Addition of Flexible Linkers to GPU-Accelerated Coarse-Grained Simulations of Protein-Protein Docking

Adrianna Pińska[1]
supervised by Michelle Kuttel[1], James Gain[1] and Robert Best[2]

3rd January 2019

[1]Department of Computer Science, University of Cape Town, Cape Town, South Africa
[2]Laboratory of Chemical Physics, National Institute of Diabetes and Digestive and Kidney Diseases, National Institutes of Health, Bethesda, United States

Plagiarism Declaration

I know the meaning of plagiarism and declare that all of the work in this thesis, save for that which is properly acknowledged, is my own.

Abstract

Multiprotein complexes are responsible for many vital cellular functions, and understanding their formation has many applications in medical research. Computer simulation has become a valuable tool in the study of biochemical processes, but simulation of large molecular structures such as proteins on a useful scale is computationally expensive. A compromise must be made between the level of detail at which a simulation can be performed, the size of the structures which can be modelled and the time scale of the simulation. Techniques which can be used to reduce the cost of such simulations include the use of coarse-grained models and parallelisation of the code. Parallelisation has recently been made more accessible by the advent of Graphics Processing Units (GPUs), a consumer technology which has become an affordable alternative to more specialised parallel hardware.

We extend an existing implementation of a Monte Carlo protein-protein docking simulation using the Kim and Hummer coarse-grained protein model [1] on a heterogeneous GPU-CPU architecture [2]. This implementation has achieved a significant speed-up over previous serial implementations as a result of the efficient parallelisation of its expensive non-bonded potential energy calculation on the GPU.

Our contribution is the addition of the optional capability for modelling flexible linkers between rigid domains of a single protein. We implement additional Monte Carlo mutations to allow for movement of residues within linkers, and for movement of domains connected by a linker with respect to each other. We also add potential terms for pseudo-bonds, pseudo-angles and pseudo-torsions between residues to the potential calculation, and include additional residue pairs in the non-bonded potential sum. Our flexible linker code has been tested, validated and benchmarked. We find that the implementation is correct, and that the addition of the linkers does not significantly impact the performance of the simulation.

This modification may be used to enable fast simulation of the interaction between component proteins in a multiprotein complex, in configurations which are constrained to preserve particular linkages between the proteins. We demonstrate this utility with a series of simulations of diubiquitin chains, comparing the structure of chains formed through all known linkages between two ubiquitin monomers. We find reasonable agreement between our simulated structures and experimental data on the characteristics of diubiquitin chains in solution.

Acknowledgements

I would like to thank my supervisors, Michelle Kuttel, Robert Best and James Gain, for their patience and assistance, and Simon Cross for his support during my many years as a full-time postgraduate student. I would like to thank my parents, Joanna Majksner-Pińska and Marek Piński, for supporting my academic career. I would also like to thank Ian Tunbridge for his work on the original implementation of the software used in this project.

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: `http://hpc.uct.ac.za`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Proteins are macromolecules which are responsible for a wide variety of functions within living organisms. Proteins consist of linear chains of amino acids which fold into three-dimensional structures. Multiple proteins can bind to each other to form multiprotein complexes. Understanding how these complexes form is crucial to many medical applications, such as drug design. Identifying protein binding sites is one of the most sought-after problems in bioinformatics [3], and has some similarities to the problem of predicting the folded tertiary structure of a protein.

Obtaining experimental data on the structure of multiprotein complexes can be challenging because of their transient nature. The simulation of multiprotein components in a virtual environment has thus become an important research tool which can be used to complement theory and experimental data and aid in its interpretation [1, 4]. A computer simulation samples the space of possible conformations of a molecular system, generating an ensemble of configurations. Analysis of the structure of samples which are in a bound state allows us to identify possible binding sites, and the proportions in which they appear in the ensemble can serve as an estimate of their binding affinity.

Two common simulation techniques are Monte Carlo (MC) algorithms, which sample conformation space by randomly mutating the system and accepting or rejecting these mutations on the basic of some criterion, and molecular dynamics (MD) algorithms, which sample the real physical dynamics of the system by applying the laws of physics to the simulated structures. Both of these types of algorithms often require the calculation of the potential energy of the system: MC simulations use this property as a scoring function for evaluating moves, and MD simulations use it to calculate the forces acting on the molecules.

Proteins are large molecules, and many simulation algorithms scale quadratically in computational cost relative to the number of bodies in the simulation. All-atom simulations of multiprotein formation are thus highly computationally expensive, and therefore limited by the availability of computing resources. Trade-offs must often be made between the size of the system which can be simulated, the time scale of the simulation, and the real-world duration of the simulation. Optimisation is thus a crucial factor in the design of simulation software, as even a modest decrease in the computational cost of a simulation can greatly increase the usefulness of the software to researchers.

The number of bodies in a simulation can be reduced significantly through the use of a coarse-grained model. Rather than modelling every individual atom, we can use an abstraction which aggregates multiple atoms together. An example of such a model was described by Kim and Hummer in 2007 [1]. In this model, each amino acid on the protein backbone is modelled as a single bead, and interactions between beads are described by a combination of short- and long-range non-bonded potential energy equations. Optionally, flexible chains of residues may be used to connect rigid domains – these segments contribute additional bonded potential energy components.

Another important optimisation technique is parallelisation: selective rewriting of portions of the code to take advantage of parallel hardware. This technique has become particularly useful given the rising popularity of graphics processing units (GPUs), parallel processors present in most modern graphics cards. Unlike CPUs, which are general-purpose processors capable of effectively executing a wide variety of functions, GPUs have a more specialised architecture which is optimised for the task of rendering graphics. They are particularly suited to specific computational problems which require the same instruction to be executed on a large number of data elements in parallel. The non-bonded potential calculation in an MC or MD simulation is a good example of such a problem, because it involves summing a large number of independently calculated pairwise interactions.

GPU programming was once made more challenging by the lack of general-purpose programming interfaces. The hardware was accessible only through low-level, graphics-specific drivers, and programmers who wished to exploit its computational power for general-purpose code, such as scientific simulations, had to work within this limited framework. However, the growing popularity of general-purpose GPU (GPGPU) applications led to the development of generic APIs such as CUDA and OpenCL.

Coarse-Grained Protein-Protein Docker (CGPPD) is a custom implementation of the Kim and Hummer coarse-grained model using Monte Carlo simulations with replica exchange [5]. It was created with the aim of improving the execution time of the simulation by performing the non-bonded potential energy calculation on the GPU. Tunbridge et al. found that the specific nature of this problem was particularly suited to the GPU architecture, and achieved impressive speedup results.

1.1 Aims

The aim of this research is to extend CGPPD to introduce optional flexible linkers: the original implementation could only model proteins as entirely rigid bodies.

The addition of the linkers makes it possible to simulate protein structures which could not be modelled adequately using the rigid implementation. To demonstrate the utility of this feature, we use our modified application to investigate the conformations of diubiquitin chains with different linkages, and compare our results to multiple sources of experimental data to gauge the accuracy with which our model can represent these structures.

Although performance is not the focus of our project, we do not want to compromise the

gains of the original implementation. We expect this change to have a minimal impact on overall execution time, and use benchmarks to confirm this.

1.2 Approach

The addition of linkers to the implementation requires a series of changes throughout the code. New Monte Carlo moves must be implemented to allow flexible proteins to deform: we add translations and crankshaft moves of individual residues within a flexible linker, and also implement a partial rotation, or flex move, which allows a protein to bend at a residue within a linker.

New potential energy components are also required to evaluate the changes in potential resulting from these deformations: we add components to represent the pseudo-bonds, pseudo-angles and pseudo-torsions between adjacent residues within a linker. These additional calculations will be performed on the CPU rather than the GPU.

The non-bonded calculation must be adjusted to include interactions between more pairs of residues – interactions internal to a molecule are no longer constant, because residues within the molecule can change position with respect to one another. Calculating these additional non-bonded components on the CPU would significantly slow down the simulation, so it is necessary to incorporate them into the calculation performed on the GPU. We must modify the GPU kernel to include or exclude pairs using a more complicated criterion than the original version.

To verify the correctess of our implementation we recalculate the non-bonded potential of the ten reference conformations used to validate the original implementation, and compare our results to the reference CHARMM implementation. We also validate our flexible linker model by performing simulations of homopolymer chains using a modified potential calculation, and comparing our results to the expected behaviour.

We perform benchmarking simulations to evaluate the effect of the addition of various proportions of flexible linkers on the execution time of the simulations.

1.3 Contribution

We add flexible linkers to an optimised coarse-grained protein-protein docking simulation, making it possible to perform fast simulations of proteins which it is valuable to model with some flexibility rather than as completely rigid bodies.

By simulating multiprotein complexes as rigid domains connected by flexible linkers, we can allow individual proteins to change orientation with respect to one another while preserving the bonds between them, effectively restricting the simulation to Monte Carlo mutations which do not break these bonds. In this way we can effectively sample the conformations of a complex with specific linkages between the component proteins.

It would not be possible to impose such a constraint in the rigid implementation while modelling these proteins as separate rigid bodies: we could only filter the samples afterwards to a subset in which the complex is still bound. If there are multiple possible bound states, those

with the lowest energy would be overrepresented in the samples, and it would be difficult to obtain a sufficient number of samples for less-favoured bound conformations.

Our benchmarks show that the addition of the linker functionality does not negatively impact the performance of the simulation, even when all of the residues are made flexible.

Our investigation of the conformations of diubiqitin chains with different linkages produces results consistent with experimental data on diubiquitin chains in solution, and demonstrates that the Kim and Hummer model may be useful in the study of longer polyubiquitin chains.

1.4 Thesis organisation

This thesis is organised as follows: Chapter 2 reviews the basic principles behind protein-protein docking simulations and the optimisation of scientific software using parallel hardware such as GPUs. In this chapter we also discuss existing simulation software and explain the context in which the original implementation of CGPPD was created. In Chapter 3 we describe the original implementation of CGPPD in greater detail. In Chapter 4 we discuss the design and implementation of our modifications to introduce flexible linkers. In Chapter 5 we describe the techniques we used to verify and validate our modified model, and also discuss the benchmarking simulations which we used to test its performance. In Chapter 6 we discuss our application of the model to diubiquitin chain conformations, summarising the results of our simulations and comparing our simulated structures to previously published experimental data. Chapter 7 presents our conclusions and our ideas for future work.

Chapter 2

Background

In this thesis we describe a set of modifications made to a custom software package for protein-protein docking, and report the results of selected simulations which were made possible by these changes. In this chapter, we provide the context necessary for understanding the evolution of this software: we summarise the algorithms commonly used in protein-protein docking simulations, highlight some factors which introduce computational complexity, and discuss how it can be mitigated through optimisation techniques such as the use of coarse-grained models and parallelisation. We particularly focus on parallel programming for graphics processing units: although the GPU portion of the code is only tangentially affected by our modifications, it forms a crucial element of the design of the original implementation.

2.1 Protein-protein docking simulations

In this section we introduce the basic principles of computer simulation of proteins, focusing on algorithms and techniques which are used in CGPPD: such as the calculation of a potential energy force field, the Monte Carlo search algorithm, the enhanced sampling technique known as replica exchange, and the Kim and Hummer coarse-grained protein model. We also explore the effects of protein flexibility on docking simulations.

2.1.1 Protein structure

The primary structure of a protein is determined by its amino acid sequence. In its unbonded state, an amino acid consists of a central carbon atom with an amine and a carboxylic acid functional group attached at either end and a varying side chain. The side chain determines the type of the amino acid – approximately 300 amino acids are found in nature, but only 20 are commonly found in proteins.

The amine and carboxylic acid groups of adjacent amino acids can react to form a peptide bond, which allows linear chains of amino acids to be assembled into polypeptides. Large polypeptides are commonly called proteins. When the peptide bond is formed the amine group loses a hydrogen and the carboxylic group a hydrogen and an oxygen: the portions of the amino acids which remain within the protein are referred to as amino acid residues.

One amine and one carboxylic acid group remain intact at either end of the protein – they are called the N-terminus and C-terminus respectively. The central carbon atom of each residue is called the C_α atom: together, these atoms form the backbone of the protein. The side chains are exposed, and are free to interact with each other and other molecules. Sidechains determine a protein's fuction, and how it folds.

As a protein folds, it forms repeating local substructures such as alpha helices, beta sheets and turns. These are referred to as its secondary structure. The tertiary structure is the overall three-dimensional shape into which the protein folds, and quaternary structure is determined by the interactions between multiple proteins as they bind with each other to form multiprotein complexes.

2.1.2 Simulations

The goal of a docking simulation is to determine the probable structure of a multiprotein complex formed from two or more component proteins. A simulation thus needs to be able to generate possible configurations of the system given a starting state in which the components are not bound, and to have some means of distinguishing more and less favourable configurations.

A molecular system has many degrees of freedom: each possible configuration of the system thus represents a point in a multidimensional conformation space. We can construct a function which allows us to gauge the fitness of these possible configurations. This function can be represented as a surface, and if we are able to map the surface in sufficient detail we can make deductions about the structure of the system by analysing significant features. For example, minima are likely to correspond to bound states, and saddles to transitional pathways.

There are two broad approaches to docking. Some techniques calculate the chemical or shape complementarity of the docking components, usually traversing the conformation space with a systematic search. The more computationally intensive physical techniques calculate the potential energy of the system, and sample the conformation space using simulations which seek to minimise the potential energy [5]. We will focus on the latter approach, which is used by CGPPD.

The potential energy surface of a simulated multiprotein system is shaped like a rugged funnel, as shown in Figure 2.1. This is analogous to the funnel encountered in protein folding simulations. There are many similarities in the approach to folding and docking simulations, and many principles which govern folding simulations are also applicable to docking [6, 7].

A successful simulation should provide a representative sample of the potential energy surface. A simulation begins with the component molecules of the system in a configuration which is similar to a known naturally occurring state. It then generates an ensemble of possible conformations, by modifying the system using a search algorithm such as molecular dynamics (MD) or Monte Carlo (MC) and sampling its state at regular intervals.

In a molecular dynamics simulation, the changing positions of the simulated bodies are calculated through the repeated application of the laws of motion. Snapshots of this process are stored as samples. MD simulations thus model realistically how a molecular system changes over time, and provide data about the system's dynamic properties [4].

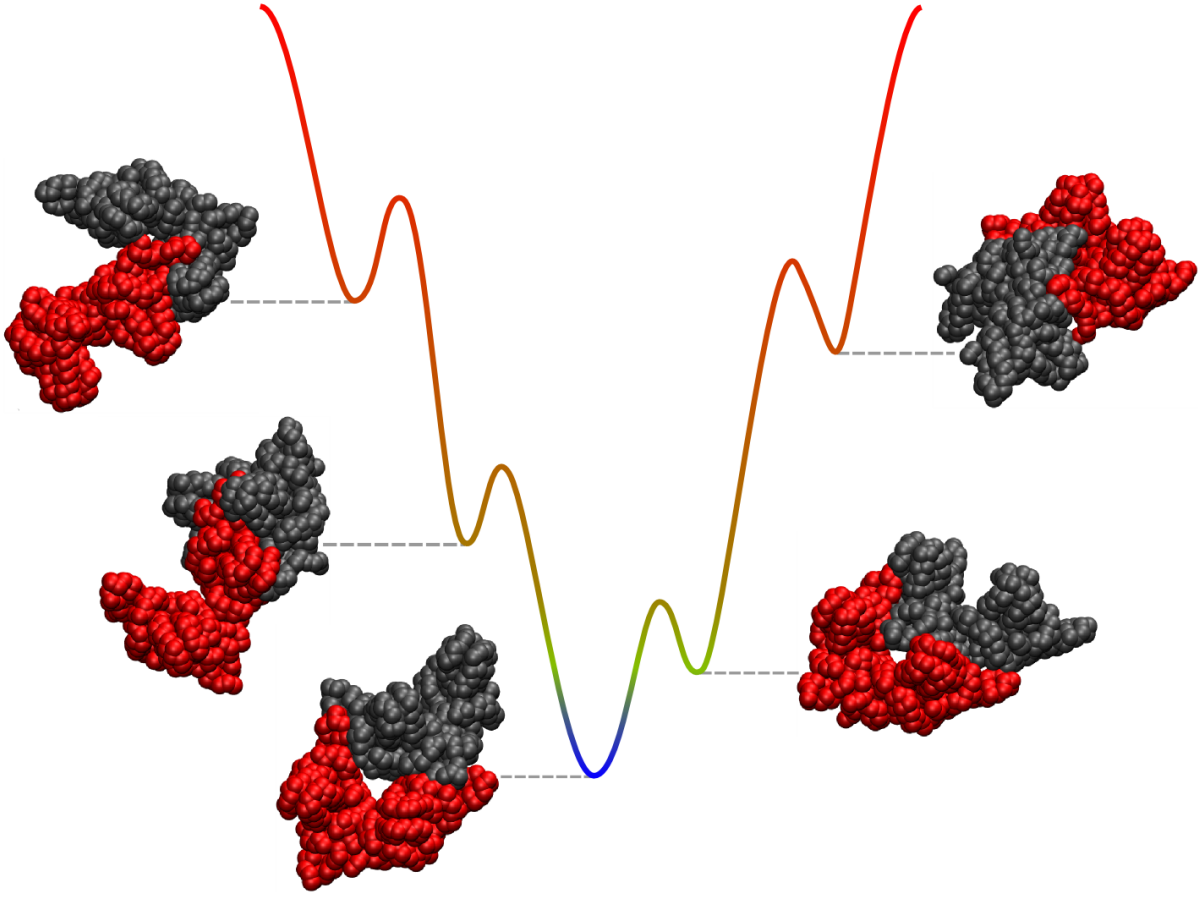


Figure 2.1: The rugged funnel of a protein-protein docking simulation *This is a simplified representation of the rugged funnel of a docking simulation, in this case the partial assembly of a viral capsid out of two component pieces. The correct configuration of the pieces corresponds to the global minimum of the funnel, and has the lowest energy state. The other, higher-energy local minima correspond to other possible stable configurations. (Figure taken from Tunbridge 2011 [5].)*

In contrast, Monte Carlo (MC) is a statistical method which selects samples of the conformation space using one of several possible criteria [8]. In molecular simulations, this is often the Metropolis sampling criterion, an example of Markov Chain Monte Carlo [9, 10]. Successive states of the system are derived from prior states through simple geometric transformations such as translations and rotations. New states are accepted or rejected using a scoring or potential function, according to a probability distribution. The sampled states do not model the motion of molecules realistically, and the order in which they appear is not meaningful.

Both of these search algorithms are directed rather than random: simulations seek to find the global minimum of the funnel-shaped potential energy surface. MC simulations do so explicitly by favouring lower-energy mutations, while in MD simulations the application of physics-based forces to the simulated particles tends to guide the system into lower-energy states.

Simulations run the risk of becoming trapped in local minima, and thus failing to explore valuable regions of the energy surface which can only be reached through unfavourable transitional states. Various enhanced sampling techniques, such as replica exchange, can be used to mitigate this risk [11].

Docking simulations can treat the component proteins as completely rigid bodies, or allow some degree of flexibility in the side chains, the backbone or both. The addition of flexibility greatly increases the number of degrees of freedom of the system, and thus the computational complexity of the simulation [12].

If a model and a simulation method are used to explore a multiprotein system for which a docked structure has previously been deduced, the quality of the simulation's results can be verified through a comparison of the simulated bound structures to the reference data. A metric such as RMSD (the root mean square distance) is commonly used to measure the similarity between two atomic structures.

It is more difficult to validate a model's suitability for use in simulations without a known reference structure. The simulation output can be checked for consistency with other experimental data, and the model's ability to predict the structure can be inferred from its effectiveness at predicting other, similar structures for which references are available.

The CAPRI experiment (Critical Assessment of Predicted Interactions) provides researchers with blind docking challenges [13] based on unpublished experimental data which is contributed to the project confidentially. Because the reference structures are not publically available, the ability to predict them correctly is a rigorous test of a model's accuracy.

2.1.3 Potential energy force fields

Calculation of the molecular system's potential energy is necessary both in molecular dynamics simulations, which use it to calculate the forces acting on each simulated body, and Monte Carlo simulations, which use it directly as a scoring function to guide their traversal of the docking funnel.

The potential is calculated using a model known as a force field, which comprises many component terms derived from various physical forces. Components commonly include two types of non-bonded interactions between pairs of atoms in different proteins, as well as bonded interactions between adjacent atoms within the same protein, representing bond stretching, bond angles and dihedral torsion.

Two types of non-bonded interactions are usually included: electrostatic interactions and van der Waals (vdW) forces. They differ in magnitude and the rate at which that magnitude varies with the distance between the atoms. Electrostatic interactions can be an order of magnitude stronger than vdW interactions. They also decay more slowly as the distance r between atoms increases, and are thus referred to as long-range, whereas vdW interactions are more short-range.

Given two atoms i and j a distance r apart, the electrostatic potential between them can be written as

$$F_{ij} = k\frac{q_i q_j}{r^2}$$

where k is Coulomb's constant, and q_i and q_j are the point charges of atoms i and j respectively. This formula is derived from Coulomb's law.

The short-range potential is typically modelled with a Lennard-Jones (LJ) type potential, which can be written as

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]$$

where ϵ is the strength of the interaction between the two atoms and σ is their interaction radius. The $1/r^6$ attractive term is derived from London dispersion forces, and the $1/r^{12}$ repulsive term approximates Pauli repulsion at short ranges, where the orbitals of the atoms overlap [4, 14].

The bond stretching and bond angle interactions are modelled as springs, and are derived from Hooke's law: $F = -kx$, where k is the spring constant, and x is the displacement of the bond length or bond angle from its equilibrium position. Similarly, the torsion potential measures the torque produced by the twisting of dihedrals, and is derived from the angular form of Hooke's law: $\tau = -\kappa\theta$.

The total potential of the system is a sum of all these components, which in turn are summed over all applicable groups of atoms. For a system of n atoms, the bonded component of the calculation has a computational complexity of $O(n)$. However, the non-bonded component has $O(n^2)$ complexity, as it requires summing the interactions between all pairs of atoms in different proteins. This makes the potential calculation very expensive computationally.

Potential energy algorithms may use various techniques to achieve performance better than $O(n^2)$. Summation may be truncated at some distance cut-off. A system of particles may be converted into a density grid or mesh which is used for the potential calculation. The long-range component of the potential may be summed in Fourier space, where it rapidly converges, rather than real space. These techniques are combined in methods such as particle-particle particle-mesh (P^3M) or particle-mesh Ewald (PME). In the multi-level summation method (MSM), long-range potentials are interpolated from multiple levels of grids [15].

2.1.4 Search algorithms: molecular dynamics and Monte Carlo

Monte Carlo (MC) and molecular dynamics (MD) are two popular search algorithms used in docking simulations. Both of these algorithms conduct a directed search of conformation space, favouring states with a lower potential energy.

MD methods simulate the evolution of the positions and velocities of atoms over time, by integrating Newton's second law of motion, $a = F/m$. The potential energy is summed separately for each individual particle, and includes the contributions from all bonded and non-bonded interactions involving that particle. These scalar values, mapped against the positions of the particles, form a potential field. The instantaneous force acting on each particle is the negative gradient of the potential field at its position: given a particle i, $F_i = -\nabla\mathcal{U}_i$. This result is used together with the particle's mass to derive its acceleration, which is integrated numerically to produce its updated velocity and again to calculate its updated position [4]. MD simulations are vulnerable to accumulated errors produced by numerical integration methods [16], which

restricts them to prohibitively short time scales. Nevertheless, MD is a very popular simulation method, implemented by many existing software packages.

A MC simulation method, such as Metropolis Monte Carlo [9,10], uses comparatively simple transformations to update the state of the system. If the proteins are treated as completely rigid bodies, mutations are restricted to whole-molecule translations and rotations, which perform the same transformation on every particle in a protein. Other mutations may be added to permit some flexibility; for example, individual particles in the protein may be translated or rotated, or rigid domains linked by a flexible segment may move with respect to each other.

The total potential energy of the system is summed after each mutation step, and used to evaluate the mutation, which is accepted or rejected. Mutations are always accepted if the potential energy of the new conformation, U_j, is lower than the potential energy of the previous state, U_j. The Metropolis Monte Carlo method also randomly accepts some higher-energy mutations: this reduces the likelihood that the simulation will become trapped in a local minimum.

In a simulation with a constant number of particles, volume and temperature (a canonical or NVT ensemble), the behaviour of the proteins should conform to the Boltzmann probability distribution, which states that the probability of an energy state ϵ occurring is proportional to $exp(-\epsilon/k_BT)$, where T is the absolute temperature and k_B is the Boltzmann constant. To adhere to this distribution, a simulation must permit transitions between all possible configurations.

The probability of a transition from state i to state j can be derived from the individual probabilities of these two states:

$$P(i \to j) = \frac{exp(-U_j/k_BT)}{exp(-U_i/k_BT)} = exp\left(\frac{-(U_j - U_i)}{k_BT}\right)$$

Thus an MC simulation can use this probability as an acceptance criterion if $U_j \geq U_i$. The full Metropolis acceptance criterion can be expressed as follows: given a random number s in the range $[0, 1]$, a transition from state i to state j will be accepted if $s < min\{exp\left(\frac{-(U_j-U_i)}{k_BT}\right), 1\}$ [17].

2.1.5 Enhanced sampling

Because of their bias towards lower-energy states, both molecular dynamics and Monte Carlo simulations run the risk of becoming trapped in local minima of the docking funnel. Several enhanced sampling approaches have been developed to mitigate this risk [11].

