Improving the performance of Molecular Modelling code in TINKER

Justs Zarins

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2014
Abstract

The goal of this project was to improve the performance of implicit solvation calculations in the molecular dynamics package TINKER. The code’s OpenMP parallel coverage was extended from about 35% to nearly 90% when using the test system with 2099 particles (3BDC). Also, the computational complexity of an electrostatic energy calculation routine was reduced by implementing a neighbor list algorithm. These changes resulted in an increase in speedup from 1.4 to 3.9 which is less than predicted by Amdahl’s law. Some evidence from parallel profiling and hardware counters was found that indicates that the cause is parallel overheads and an increase in stalled front-end cycles per instruction. Research directions for reducing these inefficiencies are offered. Increasing the problem size was found to improve efficiency of the parallelisation, but remaining serial sections grow strongly with more atoms in the system.
Contents

1 Introduction 1

2 Background 3
  2.1 Molecular dynamics ........................................... 3
  2.2 AMOEBA force field ........................................... 3
  2.3 Solvation ..................................................... 4
    2.3.1 Explicit solvation ...................................... 4
    2.3.2 Implicit solvation ...................................... 4
  2.4 TINKER ...................................................... 5
    2.4.1 OpenMP parallelisation ................................ 5
  2.5 Summary ..................................................... 6

3 Initial investigation and baseline 7
  3.1 Machine and methodology ................................... 7
  3.2 Input systems ............................................... 8
  3.3 Profiling tools ............................................... 12
    3.3.1 gprof ...................................................... 12
    3.3.2 ompP ...................................................... 12
    3.3.3 perf ....................................................... 13
    3.3.4 Doxygen .................................................. 13
  3.4 Main routines ............................................... 13
  3.5 Bugs ........................................................ 15
  3.6 Serial profiling ............................................. 16
    3.6.1 Energy ..................................................... 16
    3.6.2 Gradient .................................................. 17
  3.7 Parallel profiling ........................................... 17
    3.7.1 Parallel coverage and Amdahl’s law ................... 17
    3.7.2 Parallel scaling ......................................... 19
  3.8 Summary ..................................................... 23

4 Optimisation 24
  4.1 Optimisation roadmap ....................................... 24
  4.2 Verification ................................................ 24
  4.3 Serial optimisations ........................................ 25
  4.4 Increasing parallel coverage ................................ 27
    4.4.1 ewca() .................................................. 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2</td>
<td>ewca()</td>
<td>28</td>
</tr>
<tr>
<td>4.4.3</td>
<td>surface()</td>
<td>30</td>
</tr>
<tr>
<td>4.4.4</td>
<td>surface1()</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Algorithmic changes</td>
<td>32</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Double loop and neighbour list</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>Results and Analysis</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Strong scaling</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Weak scaling</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Neighbor list effects</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>Hardware effects</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>Future work</td>
<td>47</td>
</tr>
<tr>
<td>A</td>
<td>Key file</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Percentage of theoretical maximum speedup achieved using 3BDC and 5x3BDC as input files with initial and final optimisations. . . . . . . . . . 40
5.2 Timing breakdown of shortest run of energy() with and without neighbor lists. Times are reported in seconds for 20 executions. . . . . . . . . . . . . . . . . . . . . . 41
List of Figures

3.1 Biological assembly image of 3BDC system. Source: [1] .................. 8
3.2 Particle view of 3BDC system. Image produced by VMD. .............. 9
3.3 Change in execution time when increasing separation between 2 3BDC ”clouds”. ................................................................. 10
3.4 Particle view of 5x3BDC system. Image produced by VMD. ........... 11
3.5 Call graphs of energy() and gradient() routines. ...................... 14
3.6 energy() profile. ................................................................. 16
3.7 gradient() profile ............................................................... 17
3.8 Time spent in parallel and serial regions of energy() and gradient().
This corresponds to parallel coverage of 33% for energy() and 37% for
gradient(). Reported times are the sum over 20 runs using 3BDC system. 18
3.9 Real and theoretical max speedups of energy() and gradient() routines
before optimisation. Times with 1 thread: energy() - 27.7s, gradient() - 49.9s. Reported times are the sum over 20 runs using 3BDC system. 19
3.10 Real and ideal speedups of energy() and gradient() routines’ parallel
parts before optimisation. Ideal refers to parallel time with overhead times
removed. Times with 1 thread: energy() - 9.3s, gradient() - 18.3s. Reported times are the sum over 20 runs using 3BDC system. ............. 20
3.11 Real and theoretical max speedups of energy() and gradient() routines
before optimisation. Times with 1 thread: energy() - 353.7s, gradient() - 713.8s. Reported times are the sum over 20 runs using 5x3BDC system. 21
3.12 Overhead report for the top 3 parallel loops for a large and a small input
running energy() on 24 threads. Synchronisation and limited parallelism
overheads are not shown because they were all 0. R000## are identifiers
of particular routines. .............................................................. 22
3.13 Hardware statistics of energy() obtained with perf. This graph is almost
identical for gradient(). ............................................................... 23
4.1 Comparison of the standard optimisation flags in Intel Fortran compiler. 26
4.2 Pseudocode of ewca() routine. .............................................. 28
4.3 Pseudocode of ewca1 routine. ............................................. 29
4.4 Pseudocode of surface() routine. ....................................... 30
4.5 Pseudocode of surface1() routine. ..................................... 31
4.6 Comparison of speedup of gradient() with the surface1() function par-
allelised with critical regions or a reduction array. ........................ 32
4.7 Pseudocode of main `egk0b()` routine loop. `ipole` contains the number of the atom at each multipole site, `ne1st` stores the number of neighbors and `elst` stores a list of the neighbors.

5.1 Real and theoretical max speedups of `energy()` and `gradient()` routines with fixed problem size after optimisation. Times with 1 thread: `energy()` - 27.4s, `gradient()` - 50.0s. Reported times are the sum over 20 runs using 3BDC system.

5.2 Final parallel efficiency of `energy()` and `gradient()` for a fixed problem size.

5.3 Useful work time and overhead time in parallel regions of `energy()` and `gradient()`. Reported times are the sum over 20 runs using 3BDC system.

5.4 Real and theoretical max speedups of `energy()` and `gradient()` routines on ARCHER after optimisation. Times with 1 thread: `energy()` - 19.0s, `gradient()` - 32.6s. Reported times are the sum over 20 runs using 3BDC system.

5.5 Scaling of `energy()` and `gradient()` for a fixed problem size (524.75 atoms) per thread after optimisation. The total speedup and speedup of parallel sections alone is plotted. Timings are from 20 repeated calls to each function.

5.6 Real and theoretical max speedups of `energy()` and `gradient()` routines after optimisation. Times with 1 thread: `energy()` - 346.8s, `gradient()` - 715.0s. Reported times are the sum over 20 runs using 5x3BDC system.

5.7 Wall times of `egk0a()` (no neighbor lists) and `egk0b()` (neighbor lists) subroutines' parallel sections in `energy()` using 3BDC and 5x3BDC systems.

5.8 Instructions per cycle (IPC) and stalled cycles per instruction (SCPI) of `energy()` and `gradient()` using 3BDC system.

5.9 Instructions per cycle (IPC) and stalled cycles per instruction (SCPI) of `energy()` and `gradient()` using 5x3BDC system.

5.10 Correlation between `energy()` SCPI and aggregated parallel time spent in the `egk0b()` subroutine. The two curves labelled ”scaled” are normalised by their respective values at 1 thread. The overheads is the aggregated time across threads spent on parallel overheads in seconds.

5.11 SCPI and aggregated parallel time spent in the `surface` subroutines. The curves labelled ”scaled” are normalised by their respective values at 1 thread. The overheads is the aggregated times across threads spent on parallel overheads in seconds.

5.12 SCPI and aggregated total time spent in the main routines. The curves labelled ”scaled” are normalised by their respective values at 1 thread. The overheads is the aggregated times across threads spent on parallel overheads in seconds.
Acknowledgements

My gratitude goes to Arno, Mario and Weronika for the guidance and help throughout the project, Jay and Richard for the quick and valuable responses to my questions.
Chapter 1

Introduction

The main aim of this project was to improve the performance of a module of the molecular dynamics software toolkit TINKER [2]. The module in question performs implicit solvation calculations. This method is used to learn about the properties of molecular structures immersed in a solvent, where the solvent particles are not modelled as explicit particles but rather as an averaged field. Before starting the project parts of this code had been parallelised with OpenMP, but the execution time was not reduced significantly when run on multiple threads. Additionally, there was a wider project surrounding TINKER called APES which aimed to improve and modernise the entire codebase; one of the tasks being to reduce runtime and add parallelisation in order to use novel force fields for more realistic modelling of biomolecular systems. Observations have been made in various parts of the codebase that achieving good scaling behaviour is difficult. Thus the main tasks of this project were to investigate the current performance of TINKER’s implicit solvation code, improve it and at the same time evaluate what factors limit the scalability of the code.

