start of every time step, the particles that reside inside each non-empty cell randomly form a few particle groups. (The number of particle groups per host cell is stored as an integer variable called `numParticleGroups`.) The computation associated with forming particle groups is easily parallelized by adding `# pragma omp parallel for` ahead of a for-loop that iterates over the entire `compressedDataCurrent` array.

However, it is not straightforward to parallelize the remaining computations that are related to moving the particle groups from each host cell to the diverse target cells. To demonstrate the difficulty, let us first show the sequential implementation in Figure 6.

From the code segment shown in Figure 6, one can see that it is not feasible to directly parallelize, by using "`pragma omp parallel for`", the loop of `for (int i=0; i<nonEmptyCells; i++)`. The difficulty arises from a high possibility of race conditions. More specifically, there are two types of race conditions, (1) updating `lookup` and `NidxNext` contained in the `if` block, (2) the call to `addParticles` that reads particle groups in a host cell and respectively adds them in the corresponding target cells. For the sake of parallelization, the computations shown in Figure 7 can be divided into two steps, the first step being carried out by a single OpenMP thread, whereas the second step modifies the original sequential algorithm to enable parallel execution by multiple threads.

```
nonEmptyCellsNext = 0;
for (int i=0; i<nonEmptyCells; i++) {
  for (int j=0; j<compressedDataCurrent[i].numParticleGroups; j++) {
    targetCell = computeTargetCell( compressedDataCurrent[i].particleGroup[j] );
    if (lookup[targetCell] < 0) {
      lookup[targetCell] = nonEmptyCellsNext;
      NidxNext[nonEmptyCellsNext] = targetCell;
      nonEmptyCellsNext++;
    }
    addParticles( compressedDataCurrent[i], j,
                  compressedDataNext[ lookup[targetCell] ] );
  }
}
```

Figure 6: A serial algorithm of particle movement.

(i) First, the new storage places of all non-empty cells after the particle movements are updated in the compressed data structures by a serial code. Specifically, the master thread increases the `nonEmptyCellsNext` counter and fills arrays `lookup` and `NidxNext` with new entries. In addition, for each outgoing particle group, the master thread also calls a new function named `register`, which only records two integers into the target cell's data structure. (The first integer records the particle group's host cell index, while the second integer records the particle group's local group index within its host cell.) The `registerInTargetCell` function actually requires a small extension of the original cell data structure, necessary for parallelizing the following step.

(ii) Then, the contents of those new non-empty cells in the compressed data structure are updated in parallel. Concretely, after the master thread has rearranged places of non-empty cells in the compressed data structures, for each target cell, minimalistic information about all its incoming particle groups (i.e., two integers per incoming group), a for-loop can iterate over all the target cells one by one. This for-loop has the purpose of letting each entry of the `compressedDataNext` array to call the computationally heavy `addParticles` function, for merging all its incoming particle groups, whose complete information is fetched from `compressedDataCurrent`. There is no risk of race condition with such a for-loop, which can therefore be parallelized by enforcing `#pragma omp parallel for`.

The above two steps can be implemented as shown in Figure 7.

It is actually possible to involve multiple OpenMP threads to partially parallelize Step 1, while still avoiding race conditions, as shown in Figure 8. This only applies to calls of function `registerInTargetCell`. As before, we can first let the master thread to increment the `nonEmptyCellsNext` counter and fill new values in arrays `lookup` and `NidxNext`. Thereafter, we can parallelize the calls to `registerInTargetCell`. Time measurements show that such a partial parallelization of Step 1 does pay off, in comparison with letting a single OpenMP thread to execute the entire Step 1.

```
//Step 1
nonEmptyCellsNext = 0;
for (int i=0; i<nonEmptyCellsCurrent; i++) {
  for (int j=0; j<compressedDataCurrent[i].numParticleGroups; j++) {
    targetCell = compressedDataCurrent[i].particleGroup[j].targetCell;
    if (lookup[targetCell] < 0){
      lookup[targetCell] = nonEmptyCellsNext;
      NidxNext[nonEmptyCellsNext] = targetCell;
      nonEmptyCellsNext++;
    }
    registerInTargetCell(compressedDataNext[lookup[targetCell]], i, j);
  }
}

//Step 2
#pragma omp parallel for default(shared) private(cell_id, group_id)
for (int i=0; i<nonEmptyCellsNext; i++) {
  for (int j=0; j<compressedDataNext[i].numIncomingParticleGroups; j++) {
    getSource(compressedDataNext[i].enteredGroups[j], &cell_id, &group_id);
    addParticles(compressedDataCurrent[cell_id], group_id,
                 compressedDataNext[i]);
  }
}
```

Figure 7: A parallel implementation of particle movement.

1.3.3 Memory performance enhancements

When studying the data access patterns involved in the above OpenMP parallel implementation, we can see that the main memory is likely accessed with large and potentially irregular jumps. These memory jumps happen when recording the two integers from each particle group, whose data resides in the `compressedDataCurrent` array, into its target cell that has its data residing in the `compressedDataNext` array. Similarly, irregular jumps in the memory also happen when we later iterate over the `compressedDataNext` array and repeatedly call the `registerInTargetCell` function, which needs to fetch data from the `compressedDataCurrent` array.