The magnitude of the potential energy barriers which a simulation can overcome is related to its temperature: the higher the temperature, the easier it is for the system to reach a high-energy state. It is therefore possible, for example, to improve conformational sampling by raising the temperature of the simulation, then cooling it to allow it to reach a new equilibrium. This technique is known as simulated annealing.

However, the equilibrium properties of the canonical ensemble rely on its constant temperature. Varying the temperature of an ensemble alters its statistical properties, increasing the proportion of high-energy states in low-temperature simulations or vice versa.

An alternative approach is parallel tempering, also known as replica exchange. This technique requires multiple replicas to be simulated in parallel, at a range of different constant temperatures. At regular intervals, conformations from pairs of replicas are randomly swapped, with a probability that preserves the ensemble statistics of both replicas in the exchange. The Metropolis acceptance criterion can be used for this purpose. Through this mechanism, high-energy conformations can be introduced to low-temperature simulations, which improves their sampling [18].

Exchanges are typically attempted between replicas with adjacent temperatures. Given two replicas i and j where $T_i < T_j$, the probability that their states are exchanged is

$$p = min\{1, \frac{P(i \to j)}{P(j \to i)}\} = min\{1, exp\,[(U_i - U_j)(1/k_B T_i - 1/k_B T_j)]\}$$

Replica exchange is a computationally expensive technique, because of the necessity of running multiple additional simulations in parallel. However, some of this expense is amortized by the increased efficiency of the sampling. The algorithm also lends itself to a variety of parallel implementations because all replicas are independent apart from the exchange step [19, 20].

2.1.6 Coarse-grained models

Proteins are large molecules: a small protein might contain 20-30 amino acid residues; the largest contain hundreds or even thousands. A single amino acid residue contains 20 atoms on average. Because the cost of the algorithm used to calculate potential energy in both MD and MC simulations scales quadratically with the number of bodies in the system, all-atom simulations of proteins are very expensive computationally. Trade-offs must be made between the size of the simulated system, the timescale of the simulation, and its real-world duration.

It is possible to model proteins at a coarser resolution, reducing the number of individual bodies in the simulation. Use of such a coarse-grained model reduces the problem size by a constant factor, and thus decreases the computational cost of the simulation. The size and time scales which coarse-grained models make feasible are similar to those which can be observed through spectroscopic techniques, which potentially allows direct comparisons between simulation and experiment [21].

There are several possible coarse-grained models for proteins, which differ by the degree and type of approximation used on the all-atom structure. Groups of atoms are aggregated into single bodies: one or more beads may be used to represent a single amino acid. In a lattice model, the aggregate bodies are confined to the vertices of a cubic lattice [22], while in a continuous-space model they can be positioned anywhere in 3D space.

Specialised force fields can be adapted for use with these representations. However, the coarser the model, the more challenging it is to construct a force field which is both accurate and generically applicable to multiple systems. Many coarse-grained models are therefore parameterised using a reference conformation, and simulations that use the model tend to be biased towards that conformation [21].

Several techniques exist for moving between coarse-grained and all-atom representations of

proteins without loss of information. This makes various multi-resolution approaches possible [23]. For example, a docking simulation may use a coarse-grained model in its initial passes to identify docking sites, and refine the results with an all-atom model [24]. Regions of interest could also be modelled at a higher resolution than areas of the system where detail is less important [25].

The simplest coarse-grained representations are elastic networks (ENMs), which model amino acids as beads connected by springs. They require knowledge of the equilibrium reference conformation of the system. Although they are very simple, they correctly reproduce the topology of the system and can accurately predict its principal modes [21]. They have several applications in the study of general protein behaviour and in the analysis and refinement of low-resolution experimental data [26].

Go-like models represent proteins as chains of amino acids, originally with one bead per amino acid, with simple non-bonded attractive or repulsive interactions modelled between beads. The model is strongly biased towards the reference configuration, and can successfully reproduce some of the dynamics of the folding of a protein towards this state. However, it is less successful at determining intermediate states of the folding process, although the addition of more complex interactions to the model can mitigate this [21].

Refinement of Go-like models through the addition of more complex potentials has led to the development of a variety of models which parametrise amino acids. The coarsest of these use a single bead per amino acid and do not represent side chains explicitly, while the finest can use four to six beads. The coarser the model, the more biased it tends to be towards a reference conformation.

CGPPD uses Kim and Hummer's 2007 model. Each amino acid is represented by a single bead centered on its C_α atom, with a radius equal to the residue's van der Waals radius. Use of this model thus reduces the number of bodies in the simulation by a factor of approximately 20. Folded domains of a protein are treated as rigid bodies, which may be connected by flexible linkers modelled as polymers. The potential interactions between the proteins are calculated at the residue level, and comprise short-range and long-range non-bonded interactions, as well as additional bonded interactions within the flexible linkers. A modified LJ-type formula is used for pairs of residues which react with each other less favourably than with the solvent: this term is thus dependent on a table of interaction radii calculated for different pairs of residues [1].

This model was parametrised using experimental data for the second virial coefficient of lysozyme and the binding affinity of the ubiquitin-CUE complex. There are several possible ways to weight the potential contribution of residues depending on how much of their surface area is exposed to the solvent. The option which optimised the binding affinity of the ubiquitin-UIM complex was selected: this was the simplest model, an equal weighting.

The model was successfully used to predict the structures of several other complexes involving ubiquitin, and approximately half of the other test cases. In all cases it correctly predicted the binding sites of at least one protein in the complex.

2.1.7 Flexibility

Regardless of the search algorithm used, proteins in docking simulations may be treated as rigid bodies or be partially or entirely flexible. In molecular dynamics, the rigidity of domains is maintained through constraint algorithms. In Monte Carlo simulations, the degree of protein flexibility is determined by the types of mutations which are permitted.

A completely rigid protein has only six degrees of freedom, as it may be either translated or rotated in a three-dimensional space. Flexibility adds a very large number of degrees of freedom to each protein. Rigid simulations thus have a much smaller search space than flexible simulations, and are much less computationally expensive [12].

There are techniques which allow flexibility to be introduced into a simulation implicitly. In soft docking simulations, protein surfaces are smoothed, or some degree of interpenetration of the residues is allowed. In cross or ensemble docking, several rigid simulations are run from different starting conformations, an ensemble of which is generated in a first pass by a flexible simulation [27].

Explicit flexibility can apply to both side chains and the backbone of the protein. A partially flexible simulation may allow only mobility of side chains while keeping the backbone rigid. If the backbone is also made flexible, it becomes possible for the tertiary structure of the proteins to change, which introduces elements of protein folding into the simulation [27]. The Kim and Hummer coarse-grained model limits flexibility to linkers which connect rigid domains.

Flexibility can contribute positively to docking if it is necessary for a protein to change shape significantly from its starting conformation in order to make a docking site accessible. However, in cases where little conformational change occurs during docking, the small benefit derived from flexibility may be offset by the additional cost of running a flexible simulation [28]. There are also cases in which rigidity has a positive impact on binding [29]. Thus some knowledge of the system being simulated is required in order for the appropriate level of flexibility to be selected. The results of a rigid docking simulation may be refined with a second pass which introduces flexibility [30].

2.2 Hardware and software for computational chemistry

In this section we give a brief introduction to the use of graphics processing units to parallelise simulation software, focusing on NVIDIA's GPU hardware and the CUDA API, which are used in CGPPD. We also give an overview of existing software packages for simulation, and summarise the features of the initial implementation of CGPPD.

2.2.1 Parallelisation and graphics processing units

Optimisation is an important concern in computationally intensive applications such as molecular simulations, and a technique which can yield considerable speedups is the parallelisation of some or all of the application code. This requires access to parallel hardware, which is capable of executing multiple threads of operations simultaneously, but also some alteration of the software

so that it can make efficient use of the available hardware resources. Not all algorithms can be parallelised easily, or at all: the work to be performed must be divisible into parts which are mostly independent of each other, so that it is possible for them to be performed in parallel.

There are several hardware approaches to parallelism: a processor may have multiple cores, a single computer may have multiple processors, and several computers may be networked into a cluster or a distributed computing system. Within the past decade, an alternative processor architecture, the graphics processing unit (GPU), has become ubiquitous in commodity graphics cards. The computational capabilities of modern GPUs exceed by far those of comparable CPUs, and their relatively low price has made parallel computing more accessible.

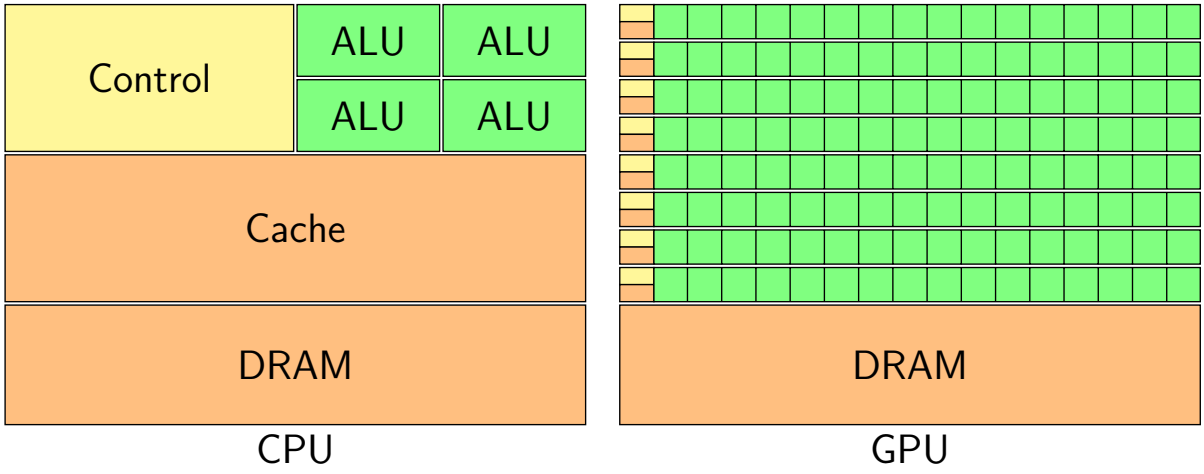


Figure 2.2: Differences between the CPU and GPU architectures *A GPU devotes more transistors to data processing than a CPU, at the cost of more limited caching and flow control.*

GPUs are SIMD (single instruction, multiple data) compute devices which execute the same function, or *kernel*, on all elements of a stream of homogeneous data simultaneously. Figure 2.2 is a simplified visualisation of the broad differences between the two processor architectures: a GPU dedicates more transistors to the processing of data. Less sophisticated flow control is required because the same instruction is executed on multiple sets of data at once. Massive multithreading and fast context-switching can be used to hide latency of memory accesses in individual threads, which reduces the need for data caching. This design makes GPUs particularly suited to algorithms which exhibit a high level of data parallelism and a high ratio of arithmetic calculations to memory accesses.

Code written for the CPU must be ported explicitly to make use of GPUs. Although GPUs could initially only be accessed through graphics APIs, in response to the growing interest in their use as general-purpose computing devices, several general-purpose programming APIs were introduced. In this section we will focus on NVIDIA's CUDA [31], which is used by CGPPD. OpenCL is a popular alternative open standard which was designed to be cross-platform [32].

Programming for the GPU remains a non-trivial task, as it requires explicit management of limited memory resources. Although naive GPU implementations are conceptually simple, achievement of good performance results often requires the use of sophisticated optimisation techniques.

GPU implementations of simulation code typically offer speedups of 10-100 times over optimised CPU implementations [15], with some examples reaching speedups of 700 or 1400 times [2, 33]. However, speedups of this magnitude are not guaranteed, and are dependent on the algorithm used, the degree of optimisation and the size of the simulation data.

2.2.2 The CUDA programming model

In this subsection we introduce the principles of general purpose GPU programming using the Compute Unified Device Architecture (CUDA), the proprietary API developed by NVIDIA for its GPU devices. We will focus on this technology because it is the API which used by CGPPD. However, this information can easily be transferred to the OpenCL API, which is very similar, although some of its terminology is different.

CUDA can be integrated into many commonly used programming languages, through language extensions, third-party wrappers or accelerated libraries. CGPPD, which is written in C++, uses CUDA's C++ language extensions.

NVIDIA's GPU hardware versions are referred to as *compute capability*, a figure comprising a major and minor version number. All specific graphics card models with the same compute capability share the same set of hardware-supported features. This version is distinct from the CUDA version number, which refers to the software version of the CUDA API.

GPUs were originally designed to render computationally intensive graphics in applications such as computer games. This did not require high floating-point accuracy, and thus early GPUs had poor or non-existent support for double precision arithmetic. All CUDA GPUs of compute capability 1.3 or higher support native double precision arithmetic [34], and are therefore more suitable for scientific applications than older models.

The basic unit of parallel CUDA code is a kernel, a specialised function which is executed on multiple threads in parallel. One of the first steps in the porting of a serial algorithm to the GPU is the elimination of serial iteration over data sequences, as shown in Figure 2.3. Instead, the data is assigned to threads according to an appropriate user-defined mapping, and each thread processes its portion of the data in parallel when the kernel is run.

On the hardware level, a GPU contains an array of streaming multiprocessors (SMs) aggregated in texture programming clusters (TPCs). Each SM in turn contains an array of scalar processors (SPs) and a smaller number of special function units. The sizes of these arrays differ between generations of the architecture, but it is unnecessary, except possibly during optimisation, for the programmer to be aware of this level of hardware detail. The abstraction provided by the CUDA API allows the same code to be run unmodified on different GPU models. Figure 2.4 illustrates both the physical processor hierarchy of an NVIDIA GPU and the virtual CUDA thread hierarchy [35].

In the CUDA thread hierarchy, threads are grouped into blocks, which are grouped into a grid. The dimensions of the grid and the blocks are configurable: the grid may be one- or two-dimensional, while blocks may be one-, two- or three-dimensional. There are upper bounds on each dimension as well as the total number of subdivisions in each level of the hierarchy: these limits vary between generations of the architecture. Thread blocks are are dynamically

```
void add_vectors(int N, float * A, float * B, float * C) {
        for (int i = 0; i < N; i++) {
                C[i] = A[i] + B[i];
        }
}
//...
add_vectors(N, A, B, C);
```

(a) Serial CPU implementation

```
__global__ void add_vectors(float * A, float * B, float * C) {
        int i = threadIdx.x;
                C[i] = A[i] + B[i];
}
//...
add_vectors<<<1, N>>>(A, B, C);
```

(b) GPU kernel

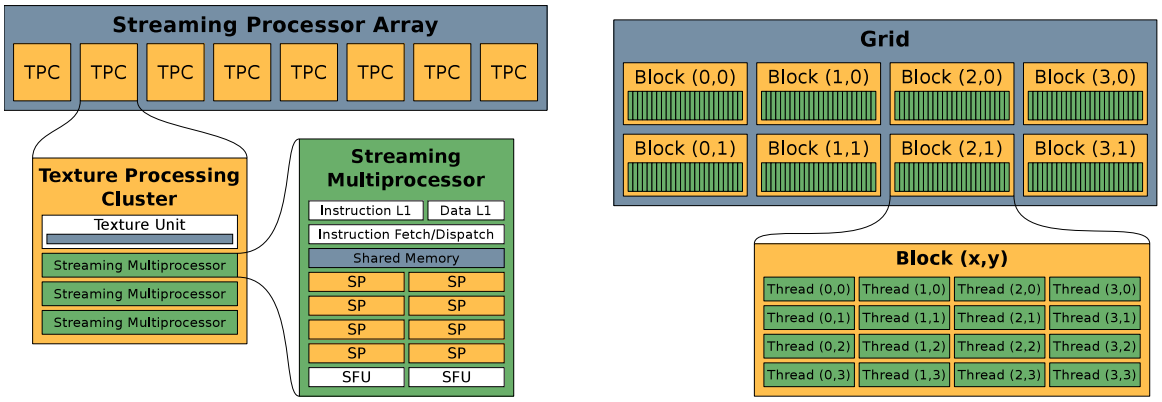
Figure 2.3: Comparison of CPU and GPU implementations of vector addition in C++ The function in code fragment (a) iterates serially over pairs of values and sums them. The GPU kernel in code fragment (b) is executed in parallel by multiple GPU threads: an extended function call syntax is used to describe the geometry of the grid and the thread blocks to be allocated to the kernel. Each thread sums one pair of elements: the thread index variable, which is different for each thread, determines which elements are accessed.

scheduled to be executed on the available hardware. Each block is assigned to an SM, which maps each thread in the block to one of its SPs. A SM may execute multiple blocks concurrently, but there is a limit on the total number of blocks and the total number of threads which may be executed concurrently on one SM.

Blocks are subdivided into warps of 32 contiguous threads each: this size is fixed in the hardware. A warp is the smallest unit of threads which can be processed simultaneously by an SM: the same instruction is guaranteed to be executed on all the threads in a warp in lockstep. It is thus important to avoid divergent conditional instructions within a single warp, as they will be executed serially.

The configurable shape of the grid and the blocks can facilitate an intuitive mapping of data to threads: for example, a two-dimensional grid of two-dimensional blocks could be used for a two-dimensional array. The selection of the block size affects how efficiently the GPU's resources are utilised, and is an important consideration during optimisation. It is common to select a block size which is a multiple of 32, so that the block divides evenly into warps.

SMs are able to hide latency by context switching between different warps. The ratio of the number of active warps per SM to the maximum number of possible active warps is known as *occupancy* [36], which can serve as a useful indirect metric during optimisation. Higher



(a) GeForce 8800 series architecture (b) CUDA thread hierarchy

Figure 2.4: Comparison of physical GPU processor architecture and the CUDA thread hierarchy *CUDA's grid and block abstraction (b) allows the programmer to write generic code which can be executed on any processor layout, such as (a). Each thread block is dynamically scheduled to run on one SM. (Figures taken from Tunbridge 2011 [5].)*

Type	Location	Cached	Access	Scope	Lifetime
Register	On chip	n/a	read/write	thread	thread
Local	Off chip	2.x only	read/write	thread	thread
Shared	On chip	n/a	read/write	block	block
Global	Off chip	2.x only	read/write	global	application
Constant	Off chip	Yes	read-only	global	application
Texture	Off chip	Yes	read-only	global	application

Table 2.1: CUDA memory spaces A summarised feature comparison of the six types of memory available on the GPU. (Table derived from the CUDA C Best Practices Guide 6.5 [36].)

occupancy does not always guarantee better performance, since there is a threshold beyond which it yields no further benefit, but low occupancy usually indicates poor latency hiding capability.

All the memory resources available to a block are shared between all threads in the block. This means that there may be insufficient resources available to support the maximum possible number of threads per block. However, a block size which is too small may lead to reduced occupancy, because the SM's block limit will be reached before the thread limit.

CUDA has a hierarchy of six different memory types: per-thread registers and local memory, per-block shared memory, and application-wide global, constant and texture memory. Each type of memory has certain advantages and disadvantages, as shown in Table 2.1.

The global memory store is the largest, and also the memory type with the highest latency, as it is physically located far from the GPU core. Registers are used to store local thread variables, and have very low latency. Local memory is used automatically by the compiler to store variables which cannot fit into registers: its name refers to its scope; it has the same physical location as global memory and is equally slow. Shared memory is physically identical to registers, and can

be accessed by all the threads in one block. Constant memory is a read-only global store which is cached by SMs. Texture memory is similar, but is cached by the texture processing units. Texture memory accesses also cache data near the accessed element: this behaviour makes it more similar to a CPU cache. Concurrently executing threads can thus make efficient use of this cache if they access spatially related data. Devices with compute capability 2.0 or higher also cache local and global memory [36].

One of the most important memory-related optimisations in CUDA programming is the coalescing of global memory accesses. If memory accesses are aligned correctly, all threads in a warp can potentially read or write a 64-bit word from or to global memory in a single coalesced transaction, which greatly reduces the latency cost. The memory addresses must be contiguous, have an order corresponding to the threads in the warp, and be offset correctly so that they fall within the same addressable segment of the memory store [35, 36].

Copying data between the host (CPU) and device (GPU) is a very high-latency operation, which can be mitigated by the asynchronous use of the GPU, which can be achieved with *streams* in CUDA. If multiple CPU threads are able to execute code on the GPU independently, it is possible to hide the latency of the data transfer by overlapping it with computation: some threads can perform calculations on the GPU while other threads are transferring data to and from it. This also allows efficient concurrent use of the CPU and GPU.

2.2.3 Existing simulation software

One of the most fundamental elements of a simulation package is the implementation of a force field, the theoretical model which describes the energy of the simulation as a function of the coordinates of its component particles [37]. Some commonly used force fields include Amber [38], CHARMM [39], GROMOS [40], and OPLS-AA [41].

Each of these force fields was originally developed for use with a specific software package, often with the same name: Amber [42], CHARMM [43], GROMOS [44], and BOSS and MCPRO [45], respectively. However, today most of the widely used packages implement several force fields. Popular packages other than those already mentioned include NAMD [46], LAMMPS [47], and GROMACS [48] (the open source reimplementation of GROMOS). Most of these packages focus on molecular dynamics simulations, but some (like BOSS and MCPro) are primarily used for Monte Carlo and some (like CHARMM) include support for it.

It is possible to use many of these packages for coarse-grained simulations, if an appropriate force field is available. For example, the MARTINI force field [49, 50] may be used with GROMACS, GROMOS, and NAMD, and the PRIMO force field [51] has been developed for use with CHARMM. There are, however, also custom packages specifically developed for other coarse-grained models, such as IBIsCO [52] or ESPResSo [53].

Both molecular dynamics and Monte Carlo packages can take advantage of the parallel nature of MC and MD algorithms to optimise performance through parallelisation. While GPUs were initially used only to visualise structure and trajectory data produced by MD simulations, they are now widely used in molecular modelling to accelerate a variety of scientific calculations, including many algorithms for potential summation, typically providing speedups of one to two

orders of magnitude [54]. Packages which make use of GPU acceleration include NAMD [15], MDGPU [55], HOOMD [56], OpenMM [33] and ACEMD [57].

Several custom packages have been written specifically for protein-protein and protein-ligand docking simulations, and implement a variety of docking algorithms. For example, AutoDock [58] and GOLD [59] are both implementations of genetic algorithms for flexible docking. RosettaDock [24] uses a multi-resolution MC approach: initial coarse-grained passes are used to identify docking sites, and subsequently the fit is refined with all-atom simulations. ZDOCK [60] uses a shape complementarity scoring function to perform initial-stage rigid-body protein docking, and a package such as ZRANK [61] or RDOCK [30] may be used to refine its results. HADDOCK [62] drives protein-protein docking simulations using biochemical or biophysical information derived from experimental data.

2.2.4 CGPPD v1

CGPPD (Coarse-Grained Protein-Protein Docker) [5] is a hybrid CPU/GPU implementation of the Kim and Hummer coarse-grained protein-protein docking model. It was initially developed to provide a faster implementation of the model than a modified version of CHARMM, and achieved a speed-up of 4 to 1400 times, depending on the size of the simulated system. To the best of our knowledge, it remains the fastest implementation of this coarse-grained model.

The focus of our research was the addition of flexible linkers to the initial implementation, which treats proteins as purely rigid bodies. Both the initial implementation and the modifications which were made are described in greater detail in the following two chapters.

CGPPD is written in C++, and NVIDIA's CUDA API is used for its GPU component. The search algorithm used is Monte Carlo with replica exchange. The potential energy calculation used in the Monte Carlo mutation evaluation step is parallelised on the GPU. Additionally, CPU multithreading is used to parallelise the simulation replicas.

Chapter 3

CGPPD: a coarse-grained protein-protein docking application

CGPPD is a C++/CUDA application for simulating protein-protein docking using a coarse-grained model and the Metropolis Monte Carlo method with replica exchange [5]. In this chapter we summarise the initial implementation of this application, which will henceforth be referred to as CGPPD v1. The following chapter will describe the modifications and additions that we made in CGPPD v2.

The focus of CGPPD v1 was to port the model and simulation method described by Kim and Hummer [1] to the parallel GPU architecture. CGPPD v1 was designed and developed iteratively, so that performance improvements could be introduced and evaluated incrementally. The final version of the code exports the costly non-bonded potential calculation to the GPU, and uses CPU multithreading for replica exchange. An option is available to execute the GPU calculations asynchronously using CUDA streams, which allows CPU and GPU computation to be performed more efficiently in parallel. The application can scale to an arbitrary number of GPUs.

3.1 Design

3.1.1 Model overview

The coarse-grained model proposed by Kim and Hummer represents each amino acid residue by a spherical bead centered on its C_α atom, and each protein by a chain of residue beads. This reduces the number of bodies in the simulation by a factor of approximately 20. CGPPD v1 treats the proteins as rigid bodies, which simplifies many aspects of the implementation. The introduction of flexible linkers is the focus of our research, and will be discussed in greater detail in the following chapter.

The simulation method used is Metropolis Monte Carlo with replica exchange, both of which use the interaction potential of the system as a scoring function. Because the proteins are rigid, the only Monte Carlo mutations which are implemented are whole-molecule translations and rotations. The cost of performing these molecule transformations scales linearly with the

number of residues in the system, and is much lower than the cost of the potential calculation. The calculation of the mutations is thus performed on the CPU.