Some organisational issues were encountered before the main research period began. Initially the project was going to be about porting a geophysics application to the Intel Xeon Phi card. Unfortunately, in the middle of the project preparation phase the geophysics company that I would have been collaborating with left Edinburgh and had to pull out of the project as well. An alternative was found to work on TINKER, specifically to enhance a prototype MPI implementation of a part of the codebase. Improving the distributed memory implementation would have ultimately been more beneficial to TINKER because it has the potential for the greatest speedup and it can be run on a wider range of machines. However, a more suitable project direction with more immediate demand was found near the start of the main research period. This was to improve the shared memory parallelisation of implicit solvation. Here the original code author and an active expert user were interested to see work done in this area and to contribute to it. Enhancing the implicit solvation part of the codebase was on the official "roadmap" of improvements thus it was likely to continue life as part of a release version. Corresponding with an end user of TINKER would give a link to real applications and example cases. In light of these benefits, it was deemed a better direction for the project. A drawback was that the work done during the project preparation phase became less relevant, so time during
the research period has had to be devoted to picking up background information and re-planning the work.

The remainder of this report is structured as follows. Chapter 2 outlines background of the chemistry concepts that the problem is based upon and the software package that is examined. Chapter 3 explains the experimental setup and tools used to probe performance and code structure. There is some discussion of encountered errors in the program followed by an initial investigation and profiling of the code. Chapter 4 details the serial and parallel optimisations that were applied. Next, in chapter 5 the effects of optimisation are evaluated and analysed. Finally chapter 6 draws conclusions and outlines future work.
Chapter 2

Background

2.1 Molecular dynamics

Molecular dynamics (MD) is a computational tool used to learn about chemical systems in great detail. The main benefit of using MD simulations is the ability to probe deeply into a system by knowing where every atom is and what state it is in at any given time. Simulation and modelling is regarded as the third pillar of science along with experiment and theory. With careful reference to empirical verification, it can indeed provide valuable insights and access to otherwise impossible experiments.

Some fields that utilise molecular dynamics are chemistry, biology and material science. Common uses include energy optimisation, particle trajectory tracing and protein folding.

2.2 AMOEBA force field

In computational chemistry, force fields are potential energy functions that define how particles in the system behave. The most popular force fields have historically been fixed charge ones where particles can not be polarised at all in response to electrostatic influences within the system. These are computationally cheap to use in computer simulations and have been successful in predicting properties of homogeneous systems or systems in equilibrium. However, they have not been able to deal with dynamic systems, ones that are out of equilibrium or heterogeneous. This is because the fixed charge assumption becomes too unrealistic in these conditions. In reality the potential of an environment can cause charge to shift inside atoms making them polarised, which in turn can change the potential. Hence there is a need to consider polarisable force fields.

One such force field is the Atomic Multipole Optimised Energetics for Biomolecular simulAtion (AMOEBA) [3] force field. It combines short range interactions, van der Waals (VdW) forces, permanent and, importantly, induced electric interactions. The last contribution allows electronic polarisation to take place. A polarised particle will polarise some of its neighbors which will further change the field until equilibrium is reached. Com-
putationally this makes the scheme more expensive, so code implementations strive for parallelisation. The reward is that it allows us to approach problems that were previously not suitable for computational methods using fixed charges.

2.3 Solvation

Most systems that are interesting for the study of biomolecular properties contain molecules suspended in a liquid, for example proteins inside a cell. Being in a solution changes the physical properties of the solute, so it is interesting to examine them in this state rather than in a vacuum. An important property to calculate is the solvation energy. When a solute is introduced into a liquid, it interacts with the liquid causing atoms to rearrange and to polarise. The new arrangement might be overall more favourable in decreasing the system’s energy, thus releasing the excess. Some of the released energy is used up in polarising the solvent in the first place and the remainder is the solvation energy. [4][5]

2.3.1 Explicit solvation

The obvious way to model solvation is with standard molecular dynamics. In this case every particle (both solvent and solute) has to be explicitly represented and modelled. In addition, the number of solvent molecules must be large to make sure there are no artifacts created by the potential of the solute reaching edges of the simulation box before being screened by the solvent. Naturally, simulating many particles is time consuming and a large proportion of the effort goes into simulating paths of the solvent molecules when the solute is the actual object of interest.

2.3.2 Implicit solvation

An alternative approach is to represent the solvent as a field. The solute can then be thought of as charged particles submerged in a homogeneous dielectric, polarisable material characterised by its dielectric constant. The solvent then can be as large as needed without adding additional particles to simulate. Implicit solvation also makes focusing on the solute easier by washing out unimportant details within the solute [6]. Computation time is generally reduced both in calibration and execution, and parallel scaling is usually better but at the expense of a more complicated model and possibly reduced precision. Moreover, implicit schemes depend on correctly set parameters which might not generalise to different systems. Hence they need to be treated carefully and compared against experimental results. Another factor that needs to be considered is representing the extent of the solute to show where the solvent field can or cannot physically be [7].

There exist many formulations for the computational calculation of implicit solvation, but in this project the generalised Kirkwood (GK) [8] model was focused on, with some reference to the generalised Born (GB) model.
The Born model was first proposed by M. Born in 1920 [9]. It treats the solute as a cavity in the medium of the solvate. The solvation energy can then be found by treating the system with standard electrostatics. This can be done with a lot of precision analytically, but an approximation can be achieved faster. One method for doing this approximation is the Generalised Born (GB) model [10].

The Generalised Kirkwood (GK) model further extends the GB model by adding support for arbitrary multipole moments [8], thus it can be used for AMOEBA whereas GB is limited to fixed charge force fields. As before, the particles of the solute are represented as spheres, but with more complicated charge distribution within them. The theory is based on Kirkwood’s original work in 1934 [11]. GK allows for more realistic and more accurate electrostatic calculations. The penalty is increased computation time.

2.4 TINKER

TINKER [2][12] is a molecular dynamics software package with a leading implementation of the AMOEBA force field. It provides a comprehensive toolset and is widely used. The code is written in Fortran 77 but is being ported to Fortran 90. Parallelisation is achieved using OpenMP and an MPI version is in development.

2.4.1 OpenMP parallelisation

OpenMP (Open Multi-Processing) is an API that allows a developer to add shared memory parallelisation to applications via compiler directives (comments that the code compiler uses to guide its work). The application launches with a master thread which creates multiple worker threads once a designated parallel region is reached in the execution path. The principle method for gaining performance benefits is by distributing the work of processing large arrays where elements can be updated independently of each other. To achieve the shortest run time, the workload should be spread evenly. OpenMP parallel loops provide different schedules of loop iteration assignment to threads to help achieve this.

In TINKER there are many sections of code that apply actions to every atom in the system. For more sophisticated functions, such as GK implicit solvation, the loop bodies are quite large and usually include - for each atom - a nested loop over all neighboring atoms. At the start of the main research period some of these were parallelised, mostly by using an OpenMP parallel loop construct over the outer loop. This means that multiple threads may try to read or write data of the same atoms because they are neighbors to multiple other atoms.
2.5 Summary

Molecular dynamics is an important technique used in many physical science disciplines. Implicit solvation allows researchers to learn about properties of molecules suspended in a liquid; it replaces the explicit modelling of many solvent particles with an average effect field. These scientific tools are present in TINKER, an MD software package, along with the AMOEBA force field and GK solver to implement polarisability in the system. Using AMOEBA leads to increased simulation quality and also computational time.
Chapter 3

Initial investigation and baseline

An initial profiling was done to find how and where the program spent its execution time. Optimisation and parallelisation is best done on the parts of a code where most of the execution time is concentrated. Making one such “computational kernel” go faster would yield a large overall improvement. Profiling reveals these computational kernels by tracking and timing a live run of the code. Plus, profiling for this program was a useful aid in understanding its structure in addition to static code analysis. The program contains branches throughout the code based on global flags which are set in an input file, on the command line or have default values. Following these by hand can be time consuming, and call trees illuminate what route the program is taking.

3.1 Machine and methodology

The profiling and optimisation work was done mainly on EPCC’s Hydra machine [13]. Specifically, the code was run on one Fermi node. A Fermi node has 4 CPUs with 6 computing cores in each, making a total of 24 cores. The chips are Intel Xeon X5650 @ 2.67GHz which have 64KB L1 cache, 256KB L2 cache and 12MB L3 cache. The code was compiled using the Intel Fortran compiler version 14.0.2 and used the fast Fourier transform library FFTW [14] version 3.3.3.

Over the duration of the project Hydra was in low demand, so the whole Fermi node could be continually used. As a consequence timing results were rather stable. Still, operating system events can jump in and cause timings to fluctuate so each experiment was run 3 times and the minimum total execution time picked, as this represents the quietest run.

Some timings were obtained on ARCHER [15]. At the time of writing this report ARCHER was the National supercomputer of the UK. The compute nodes on this machine have two Intel 12-core E5-2697 @ 2.7GHz CPUs. It will noted if data has been obtained on ARCHER.
3.2 Input systems

The main testing input in this project was the 3BDC system. It is the residue Glu57 of Staphylococcal nuclease variant Delta+PHS [1]. Figure 3.1 shows the biological assembly image of the system and Figure 3.2 shows the individual particles. The 3BDC system was chosen because it is representative of a real problem and was of interest to the expert user collaborator. It contains 2099 particles and uses the AMOEBA forcefield to do implicit GK solvation which is more computationally costly than GB and thus more interesting to improve.