Moreover, if consecutive cells within the `compressedDataCurrent` array in fact spread randomly out in the physical domain, consecutive cells within the resulting `compressed-DataNext` array will likely spread out more randomly. Such random physical locations of the consecutive cells inside `compressedDataCurrent` will aggravate the memory jumps described above.

As a partial remedy to the aggravated memory jumps, we have adopted a sorting procedure (shown in Figure 9) that reshuffles `NidxCurrent` and `compressedDataCur-rent`, such that the entries of the reshuffled `NidxCurrent` array has an increasing order. The rationale behind this simple sorting procedure is that consecutive cells within `compressedDataCurrent` should become somewhat closer to each other in the physical domain. If, in addition, particles do not move not very far (from host to target cells) per time step, consecutive cells within the resulting `compressedDataNext` array will not be too far from each other in the physical domain either. Therefore, after using the sorting procedure, the memory jumps will likely become smaller and less irregular, thus

```
//Step 1.1
nonEmptyCellsNext = 0;
for (int i=0; i<nonEmptyCellsCurrent; i++)
  for (int j=0; j<compressedDataCurrent[i].numParticleGroups; j++) {
    targetCell = compressedDataCurrent[i].particleGroup[j].targetCell;
    if (lookup[targetCell] < 0) {
      lookup[targetCell] = nonEmptyCellsNext;
      NidxNext[nonEmptyCellsNext] = targetCell;
      nonEmptyCellsNext++;
    }
}

//Step 1.2
#pragma omp parallel default(shared)
{
  //Calculating the index range for the current thread.
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();
  int index1 = (thread_id * nonEmptyCellsNext) / num_threads;
  int index2 = ((thread_id+1) * nonEmptyCellsNext) / num_threads;

  for (int i=0; i<nonEmptyCellsCurrent; i++)
    for (int j=0; j<compressedDataCurrent[i].numParticleGroups; j++) {
      targetCell = compressedDataCurrent[i].particleGroup[j].targetCell;
      if (index1 <= targetCell && targetCell < index2)
        registerInTargetCell(compressedDataNext[lookup[targetCell]], i, j);
    }
}
```

Figure 8: A partial parallelization of Step 1.

benefiting the cache usage. The sorting algorithm is depicted in Figure 9.

```
int tcounter = 0;
int tPos;
for (int i = 0; i<size; i++)
  if ((tPos = lookup[i]) > -1) {
    compressedDataNext[tcounter].data = compressedDataCurrent[tPos].data;
    NidxNew[tcounter] = i;
    tcounter++;
  }
```

Figure 9: Sorting of “Nidx” and “compressedData”.

It should be remarked that the above sorting procedure aims to improve the memory and cache performance, but the simple sorting criterion can not of course guarantee to minimize the memory jumps. Moreover, there is overhead associated with reshuffling `NidxCurrent` and `compresseDataCurrent`. This means that the sorting procedure should probably not be called every time step.

As a second memory performance enhancing strategy, which applies to NUMA systems, we have adopted the strategy of First Touch [16] to initially place the data of `compressedData` evenly across multiple NUMA regions, as shown in Figure 10. More specifically, `#pragma omp parallel for schedule(static, chunksize)` is added to the for-loop that initializes the `compressedData` array. Although subsequent particle movements will eventually destroy the perfect initial data distribution of `compressed-Data` among multiple NUMA regions, numerical experiments suggest that performing the First Touch always pays off, in comparison with letting the entire `compressedData` initially reside on a single memory region.

```
#pragma omp parallel for default(shared)
for (int i = 0; i<MaxNonEmptyCells; i++) {
  compressedDataNext[i].data = 0;
  compressedDataCurrent[i].data = 0;
}
```

Figure 10: Using "First Touch" via a parallel initialization for evenly distributing data on all NUMA regions.

1.4 Performance modeling

Parallel programs normally have a much longer life than that of parallel computers, it is therefore important to study means of evaluating programs' performance on not only existing machines but also future ones [10]. Many performance modeling approaches have been studied in previous work with different focuses. For examples, Performance Evaluating Virtual Parallel Machine (PEVPM) [15] adopts statistical means to analyze MPI communication but is only dedicated for MPI programs, while [30] is capable of coping with MPI, OpenMP and their hybrid programs but the model heavily relies on measurements and lacks depiction of estimating computing time on core level. The approach [18] used by Performance and Architecture Laboratory parameterizes source codes via a full analyzation and hardware through measurements, which is not straightforward to be applied on complex codes. The survey work [10] points out most of approaches are capable of coping with small programs meanwhile it is impractical to apply them on those complex codes. As a simpler alternative, [29] introduces a means focusing on "bound and bottleneck analysis", which may provide valuable insight into how the critical bottlenecks affect the performance of computer systems through quantified methods [19].