The interaction potential is the sum of the pairwise non-bonded potential interactions between all residues in the system. Because the proteins are rigid, contributions to the total potential sum from bonded potential components as well as non-bonded interactions between pairs of residues within the same molecule remain constant, and are not calculated. The calculation scales quadratically with the total number of residues, and is thus the most costly portion of the simulation. Because the pairwise interactions are independent of each other, this calculation can readily be parallelised and performed on the GPU: the calculation of forces for individual residues can be mapped to CUDA threads, and the components can be summed with an efficient parallel reduction. Each pairwise calculation requires a lookup of contact potential data which is dependent on the type of each amino acid. During the development of CGPPD v1, various types of GPU memory were evaluated for their suitability for the storage of this lookup table.

Replica exchange is parallelised on the CPU with the use of multithreading, which allows all cores of a multi-core CPU architecture to be utilised efficiently. Replicas are divided between CPU threads, which can run in parallel independently of each other in between exchange steps.

3.1.2 Interaction potential

To calculate the total interaction potential, CGPPD sums the non-bonded potentials between all pairs of residues in the simulation, excluding pairs of residues which lie within the same molecule.

The non-bonded interaction potential between each pair of residues comprises a short-range Lennard-Jones-type potential and a long-range electrostatic Debye-Hückel-type potential. Given two residues i and j, which are distance r apart, the total potential $\varphi_{ij}(r)$ is the sum of an LJ-type component $u_{ij}(r)$ and a DH-type component $u_{ij}^{el}(r)$.

$$\varphi_{ij}(r) = u_{ij}(r) + u_{ij}^{el}(r)$$

Kim and Hummer define the LJ-type potential as

$$u_{ij}(r) = \begin{cases} 4|\varepsilon_{ij}| \left[\left(\frac{\sigma_{ij}}{r} \right)^{12} - \left(\frac{\sigma_{ij}}{r} \right)^{6} \right] & \text{if } \varepsilon_{ij} < 0, \\ 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r} \right)^{12} - \left(\frac{\sigma_{ij}}{r} \right)^{6} \right] + 2\varepsilon_{ij} & \text{if } \varepsilon_{ij} > 0, r < r_{ij}^{0}, \\ -4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r} \right)^{12} - \left(\frac{\sigma_{ij}}{r} \right)^{6} \right] & \text{if } \varepsilon_{ij} > 0, r >= r_{ij}^{0} \end{cases}$$

where σ_{ij} is the interaction radius between residues i and j, ε_{ij} is the strength of the interaction, and $r_{ij}^{0} = 2^{1/6}\sigma_{ij}$. If $\varepsilon_{ij} > 0$, the two residues repel each other; if $\varepsilon_{ij} < 0$ they attract each other.

σ_{ij} is the average of the van der Waals radii of residues i and j. The radius of each residue depends on its amino acid type:

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}$$

The interaction strength ε_{ij} is defined as

$$\varepsilon_{ij} = \lambda(e_{ij} - e_0)$$

where e_{ij} is the contact potential between residues i and j, which is experimentally derived and depends on the amino acid type of both residues. The offset parameter e_0 adjusts the strength of the residue-residue interactions relative to residue-solvent interactions, and the scaling factor λ weights the strength of the LJ-type component relative to the electrostatic component of the potential. These parameters are derived through fitting against experimental data. The values used in CGPPD are taken from the 1996 Miyazawa and Jernigan model [63].

The DH-type component is defined as

$$u_{ij}^{el}(r) = \frac{q_i q_j exp(-r/\xi)}{4\pi D r}$$

where q_i and q_j are the charges of residues i and j corresponding to pH 7, and are equal to $-e$, e or $0.5e$ where e is the elementary charge. ξ is the Debye screening length, and D is the dielectric constant of the solvent. CGPPD uses $D = 80$, the dielectric constant of water.

Further weighting factors may be applied to the total interaction potential $\varphi_{ij}(r)$ to take into account the effect of the solvent-accessible surface area (SASA) of each residue pair. Kim and Hummer suggest six possible models; CGPPD employs the simplest of these, which weights each pairwise interaction with a constant factor of 1.

3.1.3 Monte Carlo

At each Monte Carlo step, a random molecule is selected for mutation. The type of mutation is randomly selected: it can be either a translation or a rotation of the entire molecule. The mutation is then evaluated with the Metropolis acceptance criterion. A mutation is accepted with a probability of

$$p = min\{1, \frac{-(U_j - U_i)}{k_B T}\}$$

where U_i and U_j are the interaction potentials of the conformation before and after the mutation is applied, respectively, k_B is the Boltzmann constant and T is the temperature of the conformation.

The mutation is always accepted if the new potential energy is lower than the previous potential energy. If it is higher, the mutation is accepted or rejected randomly according to the Boltzmann probability distribution. If the mutation is rejected, the mutated molecule is restored to its previous state.

3.1.4 Replica exchange

CGPPD uses replica exchange, or parallel tempering, to improve the sampling of conformations. Multiple replicas of the same conformation at different temperatures are simulated in parallel. At regular intervals during the Monte Carlo simulation, the conformations of adjacent replicas are exchanged if they meet the Metropolis acceptance criterion; a condition which ensures that the ensemble statistics of each replica are not affected [18].

The range of temperatures of all the replicas, $T_0, T_1 ... T_i$, forms a geometric progression. These values are calculated as follows: if T_{min} and T_{max}, are the minimum and maximum temperatures to be simulated, then for each i, $T_i = T_{min} r^i$, where $r = (\frac{T_{max}}{T_{min}})^{\frac{1}{N-1}}$.

At each replica exchange step, all of the replicas, ordered by temperature, are grouped into adjacent pairs. At every second step this is done with an offset of one replica. For each pair of replicas, an exchange is performed with a probability of

$$p = min\{1, exp[(U_i - U_j)(\frac{1}{k_B T_i} - \frac{1}{k_B T_j})]\}$$

where U_i and T_i are the interaction potential and temperature of replica i, respectively, and k_B is the Boltzmann constant.

3.2 Implementation

The CGPPD v1 code has a modular structure and was designed according to the object-oriented programming paradigm. Elements of the physical coarse-grained model are mapped to classes within the code which aggregate related properties and functions.

Three different versions of the potential energy calculation have been implemented: a CPU-only version, which allows the code to be run on a computer without a CUDA-capable GPU; a synchronous GPU version; and a more efficient asynchronous GPU version which uses CUDA streams, and thus requires a GPU with compute capability of 2.0 or higher. Most of the code is common to all three versions.

3.2.1 Input

Molecule data is read from files in the Protein Data Bank (PDB) format; a widely used standard. Because CGPPD uses a coarse-grained protein model, only the C_α atoms are processed. All other atoms read from the file are ignored.

Simulation properties are read from a custom configuration file with a simple format. The first portion of the file is a list of space-separated key-value pairs which specify properties such as the duration of the simulation, the sampling frequency, the temperature range or the total number of replicas. The second portion is the files section, which lists the molecules to be added to each replica. Each entry comprises a path to a PDB file and the starting position and rotation of the molecule. The position can be specified either as the absolute coordinates of the molecule centre or a translation relative to the starting position in the PDB file. The optional third section of the configuration file is used in simulations which model the effects

of macromolecular crowding, a phenomenon which causes molecules to behave differently in a solution with a high concentration of macromolecules – conditions which are frequently found in living cells. In CGPPD certain molecules in the simulation can be designated as crowders, which causes them to be modelled as spheres with a simpler, sharply repulsive interaction potential.

Properties which are set by the configuration file can be overridden by the user with commandline parameters passed to the application.

3.2.2 Data structures

The smallest element in the object hierarchy is the Residue object, which stores each amino acid's position. Both an absolute position and a position relative to the centre of the molecule are stored, because this simplifies several calculations performed during the simulation. Each residue also has properties which depend on its type: the electrostatic charge, the van der Waals radius, and the contact potential between that residue and other residues of every possible amino acid type (twenty in total). All residues of a given type share the same values for these properties. Replicating the twenty contact potentials in each residue object would be an unnecessary waste of space in memory. Each residue thus stores an amino acid type index which is used to look up these values in an external data structure: the AminoAcids class, which stores all 210 possible contact potentials. The contact potential data is duplicated so that the order of the two residues being looked up does not matter.

Residues are aggregated within a Molecule object. The residue objects are stored in a contiguous array in memory, so that they can be accessed more efficiently. Each molecule also stores the position of its centre, as well as its cumulative rotation from its starting position, and both are printed to an output file when the simulation is sampled. Because the molecules are rigid, these values are sufficient to recreate a full PDB file from each sample during post-processing of the data.

A Replica object aggregates Molecule objects. The Replica class is responsible for most of the functions for performing the steps of the Monte Carlo simulation and calculating the potential energy, although some of this functionality is delegated to the Molecule class and other helper classes such as the CUDA code for calculating the potential on the GPU.

Multiple replicas are aggregated within the top-level Simulation class, which manages the replica exchange and the global properties of the application.

3.2.3 Random number generation

Random numbers for the simulation are generated using the GNU Scientific Library (GSL) [64] implementation of the Mersenne Twister, an algorithm designed specifically for Monte Carlo simulations and which meets their need for very long non-repeating sequences of random numbers [65]. The Mersenne Twister provides a random number generator with a period of $2^{19937}-1$. Because CGPPD only consumes approximately 2^{40} random numbers per 10^{10} Monte Carlo steps, a single instance of this generator would be sufficient even for simulations several orders of magnitude longer in duration than those typically performed with CGPPD. In the implementation,

each Replica object uses a separate generator, and another generator is used for the replica exchange. This simplifies the process of testing the application, because each replica is assured not to be affected by the state of other replicas in between replica exchange steps.

3.2.4 Multithreading and replica exchange

A replica exchange Monte Carlo simulation is a series of replica exchange steps interspersed with Monte Carlo simulation steps. The MC steps form the largest portion of the total simulation. MC steps performed on different replicas are completely independent of one another – no transfer of data between replicas is required in between the replica exchange steps. Most of the simulation is thus embarrassingly parallel and can easily be sped up on a multi-core CPU architecture if the MC processes are assigned to different CPU threads or cluster compute nodes.

If there are as many processors available as there are replicas in the simulation, it is theoretically possible to achieve a linear speedup proportional to the number of replicas. Various factors affect this speedup in practice. Hardware limitations in desktop computers make it likely that each processor core will need to be shared by multiple replicas. If the code is executed on compute nodes on a cluster, the communication overhead between nodes may cause the data transfer during replica exchange steps to be more expensive. If the potential calculation is exported to a GPU, the availability of the GPU will have to be considered as well.

The number of threads used by CGPPD in a simulation is configurable, and must be smaller than or equal to the number of replicas. It would be conceptually simple to assign each replica to its own thread, but if multiple threads were to compete for the same core, the CPU would have to swap them in and out. This would lead to unnecessary computational overhead and slow the simulation. The recommended configuration is thus one thread per available processor core.

Replicas are distributed evenly between the available threads as follows: given N replicas and T threads, $\lceil \frac{N}{T} \rceil$ contiguous replicas are assigned to each thread except the last, while the last thread is assigned $N - \lceil \frac{N}{T} \rceil \cdot (T - 1)$ replicas.

A simple threading model for the simulation would use a main thread to perform replica exchange and child threads to perform the Monte Carlo steps in parallel. At each simulation step, the main thread would launch the MC threads, wait for them to join, then perform the replica exchange step. In the next step, a new set of child threads would be launched.

Unfortunately, this model triggered a bug which was present in CUDA at the time when CGPPD v1 was written. Before CUDA functions can be called from a particular thread, the CUDA runtime needs to be initialised from that thread. A bug in the initialisation function caused it to leak a small amount of memory whenever it was called. If CGPPD were to launch a new set of threads in each replica exchange step, the leaked memory would rapidly accumulate and eventually cause the program to run out of memory and crash.

To work around this limitation, CGPPD v1 reuses the same set of threads to perform the MC steps for the duration of the simulation. This means that synchronisation between the threads has to be managed explicitly.

Instead of waiting for the child threads to join, the main thread waits for a signal from the

child threads before beginning replica exchange. This signal is sent by the last child thread to complete its set of MC steps. The number of threads which have completed their work and are in a waiting state is tracked with a threadsafe counter. When it receives this signal, the main thread performs the replica exchange step and signals all the child threads to indicate that the next set of MC steps is to be performed.

During the replica exchange step, the main thread of the program iterates over all the replicas, sorted by temperature, two at a time. This loop starts at replica R_0 during every second step, and at replica R_1 during every other step. This offset makes it possible for a conformation to move between multiple different replicas over the course of several exchanges. Each pair of replicas is evaluated. If the replicas meet the Metropolis acceptance criterion, their conformations are exchanged.

It would be extremely inefficient to perform this exchange by copying the molecule data between the two Replica objects, especially if this required copying data between different threads or compute nodes. It is much simpler and less expensive to leave most of the replica data in place and only swap the temperatures (as well as some counters used to record each replica's state).

However, if the replicas remain in place while their temperatures are exchanged, an additional mechanism is required to maintain the temperature ordering for subsequent replica exchange steps. This is achieved with an ordered map of temperatures to replica positions, which is updated after every replica exchange.

This mapping does not affect which replicas are processed by which threads during the Monte Carlo step. Each MC thread iterates over the contiguous replicas which were originally assigned to it, regardless of their updated temperatures.

The minumum and maximum temperature values for the simulation, T_{min} and T_{max}, are read from the configuration, as well as the desired number of replicas N. The intermediate temperature values are calculated when the replicas are initialised. Typical values are $T_{min} = 250K$, $T_{max} = 500K$, and $N = 20$: this was the temperature progression used by Kim and Hummer [1].

3.2.5 Monte Carlo

In each Monte Carlo step, a single molecule to be mutated is selected randomly, and its state is saved so that it can be restored later if the mutation is to be discarded. Because the molecules are rigid, only two types of Monte Carlo mutations are performed: whole-molecule translations and rotations.

In both cases, a vector is randomly generated so that each of its components lies in the range $[-0.5, 0.5)$, and then normalised.

In a translation step, this vector is scaled to the translation step length of the replica, and added to the position of every residue in the molecule as well as the centroid of the molecule.

In a rotation step, a quaternion is generated with the random vector as its axis of rotation and an angle of rotation specified by the replica, applied in a clockwise direction. CGPPD uses

64-bit quaternion rotation with 64-bit axes of rotation to minimise the cumulative numerical errors which would otherwise cause gradual distortion of the rotated molecules.

The translation and rotation step size for each replica is proportional to its temperature. These values also follow geometric progressions, and are calculated with the same method as the temperature values. The minimum and maximum values are configured in the code by means of compile-time flags. Typical ranges are 0.5 Å to 5 Å for the translation step and 0.1 rad to 0.5 rad for the rotation step.

After the mutation step, the potential energy of the replica is calculated, and used to evaluate the mutation. If the mutation is rejected, the mutated molecule is rolled back to its previous state.

Depending on the program's compile-time flags, the potential calculation may be performed on the CPU or on the GPU. If the GPU version is enabled, the MC steps of parallel replicas may be executed synchronously or asynchronously.

In the synchronous version, within each Monte Carlo thread the outer loop iterates over all replicas assigned to the thread, and performs all parts of the Monte Carlo step for each replica sequentially before switching to the next replica. The longest part of each step is the potential calculation on the GPU. The GPU calls are blocking, which means that the thread must wait for each replica's calculation to be completed before beginning the calculation of the next replica's MC step. On a multithreaded system with a single GPU, threads may also block each other, since only one thread is able to access the GPU at any time.

The asynchronous version addresses this inefficiency, allowing the CPU and GPU to be utilised concurrently. Each Monte Carlo step is split into three parts: the selection and application of the mutation, the calculation of the potential, and the evaluation and possible rolling back of the mutation. Of these, only the potential calculation is performed on the GPU.

In the asynchronous Monte Carlo algorithm, the inner and outer loops in each thread are swapped: the outer loop iterates over the three parts of the MC step, and for each part an inner loop iterates over all of the thread's replicas. The calculation of the potential on the GPU is executed as an asynchronous call. Thus the CPU can begin to compute the beginning of a replica's MC step while the previous replica's potential is still being calculated on the GPU.

To prevent threads from blocking each other when they access the GPU, the asynchronous version utilises CUDA streams, a feature which allows parallel queues of operations to be executed on the GPU concurrently. Operations are launched on the GPU with asynchronous calls which specify the stream to be used. These calls are non-blocking as long as the stream is free. if it is occupied, the call will be blocking until all the previous instructions in that stream have been executed.

The CUDA architecture allows a maximum of 16 streams to be used on one GPU. CGPPD can thus use any number of streams up to this maximum (per available GPU). The total number of streams is read from the configuration, and replicas are assigned to streams dynamically when the simulation is initialised. It is recommended for the total number of replicas to be an exact multiple of the total number of streams, so that the work is distributed evenly.

3.2.6 Potential calculation on the CPU

The total interaction potential used to evaluate Monte Carlo moves and replica exchanges is the sum of the short-range and long-range non-bonded forces between all pairs of residues in the simulation. Pairs of residues which lie within the same molecule are excluded, because their contribution to the total in CGPPD v1 is constant.

The CPU implementation of the potential calculation consists of four nested loops. The two outer loops iterate over pairs of distinct molecules, and the two inner loops iterate over pairs of residues. Inside the innermost loop, the potential for each pair of residues is added to the potential total.

Aggregation of residues into molecules allows this algorithm to discard unnecessary residue pairs in the outer loop: a molecule is never paired with itself, so residues within the same molecule are never compared.

3.2.7 Potential calculation on the GPU

In order to calculate the interaction potential, the GPU implementation must copy the residue data from the host to the device, perform the calculation in a CUDA kernel, and transfer the result back from the device to the host.

The design of the implementation was influenced by various limitations of the GPU architecture. The most important concerns of GPU algorithm design are: maximising the parallel execution of the code so that all of the GPU cores can be used effectively; minimising the memory transfer through the low-bandwidth channel between the host (CPU) and the device (GPU) to avoid bottlenecks; and maximising the instruction throughput by optimising instruction use.

The data structures implemented in the CPU code cannot be used unmodified on the GPU because not all C++ language features are accessible from CUDA. Also, because both memory and bandwidth are a much more limited resource on the GPU, it is important to optimise the size of the data to be transferred to and stored on the device as much as possible. The simulation data is therefore converted to a minimalist representation which contains only the information necessary for the potential calculation to be performed on the GPU.

The GPU eliminates the molecule abstraction, and treats the residues as a single flat list. This allows the four loops of the CPU implementation to be refactored into two, which in turn results in an easy mapping of residue pairs to GPU cores when the loops are replaced by a parallel calculation distributed over GPU threads. Because it is still necessary to eliminate residue pairs within the same molecule from the calculation, a molecule identifier is added as a property to each residue.

Each residue is represented by eight floats. This data is contained in two arrays of type $float4$, a built-in CUDA vector type which aggregates four floats in a struct. This type was chosen over a custom structure because it is an efficient implementation which makes optimal use of memory alignment for parallel memory fetches. Use of this type means that some residue properties which are integer values must also be represented as floats, but IEEE 754 compliance ensures that these values are not affected by the conversion.

Three of the floats are used for the position, and one each for the amino acid type, the charge, the radius and the molecule identifier. The eight float is a flag which indicates that the residue is part of a crowder molecule: this information is used to calculate a different potential between two crowder residues, if macromolecular crowding is enabled in the simulation.

One of the optimisations introduced to the GPU code is the use of dummy residues to pad the arrays to a size exactly divisible by the total number of available threads. This removes the need for special handling of blocks of threads which are not full, and also guarantees that the potential results can be summed with a parallel reduction algorithm which requires a power of two array size. The molecule identifier property is reused as a flag to mark dummy residues as padders. Any pair which contains at least one of these padders contributes a zero value to the calculation.

The calculation of the Lennard-Jones-type potential for each residue pair depends on e_{ij}, the contact potential between the residues, which is determined by their amino acid type. Because the residue types have no predictable order, the lookup of this value is a random memory access. Unlike memory accesses which follow a predictable pattern, which can be optimised so that multiple threads can perform them in parallel, this lookup is likely to be completely serial.

The type of memory used to store this data can have a strong impact on the performance of the lookup, and all the available options were evaluated. Ultimately texture memory was selected as the best-performing option. It is less sensitive to random memory access patterns, and caches data stored near data which has already been retrieved. The type of memory used can be changed at compile time with a preprocessor directive.

The contact potential values are stored in a 1D array of floats, so that $e_{ij} = A[i + 20 * j]$. The data is duplicated in this structure as it is on the CPU, so that the two residues can be looked up in either order.

The residue arrays for each replica are initialised once at the start of the simulation. They are copied to the GPU in their entirety before the potential calculation is performed on the GPU for the first time. Thereafter, residue data is only updated when this is necessary. After each Monte Carlo move, only the positions of residues associated with the molecule which has been mutated are updated. If a Monte Carlo move is subsequently rejected, these positions are updated again so that they are restored to their previous state. The array of contact potentials is constant, and thus only has to be copied to the GPU once.

Each thread executing the potential kernel calculates the potential for several residue pairs: the same x residue is paired with a block of y residues. This is a more efficient distribution of the total work than assignment of a single residue pair to each thread. This tiled approach was adapted from GPU algorithms for molecular dynamics described by Friedrichs et al. [33] and van Meel et al. [55].

The kernel uses a two-dimensional grid of one-dimensional thread blocks. The block size $blockDim$ – the number of threads in each block – is determined at runtime to be 32 if the simulation contains fewer than 1024 residues, and 64 if it is larger. These defaults were selected to provide optimal performance, but the value may also be overridden through a configuration setting if a larger block size is required to cater for a larger number of residues. The size of the

grid, a $gridDim \times gridDim$ square, is calculated as follows: $gridDim = paddedSize/blockDim$, where $paddedSize$ is the total number of residues padded to the nearest power of 2.

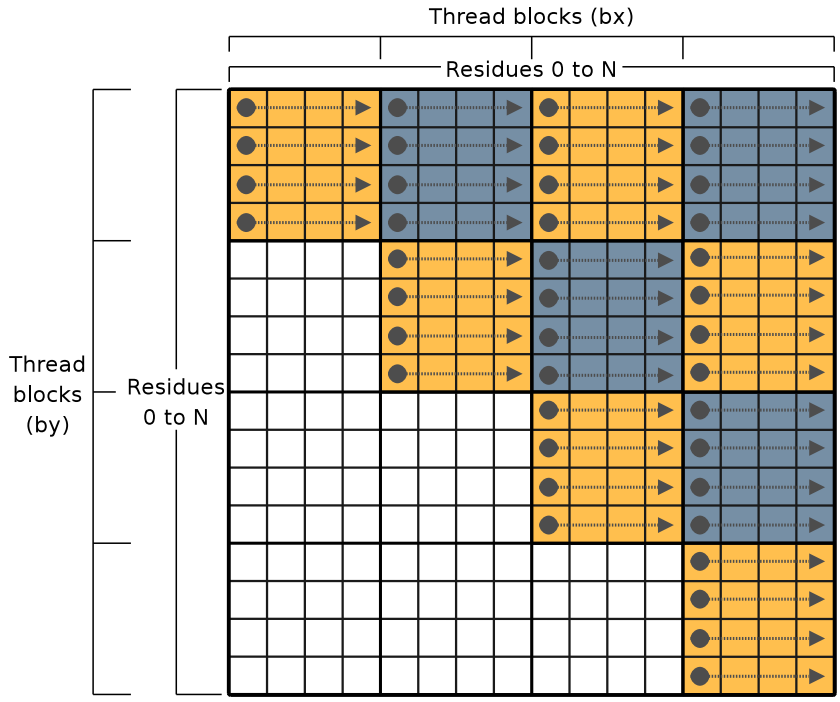


Figure 3.1: Potential kernel grid layout *Each thread calculates the potentials between one x residue and all the other residues within the same block. Blocks below the diagonal perform no calculations, and blocks which lie on the diagonal halve their totals [5].*

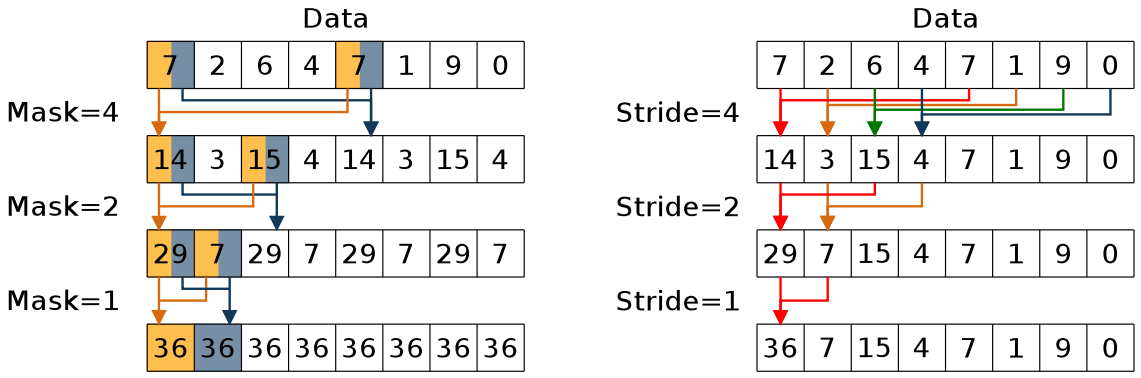
The mapping of threads to residue pairs is illustrated in Figure 3.1. Within each thread, the x residue is $R[bx \times blockDim + tx]$, where bx is the x dimension of the block index, tx is the thread index and $blockDim$ is the thread block size. The y residues range from $R[by \times blockDim]$ to $R[by \times blockDim + blockDim]$, where by is the y dimension of the block index. At the beginning of the kernel's execution, data for the x and y residues used by the thread block is copied into its shared memory. This improves the performance of the kernel, since accesses to shared memory have a much lower latency than accesses to global memory. Each thread in the block accesses the same y residue data in shared memory once it has been copied. Bank conflicts are avoided because all threads access the same residue data in lockstep.