In order to investigate the program in more depth (e.g. weak scaling), a selection of problem sizes is needed. Ideally these systems would be qualitatively the same as the original, but larger. However, it is not always obvious how to fairly quantify the problem size. It depends on what aspect one wants to investigate and how precise one wants to be. For example, the amount of work that needs to be done might not scale linearly with the number of particles. Particles interact with each other, so doubling the number of particles might more than double the work. If this is not taken into account and one expects the run time to double, false conclusions might be drawn.
Figure 3.2: Particle view of 3BDC system. Image produced by VMD.
A module of TINKER called **xyzedit** was used to create new input files from the base system 3BDC. The number of particles can be easily doubled by making a copy of the first "cloud" of atoms, displacing it and superimposing on the original. The resulting system was inspected visually using VMD [16] and it looked much like the original. However, this solution was found to produce very different energy values (opposite sign and 10000-fold increase in magnitude) which seemed unrealistic. It was probably the case that displacing the dense cloud caused some particles to be extremely close to each other, causing numerical instability. Thus a lot of energy was fed into the system making it erratic, possibly adding more work that is difficult to account for.

It was further noted that changing the separation between the two particle clouds had an effect on execution time. Apparently the number of interactions that are calculated depends on the distance between the 3BDC particle clouds. The separation between clouds was increased until the execution time stopped changing (see Fig. 3.3). At this point the energy of the entire system was double of the original system with one cloud which is intuitive if the two clouds are not interacting any more. Changing the separation vector above 100 units in magnitude produced no difference in execution time or output value beyond noise so a separation of 100 units is enough. This arrangement was deemed the best option as a qualitatively comparable, numerically plausible and more work-intensive input system. The largest system (referred to as 5x3BDC) contained 5 3BDC clouds with large separations between them, totalling 10495 particles. It can be seen in Figure 3.4.

![Figure 3.3: Change in execution time when increasing separation between 2 3BDC "clouds".](image)
Figure 3.4: Particle view of 5x3BDC system. Image produced by VMD.
3.3 Profiling tools

A selection of profiling tools was used to probe different aspects of the program’s execution. The program was examined in serial, in parallel and on the hardware level. This section describes the tools used to do this.

3.3.1 gprof

gprof [17] was used to profile the program running in serial. Instrumentation needs to be added to the code during compilation with the -p flag. This modifies the executable to record statistical samples about how and when parts of the program are called. This information is then used by gprof to produce a report of an execution of the code. The report contains a call tree of the run-through and time spent in each routine. To make the report more useful, the code should also be compiled with the -g flag to add debugging information.

For the clearest results the code should be without parallelisms. This was easy to achieve because the code only had OpenMP parallelisation, so a simple recompilation without the -openmp flag did the job. Compiling with optimisations turned off gives the highest resolution because routines are not inlined (essentially copied and pasted into the main body of code), thus they appear as separate functions in the profile and not as part of the parent.

The output presented here will be the flat profile which presents the functions where most time was spent. The columns of the output table are:

- % time - the percentage of the total time that was spent in this function
- cumulative seconds - a running sum of the number of seconds spent in this function plus the ones listed above it
- self seconds - number of seconds spent in this function
- calls - the number of times this function was called in the code
- self s/call - number of seconds per call spent in this function
- total s/call - number of seconds per call spent in this function and the ones it calls
- name - name of the function

3.3.2 ompP

ompP [18] is an OpenMP profiling tool. It comes with a tool that wraps around the compiler and adds instrumentation specific to the parallel regions. Executing the code with threads then produces a parallel report. Similar to gprof, the ompP report contains call trees and time spent inside the parallel regions. There is additional information
about overheads incurred due to starting/stopping a parallel region, work imbalance etc. A special profiler for the parallel code is needed because gprof would give misleading results, for example by reporting events only from the main thread.

### 3.3.3 perf

perf [19][20] is a hardware usage profiling tool. It reads hardware counters available on a machine and reports the event counts as requested. Things like executed cycles, instructions per cycle, cache hits and misses can be examined. These statistics are often machine specific, but they can help diagnose badly written code that would have similar issues on other machines, for example, if arrays are being accessed incoherently.

According to the perf manual, perf aggregates counter values over child processes and threads that the original process creates. Therefore it is suitable for use on the parallel program, but the counting behaviour needs to be kept in mind when interpreting its output.

### 3.3.4 Doxygen

Doxygen [21] is a static analysis tool that was used to explore the structure of the code. It creates documentation for the code with browsable code and call graphs for every function. This was useful for gaining a general overview of routines and their calling patterns.

### 3.4 Main routines

To narrow the focus of the project, two important routines were chosen to focus on. These were `energy()` and `gradient()`. They appear in most modules of TINKER and are central to the basic molecular dynamics simulations. For example, `gradient()` is called every timestep when calculating molecular trajectories. Therefore improving their performance and parallel scaling should reduce total execution time in the broader context application. A brief introduction to these routines will be given here.

The `energy()` routine calculates the numerous potential energy terms and sums them up to give the total energy of the system. The `gradient()` routine calculates the total energy of the system as well as the first derivatives of it with respect to Cartesian coordinates. These are general functions, certain parameters must be set when launching a simulation to take a GK or GB implicit solvation execution path. Figure 3.5 shows the call graphs generated by Doxygen. Obviously there are too many branches even within these two routines for complete optimisation during a short project, so focusing on parts related to implicit solvation was sensible not only due to the interest of stakeholders (see section 1) but also practically.
Figure 3.5: Call graphs of energy() and gradient() routines.
### 3.5 Bugs

Before profiling and applying optimisations, it was important that the code worked as expected. Over the course of the project a number of setups were found where the program would not work as expected. Thus a significant portion of the project time was spent on correcting these issues. Many of these were rather mundane, however one took a particularly long time to fix and is interesting enough to include here.

While running `energy()` with different combinations of compiler optimisations and thread counts a serious bug was discovered. If `energy()` was compiled with `-O0` optimisation level and run on multiple threads the calculated energy value became irreproducible. I.e., running the program multiple times with the same system and settings would produce different output values. Compiling without OpenMP enabled or running with only 1 thread made the results reproducible. Oddly, if the program was compiled with `-O1` or higher optimisation, it could be run with multiple threads and produce consistent results. Commonly higher optimisations levels gain speed by ”cutting corners” or not adhering to standards (the behaviour definitions of a programming language) strictly, hence the program can become unstable and produce incorrect output. This, however, is a case where the optimisations appear to have negated a programmer’s error which manifests only when optimisations are turned off.

The energy difference was small, less than 0.5% of the correct value. Spread over 2099 atoms it probably would not have resulted in significant qualitative difference in a simulation. Still, making scientific conclusions based on irreproducible results is not acceptable, so it was important to remove this bug.

To investigate the bug, the search area had to be narrowed down. The output was trimmed down to only contain the output that indicated the problem. Fortunately, because the bug was connected with OpenMP, it was possible to find in which file the problem lay. An ompP report showed all the files where OpenMP calls were made in the execution of `energy()`. One by one the files were recompiled without the OpenMP flag but still linked with it. It was found that switching OpenMP on or off in one file (`empole.f`) made the results irreproducible or reproducible. The ompP report showed which parallel region in particular was being called. As a result of the investigation, the problem code was narrowed down to one subroutine.

Next, attention was turned to the compiler optimisations. Since some optimisation or a combination fixed the irreproducibility bug, knowing which compiler flag did it could point to the origin in the bug. Unfortunately, the optimisation levels `-O[n]` encompass many functions within them and it is not easy to find what exactly they do. The Intel compiler’s manual describes the optimisations in words, and it is not obvious which flags it turns on. Moreover, the manual states that the compiler may or may not do some additional optimisations beyond what is described. The GNU compiler was tried as well; it explicitly lists which flags are part of an optimisation level, but it also retains the right to do some other optimisations. There was hope that by turning on the GNU compiler’s flags on one by one the target flag could be identified. It was found that `-O0` with all the flags of `-O1` still made irreproducible results while just `-O1` gave reproducible results.
The subroutine in question was about 300 lines long and it calculates the atomic multipole polarisability interaction energy. The parallel region within it contained many shared and private variables; inside it is a single, large distributed do loop. The first test was to change the loops schedule from guided to static. The bug remained, so the problem was something that did not depend on loop iteration assignment but rather something that manifested all the time. This made it seem like there was a race condition somewhere in the parallel region and hence shared variables had to be examined. In order to do this, the default clause was changed from shared to none. Doing so revealed all variables used within the parallel region. They were sifted through and made shared if they were only read and private if they were written and used in the same loop. A large number of incorrectly assigned variables were found; correcting these mistakes made the results of the program reproducible. Somehow the compiler had removed these issues, probably by doing something of its own discretion. Further investigation here would have been too time consuming especially since the compiler documentation is not detailed enough as was highlighted earlier.

### 3.6 Serial profiling

#### 3.6.1 Energy

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>self</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.33</td>
<td>15.19</td>
<td>15.19</td>
<td>100</td>
<td>0.15</td>
<td>0.15</td>
<td>ufield0d_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.27</td>
<td>24.95</td>
<td>9.76</td>
<td>20</td>
<td>0.49</td>
<td>0.49</td>
<td>ewca_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.97</td>
<td>32.73</td>
<td>7.78</td>
<td>20</td>
<td>0.39</td>
<td>0.55</td>
<td>surface_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.69</td>
<td>38.54</td>
<td>5.81</td>
<td></td>
<td></td>
<td></td>
<td>acos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.00</td>
<td>42.74</td>
<td>4.20</td>
<td>20</td>
<td>0.21</td>
<td>0.21</td>
<td>born_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.54</td>
<td>46.06</td>
<td>3.32</td>
<td>1331712</td>
<td>0.00</td>
<td>0.00</td>
<td>sort2_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.44</td>
<td>49.32</td>
<td>3.26</td>
<td>20</td>
<td>0.16</td>
<td>0.16</td>
<td>dfield0d_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.22</td>
<td>52.45</td>
<td>3.13</td>
<td>20</td>
<td>0.16</td>
<td>0.16</td>
<td>egk0a_</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: energy() profile.