In this thesis, we proposed a model framework for two practical goals: (1) a model concept can be easily used for predicting run times of various complex codes by different means of using cache bandwidths for different multi-core structures, and (2) it should be able to reflect performance bottlenecks to guide performance tuning. Different from [15], [30] and other similar work, the model concept proposed in this thesis is applicable for both sequential and parallel codes. Our model needs simply counting memory traffic and floating operations and parameterizing machines via measurements, which is similar to [18]. Meanwhile, different from [18], we don't need to fully analyze source code. In essence, our model is close to [29]. Through focusing on bandwidths of different memory hierarchies, our model predicts low-bound of run time that is either memory or computing bound.

On recent multi-core architectures, memory bandwidth is a key for performance modeling, e.g. [30] and [20]. Our model parameterizes memory system via standard benchmark software, such as STREAM and STREAM2 [4]. Although this kind of performance modeling needs some initial running on the target computer system, it is still very useful for code optimization and can be used as a reference when the target computer system is changed.

The whole model is based on a simple philosophy of adopting simplified assumptions. Thanks modern hardware features, e.g. pipeline and prefetching, the data transfer through multiple level caches and main memory can proceed simultaneously with executing floating operation on cores. Most modern multi-core processors contain up to 3 level caches that differ on reading/writing bandwidth. For the purpose of predicting bound of computing time, we assume the following parts can fully overlap with each other. The first part is the time consumed on executing floating operations on CPUs where data only come from CPU registers. The second part is the time of transferring data between L1 cache and CPU registers. The third part is thus for transferring data between L2 cache and CPU registers. Similarly, the fourth is the data transferring between L3 cache and CPU registers and the fifth part is that between main memory and CPU registers.

As we assumed the above five parts can proceed simultaneously, the whole running time should be determined by the maximum one. For generality consideration, we assume the problem size is larger than L3 cache size to ensure forementioned five parts will be all included in the following discussion.

As to the L1 cache, all data saved in registers have to be read/written from/to L1 cache, which has the maximum data transferring volume meanwhile the maximum bandwidth. Without thinking of register reuse, the L1 reading/writing is straightforward to count, which comes directly from code expression. As to the L2 and L3 caches, the intermediate levels of memory hierarchies, it is difficult to count reading/writing precisely in general when problem size is larger than L3 cache size because the cache utilization will be affected by memory access pattern, mesh size, mesh shape, etc., many factors. Regarding the main memory, which is the last memory hierarchy, one can desire a lower bound estimation on the data volume that needs to be transferred into registers. For good feasibility in practice, L2 and L3 caches can be ignored although it causes some precision loss.

The same philosophy also fits for the case, where the problem size is less than a certain level of cache. The last part of memory hierarchy can be notated as the involved Last Level Memory (LLM) and thus LLM can be either L2 cache, L3 cache or main memory depending on problem size.

1.4.1 Sequential performance model

When constructing the sequential model, hardware can be abstracted to a group of parameters as follows without considering its multi-core structure.

(i) The peak performance of the core, i.e. the number of floating-point operations can be executed per second without need of data outside of CPU registers, is denoted as F.

(ii) The bandwidth of reading data from L1 cache to CPU registers is represented as B^r_{L1}. Similarly, B^w_{L1} is used for writing register data to L1 cache.

(iii) The reading/writing bandwidth between registers and LLM is denoted as B^r_{LLM} and B^w_{LLM}.

The corresponding program characteristics also need to be abstracted in the similar way showing as follows.

(i) The total number of floating operations can be denoted by n_{flop}.

(ii) We denote the reading volume from L1 cache by n^r_{L1} and writing volume to L1 cache by n^w_{L1}.

(iii) Similarly, reading volume from LLM can be represented by n^r_{LLM} while writing volume to LLM can be represented by n^w_{LLM}.

Considering reading and writing can proceed in parallel in L1 cache for most of CPUs while they perform sequentially in other lower memory hierarchies, then we can derive the following formula from the above assumptions and definitions.

$$T_{run} = \max(\frac{n_{flop}}{F}, \frac{n^r_{L1}}{B^r_{L1}}, \frac{n^w_{L1}}{B^w_{L1}}, \frac{n^r_{LLM}}{B^r_{LLM}} + \frac{n^w_{LLM}}{B^w_{LLM}}). \tag{1}$$

Furthermore, reading is more than writing in most of scientific computing codes, e.g. solving diffusion equations and wave equations. Thus, Equation 1 can be more simplified as follows.

$$T_{run} = \max(\frac{n_{flop}}{F}, \frac{n^r_{L1}}{B^r_{L1}}, \frac{n^r_{LLM}}{B^r_{LLM}} + \frac{n^w_{LLM}}{B^w_{LLM}}). \tag{2}$$

Let's start from a concrete case to explain how to apply this model, of which the problem size is larger than the size of L3 cache, thus where LLM is the main memory. Figure 11 shows a small piece of kernel code of solving heat diffusion equation on 2D domain.