Because each pair of residues is mirrored diagonally in the matrix which represents the residue interactions, each pair potential would be calculated twice if the calculation were to be performed by all thread block. To avoid most of this duplicate work, the kernel skips the calculation in blocks below the major diagonal and halves the potential total in blocks which lie on the diagonal.

Each thread writes its potential subtotal to an array of results stored in shared memory. Within each block these results are summed with a parallel reduction. It would be inefficient for all the threads in a block to accumulate the potential total in a single variable, because the locking techniques which would have to be used to prevent threads from overwriting each other's modifications would cause the operation to become sequential. A parallel reduction

allows threads to accumulate subtotals in different variables concurrently, without conflicts.

CGPPD implements a variant of a simple parallel sum reduction algorithm which can only be used on an array which is a power of two in size. At each step, each thread reads two values from the array, sums them, and writes the result back to the array. The two variants differ in the implementation details; they are compared in Figure 3.2, which illustrates how each algorithm would be used to reduce an array of eight elements.



(a) Parallel reduction with mask

(b) CUDA reduction

Figure 3.2: Comparison of parallel reduction algorithms *Both variants of the algorithm require only three iterations to reduce an array of eight elements. The CUDA reduction (b) is more work-efficient, as fewer threads are used in each iteration and there is no duplication of work [5].*

In the original parallel reduction algorithm, given an array M of length N, the thread with index x sums elements $M[x]$ and $M[x \oplus mask]$, where $mask = N/2$, and the result is written back to $M[x]$. After each step, $mask$ is halved. The array is thus divided into pairs which are $mask$ elements apart, and each distinct sum is performed by two threads. Thus all threads are used in every iteration, but most of them perform duplicate work: after the final iteration each element of the array contains a copy of the total. The algorithm performs the reduction in $O(\log_2 n)$ time, but with $O(n \log_2 n)$ work: because the duplicate work occurs in parallel, it does not affect the running time. The sequential implementation is $O(n)$.

The modified CUDA reduction algorithm uses a stride of $N/2$ instead of a bitmask: thread x sums elements $M[x]$ and $M[x + stride]$. Only $N/2$ threads are used in the first iteration, and the number is halved in each subsequent iteration. Thus each sum is only performed once, and after the final iteration only the first element of the array contains the result. This allows the algorithm to perform $O(\log_2 n)$ work. The elimination of superfluous threads allows the reduction to be optimised in various ways on the GPU hardware: bank conflicts are reduced because each array value is only read by one thread at a time; fewer warps are in use in each iteration; and once only threads within a single warp are active no further thread synchronisation is required because warp operations are synchronous.

The final total is calculated from the partial sums produced by each block either on the device or on the host. Depending on which method is used, either a single value or an array of values is copied back from the device to the host.

3.2.8 Output

CGPPD outputs samples at regular, configurable intervals during the course of the simulation. Each sample records the positions of the molecules, as well as information about the Monte Carlo acceptance and rejection statistics and the fraction of bound conformations.

Because the molecules are rigid, it is necessary only to output a cumulative translation and rotation value for each one. If these are applied to the PDB file which contains the starting positions of the molecules, a PDB file of the sampled conformation can be reconstructed. This is done in a post-processing step after the simulation has completed. These PDB files can subsequently be used as input to an external clustering program.

Chapter 4

Design and implementation

The previous chapter summarises the original implementation of CGPPD, which focused on the acceleration of coarse-grained Monte Carlo protein-protein docking simulations on GPU hardware. The focus of our project was to extend CGPPD v1, in which the proteins could only be modelled as entirely rigid bodies, to make it possible to to model a protein as a series of rigid domains connected by flexible linkers. We aimed to make this addition without compromising the performance gains of the original implementation.

The addition of flexible protein segments required modifications throughout the CGPPD code base, with the most extensive changes required in two stages of the Monte Carlo algorithm: the selection of Monte Carlo mutations, and the calculation of the potential energy of the system in order to evaluate them.

New Monte Carlo mutations which allow the proteins to change shape were implemented: two local mutations which affect individual residues within flexible linkers, and a mutation which allows a protein to bend. The potential calculation was extended to include bonded energy components. It was also necessary to include more residue pairs in the nonbonded part of the calculation: because the proteins were no longer rigid, it could not be assumed that each protein's internal nonbonded potential would remain constant throughout the simulation.

The changes to the potential calculation had to be made both to the CPU and GPU implementations. Since GPU optimisation was not the focus of this project, we aimed to make as few changes to the GPU implementation as possible, avoiding modifications which were likely to have a major impact on performance.

Assorted other minor changes were required: the input format was extended to include the means to mark certain portions of a protein as flexible, and a new data structure was added to the model to store this information. Because the proteins could change shape during the simulation, it was also no longer sufficient to output a cumulative translation and rotation value for each protein in each sample of the simulation. Instead, each sample now includes a full PDB file containing a snapshot of all proteins in the simulation.

4.1 Input

A new section has been added to the configuration file format to allow the user to mark which portions of a molecule are flexible. The heading *segments* denotes the start of the segment section, in the same way that the headings *files* and *crowders* are used to delimit the non-crowder and crowder molecule sections, respectively.

Within the segment section, each entry takes the form *moleculenamesegment*, where *segment* can either be a space-separated list of residue indices within the molecule or the word *all*, which indicates that the entire molecule is flexible. The indices correspond to the numbering inside the PDB file (which starts from 1), not CGPPD's internal numbering (which starts from 0). *moleculename* must correspond to a valid molecule name used to label an entry in the *files* section. This optional label was added to the file format in CGPPD v2, and allows segments to refer unambiguously to specific molecules in a way which is not dependent on the ordering of the molecules within the file.

```
(...)

files

t(0,0,0) r(0,0,0,0) data/diubiquitin_lys_48.pdb diubiquitin

segments

diubiquitin 73 74 75 76 124
```

Figure 4.1: An example configuration file fragment The segment provided represents the flexible linker between the two ubiquitin molecules in Lys-48-linked diubiquitin. The tail of the first ubiquitin molecule is connected to the lysine residue in position 48 of the second ubiquitin molecule. Note that this complex is modelled as a single molecule with two chains: a single PDB file containing both structures is used, and the index of the lysine in the second chain is offset by the length of the first chain.

A segment can be made up of an arbitrary list of residues: they do not have to be adjacent within the PDB file. In particular, a segment can span multiple chains within the same protein, and the end of one chain can be attached to the middle of a second chain, thus forming a branching structure. An example configuration file fragment is shown in Figure 4.1.

4.2 Model

4.2.1 Requirements and design

If each flexible linker were confined to a single chain, it would be possible to describe the linkers with a simple linear structure, such as an array of Link objects corresponding to the array of Residue objects: each Link could have a property which determined whether the link between

two residues was flexible. However, this simple solution would not allow us to model compounds such as diubiquitin, which has a flexible linker connecting the end of one chain to the middle of another and thus includes a branch.

We therefore opted to store topological information about each molecule in an adjacency list. This approach is sufficiently generic and powerful that it allows us to model arbitrary interconnections between residues, although in practice we expect multi-chain proteins to be mostly linear.

Information about the flexible linkers is required at two points within the simulation: during the Monte Carlo mutation step, when certain mutations may only be applied to a subset of residues within flexible linkers; and during the potential calculation step, when the bonded potential contribution of each flexible linker is calculated. In both cases we require the information to be presented as an index of specific topological features: during the mutation step, a list of all residues available for a particular move, from which one may be randomly selected; and during the potential calculation step the set of all flexible pseudo-bonds, pseudo-angles and pseudo-torsions.

These indices are derived from the data stored in the adjacency map. Because the Monte Carlo transformations applied to the molecules during the course of the simulation can change residue coordinates, but not break or form links between residues, the adjacency map never changes. We can therefore calculate all the secondary information about the molecule's topology once at the start of the simulation, and thereafter use cached data instead of repeating the calculations unnecessarily during each Monte Carlo step.

4.2.2 Implementation

We encapsulate all of this topological information inside a new Graph object, which is added as a property to each Molecule object. Because the topological structure of each molecule remains constant, although each Replica must have an independent copy of every Molecule object, all replica copies of the same Molecule can share a single Graph object. This is illustrated in Figure 4.2.

For ease of implementation, within the Graph class we made extensive use of containers provided by the C++ Standard Template Library (STL), a popular library of generic algorithms and data structures [66]. Throughout this chapter, unless otherwise specified, data structures referred to as *vectors*, *sets*, *maps* and *pairs* can be assumed to be STL implementations.

In addition to the Graph class itself, we implemented a range of simple containers to store information about individual bonds, angles, torsions and arbitrary pairs of residues. These containers aggregate indices which refer to residues in the Residue array: they do not directly store any information which is dependent on residue positions, which allows the same Graph and its contents to be reused in multiple replicas. The containers also implement comparison operators and assorted helper methods.

The Graph object is initialised from a vector of Residues and a vector of segment containers, where each segment contains a vector of Residue indices demarcating a flexible segment. A flag may also be used to indicate that the entire molecule is flexible. This information is used to

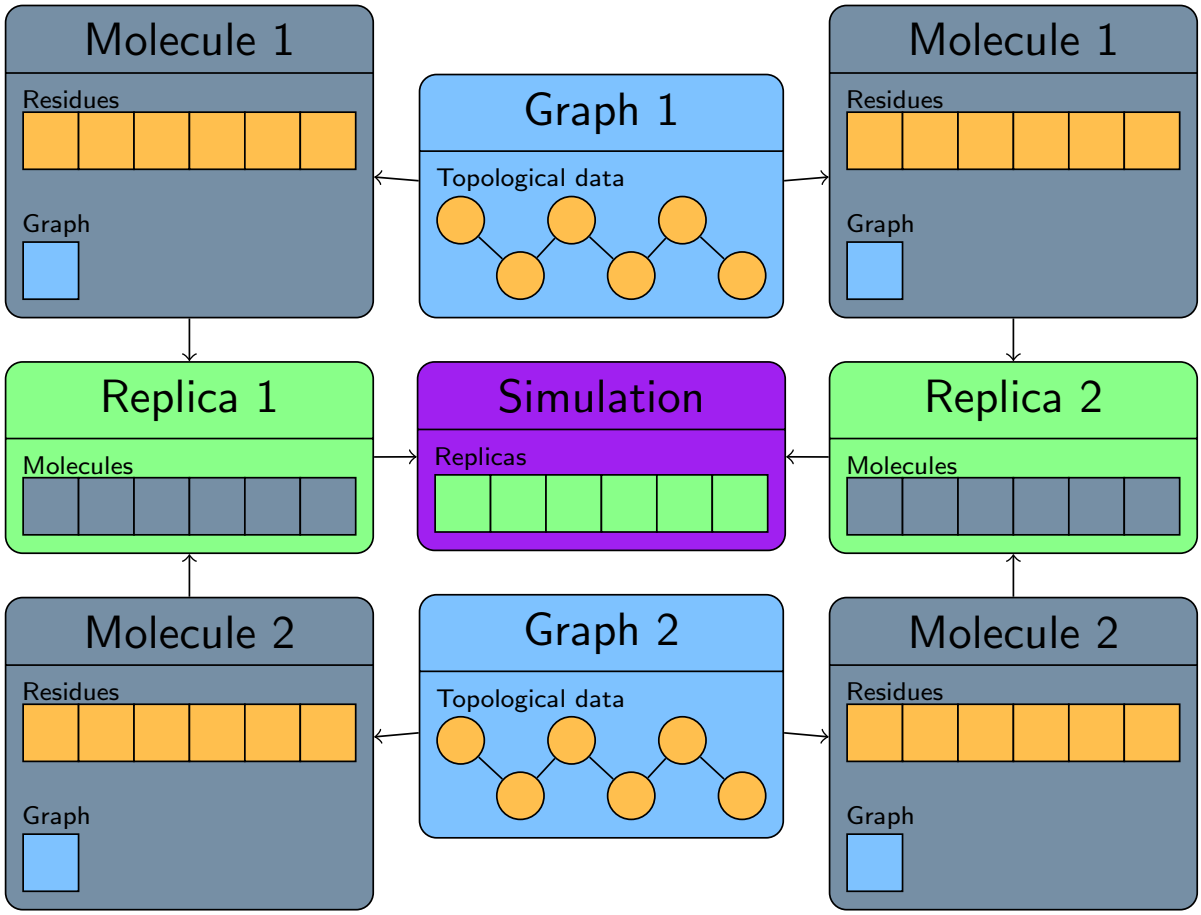


Figure 4.2: Relationship between Molecule and Graph objects *Every replica stores an independent copy of each molecule in the simulation, but all copies of a molecule share the same Graph object: the topological information that it aggregates is independent of the changing positions of the residues, and remains constant during the simulation.*

populate four private data structures which describe the graph. The vertices are stored in a set of integers, where each vertex is a Residue index. The edges are a set of pairs of integers: each edge is stored only once, as the pair (i, j) where $i < j$. The adjacency list is implemented as a map of integers to sets of integers: for simplicity of lookups, the data is duplicated, so that for each edge (i, j) the vertex i is in the set of vertices adjacent to j, and vice versa. A map of integer pairs to booleans indicates whether each edge is flexible: this map also stores each edge twice.

A set of Bonds is constructed from all the flexible edges in the graph. A set of Angles is constructed from all adjacent pairs of edges where at least one edge is flexible, and similarly a set of Torsions is constructed from all adjacent triplets of edges where at least one edge is flexible.

For each of the added flexible Monte Carlo moves, we construct a set of residues which are available to be selected for that move: although most residues within a flexible linker are common to all three sets, edge cases are handled differently. The differences are described in detail in the Monte Carlo section. These sets of residues are exposed by the object as vectors, because they are selected randomly and thus must be stored in indexable containers.

The graph class also exposes several helper data structures and functions which are used when a flex move is performed and when the potential is calculated. These elements will be described in detail later in the chapter.

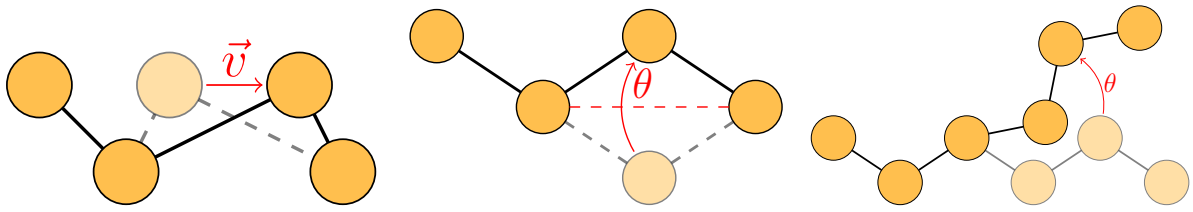
4.3 Monte Carlo

The rigid implementation of CGPPD provided two Monte Carlo mutations: translation and rotation of a single molecule. Both these moves preserve the shape of the molecules. In our implementation we make no modifications to these existing mutations, but add three new mutations which change the shape of the molecule: translation of a single residue, crankshaft rotation of a single residue, and a flex move which allows the molecule to bend.

4.3.1 Requirements and design

The local translation and crankshaft moves were suggested by Kim and Hummer in the paper which describes their model [1]. The local translation move simply translates a single linker residue in a random direction. In a crankshaft move, a single linker residue is rotated about the axis formed by its two neighbours. We implement only a crankshaft move which moves a single residue – our code could easily be extended to crankshaft moves which move multiple sequential residues [67, 68].

These local moves alone would not allow the domains of a single molecule to change position with respect to each other. To make this type of transformation possible, we have also added a move which bends the molecule at a single point: a random linker residue is selected, and all residues on one side of this residue are pivoted about that residue. Similar mutations have previously been implemented in protein folding simulations [67]. The three added moves are illustrated in Figure 4.3.

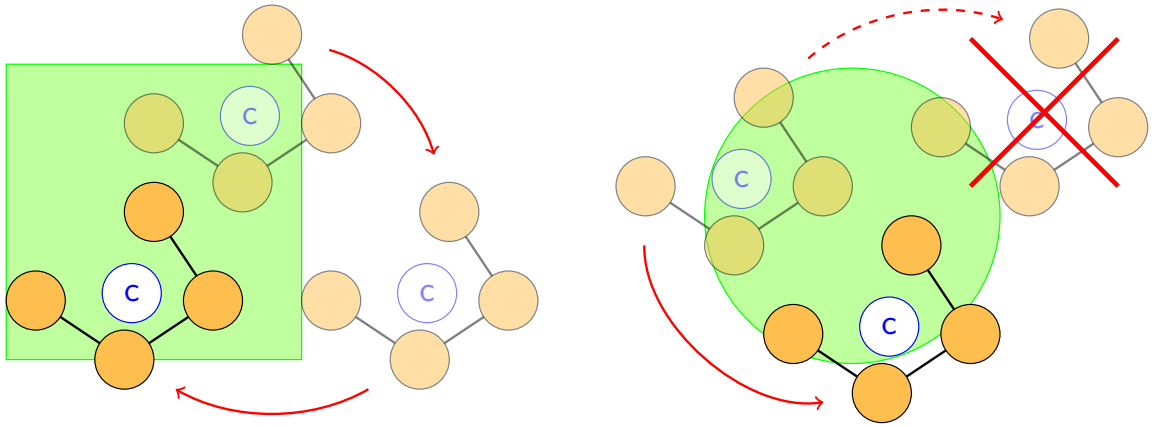


(a) Local translation (b) Crankshaft (c) Flex

Figure 4.3: Additional Monte Carlo moves for flexible linkers *In a local translation move (a), one residue is translated in a random direction. In a crankshaft move (b), one residue is rotated about the axis formed by its two neighbours. In a flex move (c), all residues on one side of a pivot residue are rotated about a random axis passing through that residue.*

We aimed to implement these moves with support for both periodic and spherical boundary conditions, which are illustrated in Figure 4.4. This proved a little more challenging than the simple approach used for the whole-molecule translation and rotation, because the three flexible moves cause the molecule to change shape, and thus require both the centre and the relative

positions of the residues to the centre to be recalculated. We wished to perform these calculations as efficiently as possible.



(a) Periodic boundary (b) Spherical boundary

Figure 4.4: Comparison of boundary conditions *If a periodic boundary (a) is used, whenever a move causes a molecule's geometric centre to move outside the boundary, the molecule is wrapped around to the other side of the bounded area. If a spherical boundary (b) is used, moves which would cause a molecule to move outside the boundary are discarded.*

Because the local translation and crankshaft moves have much less of an effect on the system than the whole-molecule translation and rotation and the flex move, we wished to include an option to aggregate several small local moves in a single Monte Carlo mutation. We also aimed to make the relative probability weightings of all five moves configurable.

4.3.2 Random selection

As in the original implementation, we use the GNU Scientific Library (GSL)'s [64] implementation of the Mersenne Twister [65] to generate all random numbers used in the Monte Carlo algorithm.

In each MC step, the selection of a molecule for a mutation remains unchanged from the rigid implementation. However, while the mutation type was previously a binary choice, there are now four possibilities: a whole-molecule translation, a whole-molecule rotation, a flex move or a series of local moves. To make a random selection, we use GSL's general discrete distribution function, *gsl_ran_discrete*, which can be initialised with an arbitrary table of weights. By default all four mutation types are weighted evenly.

If no molecules in the simulation have flexible linkers, the two flexible options are disabled and only whole-molecule translations and rotations may be selected, as in the rigid-only implementation. If there is only one molecule in the simulation, whole-molecule translations are disabled, since they have no effect on the potential energy of the system, and only whole-molecule rotations (which may change the position of the molecule relative to its neighbours across a periodic boundary) and the two flexible options may be selected.

The number of local moves to be performed in a single MC step is configurable. Each

individual local move is randomly selected to be either a local translation or a crankshaft move. By default both are selected with the same probability, but the bias is configurable.

Each of the three additional moves focuses on a particular residue within a flexible linker. A list of all residues available for each move is exposed as a vector by a molecule's Graph object, and a random element is selected from the appropriate vector when that move is made.

4.3.3 The local translation

This mutation moves a single randomly selected residue in a random direction. The residue must not be adjacent to any rigid edges.

A random translation vector is generated in exactly the same way as in a whole-molecule translation. However, the step size is scaled down substantially, as a step size of the magnitude used for the whole-molecule translation leads to rejection of the move in an overwhelming majority of cases. This is because the move changes the lengths of the bonds on either side of the residue, which has a costly effect on the bond component of the potential.

The scaling factor is configurable. By default it is set to 0.1. We selected this value to make the acceptance ratio of local translation moves have the same order of magnitude as that of crankshaft moves.

To handle the boundary conditions for this move, we need to recalculate the geometric centre of the molecule. We can do this efficiently using only the old centre and the translation vector v:

$$centre_{new} = centre_{old} + \frac{v}{N}$$

where N is the total number of residues in the molecule.

We make this calculation before applying the translation vector. If the boundary is spherical and the new centre lies outside it, we ignore the move. Otherwise, we apply the translation vector to the residue, apply the new value of the centre, and recalculate the relative positions of all the residues to the centre: we can do this by adding a constant offset to the old relative positions. Finally, if the boundary is periodic, we check whether the new centre lies outside it, and if so add a constant offset to the centre and all the absolute residue positions in order to wrap the molecule's position.

4.3.4 The crankshaft move

This mutation rotates a single randomly selected residue about the axis passing through its neighbours, thus preserving the bond lengths on either side. The residue must not be adjacent to any rigid edges, and must be adjacent to at least two flexible edges. We don't eliminate residues which are adjacent to a third flexible edge, a case which would be made possible if a linker at the end of a chain were to bind to the middle of another linker. Crankshaft moves on such a residue are likely to be rejected, however, because of the distortion of the bond along the third edge.

The normalized vector between the two neighbouring residues is used as the axis of rotation. The rotation angle is the same as the one used for the whole-molecule rotation, but we randomly select whether to flip its sign. A quaternion is constructed from the angle and axis, and used to perform the rotation: because the axis passes through both of the residue's neighbours, we can use either neighbour as the centre relative to which the residue is rotated. We thus apply the quaternion to the relative position of the selected residue to its neighbour, and the result is its rotated position relative to that neighbour.

We then calculate the translation vector which is required to move the residue to that position, thus reducing the problem to a local translation move. This allows us to re-use all of the translation move's boundary checking code.

4.3.5 The flex move

This mutation causes the molecule to flex at a single point. We randomly select any residue from the set of all linker residues, thus partitioning the molecule into two or more branches. We randomly select the branch which will move, and select all the residues included in that branch using a helper function on the molecule's Graph object. This function traverses the graph, and is intelligent enough to maintain rigidity constraints on domains by including all residues in a domain if a residue from that domain has already been included, regardless of the numbering of the residues.

We then rotate all the residues in the selected branch relative to the first selected residue using a quaternion which is constructed from a random axis of rotation and the same rotation angle that is used for the whole-molecule rotation.

Checking the boundary conditions for this move is a little more complex because more than one residue changes position: in the worst case scenario, $N - 1$ residues might be rotated. We need to calculate the new positions of all these residues in order to calculate the new geometric centre, which has to be done before the move is applied if the boundary is spherical.

We store a translation vector for each rotated residue in a temporary container. We can use these vectors to precalculate the new centre:

$$centre_{new} = centre_{old} + \frac{1}{N} \sum_{i=1}^{R} v_i$$

where R is the total number of rotated residues and N is the total number of residues. If the move is not rejected because of a spherical boundary condition, we can reuse the saved translation vectors to update the absolute position of each residue. Thereafter we can update the relative positions of all residues, and wrap the molecule if this is necessary because of a periodic boundary condition.

4.4 Potential energy

The rigid implementation of the potential energy comprises only two non-bonded components: a short-range Lennard-Jones potential, and a long-range Debye-Hückel electrostatic potential.

Each of these components is calculated for every pair of residues, excluding pairs which both lie withing the same molecule: because the molecules are rigid, their internal potential never changes. Our implementation adds pseudo-bond, pseudo-angle and pseudo-torsion components, calculated within the flexible linkers. We also extend the non-bonded components to include more pairs.

4.4.1 Requirements and design

The introduction of Monte Carlo mutations which allow the molecules in the simulation to deform required us to add components to the potential sum which measure the fitness of these mutations. Changes to the shape of the backbone place strain on the pseudo-bonds between pairs of adjacent residues, the pseudo-angles formed by adjacent triplets, and the pseudo-torsion angles between adjacent quadruplets. We added components for these three elements, as was suggested by Kim and Hummer [1]. Our torsion component includes only proper dihedrals: we do not include a component for improper dihedrals.

The bonded components are much less expensive to calculate than the bonded components: the cost scales linearly with the number of residues inside flexible linkers, and in a typical use case the flexible linkers make up a small portion of the total number of residues in the simulation. We thus implemented this calculation only on the CPU.

It was also necessary to extend the calculation of non-bonded potentials to more pairs of residues. Because the Monte Carlo and replica exchange criteria only rely on differences between potentials of different conformations, and not the exact value of the total, we can optimise the non-bonded calculation by omitting pairs of residues whose contribution is static. Thus in the rigid implementation we could ignore the contributions from all pairs of residues within the same molecule.