Figure 3.6 shows the most time consuming functions of the energy() routine as reported by gprof. The energy() routine was called a total of 20 times in the run to reduce the portion of time spent in the setup steps. The top functions are a good candidates for parallelisation as this should produce most benefit. There are two exceptions though; acos is a Fortran intrinsic so there is no access to it to parallelise. sort2 is called many times and has a very short execution time, so naive parallelisation would slow it down through overheads of starting and killing the parallel region.

Of these dfield0d, ufield0d and egk0a (and others less computationally heavy parts of the code) are already parallelised.
### 3.6.2 Gradient

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>seconds</th>
<th>self calls</th>
<th>s/call</th>
<th>self total</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.95</td>
<td>22.64</td>
<td>22.64</td>
<td>20</td>
<td>1.13</td>
<td>1.13</td>
<td>egk1a_</td>
<td></td>
</tr>
<tr>
<td>16.32</td>
<td>44.45</td>
<td>21.81</td>
<td>20</td>
<td>1.09</td>
<td>1.09</td>
<td>ewca1_</td>
<td></td>
</tr>
<tr>
<td>11.80</td>
<td>60.22</td>
<td>15.77</td>
<td>20</td>
<td>0.79</td>
<td>0.79</td>
<td>ediff1b_</td>
<td></td>
</tr>
<tr>
<td>11.50</td>
<td>75.59</td>
<td>15.37</td>
<td>100</td>
<td>0.15</td>
<td>0.15</td>
<td>ufield0d_</td>
<td></td>
</tr>
<tr>
<td>7.66</td>
<td>85.83</td>
<td>10.24</td>
<td>20</td>
<td>0.51</td>
<td>1.67</td>
<td>empole1b_</td>
<td></td>
</tr>
<tr>
<td>6.31</td>
<td>94.26</td>
<td>8.43</td>
<td>20</td>
<td>0.42</td>
<td>0.42</td>
<td>born1_</td>
<td></td>
</tr>
<tr>
<td>6.04</td>
<td>102.33</td>
<td>8.07</td>
<td>20</td>
<td>0.40</td>
<td>0.56</td>
<td>surface1_</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.7: gradient() profile

Figure 3.7 shows the heaviest functions of the gradient() routine, also reported by gprof. The situation is similar to that of the energy() routine. Overall the different functions have longer self times per call due to calculating the added calculation of the energy gradient. Subroutines with similar names but with an added 1 signify modified versions of the subroutines called by energy().

Of these empole1b, egk1a, ufield0d and ediff1b (and others less computationally heavy parts of the code) are already parallelised.

### 3.7 Parallel profiling

#### 3.7.1 Parallel coverage and Amdahl’s law

An important metric in High Performance Computing is parallel coverage. It is the proportion of the program that has been enabled to execute in parallel; the remaining portion executes in serial. Increasing the number of workers reduces the time it takes to complete the parallel parts but has no effect on the serial parts (aside from some memory rearrangements). It follows that the execution time of a parallel code is limited by the serial parts. This is expressed in Amdahl’s law as

$$T(n) = T(1) \left( \alpha + \frac{1}{n}(1 - \alpha) \right)$$

(3.1)

where $T(n)$ is execution time for $n$ threads and $\alpha$ is the fraction of the code which is serial. The speedup for $n$ threads $S(n)$ can then be expressed as

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{\alpha + \frac{1}{n}(1 - \alpha)}$$

(3.2)

Thus, knowing the proportion of the code that has been parallelised, one can estimate the theoretical maximum speedup as
Parallel coverage usually is not straightforward to estimate. For example, in MPI or PGAS language parallelised codes the workflow can be very complicated, and a small part of the code could be doing most of the work. Fortunately it is possible and simple to gain a good approximate when using OpenMP. Parallel regions are clearly labelled, and using ompP they can be instrumented too. The parallel coverage then in this project was estimated based on the profiling of ompP. ompP reports total run time and time spent inside parallel regions, hence running the program on one thread gives a good estimate of parallel coverage. The reported value might be slightly overestimated due to critical, single or master regions. These are counted as part of the parallel region even though they execute in serial.

The time spent in serial and in parallel parts before applying optimisations can be seen in Figure 3.8. The timings with 1 thread show that the parallel coverage of \texttt{energy()} and \texttt{gradient()} is 33% and 37% respectively. This corresponds to theoretical maximum speedups of 1.50 and 1.58. As expected, the serial part of the code takes roughly the same amount of time across thread counts. The parallel regions decrease in execution time reasonably well up to 8 threads, thereafter they are mostly static or even increasing. Part of the reason is increasing parallel overheads as indicated by ompP.

Figure 3.8: Time spent in parallel and serial regions of \texttt{energy()} and \texttt{gradient()}. This corresponds to parallel coverage of 33\% for \texttt{energy()} and 37\% for \texttt{gradient()}. Reported times are the sum over 20 runs using 3BDC system.
3.7.2 Parallel scaling

Figure 3.9 shows the real and theoretical maximum speedups of the routines. On the whole, it appears that the routines are not parallelised efficiently as can be seen immediately by the divergence from the linear speedup line. `energy()` speedup peaks at 1.40 and `gradient()` at 1.46, at 16 and 8 threads respectively. Beyond that the performance drops again. These results can be attributed to the parallel coverage rather than the quality of the parallelisation. Indeed, since `energy()` is 33% parallelised and `gradient()` is 37% parallelised, the theoretical maximum speedups are 1.50 and 1.58. However, the difference between the real and theoretical speedup signals some inefficiencies as well, for example parallel overheads.

![Real and theoretical max speedups of energy() and gradient() routines before optimisation. Times with 1 thread: energy() - 27.7s, gradient() - 49.9s. Reported times are the sum over 20 runs using 3BDC system.](image)

To examine how overheads affect the scaling in more detail, the ”ideal times” were obtained. The ideal case data was obtained by subtracting the time lost to overheads (as reported by ompP) from the total run time of each parallel region, summed over all parallel regions and divided by the number of threads. In reality work balance could be achieved by using appropriate OpenMP schedules for the different loops, creating custom scheduling or perhaps using tasks instead of a parallel loop. Figure 3.10 shows how the speedup of parallel regions would change in the case of ideal balancing. It can be seen that the parallel parts themselves experience better speedup than the whole (compare to Fig. 3.9). Up to 8 threads the speedup is almost linear with lesser improvements until 16 threads; afterwards it quickly drops. It can also be seen that the scaling would improve slightly for both `energy()` and `gradient()` with overheads removed but not by a great amount. Thus, increasing overheads account for some of the poor scaling of the parallel parts, but not all of it.
Figure 3.10: Real and ideal speedups of \texttt{energy()} and \texttt{gradient()} routines’ parallel parts before optimisation. Ideal refers to parallel time with overhead times removed. Times with 1 thread: \texttt{energy()} - 9.3s, \texttt{gradient()} - 18.3s. Reported times are the sum over 20 runs using 3BDC system.

A frequent cause of parallel overheads is not enough work to spread between then workers. Using a larger problem can improve this. Running the routines with an input file that has 5 times more atoms in it (5x3BDC) produces real scaling closer to the theoretical maximum (Figure 3.11) than when using 3BDC.
Figure 3.11: Real and theoretical max speedups of energy() and gradient() routines before optimisation. Times with 1 thread: energy() - 353.7s, gradient() - 713.8s. Reported times are the sum over 20 runs using 5x3BDC system.

Figure 3.12 gives a sample comparison of overheads in the parallel regions for the two inputs of different sizes. A drastic fall in overhead proportion can be seen when running the code on 5x3BDC with the work spread between 24 threads. Some overheads due to an imbalanced workload still remain though, so possibly some improvements can be found by altering loop schedules. A similar picture appears when comparing the gradient() routine. On the lower end of the thread count there are very little overheads (below 1%) in the top parallel regions for both problem sizes.

An interesting thing to note is that there is a change in parallel coverage. energy() has gone from 33% to 28% and gradient() from 37% to 19%. Evidently the existing parallelism does not cover the areas that increase in work most with a growing problem size. So, even though the overheads are amortised by adding more atoms, overall scaling has not improved with the larger input system.
Overheads wrt. each individual parallel region (3BDC):

<table>
<thead>
<tr>
<th></th>
<th>Total Ovhds (%)</th>
<th>Imbal (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00077</td>
<td>18.96</td>
<td>5.26 (27.75)</td>
<td>5.14 (27.10)</td>
</tr>
<tr>
<td>R00043</td>
<td>5.54</td>
<td>2.26 (40.80)</td>
<td>1.33 (24.08)</td>
</tr>
<tr>
<td>R00041</td>
<td>19.43</td>
<td>12.75 (65.62)</td>
<td>9.01 (46.38)</td>
</tr>
</tbody>
</table>

Overheads wrt. each individual parallel region (5x3BDC):

<table>
<thead>
<tr>
<th></th>
<th>Total Ovhds (%)</th>
<th>Imbal (%)</th>
<th>Mgmt (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00077</td>
<td>135.32</td>
<td>22.39 (16.55)</td>
<td>21.15 (15.63)</td>
</tr>
<tr>
<td>R00043</td>
<td>39.17</td>
<td>1.52 (3.87)</td>
<td>0.86 (2.19)</td>
</tr>
<tr>
<td>R00041</td>
<td>29.89</td>
<td>1.16 (3.88)</td>
<td>0.56 (1.88)</td>
</tr>
</tbody>
</table>

Figure 3.12: Overhead report for the top 3 parallel loops for a large and a small input running `energy()` on 24 threads. Synchronisation and limited parallelism overheads are not shown because they were all 0. R000## are identifiers of particular routines.

