PARALLEL PROTEIN FOLDING

BY MINIMUM ENERGY STATE EQUATION

BY

ANTONIO R. ARREDONDO, B.S.

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

Major Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

August 2009

"Parallel protein folding by minimum energy state equation," a thesis prepared by Antonio R. Arredondo in partial fulfillment of the requirements for the degree, Masters of Science, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Enrico Pontelli
Chair of the Examining Committee

Date

Committee in charge:

Dr. Enrico Pontelli, Chair

Dr. Kenneth Hacker

Dr. Joseph Pfeiffer

DEDICATION

I dedicate this work to my mother, Alicia, my dad, Melvin, my brother, Abraham, and my sisters, Evette and Vanessa.

ACKNOWLEDGMENTS

I would like to thank my advisor, Enrico Pontelli, for his encouragement, interest, and patience. I would like to thanks the following members of my family, Angie Ramos, Josefina Ramos, and Debbie Stanard. I would also like to thank my friends that have helped me along my way, Tony Grajeda, Mario Perea, Cheli Hobson, Julie Esteban, Consuelo Resendez, Elena Grajeda, Robert Hobson, Tim Mejia, Tommy Barrera, Chito Chavez, Stan Engle, Dan Cuaron, Deedee Castruita, Mike Dowell, Ken Binkley and Shane Thomas.

VITA

May 19, 1976	Born in Watsonville, California, USA.
1998-2004	B.S., San Jose State University, San Jose, CA
2004-2009	Teaching Assistant, Computer Science Department, New Mexico State University

PROFESSIONAL AND HONORARY SOCIETIES

Alpha Chi

PUBLICATIONS [or Papers Presented]

FIELD OF STUDY

Major Field: Bioinformatics and Parallel Computing

ABSTRACT

PARALLEL PROTEIN FOLDING

BY MINIMUM ENERGY STATE EQUATION

BY

ANTONIO R. ARREDONDO, B.S.

Master's of Science

New Mexico State University

Las Cruces, New Mexico, 2009

Dr. Enrico Pontelli, Chair

This project develops a parallel computing solution to the protein folding problem in a crystal lattice representation in 3D space, using a minimum energy state equation. This work continues the development of the protein folding minimum energy state equation, in order to support larger protein sequences via parallel computing. Parallel computing enables multiple computations to be carried out simultaneously. These computations are done across multiple processors. A parallel design takes advantage of being able to execute simultaneous computations by

creating as many search branches as possible. The sequential protein folding program demonstrates behavior similar to the task pattern for parallel programming. The parallel program is developed using the task pattern and implemented using a master/worker supporting structure design. The result of parallising the protein folding code is that larger amino acids can be processed in less time and the determination of their 3D structure's is accomplished with an increase in efficiency and speed in comparison to the sequential protein folding code.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Given that proteins are the building blocks of all biological processes, it can be assumed that proteins are a huge interest to the scientific community. One interesting aspect about proteins, is how they interact in a biological environment. The interaction of proteins is dictated by their native state, tertiary structure (3D structure conformation). Protein structure prediction is the process of determining a protein's tertiary structure given its amino acid sequence. Protein structure prediction is such an interesting problem, that a community-wide experiment, Critical Assessment of techniques for Protein Structure Prediction (CASP), is held every two years. This experiment helps to asses the current methods of protein structure prediction. In general, determining the tertiary structure of a protein given its sequence, is beyond our current capabilities [20].

The current methods of determining the tertiary structure (3D structure) from the primary structure (sequence of amino acids) fall into two categories. The first is assembling the structure of the protein using know structural fragments of similar sequences. The data fragments are taken from a protein structure repository (Protein Data Bank [7]) which are screened for viability of the resulting structure. The other method is to use a simplified model representing the protein chain and then checking its viability.

The second method of determining the protein's tertiary structure has

many interesting aspects[18]. One aspect is the link between kinetics, thermodynamics of the protein folding process and intra-molecular interactions The use of simplified modeling allows easier handling of this link, due to the reduced number of variables. Another aspect is that the simplified model coincides with the assertion that the overall interactions of the amino acids are more important than the individual atom interactions [6]. Another advantage that a simplified model provides is that the energy of a conformation can be determined efficiently, due to the reduced number of variables. This provides an important computational advantage.