However, if a molecule can change shape, its internal non-bonded potential can also change. We thus had to include in our implementation additional non-bonded potential contributions from all pairs of residues of which at least one was inside a flexible linker. Because our flex move allows rigid domains to move with respect to each other, pairs of residues which lay in different rigid domains also had be included. However, we could ignore pairs of residues which lay within the same rigid domain. It was also necessary to exclude pairs of residues which were close neighbours on the backbone: including them would introduce extremely high non-bonded potential values, as the potential is strongly repulsive at short distances. For the same reason we found it necessary during implementation to exclude residue pairs which were permanently in close proximity to each other because of a bond between two chains.

The GPU implementation of the non-bonded potential calculation also had to be updated. This proved challenging mostly because of the additional residue information which had to be passed to the GPU kernel to allow for the correct residue pairs to be excluded. Because GPU optimisation is not the focus of our research, we aimed to make this change with as little disruption as possible to the existing kernel code. Our implementation packs all the required data into the data structures used by the original implementation, exploiting the fact that under some circumstances two integers can be stored losslessly in a single float. We opted to

add some computational complexity to the kernel in order to unpack the data on the GPU, rather than increasing the bandwidth required for the data transfer by increasing the size of the data structures.

4.4.2 Integration of internal molecule potential with existing code

A new function on the Molecule object is used to calculate the internal potential contribution of each molecule. This always includes the bonded potential, and may include some elements of the non-bonded potential, depending on whether the GPU-enabled version of the code is used.

In the CPU-only version of the code, this function calculates the entire non-bonded potential as well as the bonded potential. This result is added to the total non-bonded potential contribution from pairs of residues in different molecules, which is calculated by a function on the Replica object, as in the rigid implementation.

In the GPU-enabled version, the potential kernel incorporates the majority of the complex conditions which are required to select the correct residues to include in the non-bonded potential calculation, including pairs within the same molecule. We do not, however, pass information to the kernel about residues which are close neighbours because of a bond between two chains, because this would require too complex a change to the GPU code. Instead, we use the molecule potential function to compensate for the inclusion of these pairs in the GPU kernel, in addition to calculating the bonded potential. This is described in greater detail in the GPU implementation subsection below.

4.4.3 Bond potential

This component represents the potential contribution from the pseudo-bond between two residues which are adjacent within a flexible linker. To calculate it, we use a harmonic potential [1],

$$E_{bond}(r) = \frac{1}{2}k(r - r_0)^2$$

where r is the distance between the residues. r_0, the reference distance, is set to the average length of a $C_\alpha - C_\alpha$ pseudobond, 3.81 Å [69]. The spring constant k is set to 378 kcal/(mol Å^2) [1].

To calculate the total bond potential we iterate over the set of all Bond objects, which is exposed by the Graph object. We accumulate only $(r - r_0)^2$ for each pair, and multiply this sum by $\frac{1}{2}k$ to obtain the final total.

4.4.4 Angle potential

This component represents the potential contribution from the pseudo-angle between three adjacent linker residues. We also calculate it for edge cases where one of the edges forming the angle lies outside the linker, because this arrangement nevertheless allows the angle to change shape during a Monte Carlo mutation.

We calculate this component using a double well potential:

$$exp[-\gamma E_{angle}(\theta)] = exp[-\gamma(k_\alpha(\theta - \theta_\alpha)^2 + \epsilon_\alpha)] + exp[-\gamma k_\beta(\theta - \theta_\beta)^2]$$

where θ is the angle between the three residues, $\gamma = 0.1$ mol/kcal, $\epsilon_\alpha = 4.3$ kcal/mol, $\theta_\alpha = 1.60$ rad, $\theta_\beta = 2.27$ rad, $k_\alpha = 106.4$ kcal/(mol rad^2), and $k_\beta = 26.3$ kcal/(mol rad^2) [69].

We iterate over the set of all Angle objects to calculate the variable component for each triplet. We multiply the individual contributions together and take the natural logarithm at the end, which is equivalent to taking the natural logarithm of each component individually and then summing them.

4.4.5 Torsion potential

This component represents the potential contribution from the pseudo-torsion angle formed by four adjacent linker residues. We only consider proper dihedrals. We include cases where one or two of the edges lie outside a flexible linker, because this arrangement also allows the dihedral to change shape.

We calculate the torsion potential as follows:

$$E_{torsion}(\varphi) = \sum_{n=1}^{4}[1 + cos(n\varphi - \delta_n)]V_n$$

where φ is the torsion angle. The constants V_n and δ_n for $n = 1, 2, 3, 4$ depend on the sequence of the middle two residues, and are taken from Karanicolas and Brooks [70].

We iterate over all Torsion objects to calculate the contribution from each group of four residues, and sum them to obtain the final total.

4.4.6 Non-bonded potential on the CPU

We exclude from the non-bonded potential calculation all pairs of residues (r_i, r_j) where r_i and r_j lie within the same rigid domain, because their contribution is static.

To avoid spikes in potential from residue pairs which are in close proximity to one another, we exclude pairs of close neighbours on the backbone, that is (r_i, r_j) where r_i and r_j lie in the same chain and $|i - j| < 4$. For the same reason we exclude pairs (r_i, r_l) where r_i and r_l may be in different chains, but because of a bond between chains are effectively permanently within three residues of one another: that is, where there is some (r_j, r_k) such that there is a bond between r_j and r_k and $|i - j| + |k - l| + 1 < 4$.

When the Graph object is initialised, we assign an integer global unique identifier (UID) to each chain, rigid domain and non-backbone bond. These values are associated with each residue belonging to a particular chain, domain or bond through new properties on the Residue class: this allows us to determine this information directly from a Residue object. Helper functions on the Residue class use these UIDs to determine whether two residues share the same chain, domain or non-backbone bond. We assume that each residue may only be involved in a maximum of one non-backbone bond.

A set of all residue pairs which are close neighbours on either side of a non-backbone bond, excluding the pairs which form the bonds themselves, is provided by the Graph object. We handle the pairs involved directly in a bond separately because of the way we implement this code on the GPU. This will be explained in greater detail in the following subsection.

We iterate over all pairs of residues internal to the molecule, and calculate the non-bonded potential contribution for each pair unless it meets one of the exclusion conditions described above. We sum these contributions to obtain the final total.

4.4.7 Non-bonded potential on the GPU

In the rigid implementation, the GPU kernel which calculates the non-bonded potential iterates over all pairs of residues in the simulation and excludes only pairs which lie within the same molecule. To make this possible, the representation of each residue in the GPU code must include a property which identifies to which molecule the residue belongs. In our implementation it is necessary to replace this condition with the more complex exclusion conditions which we describe in the previous subsection.

To make this possible, we had to include sufficient information in the data associated with each residue to allow for each of the required checks to be performed on the GPU. We set out to do this without increasing the bandwidth required for the transfer of data between the CPU and GPU; both because this bandwidth is costly and because changing the size of the data structures could affect the memory optimisation of the kernel. We thus set out to pack all the required data into the data structures which were already in use. We were able to achieve this for all the data with the exception of information about indirect neighbours, by exploiting the possibility of storing two sufficiently small integers losslessly in a single floating point value.

Each residue is represented on the GPU by two *float4* values, *pos* (position) and *meta*: a *float4* is an optimised native CUDA data structure which aggregates four floating point values. The first three elements of the *pos* value, $pos.x$, $pos.y$ and $poz.z$, are used to store the x, y and z coordinates of the residue position, respectively: these values are updated after each Monte Carlo step. The first three elements of the *meta* value, $meta.x$, $meta.y$ and $meta.z$, are used to store the residue's amino acid type index (which is used in the Lennard-Jones calculation to look up a value in the separately stored pair potential table), electrostatic charge and van der Waal radius, respectively.

This leaves two floating point values, $pos.w$ and $meta.w$, for other uses. In the original implementation, $pos.w$ is set either to a negative constant which indicates that the residue is a dummy value which has been used to pad the residue array and should be ignored, or to the index of the molecule to which the residue belongs, which is used in the check which discards residue pairs within the same molecule. $meta.w$ may be set to a negative constant which indicates that the residue is part of a crowder molecule, for which a simplified form of the potential should be calculated, or it is unused.

We use these two elements to store four integer values: the molecule identifier, as well as the indices of the chain, rigid domain (if any), and non-backbone bond (if any) to which the residue belongs.

```
// ... fetch ypos and ymeta ...
if (ypos.w > PADDER_IDENTIFIER) {
    for (int i = 0; i < blockDim.x; i++) {
        // ... fetch xpos and xmeta ...
        if (ypos.w == xpos.w || xpos.w == PADDER_IDENTIFIER) {
        } else {
            // ... calculate potential for this pair ...
        }
    }
}
```

(a) Rigid kernel

```
// ... fetch ypos and ymeta ...
if (ypos.w > PADDER_IDENTIFIER) {
    for (int i = 0; i < blockDim.x; i++) {
        // ... fetch xpos and xmeta ...
        float xdomain(0.0f);
        float xbond = modff(xpos.w, &xdomain);
        float xmol(0.0f); // not used in this kernel
        float xchain = modff(xmeta.w, &xmol);
        // ... repeat for ydomain, ybond, ymol and ychain ...
        float xresid = bx * blockDim.x + i;
        float yresid = by * blockDim.x + tx;
        if (xpos.w == PADDER_IDENTIFIER
        || (xdomain && xdomain == ydomain)
        || (xbond && xbond == ybond)
        || (xchain == ychain && fabs(xresid - yresid) < 4)) {
        } else {
            // ... calculate potential for this pair ...
        }
    }
}
```

(b) Flexible kernel

Figure 4.5: Comparison of rigid and flexible potential calculation on the GPU These code fragments show the difference between the rigid and flexible implementations of the potential calculation kernel. In the flexible implementation a more complex series of checks is required to exclude pairs of residues from the calculation.

The molecule identifier is not directly used in the kernel in which we calculate the potential for the Monte Carlo evaluation step, because the chain, domain and bond indices are derived from UIDs which are unique to the entire simulation. Thus, for example, if two residues have the same chain identifier they are implicitly also in the same molecule. However, to determine whether a sampled conformation is in a bound state, we use a different kernel, which excludes crowder molecules and all potential components other than the non-bonded potential between different molecules. This kernel makes use of the molecule identifier.

Additionally, the index used to retrieve the *pos* and *meta* values for each residue serves as a global index for the residue within the simulation. We need this value in order to calculate the distance between residues within the same chain. Because we only need the difference between the values, their offset is unimportant, and this is equivalent to using the relative indices of the residues within the molecule or chain.

We use *meta.w* to store the molecule identifier (in the integer part) and the chain UID (in the fractional part), and *pos.w* to store the domain and bond UIDs in the same way. For each pair of integers (a, b) to be packed into a single float, where a is to be stored in the integer part, b is to be stored in the fractional part, and b is in the range $\lfloor 1, B \rfloor$, we set the floating point value to $a + \frac{b}{2^{\lceil log_2(B+1) \rceil}}$. This is effectively a right shift of the significant digits of b, and we use our knowledge of the maximum range of b to ensure that we shift the digits far enough to produce a value which is smaller than zero for all values of b.

We number chains, bonds and domains starting from 1. If a residue is not a member of a rigid domain, the integer part of *pos.w* is set to zero. If a residue is not involved in a non-backbone bond, the fractional part of *pos.w* is set to zero. A residue is always a member of a chain, so the fractional part of *meta.w* is always non-zero. Molecule identifiers start from zero.

We have retained the original implementation's reuse of *pos.w* and *meta.w* for the padder constant and crowder constant: as in the original implementation, we can unambiguously distinguish between the different uses because the constants are negative and all possible index values are positive or zero.

We unpack the integer and fractional parts using the *modff* function from the CUDA library [71]. Code fragments illustrating the change between the two implementations are shown in Figure 4.5. In the non-crowder non-bonded potential kernel, since we require only the integer part of *meta.w*, we use the *truncf* function [71] to truncate the value to its integer part.

The conversion from integer to float on the CPU is IEEE 754 compliant, and *modff* introduces no rounding errors [35]. We are thus guaranteed that the original values a and b will be preserved, provided that the total number of significant digits required to represent both a and b does not exceed the maximum number that can be represented by a 32-bit floating point number, which is 23. We expect to fall well within that limit, even for simulations of very large proteins. We do not need to recover the original values of the chain UID and the bond UID from their respective shifted values, because we only need to compare whether two residues have the same non-zero value. *truncf* also introduces no rounding errors [35].

Because these values remain unchanged for the entire simulation, and because they include all the information required by both kernels, they only have to be calculated and transferred to the GPU once. However, we unpack the values during every Monte Carlo step, which adds four *modff* operations in total to each pairwise calculation. Additionally, during each sampling step we add two *truncf* operations per pairwise calculation. It may therefore be worth investigating whether a solution which increases the size of the residue data structures, perhaps by including an additional *float2* vector for each residue, would be more efficient. This would require more extensive modification of the original kernel code.

If the simulation contains no crowder molecules and no molecules with flexible linkers, the

potential used to determine the bound state is identical to the potential used in the Monte Carlo step, and we reuse this potential value instead of recalculating it.

It would be far more complicated to include information about pairs of residues which are close neighbours because of a bond between chains. If we allow bonds between chains to form an arbitrarily complicated graph, we would potentially need a full $N \times N$ matrix of boolean values to flag all possible such pairs of residues. Rather than finding a complicated way of avoiding the inclusion of this potential contribution on the GPU, we have opted to correct for it by calculating it on the CPU and subtracting it from the final total. The number of pairs affected is expected to be very small compared to the total number of residue pairs, so this does not add a lot of computational cost.

4.5 Output

Because the original implementation permitted only rigid Monte Carlo mutations, the state of the simulation at any moment could be described fully with a cumulative translation and rotation value for each molecule. This information was written out for each sample, and full PDB files for the samples were reconstructed in a post-processing step through the application of the recorded transformations to the original PDB file.

This approach does not suffice for our implementation, because the flexible Monta Carlo moves cause the molecule to deform. We must therefore write out a full PDB file for each sample during the simulation. No other modifications to the output format were required; although we have made some cosmetic changes, such as aggregating files into subdirectories to make the large number of generated PDB files easier to organise.

Chapter 5

Verification, validation and benchmarking

In the course of adding functionality to the original implementation, we had the opportunity to refactor the existing codebase. Some of our modifications changed the behaviour of the existing non-bonded potential calculation, and we re-tested it against the reference implementation. We validated the correctness of our flexible linker model with two sets of homopolymer chain simulations. We also investigated the impact of the addition of flexible linkers on the speed of our application by comparing the running times of five similar docking simulations with varying proportions of linkers.

5.1 Unit tests

The original implementation of CGPPD did not include unit tests. We added testing selectively to areas of the code which we modified, to ensure that our changes did not break existing functionality and to verify their correctness. Although test coverage is incomplete, our partial testing assisted us in making the code more robust. Most of our tests are low-level unit tests of individual code components; however, we also included a high-level comparison of the output of different builds of the application. This allowed us to verify that the synchronous and asynchronous GPU builds produce identical results, and that the CPU build is as consistent with them as possible. The output of the CPU and GPU builds eventually diverges because differences in precision between the host and device calculations inevitably lead to a different outcome of the Monte Carlo mutation acceptance test.

Adding comprehensive unit test coverage to the entire existing codebase was outside the scope of this research project; however, we hope to pursue this in the future in order to improve the code and make it easier to modify and maintain.

Conf.	U_{total} (kcal/mol)	U_{LJ} (kcal/mol)	U_{DH} (kcal/mol)
1	-0.294	-0.081	-0.213
2	-1.056	-1.323	0.266
3	-10.278	-9.095	-1.184
4	-7.584	-5.905	-1.680
5	-7.91×10^{-5}	-2.12×10^{-5}	-5.8×10^{-5}
6	-5.565	-4.812	-0.753
7	-5.453	-4.184	-1.269
8	-10.670	-9.223	-1.447
9	-9.904	-7.952	-1.952
10	-8.518	-7.448	-1.070

Table 5.1: Reference conformation energies *Conformation energies for the ten reference UBQ/UIM conformations produced by CHARMM. U_{total} is the total non-bonded potential energy. U_{LJ} and U_{DH} are the short-range Lennard-Jones and electrostatic Debye-Hückel potential energy components, respectively. Reproduced from Tunbridge et al. [5]*

5.2 Verifying correctness of existing CGPPD functionality

To ensure that our modifications to the potential calculation code did not introduce errors to the existing functionality of the application, we recalculated the potential of several rigid reference conformations which were used to test the correctness of CGPPD v1 [5]. Because of several bug fixes in our version of the application, our results are not identical to the original results. The relative error, $\eta = \frac{|x-x_{sim}|}{|x|}$, is larger. However, we believe that it is still within an acceptable margin. The discrepancy may be due to differences in precision between the constants used in our code and in CHARMM.

Tables 5.1 and 5.2 list the energy values produced by the reference CHARMM implementation and CGPPD v2, respectively. Table 5.3 shows the average and maximum relative errors between the two sets of values.

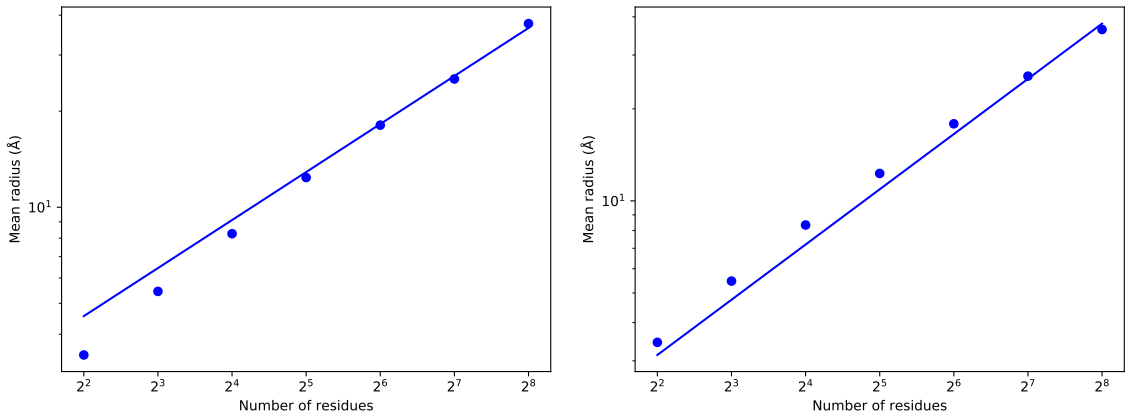
Conf.	U_{total} (kcal/mol)	U_{LJ} (kcal/mol)	U_{DH} (kcal/mol)
1	-0.292	-0.080	-0.212
2	-1.039	-1.304	0.265
3	-10.150	-8.972	-1.178
4	-7.491	-5.820	-1.672
5	-7.85×10^{-5}	-2.09×10^{-5}	-5.77×10^{-5}
6	-5.492	-4.743	-0.749
7	-5.380	-4.117	-1.263
8	-10.528	-9.087	-1.441
9	-9.777	-7.835	-1.943
10	-8.407	-7.342	-1.065

Table 5.2: Implementation conformation energies *Conformation energies for ten reference UBQ/UIM conformations produced by CGPPD v2*

	Mean relative error $\bar{\eta}$	Maximum relative error η_{max}
U_{total}	0.01209	0.01617
U_{LJ}	0.01451	0.01585
U_{DH}	0.00463	0.00488

Table 5.3: Relative errors between reference implementation and CGPPD v2 *The minimum and maximum relative errors between the individual potential energy components and the total potential energy calculated using the reference CHARMM implementation and CGPPD v2*

5.3 Validating the flexible linker model



(a) No LJ potentials (b) Repulsive LJ potentials

Figure 5.1: Mean radius of gyration of polyalanine chains of increasing length *Subfigure (a) shows mean R_g for the set of simulations with LJ potentials turned off and is fitted to the expected function $N^{\frac{1}{2}}$. (b) shows mean R_g for simulations with repulsive interactions, and is fitted to $N^{\frac{3}{5}}$. The axes are not normalised.*

To validate our flexible linker model, we performed two sets of simulations of completely flexible polyalanine chains of exponentially increasing lengths, and compared the results to the expected folding behaviour of homopolymer chains. In the first set of simulations we allowed the chain to cross itself by omitting the Lennard-Jones potential component from the potential energy calculation, and in the second we made the interactions between beads entirely repulsive by setting the offset e_0 in the Lennard-Jones potential calculation to a small negative number. We expected the mean radius of gyration R_g of the samples in the first set of simulations to scale as $N^{\frac{1}{2}}$, where N is the length of the polymer chain, and for R_g in the second set of simulations to scale as $N^{\frac{3}{5}}$ [72]. As shown in Figure 5.1, our results closely matched the expected behaviour.

5.4 Performance overhead added by flexible linkers

We expected the performance overhead added by the linkers to be small. Although the bonded potential calculation is performed on the CPU, the cost of calculating this component scales linearly with the number of residues, unlike the cost of the non-bonded potential calculation, which scales quadratically. Additionally, in a typical docking simulation we would expect only a small portion of all residues to be designated as flexible.

To investigate the overhead added by different proportions of flexible linkers we performed five simulations on the same hardware and compared their total running times as well as the times taken to perform specific tasks, as recorded by several internal timers. We used the asynchronous GPU build. All simulations were run on a cluster node with two six-core 2.10 GHz E5-2620 Intel Xeon processors and four KeplerK40M GPUs. Each simulation had access to all four GPUs and 10 CPU cores (we used 20 replicas). We performed earlier test runs on configurations with more CPUs and fewer GPUs, but discovered that the simulation time is still GPU-bound, and that a configuration with the maximum number of GPUs is optimal. On a node with 20 CPU cores and two GPUs, running times were approximately doubled.

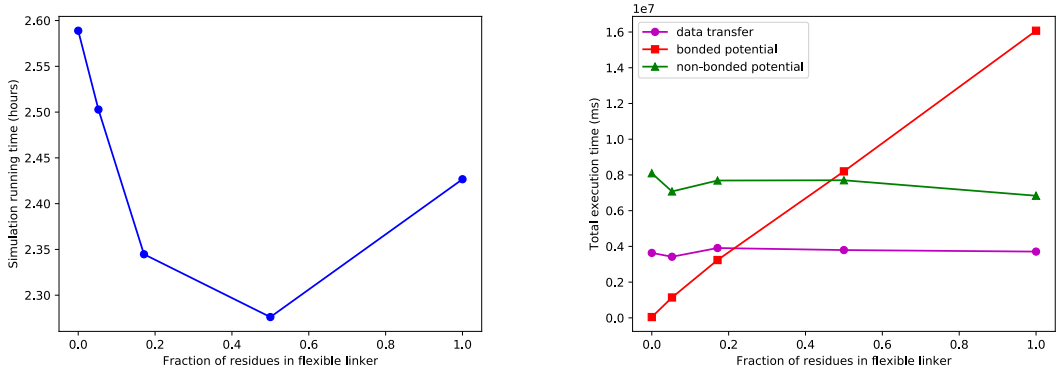
The benchmark simulations modelled the docking of ubiquitin to itself. Ubiquitin is a small protein which consists of a mostly rigid globular domain and a flexible tail. In Chapter 6 we describe our application of CGPPD v2 to the modelling of diubiquitin chains – we aimed to make our benchmarking simulations as similar to these simulations as possible, to replicate the conditions of a typical real use case, but rather than modelling a single ubiquitin dimer we modelled the docking of two separate ubiquitin monomers. This was done to provide a reasonable test case for the rigid simulation.

In the first simulation, we modelled both ubiquitins as rigid bodies. In the next two simulations we made portions of each ubiquitin flexible, using the same two linker configurations as we describe in Chapter 6: in one simulation only the last four residues in each tail were flexible; in the other, the flexible portion of the tail was extended and part of the rigid domain was additionally made flexible. In the next simulation one ubiquitin was made completely flexible and the other left completely rigid, and in the final simulation both ubiquitins were completely flexible. All other simulation parameters were the same as those used for the diubiquitin simulations described in Chapter 6.

	% residues in linkers	% interaction pairs
Rigid	0	50.33
Short tail	5.26	54.10
Long tail	17.11	68.42
Half flexible	50	75.23
All flexible	100	96.13

Table 5.4: Benchmark simulations *More detailed statistics about the selected ubiquitin docking simulations: residues within flexible linkers as a percentage of the total number of residues, and interaction pairs as a percentage of all possible interaction pairs.*

Table 5.4 shows, for each of these benchmark simulations, the percentage of residues which are inside a flexible linker and the percentage of interaction pairs which contribute to the non-bonded potential.



(a) Simulation running time (b) Task breakdown

Figure 5.2: Effect of flexible linkers on simulation running time *Subfigure (a) shows the total running time of each simulation. This is the real elapsed time, or wall time, rather than the total execution time summed over all CPU cores. Subfigure (b) shows a breakdown of the execution time taken by various tasks individually timed within the simulation: copying data into GPU memory, calculating the non-bonded potential on the GPU, and calculating the bonded potential on the CPU. This is the total execution time summed over all 20 replicas.*

Figure 5.2 shows the results of this benchmark. Somewhat surprisingly, adding flexible linkers can cause the simulation to complete more quickly (Figure 5.2(a)), despite the addition of interaction pairs to the non-bonded potential calculation on the GPU and the addition of the bonded potential calculation on the CPU. This trend is only reversed in the final, entirely flexible, simulation. Figure 5.2(b) shows the expected linear progression of the bonded potential execution time, and also indicates that there is no significant change in the time spent performing the non-bonded calculation or transferring data from the host memory to the device. We can conclude that the addition of flexible linkers has a negligible effect on simulation running time.