It is very important in our model to distinguish reading/writing in L1 cache and LLM, which incurs different values. From Figure 11, one can easily count 5 readings and 1 writing for each grid point. But what we want to emphasize is that without thinking of register reuse, this kind of counting actually gives the values of reading or writing data from/to in L1 cache level. As to LLM level, i.e. main memory in this case, there are

```
// time loop
for ( it = 1; it <= num_steps; it++ ) {
   // spatial loop on a 2D grid
   for ( j = 1; j <= Y_WIDTH; j++ )
     for ( i = 1; i <= X_WIDTH; i++ )
       un[j][i] = C1*(u[j-1][i]+u[j][i-1]+u[j][i+1]+u[j+1][i])+C2*u[j][i];

   //swap array pointers
   swap(&u, &un);
}
```

Figure 11: The computing kernel of heat diffusion on a 2D domain.

one read for u and one write for un. As to the array u, we count it as one read based on a low-bound estimation that the whole array should be read from memory to CPU registers at least one time in each time step. In this case, we thus have the following values:

(i) $n_{flop} = 6 \times X\_WIDTH \times Y\_WIDTH \times num\_steps$,

(ii) $n^r_{L1} = S \times 5 \times X\_WIDTH \times Y\_WIDTH \times num\_steps$,

(iii) $n^w_{L1} = S \times X\_WIDTH \times Y\_WIDTH \times num\_steps$,

(iv) $n^r_{LLM} = S \times X\_WIDTH \times Y\_WIDTH \times num\_steps$,

(v) $n^w_{LLM} = S \times X\_WIDTH \times Y\_WIDTH \times num\_steps$,

where S is the size of each array element normally having value of 4 or 8 bytes for single or double precision.

1.4.2 Parallel performance model

The parallel model is derived from its sequential counterpart. More hardware parameters reflecting multi-core structure are added. Although multi-core structures vary, the way of constructing a corresponding model is similar. Here, we adopt a typical multi-core structure that each core has its own L1 and L2 caches while all cores in the same processor share a L3 cache. One of the popular multi-core based processor families using this structure is Nehalem from Intel. The following are updated hardware parameters for the parallel performance model.

(i) The total number of cores in each node is denoted as p while its peak performance of each core is still represented as F.

(ii) The bandwidth of reading data from each private L1 cache to CPU registers is represented as B^r_{L1}. Writing bandwidth is ignored for the same reason stated in sequential model.

(iii) The bandwidth of reading and writing data from LLM to CPU registers is respectively represented as $B^{r,p}_{LLM}$ and $B^{w,p}_{LLM}$, where p represents the bandwidth when p cores launch reading simultaneously. More specific, in the case of a non-uniform memory access architecture when LLM is the main memory, we only consider the the B_{LLM} as transferring data between registers and its closest main memory module.

When part of cores in a node are used, these working cores can be on either one processor or all processors, which depends on mapping strategy of the OS and causes the model expression correspondingly changed. In addition, considering end users are more interested in full-core performance, the formula using all cores of one node will be depicted as follows. As we have mentioned, the multi-core architecture enables a private L1 for each core, thus the aggregate effect of L1 cache scales linearly. The model varies when LLM respectively stands for L2, L3 or main memory. When the problem size of each core is large enough to get involved in using the main memory (i.e. LLM is the main memory), a parallel performance model can be found as

$$T_{run} = \max\left(\frac{n_{flop}}{pF}, \frac{n^r_{L1}}{pB^r_{L1}}, \frac{n^r_{LLM}}{B^{r,p}_{LLM}} + \frac{n^w_{LLM}}{B^{w,p}_{LLM}}\right). \tag{3}$$

The modeling concept proposed here follows a simple philosophy, which is capable of identifying the performance bottleneck without analyzing cache misses. For specific cases, the formula may be refined but is still derived from the same idea.

2 Summary of Papers

All the following reviews will mainly focus on my work related to parallel programming and parallel computing. The details about the physics, mathematics and numerics of the simulations can be found in the subsequent chapters.

2.1 Paper I

Mathematical modeling of cardiac cells has been a great success, where models are highly nonlinear systems containing a large number of ordinary differential equations (ODEs) to describe the kinetics of ion channels, pumps, exchangers and ionic homeostasis. Such level of computing requires state-of-the-art parallel computing techniques and hardware. In addition, the scan resolution of cardiac cells improves fairly fast, which further boosts computing demand.

In this paper, we conducted a three dimensional full-scale simulation of propagating Ca^{2+} waves in a ventricular myocytes, which was the first of this kind in the world. The real data used in our case, stored within a 3D mesh of $120 \times 120 \times 570$ points, was obtained by using a combination of confocal imaging and image processing [25]. The main difficulties of obtaining the data were related to the limitations of confocal imaging. The main unknown, the Ca^{2+} concentration, is modeled by a diffusion-based PDE coupled with two ODEs. (Some of the computation is governed by a probability function based on the Ca^{2+} concentration.) Through using FDM and the Forward Euler method, respectively, in the spatial and temporal discretization, a serial code and its MPI-based parallel counterpart were developed, where the probability function was implemented as a self-developed rand48 function. Running the parallel code on a multi-core based cluster with 256 cores, the simulation time was reduced from about five days to less than one hour.