Additional insight is provided by perf. Figure 3.13 shows the number of instructions per cycle (IPC) and number of stalled cycles per instruction (SCPI) for a run of `energy()`. Up to 8 threads IPC is above 1 which indicates good utilisation of the CPU. Also, SCPI, while non-zero, stays roughly constant at 0.47. Afterwards there is a relatively large change in both metrics. IPC drops to 0.89 at 12 threads and 0.83 at 24 threads; SCPI climbs to 0.65 at 12 threads and 0.72 at 24. It appears that increasing the number of threads makes the cores work at lower efficiency as there is more contention for resources on the CPUs. One possibility is memory contention where multiple threads try to access a shared variable at the same time and end up queueing. Consequently parallel speedup is decreased because, although each core has less to do, it is doing it at a slower rate. More evidence is needed to establish whether this is truly happening. `gradient()` shows the same trend.
3.8 Summary

Most of the optimisation work was done and tested on EPCC Hydra machine’s shared memory “Fermi” node with 24 cores. The main input system was 3BDC with 2099 atoms used in GK implicit solvation experiments; larger inputs were generated by superimposing clones of 3BDC with a large separation between them. gprof, ompP and perf were the main tools used for probing the code’s performance. The core routines of GK solvation are \texttt{energy()} and \texttt{gradient()}, they calculate a solvate’s energy and the gradient of that energy. The focus of the project was placed on reducing the runtime of these routines. Before profiling and optimisation work could begun, a number of bugs, including a subtle parallelisation bug, were corrected. Serial profiling flagged up a few work intensive subroutines inside \texttt{energy()} and \texttt{gradient()}. Parallel profiling showed that the the existing parallelisation suffered from poor scaling that peaked at less than 1.5 speedup. The main causes of this appeared to be limited parallel coverage, parallel overheads and inefficient use of the CPU.
Chapter 4

Optimisation

In this chapter the general approach taken to optimisation and specific code changes applied will be discussed.

4.1 Optimisation roadmap

Early on in the project, an "optimisation roadmap" for the implicit solvation part of TINKER was obtained. This outlined the steps that the code’s original author considered worth taking in order to reduce execution time. With small additions, the list was as follows:

1. Investigate serial and compiler optimisations.
2. Investigate current OpenMP parallelisation for improvements.
3. Add OpenMP to energy routines that are computation heavy.
4. Add OpenMP calls to the routines that compute the Born radii and their derivatives.
5. Convert remaining $N^2$ double loop energy and gradient code to use neighbor lists.
6. Explore the use of distance-based cutoffs for the CPU intensive parts of the implicit solvent code.

It was not expected to complete everything on this list, but some attention was given to each item with the exception of the research-oriented last point which required domain-specific expertise.

4.2 Verification

An essential step in the optimisation process is verification - testing that the program still behaves correctly. Optimisation often is about removing wasteful operations and
sometimes about "cutting corners". Code changes of this nature can lead to incorrect results. Thus, continuously changing a complicated codebase without verification leads to an uneasy feeling for the programmer and no good objective evidence that the results of the program are trustworthy. How to check correctness varies from program to program; it is best if there is a suite of automatic tests to run or a single value to check. Testing for correctness could even be made the last step of compilation.

Ensuring correctness is complicated further when parallelism enters the picture. Code can be formally incorrect, but still work correctly 99% of the time. The result might be as expected one time, but different another time. Parallel verification is still an area of research and debate; the onus is on the programmer to parallelise carefully.

The part of the program focused on in this project had no verification in place. However, the program was used by researchers and results have been tested empirically, so it was assumed that the output was correct. Hence the verification test was chosen to be that results should stay as they were in a past stable version. The energy routine output a single number and the gradient routine output an array of gradient values for all particles. The energy output and the sum of the absolute values of the gradients were taken as the two verification values. These numbers represent an ensemble of atoms, but since there were many thousands of particles in the test systems, it is unlikely that another arrangement would yield exactly the same values.

The chosen values were tested across a number of reference version commits, with and without OpenMP parallelisation. It was found that the values stayed constant in all cases, this gave confidence in their correctness at the outset.

### 4.3 Serial optimisations

Relatively little time was given to serial optimisation given that TINKER has been in development for many years. Also, the small amount of parallel coverage meant that more performance gains might be achieved by adding more OpenMP.

There were a number of compilation flags that were set by default when compiling for an Intel architecture. These are:

- **-xHost** allows generation of the most advanced instructions allowed by the host processor. For example, this would enable fused multiply-add functionality if the hardware supports this. The resulting executable possibly runs faster on the machine it was compiled on, but might not run at all on different architectures.

- **-no-ipo** disables inter-procedural optimisations. TINKER is a large codebase, so performing whole-program optimisations would increase compilation time significantly.

- **-no-prec-div** reduces the precision of divisions in favour of quicker execution. This done by allowing A/B to be expressed as A * 1/B which is easier for the processor to do.
- **-recursive** makes the compiler compile routines for possible recursive execution.
- **-openmp** enables code parallelisation with user inserted compiler directives.
- **-static-libgcc** links the GNU library statically.
- **-static-intel** links the Intel library statically.
- **-O3** which is a large collection of serial optimisations.

These flags were left as they were, except for **-O3** which was changed to **-O2**. As seen in Figure 4.1 execution time is the same with **-O2** and **-O3** (marginally better than **-O1**). Using **-O2** the compilation time is decreased and less potentially unsafe optimisations are used. Switching on inter-procedural optimisations did not change the runtime so they were left off in the interest of quicker compilation. The **-recursive** flag indicates that part of the code are written recursively; such code is usually difficult for the compiler to optimise, so some improvements might be gained if they were rewritten for non-recursive execution.

To investigate further, the compiler flag **-opt-level=2** was enabled. This made the Intel compiler output a report of the optimisations it was performing. One notable feature is that the frequently called **sort2()** function (see Fig. 3.6) is not being inlined, which was to be expected because sorting is a relatively involved procedure. Function calls incur overheads so it is desirable to avoid calling busy functions by inlining them, however it is not certain that it would be beneficial in this case.

Another notification that came up in several places was: "Poor memory locality: Non-unit-stride memory reference detected. Advice: Data transposing might help loopnest."
One example that triggered the notification is the following code fragment:

```plaintext
do i = 1, npole
   ii = ipole(i)
   pbpole(1,ii) = rpole(1,i)
   do j = 2, 4
       pbpole(j,ii) = rpole(j,i)
   end do
   do j = 5, 13
       pbpole(j,ii) = 3.0d0 * rpole(j,i)
   end do
end do
```

It can be seen that the accesses to `pbpole` are governed by the `ii` index obtained from `ipole` at index `i`. An access pattern like this can indeed lead to the program looking up physically distant memory locations irregularly which could indeed have negative impact on performance. Changing this would be a large undertaking and beyond the scope of this project.

### 4.4 Increasing parallel coverage

It was identified in section 3.7.2 that the parallel scaling of `energy()` and `gradient()` could be improved by increasing their parallel coverage. The routines contributing most to execution time as found via the serial profiling were chosen to be parallelised with OpenMP. The approach taken here was to take the main work loop of each routine and add OpenMP parallelisation directives. To ensure correctness, all the variables appearing in the loop had to be scoped, i.e. they had to be examined and declared `shared`, `private` or sometimes `firstprivate`. For example, variables that are only read in the loop can be `shared`, variables that are both read and written and depend only on the distributed loop’s iteration index can be `private` and variables that enter the parallel region with a value already assigned can be `firstprivate`. In some cases variables have ambiguous meaning or are read and written by multiple threads, special synchronisation was added in these scenarios. If the routines are changed in the future, it is important to reevaluate the parallelisation.

#### 4.4.1 ewca()

The `ewca()` routine finds the Weeks-Chandler-Andersen dispersion energy of a solute. It is a relatively short routine with the main body of work being done in a 100 line long double loop. Every atom is checked against every other atom for an energy contribution that is then added into a total displacement energy variable. Parallelising this routine was a relatively straightforward task of adding a parallel do loop and sifting through the variables. The pseudocode of the parallelised routine can be seen in Figure 4.2. Parallelising `ewca()` increased the parallel coverage of `energy()` by 0.18.
$OMP PARALLEL DO reduction(+:edisp)
DO i in 1 to number_of_atoms
  calculate variables dependent on i
  sum = 0
  DO k in 1 to number_of_atoms
    "remove contribution due to solvent displaced by solute atoms"
    ...mathematical manipulations...
    increment sum
  END DO
edisp += function of sum
END DO
$OMP END PARALLEL DO

Figure 4.2: Pseudocode of ewca() routine.