It is possible that the unexpected speed-up which we observed is a quirk of the specific GPU

architecture that we used for testing, or the size of this test simulation, and is caused by an edge case in the way that different amounts of work are allocated to the GPU. It may be of interest in future work to test varying simulation sizes on a broader range of CPU and GPU hardware. This may clarify the circumstances under which the addition of linkers improves the simulation speed, and would also serve as a valuable update to the extensive benchmarks originally performed on CGPPD v1 [5].

Chapter 6

Application: exploring conformations of diubiquitin

The introduction of flexible segments to our implementation allows us to model a wider variety of protein interactions.

In the prevous implementation, if we wished to model a multiprotein complex consisting of multiple covalently bonded protein monomers, and to permit these component monomers to change orientation with respect to one another, in order to investigate their preferred conformations, we could only have modelled the monomers as individual rigid bodies. Because we had no means of marking and discarding Monte Carlo mutations which broke specific covalent bonds, we would have needed to filter our samples afterwards to select only conformations in which the bonds were preserved.

If multiple covalent linkages between the components were possible, and some were strongly favoured over others by our potential energy function, this would have made it difficult to sample the conformations for all linkages effectively, since conformations for the most favoured linkages would have been overrepresented in the samples.

However, in the new implementation it is possible for us to model such a structure as a single unit comprising multiple rigid domains connected by flexible linkers. This allows us to restrict the simulation to perform only Monte Carlo moves which change the relative positions of the protein monomers within the complex, without breaking the bonds. We can thus fix specific covalent linkages between component proteins in each simulation.

In this chapter we describe our application of this technique to investigate the conformations of several types of diubiquitin chains.

Ubiquitin is a 76-residue protein which is found in almost all living tissue, and has a role in many regulatory functions within the cell. It has a mostly rigid globular domain, with a flexible tail at the C-terminus which is variously described as comprising the last four [73] or last six [74] residues.

Notable surface features of ubiquitin include two hydrophobic patches, one centered on the Ile36 residue and the other on Ile44, which play a role in ubiquitin chain formation. The $\beta 1/\beta 2$ loop spanning residues 6 to 12 is more flexible than the rest of the rigid domain. This flexibility

allows the Leu8 residue at the tip of the loop to contribute to either the Ile36 patch or the Ile44 patch, depending on the position of the loop, and this appears to have an impact on the formation of certain polyubiquitin structures [75]. These features are shown in Figure 6.1(a).

The C-terminus of ubiquitin can form an isopeptide bond with a lysine (Lys) residue of a substrate protein or another ubiquitin. Multiple ubiquitins can thus form polyubiquitin chains, which vary in length and by the types of linkages between consecutive ubiquitin monomers. Eight distinct diubiquitin linkage types have been identified: The C-terminus of one ubiquitin can bond either to one of seven lysine residues or to the N-terminal methionine (Met) residue of another ubiquitin. This last linkage is the form in which ubiquitin is synthesised by the cell.

The attachment of monoubiquitin or polyubiquitin chains to substrate proteins is known as ubiquitylation, and it marks the substrates for targeting by various cellular processes: Lys48-linked polyubiquitin chains are known to act as markers for degradation of the substrate through proteolysis, but other types of chains have different functions, not all of which are well-understood.

The structures of different diubiquitin linkages play an important role in determining their function: whether the chain is compact or open, and the orientation of the hydrophobic patches on the component ubiquitins, determine how the chain can interact with UBDs (ubiquitin-binding domains) on other proteins, and thus what effect it has on the substrate. Exploring the similarities and differences between the structures of these linkages can thus help us to understand the roles that different polyubiquitin chains play in the cell [74, 76].

The structures of different diubiquitin linkages have been investigated experimentally through various techniques. X-ray crystallography has been used to derive structures for all linkages except Lys27. Lys48-linked diubiquitin is the most thoroughly understood, and is known to adopt highly compact and symmetrical conformations. Lys63- and Met1-linked diubiquitin have been found to be more open and elongated. While all linkages have been detected in the cell, the structure and function of the remaining linkages, sometimes described as *non-canonical* [77], is less well-understood, and in particular there are no crystal structures available for Lys27-linked diubiquitin.

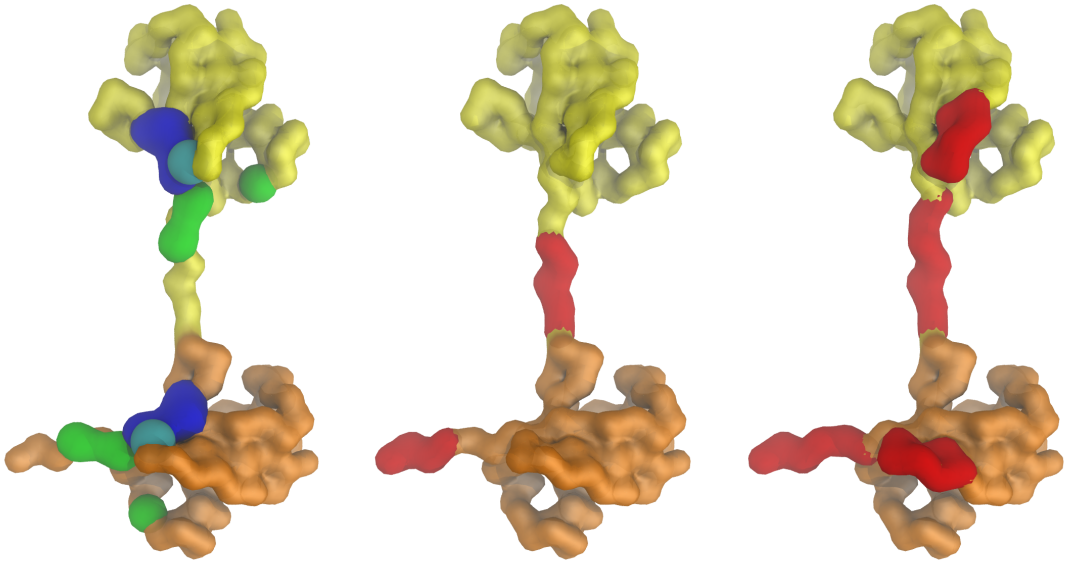
Nuclear Magnetic Resonance (NMR) spectroscopy has also been used to study the structure of diubiquitin. NMR spectra of residues in both diubiquitin domains can be compared to the spectra of corresponding monoubiquitin residues. This produces a map of the chemical shift perturbations (CSPs) of all residues in the protein. CSPs are a measure of the extent to which each residue is involved in interactions with residues within the other ubiquitin domain, and this mapping of the interaction surface can offer insight into the structure of the protein. Castañeda et al. used several experimental and simulation techniques to explore the conformations of all lysine-linked diubiquitin chains. This included the use of NMR spectroscopy to find CSPs for all the linkages, which we reproduce in Figure 6.6(a). They found a high degree of interaction between the domains of Lys48- and Lys6-linked diubiquitin, but little inter-domain interaction in the other five linkages [77].

Förster resonance energy transfer (FRET) efficiency is a measure of the energy transfer between a pair of compatible chromophores (light-sensitive compounds). If these two components are attached to either end of a protein, the observed FRET efficiency measurement can be used

as an estimate of the distance between the ends of the protein, and thus be used to deduce the protein shape. High-FRET conformations are more compact, and low-FRET conformations are more open. For the most elongated conformations no FRET reading can be detected. The proportion of these types of conformations can be estimated through the use of a complementary method such as two-colour coincidence detection (TCCD), which can detect the presence of the individual chromophores. Ye et al. performed this analysis on Lys48-, Lys63-, and Met1-linked diubiquitin. They found that Lys48-linked diubiquitin predominantly forms compact (high-FRET) structures, whereas Lys63- and Met1-linked diubiquitin appears to favour semi-compact (low-FRET) structures, as shown in Figure 6.5(a). They suggested that the result for Met1-linked diubiquitin could be consistent with the recent discovery of a more compact crystal structure for this linkage (*3axc*, shown in Figure 6.2(a) *ii*), and that the more surprising result for Lys63-linked diubiquitin could be explained by multiple compact and semi-compact conformations contributing to the low-FRET population in aggregate [78].

There have been several previous computer simulations of diubiquitin. Cummings et al. used BOXSEARCH to perform a Monte Carlo simulation with simulated annealing, reconstructing Lys48-linked diubiquitin from its component halves as well as from two unbound ubiquitin monomers. They correctly predicted the known crystal structure [79]. Van Dijk et al. used HADDOCK, a docking application which is driven by Ambiguous Interaction Restraints (AIRs) derived from experimental data such as NMR chemical shift perturbations, to model Lys48-linked diubiquitin. They found a solution structure which differs from the crystal structure by a rotation of 20°between the two ubiquitin domains [80]. Fushman and Walker used HADDOCK to model all the linkages of diubiquitin, and found that Lys6-, Lys11-, Lys27- and Lys48-linked diubiquitin tend to form similar closed conformations, while the remaining linkages are unable to do so [81]. Dresselhaus et al. used hybrid quantum mechanical and molecular mechanical (QM/MM) molecular dynamics simulations to compare Lys48-linked diubiquitin with a synthetic linker to the naturally bonded compound, and found them to have a similar structure [82]. Castañeda et al. used Monte Carlo simulations performed using SASSIE to generate structures for all lysine linkages of diubiquitin, producing structures more consistent with their experimental data than the crystal structures, having concluded that the crystal structures are not sufficient for an accurate description of diubiquitin linkages in solution [77].

We performed simulations of diubiquitin chains with each of the eight linkage types. We compared subsets of these results to existing data describing the structure of these chains: crystal structures for diubiquitin chains [74, 77, 78], NMR analysis of the interactions between residues in all lysine-linked diubiquitin types [77], as well as FRET analysis of Lys48-, Lys63- and Met1-linked diubiquitin [78]. We hoped that our simulated models would agree with these previously published results, especially for the well-understood Lys48 diubiquitin linkage, and that we could offer new information about the possible structure of linkages which have been investigated less thoroughly. We found some similarities between our models and the reference data, and we provide our simulated structures for all eight linkages.



(a) Surface features (b) Simulation A (c) Simulation B

Figure 6.1: Surface features of diubiquitin and different flexible linker configurations *All subfigures show our starting conformation of Lys48-linked diubiquitin, with the distal ubiquitin coloured yellow and the proximal ubiquitin coloured orange. (a) shows the surface features of both ubiquitin domains: the Ile36 hydrophobic patch is shown in green and the Ile44 hydrophobic patch is blue. The Leu8 residue is shown in cyan. (b) shows the flexible linkers of simulation A in red: the last four residues of each tail, plus the additional alanine inserted at the binding site between the two domains. As shown in (c), simulation B extends these linkers to two more residues in the tail, and adds a linker for each $\beta 1/\beta 2$ loop.*

6.1 Methods

To distinguish between the two component ubiquitins within an diubiquitin chain, we refer to the ubiquitin which contributes a methionine or lysine side chain to the isopeptide bond as the *proximal* ubiquitin, and to the ubiquitin which contributes its C-terminus to the linkage as *distal*. This is consistent with the terminology used in previous publications.

We programmatically generated a PDB file for the starting conformation of each diubiquitin chain by combining two individual ubiquitin molecules, including only the C_α atoms. We modelled each diubiquitin as a single molecule with two chains: each tail was a flexible linker. The bonds between the ubiquitins would thus be fixed for the duration of the simulation, but the individual ubiquitins would be able to change position with respect to one another. To represent more accurately the space taken up by the side chains of the residues at the binding sites, within the limitations of our coarse-grained model, we inserted an additional alanine residue between the C-terminus of the distal ubiquitin and the binding site on the proximal ubiquitin.

The chains were assembled starting with the proximal ubiquitin. The distal ubiquitin was rotated and translated with respect to the proximal ubiquitin so that the end of its tail was 3.8 Å away from the residue at the binding site, and the tail was approximately perpendicular

to the surface at the binding site. We had to modify this initial approach for the Lys27-linked diubiquitin, where the lysine residue is recessed within the rigid domain, to avoid collisions between the residues in the tail and nearby residues on the surface. We then wrote the diubiquitin chains to the PDB file in reverse order, starting with the distal ubiquitin, to allow for a more intuitive and consistent ordering of the residues.

We prepared two sets of simulations with varying configurations of linkers in each ubiquitin. In the first set, which we will refer to as simulation A, the linker at the tail spanned only the last four residues, in addition to the simulated side chain, and the rest of the ubiquitin was left rigid. In the second set, simulation B, the linker at the tail spanned the last six residues, and additionally the $\beta 1/\beta 2$ loop was modelled as another linker. These configuration options are visualised in Figures 6.1(b) and 6.1(c).

We ran each simulation for 10^7 Monte Carlo steps, sampling every 1000 steps beginning after 10^6 steps, and performing replica exchange every 5000 steps. We used 20 replicas spanning a temperature range from 240K to 420K. We gathered data from the 303.8K replica, which was nearest in temperature to 300K. For each diubiquitin simulation we used a bounding box of 160.7 Å a side, corresponding to a molar concentration of 400μM.

6.1.1 Analysis of results

We used the Visual Molecular Dynamics (VMD) package [83] to cluster the samples from each simulation into similar structures. We aligned the samples according to the distal ubiquitin, and clustered the aligned samples using a clustering plugin [84], selecting all atoms in the molecule and calculating 10 clusters with a distance cutoff of 7 Å. Figure 6.2 compares representative structures from all clusters which constitute at least 10% of the sample population to reference crystal structures labelled with their Protein Data Bank identifiers. We show the most common structures from all simulations in greater detail in Figure 6.4.

Additionally, in Figure 6.3 we show the RMSD distribution between each simulation and each reference structure available for that linkage. In the absence of a crystal structure for Lys27-linked diubiquitin we calculated the RMSD between these structures and *1aar*, the compact structure of Lys48-linked diubiquitin, which we expect them to resemble (Figure 6.3(d)). Because some of the crystal structures have truncated tails, and because they all lack the extra alanine residue we inserted into the backbone to simulate a side chain, we included only residues 1 to 72 of each chain when aligning the structures and calculating the RMSD.

To compare our results for Lys48-, Lys63- and Met1-linked diubiquitin to the FRET data published by Ye et al. we approximated the FRET efficiency of each sample using the formula

$$E = 1/(1 + (\frac{R}{R_0})^6)$$

where R is the distance between the termini of the diubiquitin, and R_0 is the Förster radius. We used a value of 50 Å for R_0, and added a padding value of 20 Å to R to account for the length contribution of the two chromophores and the linkers used to attach them to the diubiquitin. We also calculated the approximate percentages of high-, low- and no-FRET populations in each

simulation, classifying samples according to the criteria described in the original paper: samples with a FRET efficiency below 0.1 are considered no-FRET, those with an efficiency above 0.6 are high-FRET, and efficiency values between these limits are low-FRET [78]. These results are presented in Figure 6.5.

To produce a measure comparable to the chemical shift perturbations found by Castañeda et al., we calculated the average number of contacts between each lysine-linked diubiquitin residue and residues in the opposite domain. We defined this value to be the number of opposite-domain residues within a 8 Å distance cutoff of the selected residue, averaged over all samples in each simulation. We show this comparison in Figure 6.6.

6.2 Results

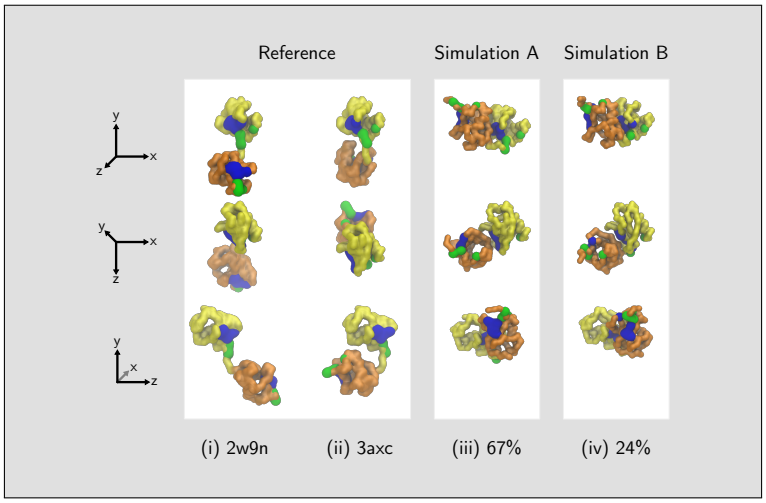
For some linkages, our simulation results are similar to the reference crystal structures, while others bear less resemblance. Our model appears to favour compact and semi-compact conformations over more elongated and open structures.

Of particular interest are both of our Lys48 simulations, where the dominant structures (Figures 6.2(g) *iv, v* and 6.4(g) *i, ii*) were compact and symmetric, with the Ile44-centered hydrophobic patches facing each other in an orientation which resembled the reference structure *1aar* (Figure 6.2(g) *i*). Figure 6.3(g) shows that the distance between simulation A of Lys-linked diubiquitin and *1aar* is the shortest on average out of all simulations and their respective reference structures (disregarding Lys27, which was compared to *1aar* as well, as explained below), with a significant peak in the RMSD distribution near 6 Å.

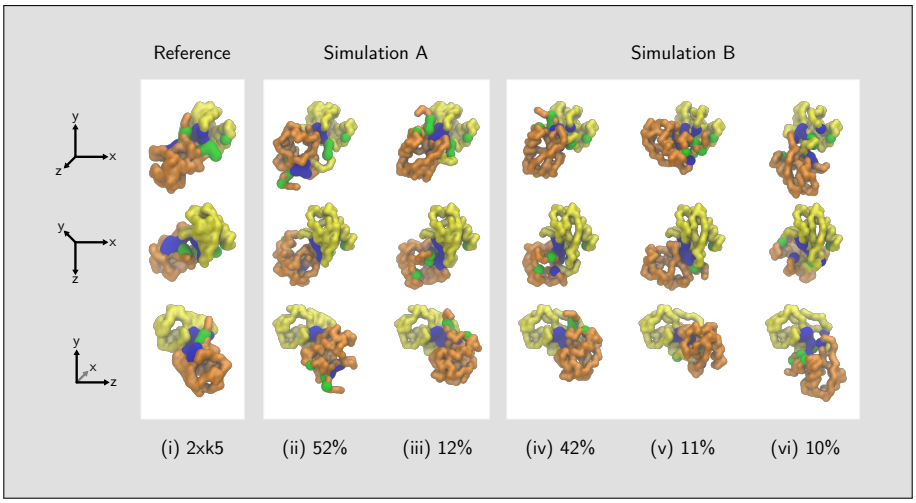
Our Lys6 simulation B produced a dominant structure with a similar orientation (Figures 6.2(b) *iv* and 6.4(b) *ii*), which resembles the reference structure *2xk5* (Figure 6.2(b) *i*), while simulation A favoured a conformation with the hydrophobic patches of the proximal ubiquitin facing outwards (Figures 6.2(b) *ii* and 6.4(b) *i*). In the structure from simulation B, the Leu8 residue is separated from the rest of the Ile44-centered patch in both the distal and the proximal ubiquitin. This is consistent with findings that changes in the position of this residue may play a role in the formation of this linkage [75].

There are currently no reference structures available for Lys27-linked diubiquitin (Figures 6.2(d) and 6.4(d)). Both of our simulations of this linkage produced compact structures similar to Lys48-linked diubiquitin (Figures 6.2(g) and 6.4(g)), which is consistent with the proposed structures described by Castañeda et al. [77]. Figure 6.3(d) shows that simulation A of this linkage has a notable subpopulation of structures which are nearer to *1aar*, the compact crystal structure of the Lys48 linkage, than the results of either of our Lys48 simulations, with a peak near 2.5 Å. Both Lys27 simulations also show peaks near 7.5 Å which suggest large subpopulations of structures nearer to *1aar* than most of the other simulations are to their respective reference structures.

Many of our simulations for different linkages favoured very similar semi-compact conformations in which the distal Ile44 patch is facing the surface of the proximal ubiquitin and the proximal Ile44 patch is on the opposite face of the proximal ubiquitin and facing outwards (as

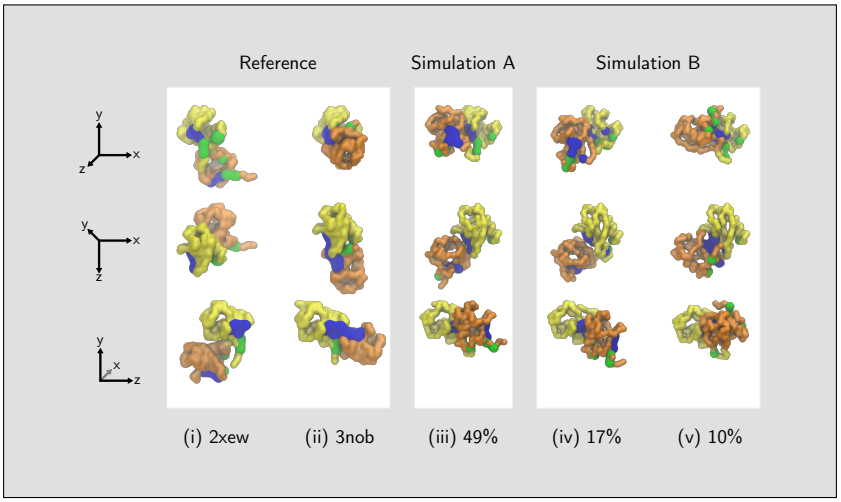


(a) Met1-linked diubiquitin

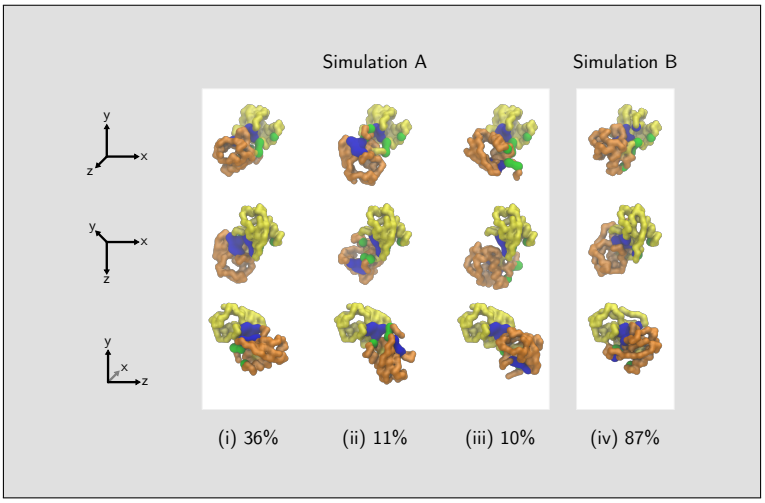


(b) Lys6-linked diubiquitin

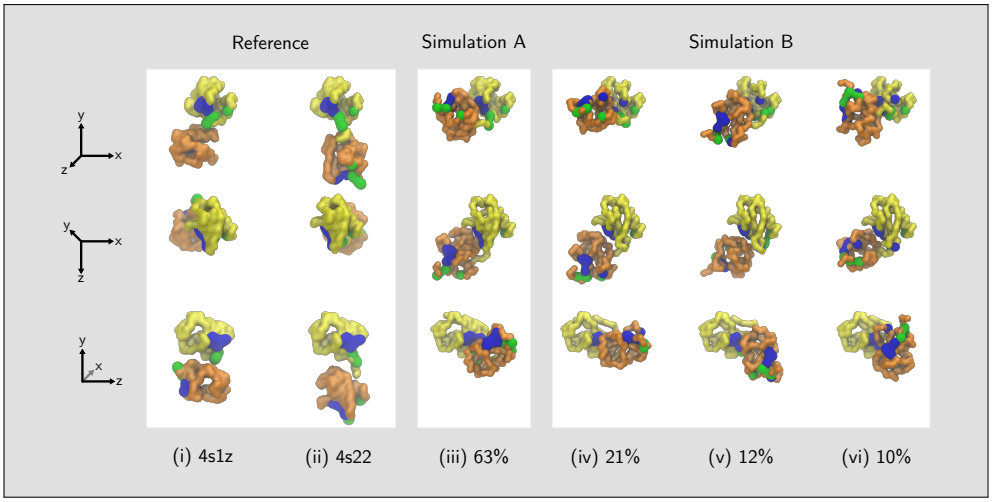
Figure 6.2: Comparisons of diubiquitin simulation clusters to reference structures *Each subfigure shows a single diubiquitin linkage. Different structures are arranged horizontally: first any available reference structures, labelled with their Protein Data Bank identifiers, then the largest clusters from the two simulations, labelled with their size as a percentage of the total number of collected samples. Clusters smaller than 10% were omitted.*
All structures in all subfigures are aligned with each other on the distal ubiquitin (shown in yellow). The hydrophobic patches centered on Ile36 and Ile44 are shown in green and blue, respectively. Each structure is shown from three different angles, arranged vertically: the top orientation was selected so that the hydrophobic patches on the distal ubiquitin would be visible, and the remaining orientations are rotations by 90° about the X and Y axis, respectively.
The largest clusters from each simulation are shown in greater detail in Figure 6.4.