Given the reduced number of variables, we can adopt the simplified model into a lattice structure, as has been done in various other fields [8, 14] and previously shown [13]. The application of lattices to the protein folding problem in 3D has been previously pursued [11, 5, 1, 2, 14]. A constraint framework is used to solve the 3D crystal lattice, where the constraint domain is represented by individual lattice points, and the primitive constraints are introduced to represent the spatial relationships of the lattice structure. Constraint solving is performed using the framework mentioned above, with a focus on propagation and search strategies in order to generate a 3D structure given its amino acid sequence.

1.1 MOTIVATION

Knowledge of the 3D structure is key to determining its biological interactions. This knowledge is beneficial for understanding the protein's possible applications (e.g. medical applications). An example of a medical application is the production of new pharamcutical drugs. Creation of effective pharamcuticals, is dependent on knowing the structure of the intended protein that the drug will interact with.

The motivation behind this work is to increase the scalability of the sequential program for solving the protein folding problem in crystal lattice in 3D space by Dal Palù [3]. This is accomplished by developing a parallel program based off the previous work [3]. In the original research, a solution was developed to the protein folding problem by using an energy function that would find the lowest energy state of the protein's natural structure[3]. His work has already shown that a constraint logic programming (CLP) approach is useful in studying protein models[1]. Constraint logic programming is a programming paradigm where the relationship between variables is defined by a set of constraints. This approached worked well, however, it is limited in the problem space (protein size) that could be searched. This limitation is a result of the sequential nature of the search algorithm [4, 3].

1.2 THE PROBLEM

The protein structure prediction problem can be defined as, given the primary structure of a protein determine the tertiary structure of the protein assuming normal conditions in a biological environment. The protein folding problem is one of the most useful to biologists due to the fact that all of a protein's functional properties depend on its tertiary structure. The foundation of structure prediction is based on the premise that, from the primary structure of a protein its tertiary structure can be derived. According to this theory, the 3D conformation that yields the lowest energy state represents the protein's native shape.

The energy conformation, the proteins minimum energy state, of the protein is determined using energy functions. These functions determine the energy level based on the interaction between any pair of amino acids [15]. As a result, the protein folding problem can be reduced to an optimization problem, where the energy function is minimized under a set of constraints derived from known chemical and physical properties [9]. The result of reducing, simplifying, the protein folding problem to an optimization problem allows us to reduce some of its overall complexity. One simplification technique is to use lattice space models, which restrict the possible positions of the amino acids in space [12, 19, 18]. In this discrete space framework, a constraint solving technique can result in an effective solution [5, 3]. In work done previously [1], it was shown that highly optimized

constraints and propagates implemented in CLP allow the achievement of satisfactory performances on small to medium size instances, improving on the precision over previous models [5]. However, this method had some limitations in proving effective for larger instances of the protein folding problem [2]. This limitation of CLP-finite domain(FD) was overcome by using the lattice constraint programming framework in [3]. The work presented here extends the solvers ability to handle larger amino acid chains.

The ability to predict large protein structures becomes difficult due to computational complexity of the protein folding problem. In general, protein structure prediction with lattice models is NP-hard. NP-hard problems are one's where the algorithms grows too fast to expect to be able to compute exact solutions in all cases. Due to this complexity, there is no one solution that can be applied to all protein structures that will yield a valid result. Given this limitation, a significant improvement in the search algorithm can result in a speed up of the structure prediction process. Parallising a search algorithm can provide just such an improvement.

1.3 CONTRIBUTION

The work done here demonstrates the benefit of taking a sequential program and modifying the algorithms to create a parallel program. In this work, the advantage of converting the program to a parallel program has significant bene-

fits. The benefit can be seen in both short (35 amino acids) and long (249 amino acids) amino acid sequences. In regards to a short chain amino acid, processing demonstrates an one and a half times increase in efficiency, and more than a four and a half times speed up increase. Results are similar on a long amino acid chain, with a seventy nine times increase in efficiency, and a two hundred thirty eight times increase in speed up.

2 METHODOLOGY

In order to describe the methods used in the program, a few concepts will need to be explained about the constraint framework. The constraint framework solver defines lattice variables with associated domains, constraints over them, and a search of the space of admissible solutions.