4.4.2 ewca1()

The ewca1() routine find the same Weeks-Chandler-Anderson dispersion energy of a solute as the ewca() routine, and also its spatial derivatives. The additional computations make the main double loop about 200 lines long. Now, however, the nested looping over all atoms causes a problem because of the shared array which stores the derivatives. The problematic part of the code is:

des(1,i) = des(1,i) + dedx
des(2,i) = des(2,i) + dedy
des(3,i) = des(3,i) + dedz
des(1,k) = des(1,k) - dedx
des(2,k) = des(2,k) - dedy
des(3,k) = des(3,k) - dedz

where des is the shared array, i is the outside loop index and k is the inner loop index. If the outer loop was parallelised, editing the ith array entries would be safe but there is nothing stopping one of the kth edits to be the same as a ith edit on a different thread. Indeed, running the program with the outer loop of ewca1() parallelised shows that this scenario can happen, giving inconsistent results between runs. A simple way to solve this is to wrap the block in omp critical directives or to use omp atomic statements for the increments. This works and produces the correct results, but the performance was found to be worse than for the serial version.

The chosen solution was to parallelise only the inner loop, pseudocode can be seen in Figure 4.3. Because the inner loop would be executed number of atoms times, the outer loop was wrapped in a parallel region to reduce thread spawning and killing overheads. The kth edits could happen unimpeded because the loop was distributed over k and reduction variables were used for the ith edits because within one execution of the parallel do i is constant. After each completion of the inner loop, the reduction variables have to be added into the shared array once so an omp single region was used. Similarly, the reduction variable has to be reset before each execution of the inner loop. Parallelising ewca1() increased the parallel coverage of gradient() by 0.27.
Tests of the parallelisation showed that it performed about as well as the (incorrect) version with no reduction variables nor critical regions. This approach, however, has introduced more synchronisation which could hinder performance on larger systems. Fortunately, most of the work is contained in the inner loop so the proportion of wasted time (waiting for other threads to reach a synchronisation point) to useful working time should be small.

```plaintext
!$OMP PARALLEL
DO i in 1 to number_of_atoms
    calculate variables dependent on i
    sum = 0
    !$OMP SINGLE
    zero out reduction variables: sum, dedxRed, dedyRed, dedzRed
    !$OMP END SINGLE
    !$OMP DO reduction(+:sum, dedxRed, dedyRed, dedzRed)
    DO k in 1 to number_of_atoms
        "remove contribution due to solvent displaced by solute atoms"
        ...mathematical manipulations...
        increment sum
        "increment the individual dispersion gradient components"
        dedxred = dedxred + dedx
        dedyred = dedyred + dedy
        dedzred = dedzred + dedz
        des(1,k) = des(1,k) - dedx
        des(2,k) = des(2,k) - dedy
        des(3,k) = des(3,k) - dedz
    END DO
    !$OMP END DO
    !$OMP SINGLE
    des(1,i) = des(1,i) + dedxred
    des(2,i) = des(2,i) + dedyred
    des(3,i) = des(3,i) + dedzred
    edisp += function of sum
    !$OMP END SINGLE
END DO
!$OMP END PARALLEL
```

Figure 4.3: Pseudocode of `ewca1` routine.
4.4.3 surface()

The **surface()** routine calculates the area which is accessible by the solvent of each atom. The structure of this routine is similar to that of **ewca()** with two loops nested over all atoms and a reduction variable which is the total accessible surface area (Figure 4.4). In principle the parallelisation then was straightforward, though this routine is considerably longer with the main loop body being about 400 lines long. Hence it was decided to use Cray Reveal’s [22] automatic scoping functionality. Automatic scoping statically analyses a loop and attempts to generate correct private, shared etc. assignments for the variables present in the loop. It was found that even the incomplete OpenMP directive generated (because Reveal could not ascertain the use of some variables, like global ones) were better than starting the scoping from scratch. Once the missing variables were assigned their correct scope, the parallelisation of **surface()** was complete. This increased the parallel coverage of **energy()** by a further 0.37.

```c
!$OMP PARALLEL DO (+:total)
DO ir in 1 to number_of_atoms
    calculate variables dependent on ir
    DO i in 1 to number_of_atoms
        check how much the irth and ith atomic spheres overlap
        calculate intersection geometry and resulting area
        increment total by the amount of accessible area on the irth atom
    END DO
END DO
!$OMP END PARALLEL DO
```

Figure 4.4: Pseudocode of **surface()** routine.

4.4.4 surface1()

Just like **surface()**, **surface1()** calculates the area which is accessible by the solvent that the atoms are implicitly immersed in and also the Cartesian derivatives of this area. The additional derivative calculations raise the size of the main loop to 500 lines and, more importantly, add simultaneous editing of the shared derivatives array by many threads. The following section appears in three places in the code:

```c
darea(1,ir) = darea(1,ir) + dax*wght
darea(2,ir) = darea(2,ir) + day*wght
darea(3,ir) = darea(3,ir) + daz*wght
darea(1,in) = darea(1,in) - dax*wght
darea(2,in) = darea(2,in) - day*wght
darea(3,in) = darea(3,in) - daz*wght
```

Here **darea** is the shared array, **ir** and **in** are indices of overlapping atomic spheres and the remaining factors are individual gradient increment amounts. The **ir** and **in** indices may overlap between different threads in a way that would cause writing to the
same darea entry simultaneously, creating erroneous results. One solution is to add `omp critical` directives to this region to avoid this. Another solution is to use effectively reduction arrays. Each thread can have its own copy of `darea` to work with; in the end the individual copies are added into the global gradients array. The second solution was found to be slightly better (Fig. 4.6) and it can be seen in pseudocode in Figure 4.5. Parallelising `surface1()` increased the parallel coverage of `gradient()` by 0.21.

One concerning factor of the reduction array approach is memory cost. For example, the test system holds 2099 atoms in Cartesian coordinates, so the memory requirement of the array is 49KB assuming 8 byte reals. Having more threads does not spread the cost because each thread needs a copy of this array in addition to other variables. Depending on how many threads share a processor and how many particles are in the system, the memory space required could become larger than the cache size which would cause the performance to drop. With the current setup using the reduction arrays gives better performance than using critical sections, but this might warrant further investigation.

```c
!$OMP PARALLEL reduction(+:total) firstprivate(dareaRed)
!$OMP DO
  DO ir in 1 to number_of_atoms
    calculate variables dependent on ir
    DO i in 1 to number_of_atoms
      check how much the irth and ith atomic spheres overlap
      calculate intersection geometry and resulting area

      in = function of sphere overlapping the irth sphere
      "increment the surface area gradient components"
      dareaRed(1,ir) = dareaRed(1,ir) + dax*wght
      dareaRed(2,ir) = dareaRed(2,ir) + day*wght
      dareaRed(3,ir) = dareaRed(3,ir) + daz*wght
      dareaRed(1,in) = dareaRed(1,in) - dax*wght
      dareaRed(2,in) = dareaRed(2,in) - day*wght
      dareaRed(3,in) = dareaRed(3,in) - daz*wght
    END DO
    increment total by the amount of accessible area on the irth atom
  END DO
!$OMP END DO

!$OMP CRITICAL
  !add each thread’s calculated area gradient contributions into the total
  darea = darea + dareaRed
!$OMP END CRITICAL

!$OMP END PARALLEL
```

Figure 4.5: Pseudocode of `surface1()` routine.
Figure 4.6: Comparison of speedup of \texttt{gradient()} with the \texttt{surface1()} function parallelised with critical regions or a reduction array.

4.5 Algorithmic changes

The most powerful but often also the most difficult performance optimisations involve algorithmic changes. In some cases it is enough to rearrange, group etc. steps of the algorithmic, in others one needs to go back to the mathematical basics. Verification is very important here, especially with edge cases.

Algorithmic changes, if done without reference to parallel computing, could be overall detrimental. For example, if numerous simple steps are made into more efficient but larger and interconnected steps, the amount of available parallelism would decrease and synchronisation would increase. In such a case the code could run quicker in serial, but it would prohibit even faster (but more wasteful) parallel execution. It is also easier to introduce race conditions and other parallel programming errors in more complicated code. As usual, a balance needs to be found.

4.5.1 Double loop and neighbour list

One opportunity to do algorithmic changes in the implicit solvation TINKER codepath was to extend the use of neighbor lists. Formally long term electronic interactions mean that every particle has an effect on every other. This leads to $O(N^2)$ complexity which severely limits the number of particles that can be simulated in a reasonable time frame, even with parallelisation. However, some electronic interactions inevitably are between
far away particles. These are weak (sometimes even below the precision representable by computers) and an approximation can be made to ignore them. A way to find the particles that should be taken into account is by using neighbour lists.

There are different ways of implementing neighbour lists, but the most obvious one is based on proximity in space. Each atom can be imagined to have a sphere around it with a radius of some cutoff value for the particular force. A list is constructed for how many atoms are in this sphere and what their indices are. Force contributions can then be calculated over this subset of atoms, rather than every atom. A drawback is that memory accesses are needed to find out which particles are to be interacted with, which is significantly slower than just iterating over an index. As the system evolves, neighbor lists might need to be recalculated with a frequency that balances the cost of recalculation and accuracy of the simulation.