(c) Lys11-linked diubiquitin

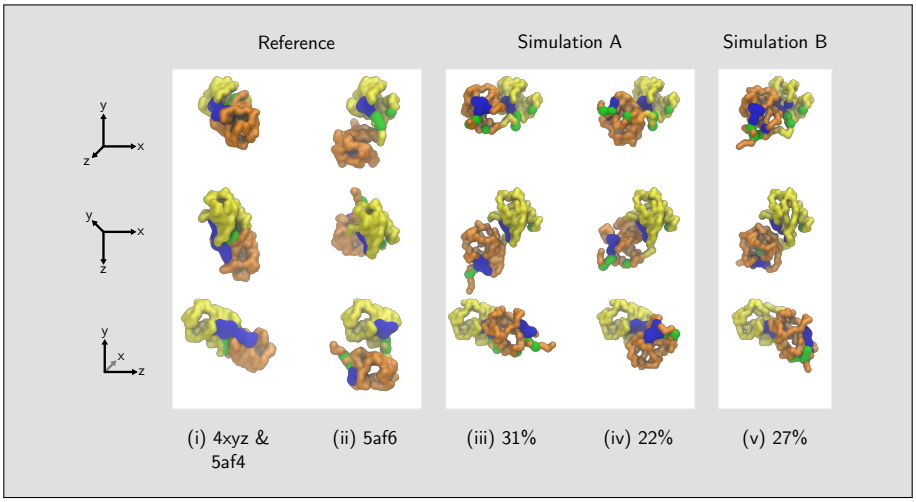


(d) Lys27-linked diubiquitin

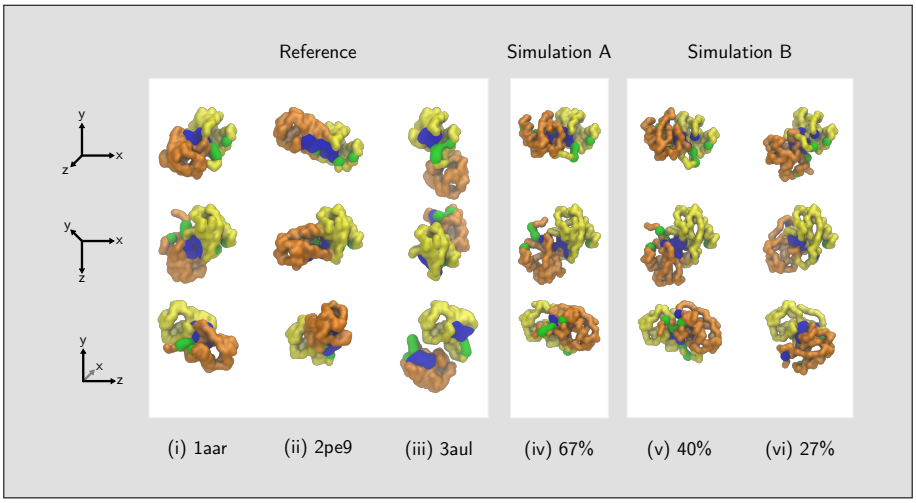


(e) Lys29-linked diubiquitin

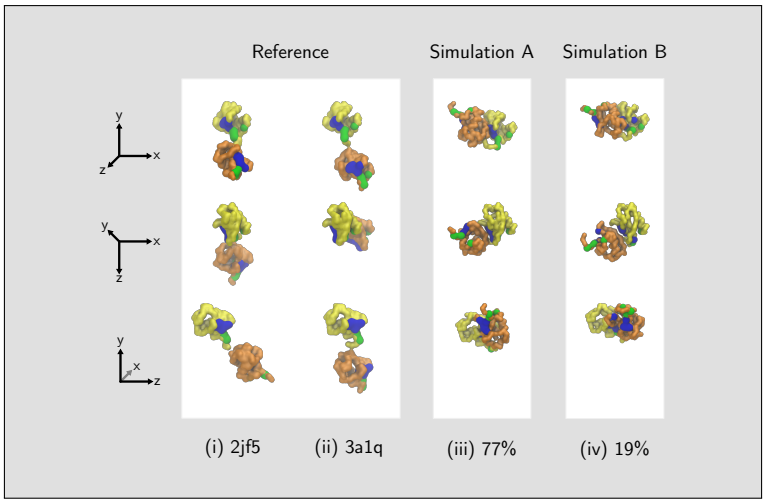
Figure 6.2: Comparisons of diubiquitin simulation clusters to reference structures (continued)



(f) Lys33-linked diubiquitin

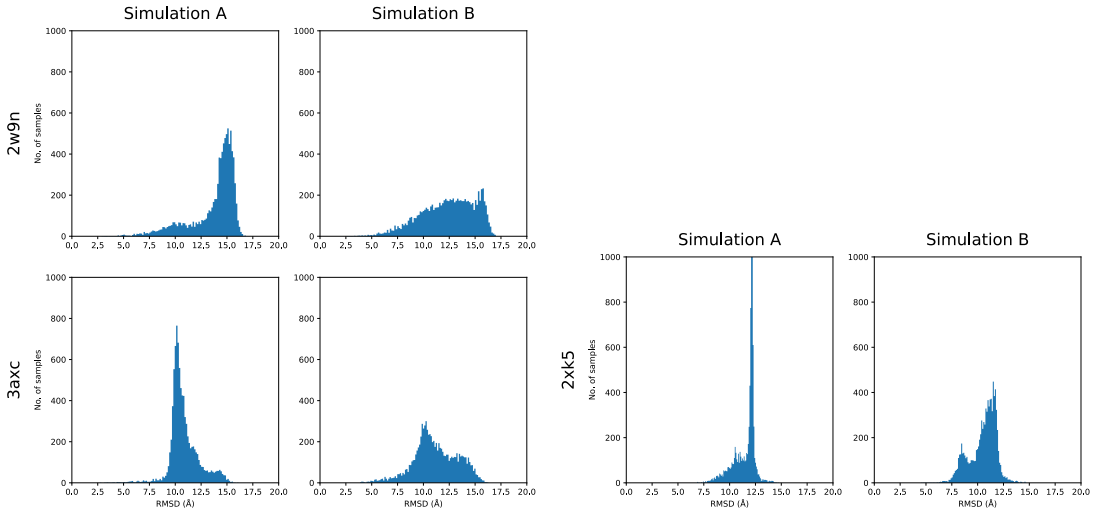


(g) Lys48-linked diubiquitin



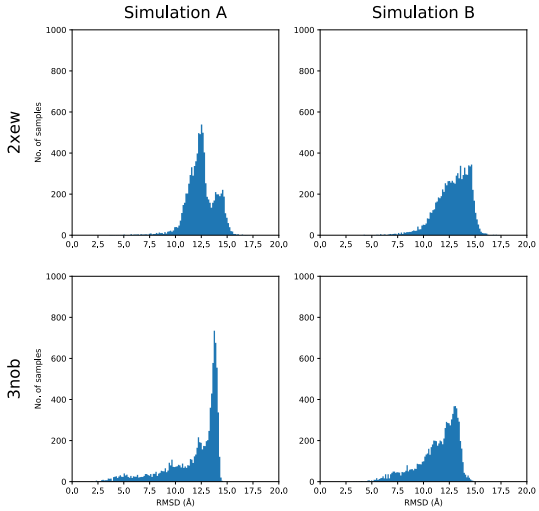
(h) Lys63-linked diubiquitin

Figure 6.2: Comparisons of diubiquitin simulation clusters to reference structures (continued)

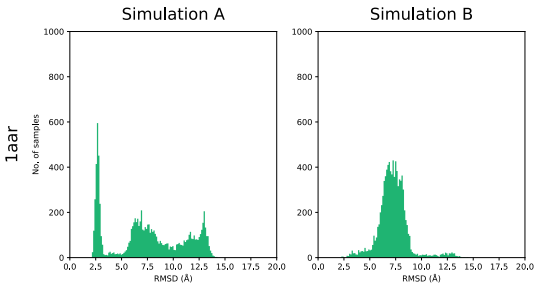


(a) Met1-linked diubiquitin

(b) Lys6-linked diubiquitin

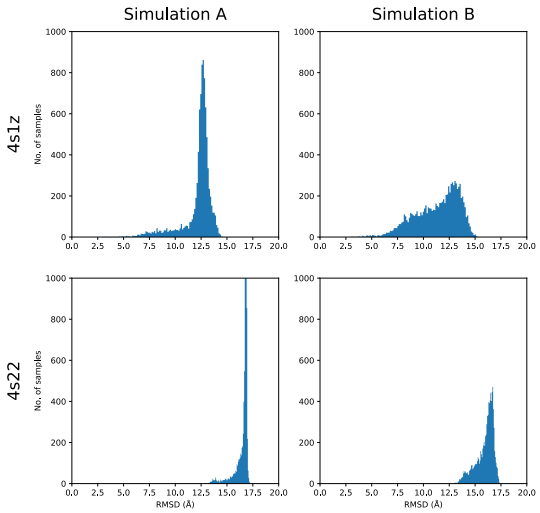


(c) Lys11-linked diubiquitin

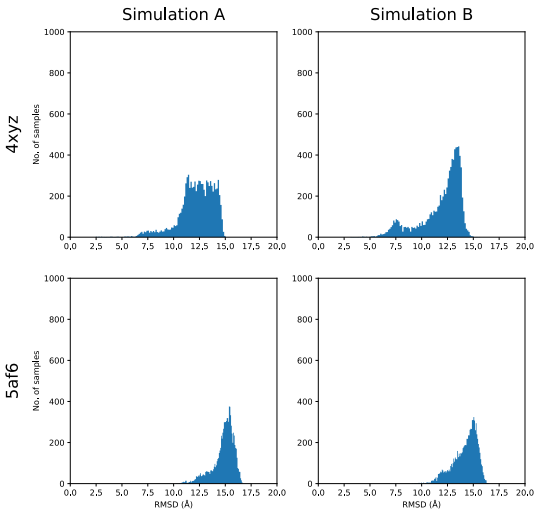


(d) Lys27-linked diubiquitin

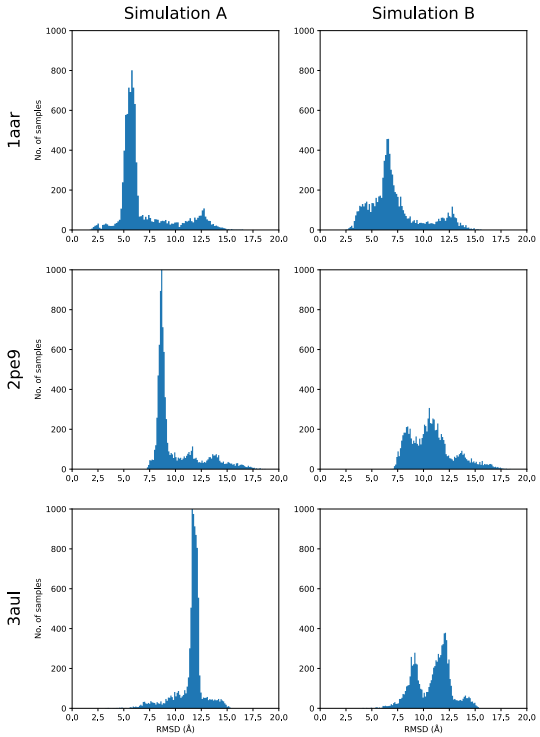
Figure 6.3: Distributions of RMSD between simulations and reference structures *The sub-figure for each linkage shows the distributions of RMSD between each simulation (A and B) and each reference crystal structure available for that linkage. No crystal linkages exist for Lys27-linked diubiquitin; (d) shows the RMSD distributions between these simulations and* 1aar*, the compact structure of Lys48-linked diubiquitin. This figure is shown in green. All histogram bins were truncated to* 1000 *samples.*



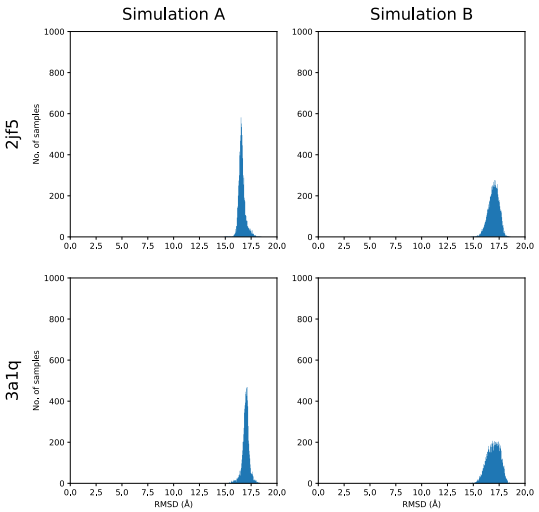
(e) Lys29-linked diubiquitin



(f) Lys33-linked diubiquitin



(g) Lys48-linked diubiquitin



(h) Lys63-linked diubiquitin

Figure 6.3: Distributions of RMSD between simulations and reference structures (continued)

shown in Figures 6.4 (a), (b) *i*, (c), (e), (f) and (h)). These similarities across linkages can also be seen in the average contact plots shown in Figure 6.6.

The reference structures for Met1- and Lys63-linked diubiquitin are open and elongated, whereas ours are more compact, as shown in Figures 6.2(a) and 6.2(h). Figure 6.3(h) shows that our results for Lys63 are the most dissimilar from all known crystal structures. However, our results for these linkages appear to be more consistent with the distribution of compact, half-compact and open structures reported by Ye et al. [78], as shown in Figures 6.5(a) *iii* and *ii*, respectively.

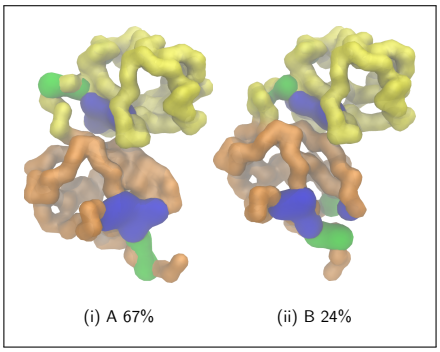
In most cases the added flexibility of the structures in simulation B added noise to the samples, which resulted in the clusters being smaller and less well-defined than in simulation A. A notable exception is simulation B of Lys27 (Figures 6.2(d) *iv* and 6.4(d) *ii*), where a single cluster constitutes 87% of the total samples, whereas in simulation A (Figures 6.2(d) *i* and 6.4(d) *i*) the largest cluster makes up only 36% of the population. In the structure from simulation B, the Leu8 residue is also separate from the rest of the Ile44-centered patch, which suggests that the orientation of this flexible loop may also be significant to this linkage.

Both reference structures for Lys29-linked diubiquitin (Figure 6.2(e) *i, ii*) and the Lys33-linked structure *5af6* (Figure 6.2(f) *ii*) are similarly more elongated and open than our results for these structures (Figures 6.2(e) *iii-vi* and 6.2(f) *iii-v*). However, the Lys33-linked *4xyz* or *5af4* (Figure 6.2(f) *i*) is more compact, but differs in orientation to our results. The two Lys11-linked reference structures likewise have a similar degree of compactness, but a different orientation of domains (as seen in Figure 6.2(c)).

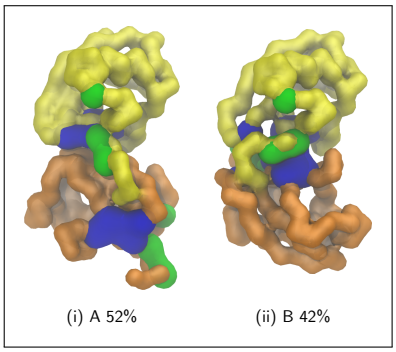
Figure 6.5 shows approximate FRET efficiency histograms for our simulations of Lys48-, Lys63- and Met1-linked diubiquitin, and compares them to the reference histogram described by Ye et al. [78]. The widths of of the peaks in our histograms (b) and (c) are not directly comparable to those in the reference (a), because the source of the width in the experiment is the finite number of photons detected, whereas in our simulation it's the heterogenous distribution of conformations. However, we can compare the positions of the peak maxima, and the apparent FRET populations.

There is some similarity between our histograms and the reference: the histograms for the Lys48 simulations (*i*) appear to show two peaks, while the others (*ii, iii*) appear to have only one. There is also a high degree of similarity in our estimated percentages for the three FRET subpopulations in our Lys63 and Met1 simulations. However, a much higher proportion of our Lys48-linked structures fall within the low-FRET population, as is shown by the proportionally higher peak near 0.1 in (b) *i* and (c) *i*. We did not find a single large cluster in either of the Lys48 simulations that corresponds to this low-FRET peak – it appears to be an aggregate of several small clusters in which the structures differ in orientation but have a similar degree of compactness.

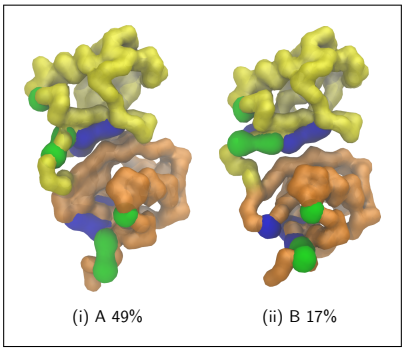
Figure 6.6 shows the degree to which specific residues in each lysine-linked diubiquitin interact with residues in the other domain: (a) shows chemical shift perturbations calculated for these linkages by Castañeda et al. from NMR data [77], and (b) and (c) show contact averages calculated from the relative positions of the residues in our simulation samples. Peaks in these



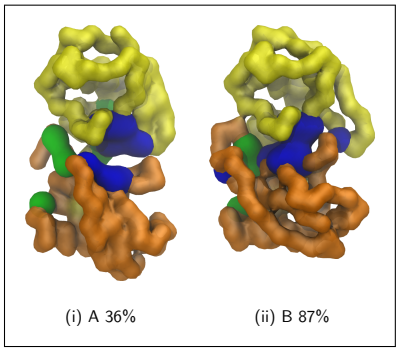
(a) Met1-linked diubiquitin



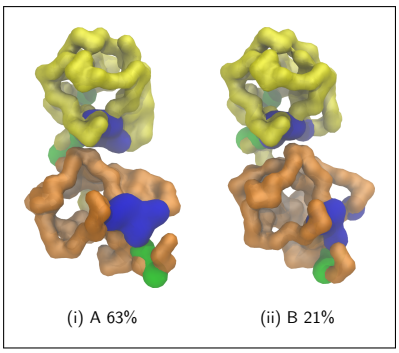
(b) Lys6-linked diubiquitin



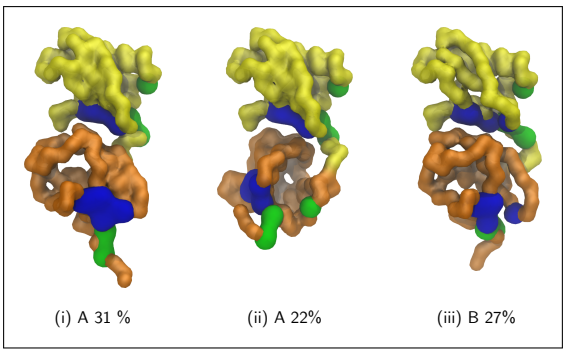
(c) Lys11-linked diubiquitin



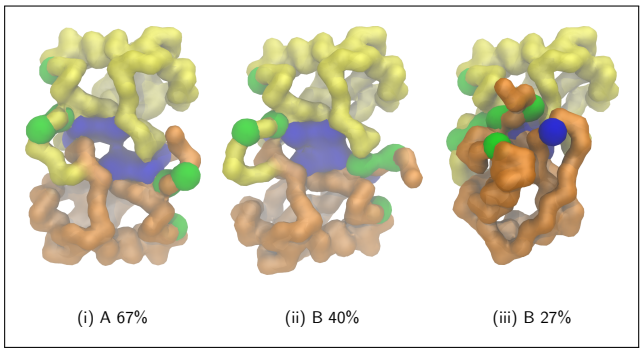
(d) Lys27-linked diubiquitin



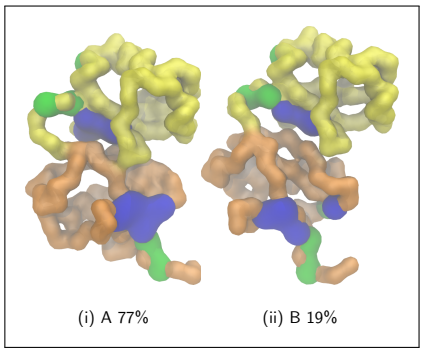
(e) Lys29-linked diubiquitin



(f) Lys33-linked diubiquitin



(g) Lys48-linked diubiquitin



(h) Lys63-linked diubiquitin

Figure 6.4: Representative structures from the largest clusters within each simulation

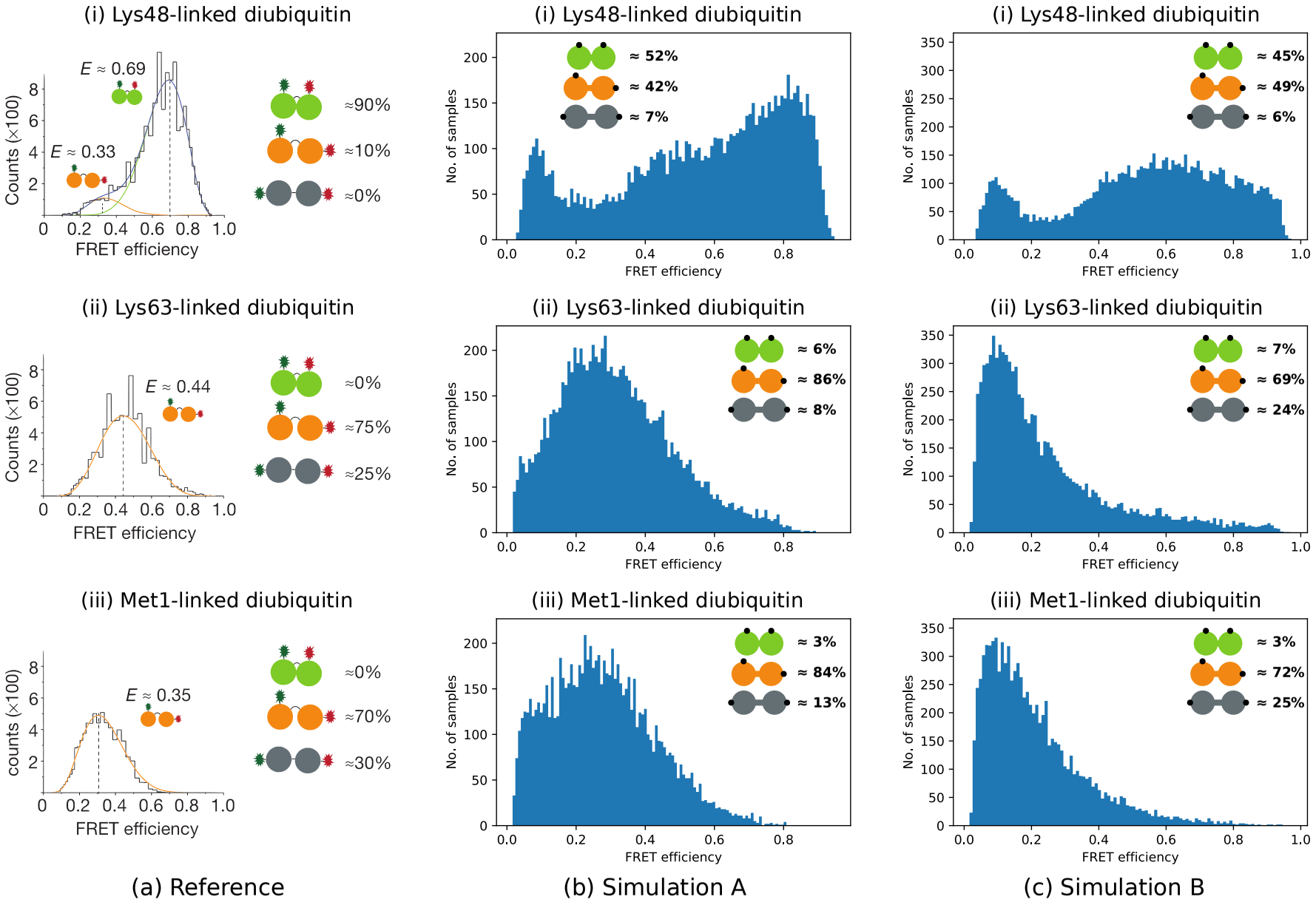


Figure 6.5: Comparison of FRET efficiency of Lys48-, Lys63- and Met1-linked diubiquitin *Reference (a) adapted from Ye et al. 2012 [78]. Used with permission. The reference (a) shows a FRET efficiency histogram for each linkage, fitted to Gaussian functions corresponding to subpopulations of high-FRET, low-FRET and no-FRET conformations. Subfigures (b) and (c) show our estimated FRET efficiency histograms for simulations A and B, respectively. The axes have not been normalised. Each subfigure also shows estimated percentages of high-FRET, low-FRET and no-FRET subpopulations.*

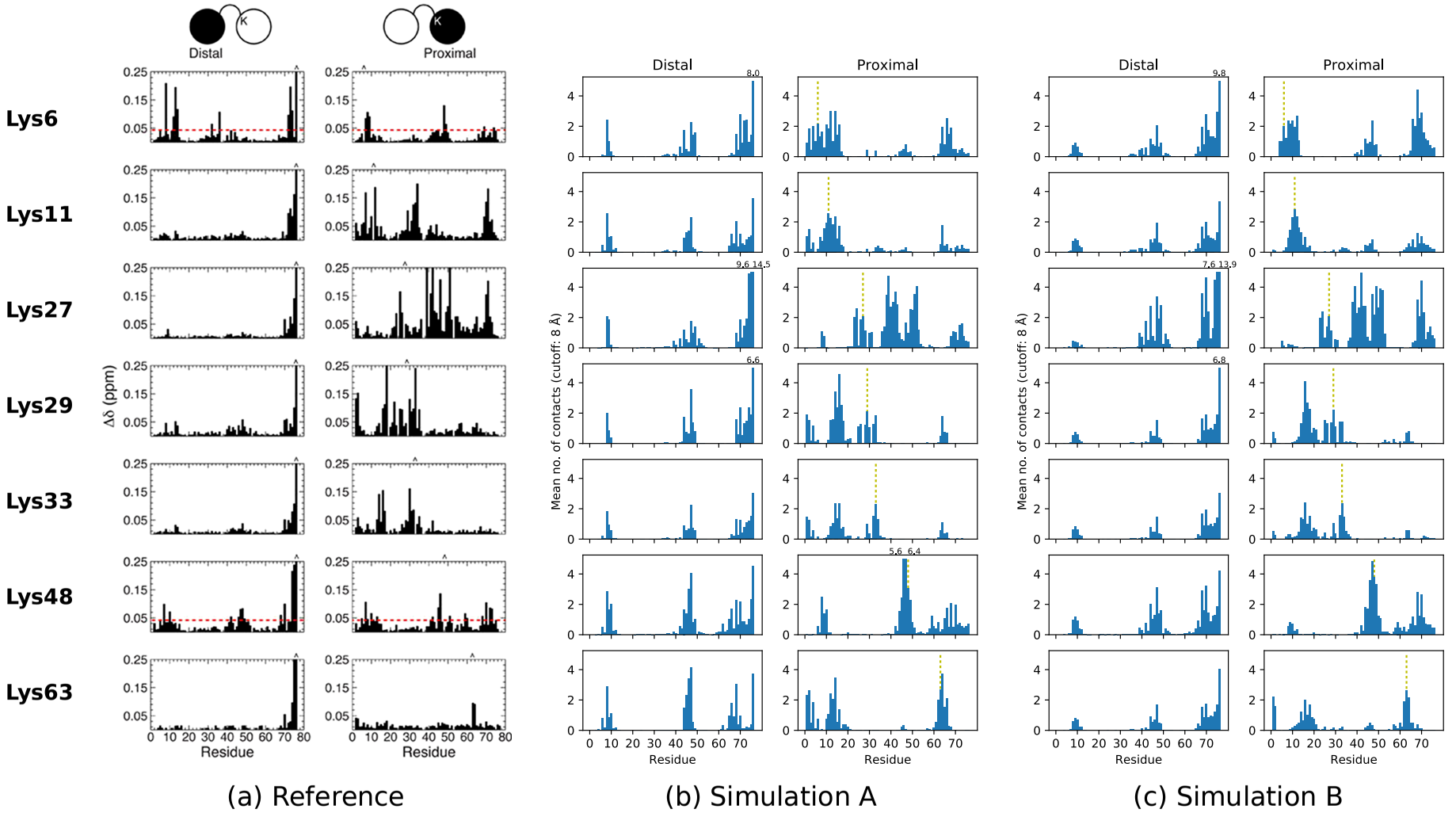


Figure 6.6: Comparison of chemical shift perturbations and average contacts for lysine-linked diubiquitin *Reference (a) adapted from Castañeda et al. 2016 [77]. Used with permission. In each subfigure the distal ubiquitin is shown in the left column and the proximal ubiquitin on the right. In (a) the residue on the proximal ubiquitin which is the binding site for the distal ubiquitin is marked with a carat; in (b) and (c) this residue is indicated with a dashed yellow line. In (b) and (c) the average contact value shown for the last residue of the distal ubiquitin is the mean of the values for that actual residue and the value for the additional alanine "side chain" which we use for padding in our simulations. We truncate the values to 5; the actual values for residues which exceed this ceiling are marked on the plot.*

plots correspond to residues in each ubiquitin domain which are in close proximity to the other ubiquitin domain.