In order to discuss the constraint framework, we need to discuss how the amino acids are abstracted into the lattice structure. First, the amino acids are considered as spheres and placed onto the side of the lattice cube (represented by $< x, y, z >$). A contact is defined when two nonadjacent amino acids are placed on two vertices of the cube (have a Euclidean distance of 2). Euclidean distance of two points, p_1 and p_2 is defined as, $\|p_1 - p_2\|$ where $\| < x, y, z > \| = \sqrt{|x|^2 + |y|^2 + |z|^2}$.

A domain ,D, used in the constraint framework is represented by a pair of lattice points $\langle \underline{D}, \overline{D} \rangle$, where $\underline{D} = (\underline{D}_x, \underline{D}_y, \underline{D}_z)$ and $\overline{D} = (\overline{D}_x, \overline{D}_y, \overline{D}_z)$. D defines a box where $Box(D) = \{(x, y, z) \in \mathbb{Z}^3 : (\underline{D}_x \leq x \leq \overline{D}_x) \wedge (\underline{D}_y \leq y \leq \overline{D}_y) \wedge (\underline{D}_z \leq z \leq \overline{D}_z)\}$. $\underline{D}$ represents the lower limits of the box, while $\overline{D}$ represents the the upper limits of the box. The defined bounds of Box(D) are used to determine whether or not a lattice point will be acceptable. The domain D is acceptable if Box(D) contains at least one lattice point. D is ground if its acceptable and $\underline{D} = \overline{D}$; D is empty (fails) if D is not acceptable. The domain is then searched

for possible positions, which are then checked against the constraints for validity.

The constraints used in the framework fall under the category of Constraint Satisfaction Problem (CSP). A CSP is defined as a triple $\langle V, D, C \rangle$, where V is a set of variables, D is a domain of values, and C is a set of constraints. V is defined as a point in the lattice and D will be the domain Box(D). For C, our constraints are defined as follows, the first constraint states that variables V will be no further than one lattice unit (Euclidean distance 2). The second constraint states that variables V will be most one lattice unit apart. This sphere will define the points that will be acceptable given the domain D. The data (variables, constraints, and the domains) used in CSP are stored in the constraint store data structure. This structure is a dynamic array and each variable has its own adjacency list that stores links to all the constraints, for increased efficiency. The next step is that the constraint solver takes the data that is stored, in order to begin the search process.

The constraint solver is used to apply the constraints above in order to sensibly prune the search results. The search results are found using constraint propagation and backtracking. Constraint propagation reduces the domains of the variables, by eliminating those that cannot lead to a constraint solution. For instance, we take $G = \{V_1, \ldots, V_{k-1}\}$ which have been bound to specific values, variable V_k (next variable to be assigned) and $NG = \{V_{k+1}, \ldots, V_n\}$ as the remaining variables. Once V_k has been labeled, its consistency is checked against

the constraints $C(V_k, V_j)$, where $V_i \in G$. The next step is the propagation phase, which is divided into two steps. First, all the constraints $C(V_k, V_j)$, where $V_j \in NG$ are processed. New bounds of V_k result from the processing, which are propagated to the variables that are not yet labeled. The data is stored using a combination of an array, data storage, and a flag (flag for being labeled) that are checked during propagation and backtracking.

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables, where all variables are initially unassigned. At each step a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned.

Now that the constraints have been set up, the search phase begins. The search results are pruned in one of the following ways. V_i is the variable that is in the process of finding a placement. The possible placement positions that can occur are dictated by the previous variables. If V_{i-1} is ground, then we have 12 possible placement positions for V_i (normal indomain processing). If V_{i-1} and V_{i-2} are ground, then the constraints limit the possible placements to 6 positions (fast indomain processing).

Now that the search tree has been generated, the next step is to consider

how the search algorithms perform their tasks.

2.1 OVERVIEW OF SEQUENTIAL METHOD

A flow diagram of the `search_best` algorithm can be found in Figure 1. This figure illustrates the components that get called in loops and how the search progresses.

The sequential approach of the program is to first load the necessary information to process the protein. Then data structures are created to hold the positions of the amino acids that will be placed in the lattice. The structures themselves are simple and just maintain information, such as placement, labeled status, and stack handling information. The program begins by placing the first amino acid and then checking the limits to begin placing all of the remaining amino acids in the sequence.