In this application scenario, the 3D computational domain has an irregular shape. In order to allow the finite difference approximation involving a standard seven-point stencil, the 3D irregular domain is "immersed" into a box-shaped uniform mesh of $120 \times 120 \times 570$ points. In Figure 12, the red part in the 3D cube represents the irregular solution domain. (Each mesh point that lies outside the solution domain is marked as non-computational.) A general 3D partitioning of the box-shaped uniform mesh can lead to a work load imbalance because the sub-meshes may contain substantially different numbers of computational mesh points. Meanwhile one can find that the shape of the computational domain in the longitudinal axis doesn't change sharply (see Figure 12), therefore a 1D partitioning of the 3D box-shaped mesh by using cutting planes orthogonal to the longitudinal axis was adopted. Our parallel solution is to use the flat MPI approach for the sake of simplicity, which ignores the shared-memory property of each multi-core based cluster node.

Main Results. Because it could take several days for the serial code to finish a complete simulation, a serial simulation of 20ms was adopted for estimation, which took 3332.5 seconds wall time on a single core of a Xeon E5420 2.5 GHz processor. For a full-scale simulation of 2.5 seconds, the serial code would have taken $2.5/0.02 \times 3332.5 = 416562$ seconds wall time. For the corresponding parallel simulation, 32 compute nodes interconnected via a Gigabit ethernet were used. Each node was equipped with two quad-core Xeon E5420 2.5 GHz processors, thus in total 256 cores were used. The full-scale parallel simulation of 2.5 seconds using 256 cores took 2501.71 seconds wall time, which can be translated to a speedup of 166.5.

2.2 Paper II

The work of this paper is an extension of Paper I using the same mathematical model and data. The extension adopted mixed programming, i.e. MPI combined with OpenMP, and associated optimizations for a better computing load balance.

For the same reason as explained in Paper I, a 1D partitioning along the longitudinal axis was used with 2D cutting planes. Meanwhile, mixed programming is adopted for better load balance, which has been explained in Section 1.2. Specifically, if flat MPI is

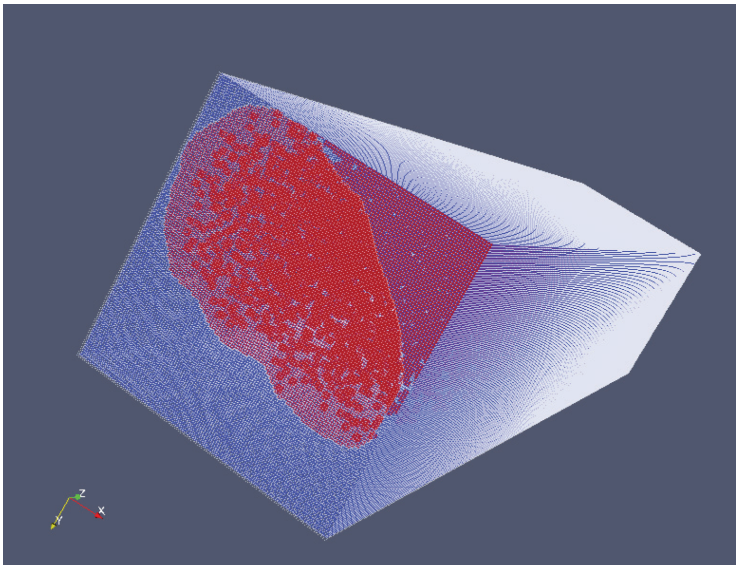

Figure 12: The irregular solution domain .

applied to such a case, where the number of mesh points in the longitudinal direction is fixed as 570, the maximum number of MPI processes cannot exceed this number. Another deficiency of using flat MPI in such a case is that the partitioning can be quite imbalanced when many CPU cores are used. For an example, when 32 computer nodes are used, each having 8 cores (i.e. 256 cores in total), most cores are given two x-y planes while others get three. This leads to a 50% load difference. This deficiency can be improved by a mixed programming approach dedicated to multi-core architecture consideration.

When using a mixed programming mode on 32 nodes, one MPI process is typically assigned to each node, where 8 OpenMP threads will be spawned. Instead of using 256 cores to partition the 570 x-y planes, the mixed programming code uses 32 nodes to do the partitioning. The number of planes allocated for each MPI process is 17 or 18, which only has a 5.88% load difference.

OpenMP Optimization for Multi-core. For the reasons we explained in Section 1.2, a loop nest of three levels iterating the 3D subdomain should be collapsed to one long for-loop for finer parallel granularity of OpenMP. A further optimization called "compacted indices" well explained in Section 1.2 is adopted here mainly for a better load balance, which also reduces the overhead of checking if a mesh point is computational or not.