As expected, all of our plots show peaks at the residues nearest the isopeptide bond between the two ubiquitin monomers: the tail of each distal ubiquitin, and the corresponding binding site of each proximal ubiquitin.

Our plots of the distal ubiquitin residues are much more similar to each other than the corresponding plots in the reference: each distal ubiquitin has a small peak corresponding to the Ile44 hydrophobic patch, and a smaller peak at the Leu8 residue. Unlike Castañeda et al., we do not observe notably higher interaction values in the distal domains of Lys6- or Lys48-linked diubiquitin.

The proximal ubiquitin plots reflect the similarity in structures across linkages which can be seen in Figure 6.4. The plots for Lys27 and Lys48 show a peak near the proximal Ile44 patch in both simulations. This is consistent with the dominant conformations in both these simulations, which are compact, with Ile44 patches facing each other (as shown in Figures 6.4(d) and 6.4(g)). In Simulation A of Lys6 this peak is almost absent, but a small peak is visible in Simulation B – this corresponds to the compact structure shown in Figure 6.4(b) *ii*. The other linkages show little or no peak around the proximal Ile44, which is consistent with the semi-compact structures found for these linkages, in which the distal Ile44 patches are facing the proximal ubiquitin and the proximal Ile44 patches are facing outwards and away from the distal ubiquitin (Figures 6.4(a), 6.4(b) *i*, 6.4(c), 6.4(e), 6.4(f) and 6.4(h)).

Our plot of the proximal ubiquitin residues in Lys11-linked diubiquitin is the most dissimilar to the experimental chemical shift data, since it lacks the prominent peak near the Ile36 hydrophobic patch. For all the other linkages, our proximal ubiquitin plots resemble the experimental data more closely, although some of the peaks differ in size. This suggests that our structures may be a better match for diubiquitin chains in solution than for the crystal structures shown in Figure 6.2.

6.3 Conclusions

Our simulation results are most consistent with the reference crystal structures for Lys48- and Lys6-linked diubiquitin, for which we found very compact structures with Ile44 hydrophobic patches facing each other. We obtained a similar result for Lys27-linked diubiquitin, for which no reference structures currently exist, but for which a structure similar to Lys48-linked diubiquitin has been proposed [77].

The distribution of FRET populations in our Met1-, Lys48- and Lys63-linked diubiquitin simulations shows some similarity to the statistics proposed in Ye et al. [78], although we found a higher proportion of low-FRET populations in both Lys48 simulations.

Most of our simulations across all diubiquitin linkages produced a very similar semi-compact conformation in which the proximal Ile44 hydrophobic patch faces away from the distal ubiquitin. These structures are not very similar to the more open and elongated crystal structures which have been proposed for these linkages, but they correspond more closely to NMR data derived

from diubiquitin chains in solution [77].

These results may support previously published findings that the crystal structures for diubiquitin do not fully describe the structure of diubiquitin in solution, which may adopt a wider range of different conformations.

Modelling the $\beta 1/\beta 2$ loop of each ubiquitin as a flexible linker and extending the flexible portion of the tail appeared to have little effect on most of the simulation results: for most linkages these added degrees of freedom resulted only in adding noise to the samples. However, in a few simulations we obtained noticeably different results; particularly in simulation B of Lys27-linked diubiquitin, which has a much larger dominant cluster of structures than Simulation A. There may therefore be specific polyubiquitin chains which can be modelled more effectively with these additional linkers.

Chapter 7

Conclusions

In this work we have described CGPPD, a custom parallel implementation of the Kim and Hummer coarse-grained model for protein-protein docking simulations using replica exchange Monte Carlo. We summarised how the original work on this implementation focused on improving performance by making use of the parallel GPU architecture to speed up the most computationally expensive portion of the simulation.

We then introduced our modifications to the implementation. Our aim was to extend its rigid protein model to allow for optional flexible linkers. This required the addition of new Monte Carlo mutations which allowed flexible proteins to deform, and new potential calculation components to evaluate these mutations. While further performance gains were not the focus of our research, we did not wish to compromise the performance of the rigid implementation, and expected that our changes would not add a significant overhead to the simulation running time.

To test the performance of CGPPD v2 we performed a set of benchmarking simulations using parameters similar to the diubiquitin simulations we describe below. Our benchmark results show that not only does the addition of the linkers not increase the running time of the simulation significantly, but sometimes adding more linkers causes the simulation to run faster. We speculate that the reason for this may be more efficient latency hiding between the CPU and GPU.

The performance of CGPPD may be improved in the future with the introduction of multiple specialised kernels which are selected at runtime according to the properties of the system being modelled. For example, an entirely rigid simulation does not require the additional complexity we introduced to the generic kernel. A homopolymer simulation, such as the polyalanine folding simulations we used to validate our model, does not require a lookup table for bead interactions, since all the residues are the same.

We performed one case study using our modified implementation: investigating the conformations of all eight possible linkages of diubiquitin chains. We compared our results both to known crystal structures of diubiquitin, and to NMR and FRET studies of diubiquitin in solution. The structures we produced for Lys48-linked diubiquitin, the most thoroughly understood linkage, show some similarity to the reference crystal structures, and we found a similar conformation for Lys27-linked diubiquitin, for which no crystal structures currently exist. Our other

structures were less similar to the crystal structures. However, we found them to be in closer agreement with the NMR and FRET data, which describes a broader range of conformations. We can thus conclude that our model can reproduce some existing experimental results. Our findings suggest that the crystal structures do not provide an accurate representation of the structure of diubiquitin in living cells.

Flexible linkers not only allowed us to perform a simulation in which the two ubiquitin monomers could move with respect to one another while constraining the conformation to a specific linkage, but also to compare two sets of simulations with different degrees of flexibility. We found that modelling the $\beta 1/\beta 2$ loop of each ubiquitin as a linker had a notable effect on the distribution of conformations of Lys27-linked diubiquitin. It may be of value to investigate the effect of this linker on other simulations involving ubiquitin.

In future this model could be used to simulate longer polyubiquitin chains, as well as interactions between monoubiquitin or polyubiquitin and various UIMs. It may be useful to revisit the case studies previously investigated with the rigid implementation.

Bibliography

[1] Y. C. Kim and G. Hummer, "Coarse-grained models for simulations of multiprotein complexes: Application to ubiquitin binding," *Journal of Molecular Biology*, vol. 375, pp. 1416–1433, 2007.

[2] I. Tunbridge, R. B. Best, J. Gain, and M. M. Kuttel, "Simulation of coarse-grained protein-protein interactions with graphics processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 11, pp. 3588–3600, 2010.

[3] A. Tramontano, *The Ten Most Wanted Solutions in Protein Bioinformatics*, ch. 7, pp. 117–139. Chapman & Hall/CRC, 2005.

[4] C. Mura and C. E. McAnany, "An introduction to biomolecular simulations and docking," *Molecular Simulation*, vol. 40, pp. 732–764, Aug. 2014.

[5] I. Tunbridge, *Graphics Processing Unit Accelerated Coarse-Grained Protein-Protein Docking*. PhD thesis, University of Cape Town, 2011.

[6] C.-J. Tsai, S. Kumar, B. Ma, and R. Nussinov, "Folding funnels, binding funnels, and protein function," *Protein Science*, vol. 8, pp. 1181–1190, Jan. 1999.

[7] G. Marshall and I. Vakser, "Protein-protein docking methods," in *Proteomics and Protein-Protein Interactions* (G. Waksman, ed.), vol. 3 of *Protein Reviews*, pp. 115–146–146, Boston, MA: Springer US, 2005.

[8] D. J. C. MacKay, "Introduction to Monte Carlo methods," in *Proceedings of the NATO Advanced Study Institute on Learning in Graphical Models*, (Norwell, MA, USA), pp. 175–204, Kluwer Academic Publishers, 1998.

[9] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[10] W. K. Hastings, "Monte carlo sampling methods using Markov chains and their applications," *Biometrika*, vol. 57, pp. 97–109, Jan. 1970.

[11] "Enhanced sampling algorithms," in *Biomolecular Simulations* (L. Monticelli and E. Salonen, eds.), no. 924 in Methods in Molecular Biology, Humana Press, 2013.

[12] K. Lee, "Computational study for protein-protein docking using global optimization and empirical potentials.," *International Journal of Molecular Sciences*, vol. 9, pp. 65–77, Jan. 2008.

[13] J. Janin, "Assessing predictions of protein–protein interaction: The CAPRI experiment," *Protein Science*, vol. 14, no. 2, pp. 278–283, 2005.

[14] J. E. Jones, "On the determination of molecular fields. II. from the equation of state of a gas," *Proceedings of the Royal Society of London. Series A*, vol. 106, pp. 463–477, Oct. 1924.

[15] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, Dec. 2007.

[16] V. E. Lamberti, L. D. Fosdick, E. R. Jessup, and C. J. C. Schauble, "A hands-on introduction to molecular dynamics," *Journal of Chemical Education*, vol. 79, p. 601, May 2002.

[17] B. Widom, *Statistical Mechanics: A Concise Introduction for Chemists*. Cambridge University Press, 1 ed., May 2002.

[18] D. Earl and M. Deem, "Parallel tempering: Theory, applications, and new perspectives," *Physical Chemistry Chemical Physics*, vol. 7, pp. 3910–3916, 2005.

[19] M. Eleftheriou, A. Rayshubski, J. W. Pitera, B. G. Fitch, R. Zhou, and R. S. Germain, "Parallel implementation of the replica exchange molecular dynamics algorithm on Blue Gene/L," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, vol. 0, p. 281, 2006.

[20] Y. Li, M. Mascagni, and A. Gorin, "A decentralized parallel implementation for parallel tempering algorithm," *Parallel Computing*, vol. 35, pp. 269–283, May 2009.

[21] V. Tozzini, "Coarse-grained models for proteins," *Current Opinion in Structural Biology*, vol. 15, pp. 144–150, 2005.

[22] A. Kolinski and J. Skolnick, "Reduced models of proteins and their applications," *Conformational Protein Conformations*, vol. 45, pp. 511–524, Jan. 2004.

[23] C. Clementi, "Coarse-grained models of protein folding: toy models or predictive tools?," *Current Opinion in Structural Biology*, vol. 18, pp. 10–15, Feb. 2008.

[24] R. Das and D. Baker, "Macromolecular modeling with Rosetta," *Annual Review of Biochemistry*, vol. 77, no. 1, pp. 363–382, 2008.

[25] C. Chen, R. Saxena, and G.-W. Wei, "A multiscale model for virus capsid dynamics," *International Journal of Biomedical Imaging*, vol. 2010, pp. 3:1–3:9, Mar. 2010.

[26] F. Tama, O. Miyashita, and C. L. Brooks, "Normal mode based flexible fitting of high-resolution structure into low-resolution experimental data from cryo-EM.," *J Struct Biol*, vol. 147, pp. 315–326, Sept. 2004.

[27] A. M. Bonvin, "Flexible protein–protein docking," *Current Opinion in Structural Biology*, vol. 16, pp. 194–200, Apr. 2006.

[28] L. P. Ehrlich, M. Nilges, and R. C. Wade, "The impact of protein flexibility on protein–protein docking," *Proteins: Structure, Function, and Bioinformatics*, vol. 58, pp. 126–133, Jan. 2005.

[29] D. Alvarez-Garcia and X. Barril, "Relationship between protein flexibility and binding: Lessons for structure-based drug design," *Journal of Chemical Theory and Computation*, vol. 10, pp. 2608–2614, June 2014.

[30] L. Li, R. Chen, and Z. Weng, "RDOCK: Refinement of rigid-body protein docking predictions," *Proteins: Structure, Function, and Bioinformatics*, vol. 53, pp. 693–707, Nov. 2003.

[31] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[32] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[33] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, pp. 864–872, 2009.

[34] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," white paper, NVIDIA Corporation, Oct. 2011.

[35] NVIDIA Corporation, "CUDA C Programming Guide 6.5." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` Last accessed: 2015-01-30, August 2014.

[36] NVIDIA Corporation, "CUDA C Best Practices Guide 6.5." `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/` Last accessed: 2015-01-30, August 2014.

[37] O. Guvench and A. D. M. Jr, "Comparison of protein force fields for molecular dynamics simulations," in *Molecular Modeling of Proteins* (A. Kukol, ed.), no. 443 in Methods Molecular Biology™, pp. 63–88, Humana Press, Jan. 2008.

[38] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz, D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman, "A second generation force field for the simulation of proteins, nucleic acids, and organic molecules," *Journal of the American Chemical Society*, vol. 117, no. 19, pp. 5179–5197, 1995.

[39] A. D. MacKerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiórkiewicz-Kuczera, D. Yin, and M. Karplus, "All-atom empirical potential for molecular modeling and dynamics studies of proteins," *The Journal of Physical Chemistry B*, vol. 102, no. 18, pp. 3586–3616, 1998.

[40] C. Oostenbrink, A. Villa, A. E. Mark, and W. F. van Gunsteren, "A biomolecular force field based on the free enthalpy of hydration and solvation: the GROMOS force-field parameter sets 53a5 and 53a6," *Journal of Computational Chemistry*, vol. 25, pp. 1656–1676, Oct. 2004.

[41] W. L. Jorgensen, D. S. Maxwell, and J. Tirado-Rives, "Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids," *Journal of the American Chemical Society*, vol. 118, no. 45, pp. 11225–11236, 1996.

[42] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The AMBER biomolecular simulation programs," *Journal of Computational Chemistry*, vol. 26, pp. 1668–1688, Dec. 2005.

[43] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations," *J. Comput. Chem.*, vol. 4, pp. 187–217, Feb. 1983.

[44] W. R. P. Scott, P. H. Hünenberger, I. G. Tironi, A. E. Mark, S. R. Billeter, J. Fennen, A. E. Torda, T. Huber, P. Krüger, and W. F. van Gunsteren, "The GROMOS biomolecular simulation program package," *The Journal of Physical Chemistry A*, vol. 103, no. 19, pp. 3596–3607, 1999.

[45] W. L. Jorgensen and J. Tirado–Rives, "Molecular modeling of organic and biomolecular systems using BOSS and MCPRO," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1689–1700, 2005.

[46] L. Kalé, R. Skeel, M. Bh, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," *Journal of Computational Physics*, vol. 151, pp. 283–312, 1999.

[47] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, Mar. 1995.

[48] E. Lindahl, B. Hess, and D. van der Spoel, "GROMACS 3.0: a package for molecular simulation and trajectory analysis," *Journal of Molecular Modeling*, vol. 7, pp. 306–317, Aug. 2001.

[49] S. J. Marrink, H. J. Risselada, S. Yefimov, D. P. Tieleman, and A. H. de Vries, "The MARTINI force field: Coarse grained model for biomolecular simulations," *The Journal of Physical Chemistry B*, vol. 111, no. 27, pp. 7812–7824, 2007.

[50] L. Monticelli, S. K. Kandasamy, X. Periole, R. G. Larson, D. P. Tieleman, and S.-J. Marrink, "The MARTINI coarse-grained force field: Extension to proteins," *Journal of Chemical Theory and Computation*, vol. 4, no. 5, pp. 819–834, 2008.

[51] P. Kar, S. M. Gopal, Y.-M. Cheng, A. Predeus, and M. Feig, "PRIMO: A transferable coarse-grained force field for proteins," *Journal of Chemical Theory and Computation*, vol. 9, no. 8, pp. 3769–3788, 2013.

[52] H. A. Karimi-Varzaneh, H.-J. Qian, X. Chen, P. Carbone, and F. Müller-Plathe, "IBIsCO: a molecular dynamics simulation package for coarse-grained simulation," *Journal of Computational Chemistry*, vol. 32, pp. 1475–1487, May 2011.

[53] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Roehm, P. Košovan, and C. Holm, "ESPResSo 3.1: Molecular dynamics software for coarse-grained models," in *Meshfree Methods for Partial Differential Equations VI* (M. Griebel and M. A. Schweitzer, eds.), no. 89 in Lecture Notes in Computational Science and Engineering, pp. 1–23, Springer Berlin Heidelberg, Jan. 2013.

[54] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-accelerated molecular modeling coming of age," *Journal of Molecular Graphics and Modelling*, vol. 29, pp. 116–125, Sept. 2010.

[55] J. A. van Meel, A. Arnold, D. Frenkel, O. Portegies, and R. G. Belleman, "Harvesting graphics power for MD simulations," *Molecular Simulation*, vol. 34, pp. 259–266, 2008.

[56] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, pp. 5342–5359, May 2008.

[57] M. J. Harvey, G. Giupponi, and G. D. Fabritiis, "ACEMD: Accelerating biomolecular dynamics in the microsecond time scale," *Journal of Chemical Theory and Computation*, vol. 5, pp. 1632–1639, June 2009.

[58] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, and A. J. Olson, "Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function," *J. Comput. Chem.*, vol. 19, pp. 1639–1662, Jan. 1998.

[59] G. Jones, "Development and validation of a genetic algorithm for flexible docking," *Journal of Molecular Biology*, vol. 267, pp. 727–748, Apr. 1997.

[60] R. Chen, L. Li, and Z. Weng, "ZDOCK: An initial-stage protein-docking algorithm," *Proteins: Structure, Function, and Bioinformatics*, vol. 52, no. 1, pp. 80–87, 2003.

[61] B. Pierce and Z. Weng, "ZRANK: reranking protein docking predictions with an optimized energy function," *Proteins*, vol. 67, pp. 1078–1086, June 2007.

[62] C. Dominguez, R. Boelens, and A. M. J. J. Bonvin, "HADDOCK: A protein-protein docking approach based on biochemical or biophysical information," *Journal of the American Chemical Society*, vol. 125, no. 7, pp. 1731–1737, 2003.

[63] S. Miyazawa and R. Jernigan, "Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term for simulation and threading," *Journal of Molecular Biology*, vol. 256, pp. 623–644, 1996.

[64] B. Gough, *GNU Scientific Library Reference Manual - Third Edition.* Network Theory Ltd., 3rd ed., 2009.

[65] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, 1998.

[66] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Library.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2000.

[67] A. Kolinski and J. Skolnick, "Monte Carlo simulations of protein folding. I. Lattice model and interaction scheme," *Proteins: Structure, Function, and Bioinformatics*, vol. 18, no. 4, pp. 338–352, 1994.

[68] F. Liang and W. H. Wong, "Evolutionary Monte Carlo for protein folding simulations," *The Journal of Chemical Physics*, vol. 115, no. 7, p. 3374, 2001.

[69] R. B. Best, Y.-G. Chen, and G. Hummer, "Slow Protein Conformational Dynamics from Multiple Experimental Structures: The Helix/Sheet Transition of Arc Repressor," *Structure*, vol. 13, pp. 1755–1763, Dec. 2005.

[70] J. Karanicolas and C. L. Brooks, "The origins of asymmetry in the folding transition states of protein L and protein G," *Protein Science*, vol. 11, pp. 2351–2361, July 2002.

[71] NVIDIA Corporation, "NVIDIA CUDA Math API 6.5." `http://docs.nvidia.com/cuda/cuda-math-api/` Last accessed: 2015-03-03, August 2014.

[72] P. de Gennes, *Scaling Concepts in Polymer Physics.* Cornell University Press, 1979.

[73] K. E. Sloper-Mould, J. C. Jemc, C. M. Pickart, and L. Hicke, "Distinct functional surface regions on ubiquitin," *Journal of Biological Chemistry*, vol. 276, pp. 30483–30489, Aug. 2001.

[74] D. Komander and M. Rape, "The Ubiquitin Code," *Annual Review of Biochemistry*, vol. 81, pp. 203–229, July 2012.

[75] M. K. Hospenthal, S. M. Freund, and D. Komander, "Assembly, analysis and architecture of atypical ubiquitin chains," *Nature structural & molecular biology*, vol. 20, pp. 555–565, May 2013.

[76] D. Fushman and K. D. Wilkinson, "Structure and recognition of polyubiquitin chains of different lengths and linkage," *F1000 Biology Reports*, vol. 3, Dec. 2011.

[77] C. A. Castañeda, A. Chaturvedi, C. M. Camara, J. E. Curtis, S. Krueger, and D. Fushman, "Linkage-specific conformational ensembles of non-canonical polyubiquitin chains," *Physical Chemistry Chemical Physics*, vol. 18, no. 8, pp. 5771–5788, 2016.

[78] Y. Ye, G. Blaser, M. H. Horrocks, M. J. Ruedas-Rama, S. Ibrahim, A. A. Zhukov, A. Orte, D. Klenerman, S. E. Jackson, and D. Komander, "Ubiquitin chain conformation regulates recognition and activity of interacting proteins," *Nature*, vol. 492, pp. 266–270, 2012.

[79] M. D. Cummings, T. N. Hart, and R. J. Read, "Monte Carlo docking with ubiquitin," *Protein Science*, vol. 4, pp. 885–899, May 1995.

[80] V. Dijk, A. D.j, D. Fushman, and A. M. J. J. Bonvin, "Various strategies of using residual dipolar couplings in NMR-driven protein docking: Application to Lys48-linked di-ubiquitin and validation against 15n-relaxation data," *Proteins: Structure, Function, and Bioinformatics*, vol. 60, pp. 367–381, Aug. 2005.

[81] D. Fushman and O. Walker, "Exploring the Linkage Dependence of Polyubiquitin Conformations Using Molecular Modeling," *Journal of Molecular Biology*, vol. 395, pp. 803–814, Jan. 2010.

[82] T. Dresselhaus, N. D. Weikart, H. D. Mootz, and M. P. Waller, "Naturally and synthetically linked lys48 diubiquitin: a QM/MM study," *RSC Advances*, vol. 3, pp. 16122–16129, Aug. 2013.

[83] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[84] Luis Gracia, "Clustering plugin for VMD." `http://physiology.med.cornell.edu/faculty/hweinstein/vmdplugins/clustering/` Last accessed: 2018-01-30, June 2014.