In processing the amino acids, the search method uses recursion as a means of storing information of where the search process is currently located in the amino acid sequence. Recursion is defined as a process or procedure that calls itself directly or indirectly. The search process uses the direct call method to process the amino acid sequence. When the search process completes and has placed the entire amino acid's sequence, the search exits with the completed tertiary structure information. Finally, the stored structure information is printed to a file, and some statistics are output to the user. The results can then be viewed

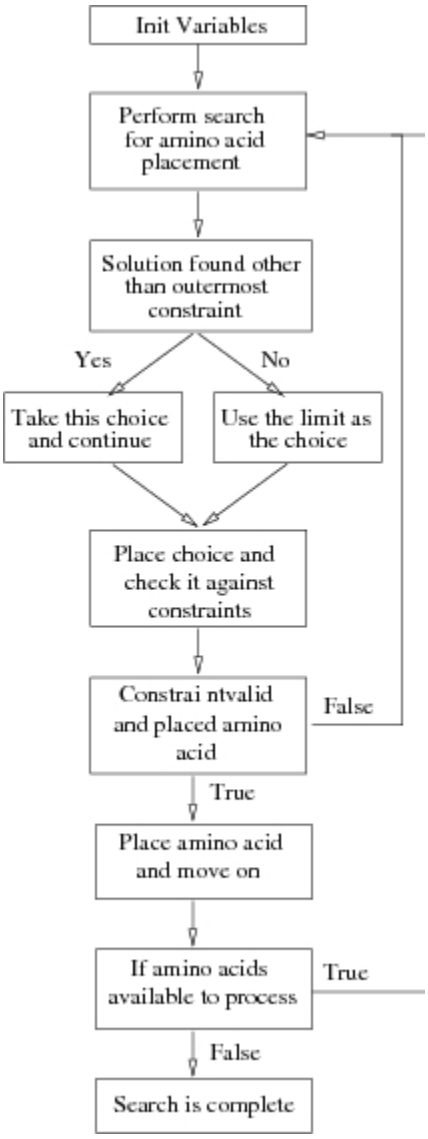

Figure 1: Flow diagram of original search algorithm

using a 3D modeling program to view the output[10].

The following Algorithms (1-6), demonstrate the how the sequential protein folding problem program completes the folding process for a given amino acid sequence, from reading in a protein, to printing out the amino acid's tertiary structure.

Algorithm 1 - main procedure pseudocode

```
1: procedure main( ):
2: load energy table
3: load protein
4: initialize the variables for processing the protein amino acid sequence
5: set the constraint method to be used
6: call search
7: end procedure
```

Algorithm 1 lists the `main` procedure of the program. This is the starting point of the program. The program will run all of the procedures listed in the `main` procedure and then will exit once it has completed. The code starts be first loading the energy table (line 2) followed by loading the sequence of the amino acid chain (line 3). Next, the variables for processing the amino acid are initialized (line 4). This includes the variables that will store the completed sequence and the constraints for processing the protein. Finally, a call is made using the `search` procedure (line 6). Once the call from the `search` procedure returns, the program terminates.

In Algorithm 2, the code initializes the data structure of the amino acid positions, that will later be determined by placement (line 2). The variable

Algorithm 2 - search procedure pseudocode

```
1: procedure search( ):
2: initialize the positions for amino acids
3: initialize the best_energy level to a big number
4: initialize & create the stack for backtracking and propagation of the search
   tree
5: initialize the lattice
6: initialize the constraint structure
7: call start_search
8: end procedure
```

best_energy is initialized to a large number (line 3), so that it will be replaced on the first calculation of the minimum energy state. A stack is created for the `search` procedure to support backtracking and propagation (line 4). A stack is a common data structure that works by only allowing insertions/deletions to occur at the top of the stack. Conceptually, it behaves like a stack of plates in a cafeteria. Plates are placed in a stack, where the size of the pile increase/decreases as plates are added/removed from the top of the pile.