The use of neighbor lists was added into the \texttt{egk()} routine which calculates the electrostatic energy within the GK implicit solvation model. This was done by modifying a copy of the existing \texttt{egk()} routine which iterated over all pairs of atoms, checked whether they are within the cutoff distance and then calculated their interaction energy accordingly. In the new version there are still two loops nested, but the inner looks up the neighbours of that atom and iterates over those instead of all remaining atoms. Additionally, self energy energy is calculated for each atom because the neighbour list does not include the atom itself. It is effectively the same calculation that takes place in the inner loop, but mathematical simplifications can be made because, for example, the distance of the atom to itself is zero. Figure 4.7 illustrates the loop using neighbor lists. Parallelisation was added on the outer loop with practically the same OpenMP directive as for the non neighbor list version.

\begin{verbatim}
DO ii in 1 to number_of_multipole_sites
  i = ipole(ii)

    compute self energy
    DO kkk in 1 to nelst(ii)
      kk = elst(kkk,ii)
      k = ipole(kk)
      compute interaction energy between ith and kth atoms
    END DO
  END DO
\end{verbatim}

Figure 4.7: Pseudocode of main \texttt{egk0b()} routine loop. \texttt{ipole} contains the number of the atom at each multipole site, \texttt{nelst} stores the number of neighbors and \texttt{elst} stores a list of the neighbors.
4.6 Summary

Verification values were established to ensure continued scientific correctness while optimising the code. The serial performance was looked at and a conclusion was made to use the -O2 optimisation flag instead of -O3 and to leave the other flags as they were. Parallel coverage was increased in \texttt{energy()} and \texttt{gradient()} by adding OpenMP to the \texttt{ewca} and \texttt{surface} subroutines in both. The \texttt{gradient()} subroutines required careful synchronisations due to the shared energy gradient arrays. Neighbor lists with parallelisation were added into the \texttt{egk()} subroutine within \texttt{energy()}. 
Chapter 5

Results and Analysis

In this chapter the effect of the applied optimisations will be measured and analysed.

5.1 Strong scaling

Strong scaling is evaluated by measuring how much speedup a program experiences when increasing the thread count, but keeping the problem size constant. Achieving good speedup in this mode is difficult because the amount of work per thread shrinks and any efficiencies in the parallelisation can quickly take over.

Applying the parallelisations described in section 4.4 increased the parallel coverage of energy() from 33% to 88% and of gradient() from 37% to 85%. According to Amdahl’s law, this should translate to a maximum speedup of 8.3 and 6.5 respectively. However, Figure 5.1 shows that the maximums speedup achieved for energy() was 4.9 and for gradient() it was 3.6. It is a 3-fold to 4-fold improvement over the original 1.3 and 1.2, but only a little more than half of the theoretical maximum. Initially the scaling problem was chiefly a question of parallel coverage, now the question of quality is equally important. For a different perspective, parallel efficiency (which is speedup divided by number of threads) is plotted in Figure 5.2.
Figure 5.1: Real and theoretical max speedups of `energy()` and `gradient()` routines with fixed problem size after optimisation. Times with 1 thread: `energy()` - 27.4s, `gradient()` - 50.0s. Reported times are the sum over 20 runs using 3BDC system.

Figure 5.2: Final parallel efficiency of `energy()` and `gradient()` for a fixed problem size.

Work imbalance overheads as reported by ompP were relatively minor up to 16 threads (see Fig. 5.3). If they were completely removed, `energy()` could achieve speedup of 5.2 and `gradient()` could achieve 3.7. There is, though, a notable jump in parallel overheads at 18+ threads, especially for `gradient()`. In Figure 5.3b the overhead in the parallel region suddenly jumps from 10% to 47% of total parallel time. Most of this increase
comes from the `ewca1` routine (see section 4.4.2), which contains many synchronisation points. While removing these overheads would not improve performance overall, the sudden jump is worthy of attention. On the execution platform each CPU has 6 cores and 16 threads is just below filling up 3 CPUs. Further investigation into how the threads are assigned to cores would be needed to draw conclusions about hardware effects in this situation.

![Figure 5.3: Useful work time and overhead time in parallel regions of `energy()` and `gradient()`](image)

Running the program without ompP profiling enabled gave speedups of 4.9 and 4.1; still a distance away from the theoretical maximum.

Strong scaling was also measured on ARCHER (Fig. 5.4). The measured and theoretical scaling behaviour is still relatively limited, but there are some interesting features. The chips on ARCHER are newer models than the ones on Fermi, so the time of execution on one thread has decreased. However the serial and parallel time has shifted in favour of the parallel part indicated by the increase in theoretical maximum speedup. In addition, the jump in overheads at 18 threads is no longer there but the real scaling still diverges from the theoretical. This gives support to thinking that the factor that prevents scaling to follow Amdahl’s law is code related rather than machine specific.
Figure 5.4: Real and theoretical max speedups of `energy()` and `gradient()` routines on ARCHER after optimisation. Times with 1 thread: `energy()` - 19.0s, `gradient()` - 32.6s. Reported times are the sum over 20 runs using 3BDC system.

5.2 Weak scaling

Weak scaling tests how a program performs in parallel when the amount of work is grown proportionally to the amount of parallel execution units available to the program. See section 3.2 for a discussion of problem sizes.

Figure 5.5 shows how `energy()` and `gradient()` perform when the problem size grows together with the number of execution threads. In the ideal case execution time would stay nearly fixed with increasing thread count. This would mean that one could solve larger problems in the same amount of time by increasing the available workforce proportionally. Here, however, it can be seen that the execution time grows quickly.
A feature to note is the widening gap between total and parallel only execution times. The execution time spent in parallel regions for energy() grows much slower than the time spent in total. This indicates that the parallel regions of energy() do handle weak scaling quite well, but the remaining serial regions grow rapidly with increased system size. A similar trend can be seen for gradient() but the parallel regions show poorer scaling with increased workload than energy().

Looking separately at the speedup with 5x3BDC as input in Figure 5.6 one can observe largely the same pattern as with 3BDC (see Fig. 5.1). Measured speedup is far from linear with energy() peaking at 3.6 with 22 threads and gradient peaking at 3.0 at 16 threads. As already observed during profiling, the parallel coverage with 5x3BDC is relatively smaller, 78% and 74% respectively.
Figure 5.6: Real and theoretical max speedups of \texttt{energy()} and \texttt{gradient()} routines after optimisation. Times with 1 thread: \texttt{energy()} - 346.8s, \texttt{gradient()} - 715.0s. Reported times are the sum over 20 runs using 5x3BDC system.

However, less parallel time is wasted on overheads. \texttt{energy()} has at most 8% overhead with 24 threads; \texttt{gradient()} again experiences a jump in overhead time at 18 threads, but being 25% it is about half of the jump seen with 3BDC as the input file. The net result is that both \texttt{energy()} and \texttt{gradient()} achieved a higher percentage of the theoretical maximum speedup with 5x3BDC as the input file (see Table 5.1). Thus having a larger system makes the parallelisation more efficient, but that does not translate to good weak scaling because the parallel coverage has decreased.

<table>
<thead>
<tr>
<th></th>
<th>energy()</th>
<th>gradient()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>93%</td>
<td>93%</td>
</tr>
<tr>
<td>Final</td>
<td>59%</td>
<td>55%</td>
</tr>
</tbody>
</table>

(a) 3BDC system

<table>
<thead>
<tr>
<th></th>
<th>energy()</th>
<th>gradient()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>96%</td>
<td>98%</td>
</tr>
<tr>
<td>Final</td>
<td>78%</td>
<td>77%</td>
</tr>
</tbody>
</table>

(b) 5x3BDC system

Table 5.1: Percentage of theoretical maximum speedup achieved using 3BDC and 5x3BDC as input files with initial and final optimisations.

### 5.3 Neighbor list effects

Adding a parallelised neighbor list version of the \texttt{egk} routine did not make a significant improvement when using the 3BDC test file. Still, further insight can be gained by testing the 5x3BDC input system and examining program wide effects of neighbor lists.
The settings file for TINKER allows to switch neighbor lists on or off for the entire program. Some caution when interpreting the timings is needed, because at the current state there are differing amounts of parallelisations, i.e. not every double list routine is parallelised if its equivalent neighbor version list is. For GK solvation in the energy() routine the different subroutines account for a few percent of total runtime, while in gradient() it is closer to 10%. Comparisons where 1 thread is used are not affected.

Overall, using neighbor lists reduces runtime by about 20% depending on problem size. Table 5.2 shows how the shortest runtimes in the energy() routine compare. Interestingly, majority of the reduction is happening in serial sections of the code with small differences in parallel time. A similar pattern is observed for gradient().

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>no NL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min runtime</td>
<td>5.56</td>
<td>6.99</td>
</tr>
<tr>
<td>Serial time</td>
<td>3.45</td>
<td>4.99</td>
</tr>
<tr>
<td>Parallel time</td>
<td>2.11</td>
<td>2.00</td>
</tr>
</tbody>
</table>

(a) 3BDC system

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>no NL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min runtime</td>
<td>96.64</td>
<td>125.79</td>
</tr>
<tr>
<td>Serial time</td>
<td>76.42</td>
<td>104.14</td>
</tr>
<tr>
<td>Parallel time</td>
<td>20.22</td>
<td>21.65</td>
</tr>
</tbody>
</table>

(b) 5x3BDC system

Table 5.2: Timing breakdown of shortest run of energy() with and without neighbor lists. Times are reported in seconds for 20 executions.