For further performance enhancement, we have used non-blocking MPI send and receive calls (instead of blocking MPI calls) to overlap communication with computation. Table 1 compares the speed of the three parallelization approaches for carrying out a 2.5-second simulation of calcium wave propagation. The time step is $\Delta t = 10^{-5}$s, the box-shaped global computational mesh has 570 points in the longitudinal direction, and 120 points in each of the two other spatial directions. Time usage measurements were obtained on a cluster using Gigabit Ethernet as the interconnect. Each compute node has two Xeon E5420 2.5GHz quad-core processors, i.e., eight cores per node. We can see from Table 1 that the first mixed MPI-OpenMP approach using blocking MPI calls is only advantageous over the flat MPI approach from [21] when the number of subdomains is large. The second approach of mixed MPI-OpenMP parallelization, which overlaps communication with computation, consistently outperforms the other two parallelization approaches.

Number of	Flat MPI		MPI+OpenMP		MPI+OpenMP	
	blocking MPI calls		blocking MPI calls		nonblocking MPI calls	
cores used	subdomains	wall time	subdomains	wall time	subdomains	wall time
32	32	14590.80	4	14136.46	4	13750.94
64	64	7330.12	8	7343.90	8	7075.45
128	128	4130.34	16	3932.86	16	3673.53
256	256	2502.88	32	2258.35	32	2058.95

Table 1: Comparison of three parallelization approaches, where the approach 1 and 2 respectively stands for non-overlapped and overlapped communication.

Main Results. In this paper, we have studied how to improve and optimize a parallel code using mixed programming with multi-core consideration, for a case coming from the real world. The new code using mixed programming achieves a much better computing load balance, which can run considerably faster than the flat MPI version used in Paper I.

2.3 Paper III

A new numerical strategy, denoted as the *lumped particle model*, is proposed in [8]. It can be used to simulate particle movements that are caused by dispersion and diffusion. In both types of movement, particles are studied in term of groups rather than individually. The new model inherits the advantages of both continuous and discrete models, which are the traditional numerical strategies for simulating particle movements. Parallel computing is also of great importance for this lumped particle model, due to the following reasons. (1) The computation load may rise significantly when multiple particle-laden flows are considered at the same time. (2) High grid resolution is necessary for accurate simulations of debris flows. (3) 3D simulations, which are considerably more computationally challenging, are needed for many realistic applications.

In this paper, we effected an OpenMP parallel implementation of the lumped particle model in a 2D scenario. Optimizations were adopted for multi-core systems, and performance comparison was conducted on three different hardware platforms.

Parallel Computing and Optimization. As mentioned, the lumped particle model describes two physical phenomena, dispersion and diffusion. Both phenomena cause particle movements that can be simulated using the parallel algorithm introduced in Section 1.3. It should be remarked that the particle movement caused by diffusion is more regular than that caused by dispersion. For diffusion-induced movement, the particles in each cell split up symmetrically and move into its eight nearest neighboring cells. The entire physical process, containing both dispersion and diffusion, is divided into five software tasks in each time step. Here we briefly explain these tasks relative to the parallel algorithm described in Section 1.3.

(i) Related to dispersion, this step calculates the target cell for each particle group and temporally stores this information in `compressedDataCurrent`. Its OpenMP parallelization is straightforward and corresponds to the initial process before Step 1 mentioned in Section 1.3.

(ii) Similar to Step 1 described in Section 1.3, the actual movement of dispersed particle groups is further divided into two sub-steps, corresponding to Step 1.1 and Step 1.2. The former is completely serial while the latter is parallelized.

(iii) This step slightly extends Step 2 of Section 1.3, which for each non-empty cell first merges all the incoming particle groups and then splits the particles again into a certain number of groups, following the physical principle of dispersion.

(iv) Related to diffusion, this step uses a simplified version of Step 1 from Section 1.3. In essence, each particle group is symmetrically split up and ready to move into the nearest neighboring cells.

(v) The step concludes diffusion and uses a simplified version of Step 2 from Section 1.3.

Main Results. The OpenMP implementation gives decent parallel performance and shows that the implicit control on data locality via Parallel First Touch is important for OpenMP scalability on the NUMA architecture. The "sorting" optimization results in a better cache utilization, which further improves the overall performance.

2.4 Paper IV

In this paper, we discretized and parallelized a dual-sediment transport model, which is constructed by two coupled diffusion based PDEs. Specifically, we developed two numerical methods, i.e. fully-explicit and semi-implicit methods and effected their parallel implementations using MPI, to solve the non-linear sedimentation model. The numerical stability, scalability and speed of the two parallel implementations are compared on a multi-core based cluster.

The parallelization strategy is straightforward, where we apply 2D partitioning on the whole solution domain. MPI is adopted for inter-subdomain communication. A third party parallel numerical package Trilinos [5] is used in the semi-implicit method to solve linear systems of the two PDEs respectively with the CG and GMRES solvers.

Main Results. Through a case adopting real bathymetry data of Lake Okeechobee, southern Florida, a $10,000$-year simulation successfully diffused material along a river-channel and into the lake. The semi-implicit method has demonstrated a superior stability in all our numerical experiments.

The speed of the fully-explicit scheme is around 100 times faster than that of semi-implicit because the latter is much more complex due to solving two linear systems per time step, which are typically memory bound. The fully-explicit scheme has better parallel efficiency when using small numbers of cores, while the semi-implicit scheme scales better on large numbers of cores. This is due to the unfavorable computation-communication ratio of the fully-explicit scheme.