There are benefits for parallel execution as well. For instance, the subroutine egk0b() which was added as part of optimisation performs better than its counterpart egk0a() without neighbor lists for every thread count. Figure 5.7 shows also that, when the system size increases 5 times, egk0b() takes about 5 times longer to execute, while egk0a() experiences a near 10-fold increase in parallel execution time. As expected, the lower complexity NL algorithm shows its good properties when the problem size is larger. However, other parallel subroutines must be benefiting less because overall the largest contribution to reduced runtime comes from serial parts of the code.
Figure 5.7: Wall times of \texttt{egk0a()} (no neighbor lists) and \texttt{egk0b()} (neighbor lists) subroutines’ parallel sections in \texttt{energy()} using 3BDC and 5x3BDC systems.

On the whole using neighbor lists is a worthwhile optimisation over double loops in TINKER’s implicit solvation code. It reduces serial execution time and parallel performance is not hindered, which seems reasonable given that the code of routines that exploit neighbor lists does not change much structurally and has fewer atom pairs to process. The creation and maintenance time of the neighbor lists is not prohibitive either. Since this algorithm is not geared at parallelism explicitly, this is a good result. It would be interesting to see whether it is possible that more parallelism can be exposed by identifying independent groups of atoms based on neighbor lists, since they contain more conceptual information than a blind search over every other atom.

### 5.4 Hardware effects

To continue the hardware investigation begun during the profiling phase, perf was used to see how efficiently CPU cycles are used. Figure 5.8 shows these results for the 3BDC system. It differs from the perf statistics of the initial code (see Fig. 3.13) in two aspects. First, the \texttt{energy()} and \texttt{gradient()} profiles now diverge from each other. Second, both profiles are showing worse use of the CPU than before; for \texttt{energy()} the largest drop is at 24 threads, while \texttt{gradient()} is less efficient for almost every thread count. This seems reasonable given that the \texttt{gradient()} subroutines require more synchronisation to run correctly than \texttt{energy()} ones. Also, they work with larger data structures which can degrade memory usage. Oddly though, the perf patterns seem to reflect neither speedup (Fig. 5.1) nor parallel efficiency (Fig. 5.2).
Figure 5.8: Instructions per cycle (IPC) and stalled cycles per instruction (SCPI) of `energy()` and `gradient()` using 3BDC system.

Figure 5.9 shows the CPU usage for the 5x3BDC system. The shape is similar to the ones already seen, but `energy()` performs better while `gradient()` does worse.

Figure 5.9: Instructions per cycle (IPC) and stalled cycles per instruction (SCPI) of `energy()` and `gradient()` using 5x3BDC system.

Promising information can be gained by probing deeper into individual functions. A good example is the `egk0b()` subroutine. Figure 5.10 shows the performance metrics obtained
with perf and ompP relevant to this routine. Up to 18 threads the scaled energy() SCPI follow the scaled egk0b() parallel execution time aggregated over the running threads. I.e. when there are 50% more stalled cycles, egk0b() takes 50% more time to execute and so on. At 20 threads the egk0b() time taken rises quickly; at the same point parallel overheads start growing. Note that the perf data is calculated from the total number of stalled cycles and instructions across the whole program and all threads so it is an average.

The total amount of work (which can be expressed in terms of number of instructions) should stay constant within the parallel loop; the time it takes to do that work should be proportional to (among other factors) the amount of delay each instruction incurs. Initially, as more threads are added, the work gets divided between them equally. Since SCPI is constant, each thread completes its work in less time proportionally to the smaller amount of work it has. Thus the aggregate time stays the same. When SCPI increases, so does the aggregate time because every bit of work takes a little longer to finish. Above 18 threads the total time diverges because more work has appeared in the form of overheads. Here SCPI increases by a lesser amount, probably because the overheads (for example, waiting at barriers) do not trigger the stalled cycle counter on the CPU, it could be doing do-nothing instructions that technically finish normally.

The trend appears in other parts of the program too, but not always as cleanly as for the egk0b() subroutine. See Figure 5.11 for two more example routines. Other hardware events like cache misses could be playing a bigger role in those. Figure 5.12 shows the investigated metrics applied to the entire energy() and gradient() routines. gradient()
shows reasonable agreement with the proposed analysis, but \texttt{energy()} does so to a lesser degree.

Figure 5.11: SCPI and aggregated parallel time spent in the \texttt{surface} subroutines. The curves labelled ”scaled” are normalised by their respective values at 1 thread. The overheads are the aggregated times across threads spent on parallel overheads in seconds.

Figure 5.12: SCPI and aggregated total time spent in the main routines. The curves labelled ”scaled” are normalised by their respective values at 1 thread. The overheads are the aggregated times across threads spent on parallel overheads in seconds.

Modern retail CPUs, despite physical effects limiting their frequency, provide in general excellent performance which is extracted using clever circuits and logic. On the flip side
this makes analysing their behaviour difficult, especially in a parallel environment. The analysis presented here suggests that at least parts of the program are limited by stalled cycles. The ones reported by perf are front-end cycles which mostly handle instruction fetching and decoding. Stalling of these could be caused by branching, which the implicit solvation code does do frequently and in ways which are difficult to predict by the CPU. However, the real test would be to change the source code and see whether these trends alter. This is a large scale change that was beyond the scope of this project.

5.5 Summary

Parallel coverage was increased from 33% to 88% and from 37% to 85% for energy() and gradient() respectively when tested with the 3BDC input. However, this did not translate to the theoretically possible 8.3 and 6.5 speedup but instead 4.9 and 3.6. Part of the reason appeared to be an increase in overheads when more than 16 threads were used. Weak scaling was found to be poor despite relatively good weak scaling in parallel regions alone. It was concluded that the remaining serial sections were growing strongly with the increase in problem size thus reducing the benefits of parallelisation. Neighbor lists were found to improve performance overall by reducing serial time while also parallelising as efficiently as double loops. CPU usage was noted to decrease in efficiency with increased thread numbers. A tentative correlation was found between the stalled cycles per instruction, aggregated execution time and onset of parallel overheads. The stalled CPU cycles were mostly front-end suggesting problems with instruction delivery.
Chapter 6

Conclusions

This project has provided immediate and future value. Removing errors from and improving the performance of the Generalised Kirkwood implicit solvation codebase will benefit the researchers using TINKER for these types of numerical experiments. The investigations carried out in this work into performance profiles and problems should enable further optimisation to take place. An OpenMP parallel implementation is useful to maintain because shared memory machines are more common than tightly coupled distributed memory machines and in the future it likely will be part of hybrid MPI/OpenMP parallelisation.

Parallelising the most computationally heavy parts of the implicit solvation code has enabled a 3 to 5 fold reduction in the execution time. The speedup is far from ideal, reaching less than 60% of the theoretical possible given the parallel coverage. There is certainly room for improvement both in terms of additional parallelisation and optimisation of the existing parallel regions.

TINKER, being a large molecular dynamics code, is non-trivial to optimise. Frequent branching, global variables and modular design rule out straightforward comprehensive parallelisation. The domain problem is intrinsically inter-linked as particles have to interact with each other so parallelisation benefits have to contend with synchronisation costs. Still, the work done here shows that reductions in the execution time can be achieved.

6.1 Future work

There is a considerable amount of possible optimisation work that has not been done due to project changes and time spent on bug fixes. Some of these tasks would be relatively easy to accomplish, while others are more open-ended.

Neighbour lists could be added to the Born routines and the egk1() equivalent in gradient() of egk0b() in energy(). These routines should then also be parallelised, including their double loop versions. The ones in the gradient() execution path are usually more difficult to parallelise efficiently due to editing shared arrays. There are some other routines that appear on different execution paths which could be given the
same treatment. For example, a different energy calculation routine \texttt{ehpmf()} which is used if the solvation type is GK-HPMF instead of GK.

Program wide parallelisation could be looked at as well. Sometimes partially parallelised applications perform significantly worse than entirely parallelised applications, even if the non-parallelised parts are not work intensive. This could happen in TINKER if there is some serial code that invalidates useful cache between parallelised routines.

Most of the parallelised routines suffered from some work imbalance and synchronisation overheads at high thread counts. This could be addressed with a detailed experimentation with OpenMP schedules. An alternative is to try using tasks instead of loop parallelisation.

There is some evidence that parallel scalability is limited by stalled CPU cycles. One possible culprit is the frequent branching in the code. The program could be adapted to parallel execution where these branches are either removed or all executed and the unneeded results thrown away. However, more investigation is needed before committing to a large scale rewriting of the code.

Two possible research-based tasks are cutoff optimisation and neighbor list exploitation. Some accuracy of results could be traded off for decreased run time. The conceptual information stored in neighbor lists could be used to identify independent work. Perhaps it could be basis for an MPI spatial distribution of the work.
Appendix A

Key file

This is the key file which was used in all runs of the the implicit solvation code. There is no line for the number of OpenMP threads because initial.f was altered to accept thread count via the environment variable OMP_NUM_THREADS.

parameters /home/h021/zarins/tinker/params/amoebapro13.prm

polar-eps 0.01
mpole-cutoff 9.0
vdw-cutoff 9.0
#(ele cutoff is infinite in nonperiodic systems)
#vdw-correction
vdw-taper

solvate GK
thermostat nose-hoover
integrator verlet

polarization mutual
neighbor-list

archive
printout 100
maxiter 2500
Bibliography