2.5 Paper V

This paper is an extension of Paper IV. The new work mainly consists of two parts, which respectively focus on numerical methods and performance modeling.

In the first part, in addition to the fully-explicit and semi-implicit methods from Paper IV, we developed another semi-implicit method using the Crank-Nicolson scheme, which can achieve second-order accuracy in the temporal direction. Meanwhile, two fully-implicit methods based on the Newtwon-Raphson method were also developed, which are first/second-order accurate in the temporal direction. In this part, we also developed a set of methodologies about how to select a suitable numerical method and find the corresponding time step size when the total simulation error is indirectly given. Through the set of methodologies, specifically, we showed how to analyze the spacial and temporal errors of related numerical methods and compared their numerical stability through numerical experiments.

In the second part, we proposed a simple performance model to predict the overall computation time on the multicore architecture, applicable to many numerical implementations. As we have explained in Section 1.4, the main philosophy of the performance model is to predict a lower bound of time usage by a parallel application.

Main Results. High-resolution bathymetric data of Monterey Bay is adopted as a case for all the numerical experiments. Regarding the implementation, the fully-explicit method is the easiest and does not need the external software, while it has the worst stability that restricts the time step size seriously. The two fully-implicit methods are complex to implement and heavy for execution. The semi-implicit scheme without Crank-Nicolson has the best numerical stability that allows large time steps. Further more, both semi-implicit methods are practical to be extended to a multi-lithology sedimentation model. Their memory usage does not increase when solving more sediments, because the linear system of each sediment will be solved one by one while reusing the same data structures.

Tianhe-1A Hunan Solution [6], a supercomputer ranked No. 26 in TOP 500 of 2012, was adopted to verify the performance model. The predicted result of speed comparison between the full-explicit and two semi-implicit methods matches the actual measurement.

3 Future work

The following aspects of the PhD work may be worth pursuing in future.

With the development of scan techniques of cardiac cells, super-high resolutions of nano-scale structures have been available [17]. The work of Paper I and Paper II may be extended to the super-high resolutions, which may need dozens of thousands of processor cores. Meanwhile, another promising hardware for parallel computing, Graphics Processing Unit (GPU), should be adopted by using Compute Unified Device Architecture (CUDA) [7] or Open Computing Language (OpenCL) [1].

As we have learnt from Paper III, the simple sorting procedure introduced there can considerably improve the performance. However, we lack a quantitative understanding. Further analyses, together with a quantitative model of the incurred data access patterns, can be carried out as future work. The purpose is to improve the sorting procedure for achieving even better performance.

The work of Paper IV and Paper V can be extended from three avenues. First, the semi-implicit methods should be applied to cases with three or more lithologies. The semi-implicit strategy is inherently superior to its fully-implicit counterpart in this respect, because adding one lithology brings an additional equation of the same type as the s-equation. The software components of the semi-implicit methods can thus be readily reused, unlike the fully-implicit methods that have to solve larger and larger nonlinear/linear systems. Second, for computations whose L1$\leftrightarrow$L2 and L2$\leftrightarrow$L3 data traffic can be quantified, the proposed performance prediction model should be extended to consider these data movements as well. Such an extended model avoids gravely optimistic predictions, when the actual performance bottleneck is between L1 and L2 or between L2 and L3. Third, the same simple philosophy of performance prediction should be extended to GPUs. In the collaboration work [28] that is not included in this thesis, a pure GPU implementation and a GPU-CPU hybrid implementation of the fully-explicit method have been developed. In order for the GPU-CPU hybrid implementation to automate the work division between CPUs and GPUs for the best load balance, performance models for both devices are needed.

In conclusion, possible long-term targets are (1) a quantitative model of the data access patterns involved in computations related to Paper III, (2) improvements of the performance prediction model for CPUs, while also extending it to cover GPUs, and (3) extensions of the parallel semi-implicit methods to handle challenging real-world sedimentation models that involve more than two lithologies.

Bibliography

1. *OpenCL - The open standard for parallel programming of heterogeneous systems.* http://www.khronos.org/opencl.

2. *OpenMP Specifications.* http://openmp.org/wp/openmp-specifications.

3. *PAPI: Performance application programming interface.* http://icl.cs.utk.edu/papi/.

4. *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* http://www.cs.virginia.edu/stream/.

5. *The Trilinos Project Home Page.* http://rilinos.sandia.gov/.

6. *Tianhe-1A Hunan Solution on TOP500 website.* http://i.top500.org/system/177448.

7. *What is CUDA.* http://developer.nvidia.com/cuda/what-cuda.

8. O. Al-Khayat, A. M. Bruaset, and H. P. Langtangen, *A lumped particle modeling framework for simulating particle transport in fluids*, Communications in Computational Physics, 8 (2010), pp. 115–142.

9. D. Bell and G. Wood, *Multicore programming guide*, Application Report SPRAB27A, Texas Instruments, 2009.

10. I. Brandic, S. Benkner, and S. Pllana, *Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art*, in Proceedings of the First International Conference on Complex, Intelligent and Software Intensive Systems, April 2007, pp. 279–284.

11. B. M. Chapman, *The multicore programming challenge*, in Advanced Parallel Processing Technologies, vol. 4847 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 3–3.

12. D. R. Cheriton, H. A. Goosen, and P. D. Boyle, *Multi-level shared caching techniques for scalability in VMP-MC*, in Proceedings of the 16th Annual International Symposium on Computer Architecture, June 1989, pp. 16–24.

13. J. Diamond, M. Burtscher, J. D. McCalpin, B. Kim, S. W. Keckler, and J. C. Browne, *Evaluation and optimization of multicore performance bottlenecks in supercomputing applications*, in 2011 IEEE International Symposium on Performance Analysis of Systems and Software, 2011, pp. 32–43.

14. J. Elder, A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, *Issues related to mimd, shared-memory computers: The NYU ultracomputer approach*, in Proceedings of the 12th Annual International Symposium on Computer Architecture, June 1985, pp. 126–135.

15. D. Grove and P. Coddington, *Performance modeling and evaluation of high-performance parallel and distributed systems*, Performance Evaluation, 60 (2005), pp. 165–187.

16. R. Iyer, H. Wang, and L. N. Bhuyan, *Design and analysis of static memory management policies for CC-NUMA multiprocessors*, Journal of Systems Architecture, 48 (2002), pp. 59–80.

17. I. D. Jayasinghe, D. Baddeley, C. H. Kong, X. H. Wehrens, M. B. Cannell, and C. Soeller, *Nanoscale organization of junctophilin-2 and ryanodine receptors within peripheral couplings of rat ventricular cardiomyocytes*, Biophysical Journal, 102 (2012), pp. 19–21.

18. D. Kerbyson, A. Hoisie, and H. Wasserman, *Use of predictive performance modeling during large-scale system installation*, Parallel Processing Letters, World Scientific Publishing Company, 15 (2005), pp. 387–395.

19. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*, Prentice-Hall, Inc., 1984.

20. J. Levesque, J. Larkin, M. Foster, G. G. J. Glenski, S. Whalen, B. Waldecker, J. Carter, D. Skinner, H. He, H. Wasserman, J. Shalf, H. Shan, and E. Strohmaier, *Understanding and mitigating multicore performance issues on the AMD Opteron architecture*, (2007). http://escholarship.org/uc/item/9k38d8ms.

21. P. Li, W. Wei, X. Cai, C. Soeller, M. Cannell, and A. V. Holden, *Evolution of intracellular Ca2+ waves from about 10,000 RyR clusters: Towards solving a computationally daunting task*, in Proceedings of the 5th International Conference on Functional Imaging and Modeling of the Heart, 2009, pp. 11–20.

22. Z. Majo and T. R. Gross, *Matching memory access patterns and data placement for NUMA systems*, in Proceedings of the 10th International Symposium on Code Generation and Optimization, March 2012, pp. 230–241.

23. S. K. Moore, *Multicore is bad news for supercomputers*, IEEE Spectrum, 45 (2008), p. 15.

24. D. M. Pase and M. A. Eckl, *A comparison of single-core and dual-core Opteron processor performance for HPC*, tech. rep., International Business Machines, 2005.

25. C. Soeller, D. Crossman, R. Gilbert, and M. B. Cannell, *Analysis of ryanodine receptor clusters in rat and human cardiac myocytes*, Proceedings of the National Academy of Sciences of the United States of America, 104 (2007), pp. 14958–14963.

26. S. Suntorn, *Analysis of false cache line sharing effects on multicore CPUs*, Master's thesis, San José State University, January 2010.

27. J. Torrellas, H. S. Lam, and J. L. Hennessy, *False sharing and spatial locality in multiprocessor caches*, IEEE Transactions on Computers, 43 (1994), pp. 651–663.

28. M. Wen, H. Su, W. Wei, N. Wu, X. Cai, and C. Zhang, *Using 1000+ GPUs and 10000+ CPUs for sedimentary basin simulations*, in Proceedings of IEEE International Conference on Cluster Computing 2012.

29. S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Communications of the ACM - A Direct Path to Dependable, 52 (2009), pp. 65–76.

30. X. Wu and V. Taylor, *Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore cluster systems*, in Proceedings of the 14th IEEE International Conference on Computational Science and Engineering, 2012, pp. 181–190.

Paper I:

Evolution of Intracellular Ca^{2+} Waves from about 10,000 RyR Clusters: Towards Solving a Computationally Daunting Task

Paper II:
Computational Modeling of the Initiation and Development of Spontaneous Intracellular Ca^{2+} Waves in Ventricular Myocytes

Paper III:
An OpenMP-enabled Parallel Simulator for Particle Transport in Fluid Flows

Paper IV:
Numerical Analysis of a Dual-sediment Transport Model Applied to Lake Okeechobee, Florida

Paper V:
Balancing Efficiency and Accuracy for Sediment Transport Simulations