The `search` procedure works by constructing a search tree, where the internal nodes correspond to the value of a variable (placement of an amino acid) with the edges corresponding to the propagation to the other variables (via the constraints). Variables are selected using two strategies. The first strategy, leftmost, selects the leftmost uninstantiated variable for the next labeling step. The second, first-fail, selects the variable with the smallest domain size (i.e. the box with the smallest lattice points) for the next labeling step. The next step is to initialize the lattice with initial directions (line 5) to allow for faster labeling (see

figure 2). The labeling is done to cut down on the search for possible placements that the amino acids can have. This is possible, due to the fact that amino acid labeling uses a pre-computed list of common torsional angles, based on the data extracted from the PDB information. Given a labeled amino acid, the search attempts to use the most common torsion angles in order to reduce the search time, if the torsional frequencies are ignored. Next, the constraint's data structure is initialized (line 6). Finally, the `start_search` procedure is called to setup the search to begin the protein folding procedure (line 7).

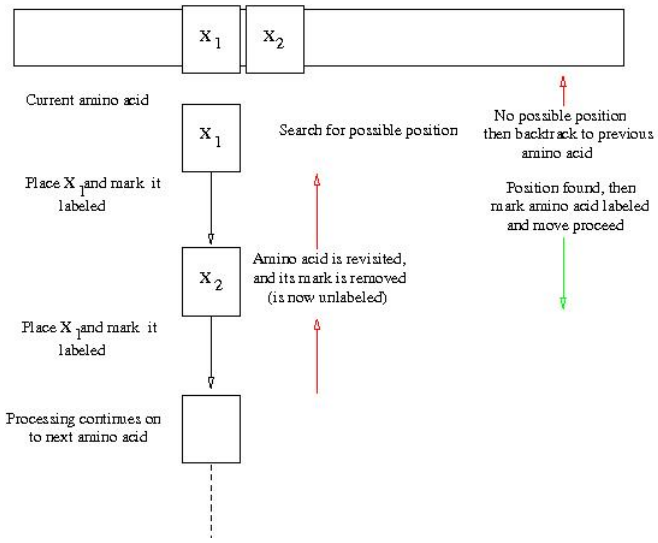

Figure 2: Labeling of an amino acid during the search process

Algorithm 3 sets up some variables before the actual search process begins. First, the start time for the search is initialized in line 2. Next, the `search_best` procedure is called. This procedure is where the actual searching for the amino acids takes place. The procedure is called with a parameter, that specifies at which

position in the amino acid chain to begin. In this case, 1 is passed in, to start the search process with the first amino acid. The `search_best` procedure returns once it has successfully folded the protein and stored its structure. An end time is recorded for the procedure (line 4) so that a total runtime can be computed by taking the difference of the end time from the start time. The results of the search (folding procedure) are written to a file (line 5). Among other things, the file contains protein structure information that can be used in a visualization program such as chimera. Finally, a brief summary of results is printed (line 6).

Algorithm 3 - start_search procedure pseudocode

```
1: procedure start_search( ):
2: initialize the start time
3: call search_best(1)
4: record the end time
5: write output to the file
6: print summary information
7: end procedure
```

The following Algorithms, 4-6, describe the search algorithm that is executed for processing the primary structure in order to determine its tertiary structure. These algorithms are where the program spends the majority of its execution time. They are responsible for sequence processing and amino acid position placement (see figure 3).

The placement algorithm can be easily understood given an example. For instance, the placement for the search algorithm uses the upper and lower constraints for the amino acid to be placed. The limits of placing the amino acid are

Algorithm 4 - search_best procedure pseudocode

```
1: procedure search_best( amino_acid_position ):
2: if amino_acid_position is a leaf then
3:    calc_energy = calculate current energy
4:    if calc_energy < best_energy then
5:       set positions to currently found position
6:       save best solution state
7:    end if
8:    return
9: end if
10: put current amino acid into stack
11: if left two amino acids are labeled for fast indomain then
12:    fast_indomain ← true
13: end if
14: if right two amino acids are labeled for fast indomain then
15:    fast_indomain ← true
16: end if
17: if fast_indomain = true then
18:    search past labeled amino acids to place next protein
19: else
20:    while done ≠ true do
21:       if check_position is valid then
22:          done ← true
23:       else
24:          component_x ← component_x + 1
25:          if component_x > upper_limit then
26:             component_x ← component_x_lower_limit
27:             component_y ← component_y + 1
28:          end if
29:          if component_y > upper_limit then
30:             component_y ← component_y_lower_limit
31:             component_z ← component_z + 1
32:          end if
33:          if component_z > upper_limit then
34:             current_position is invalid
35:             done ← true
36:          end if
37:       end if
38:    end while
39: end if
```

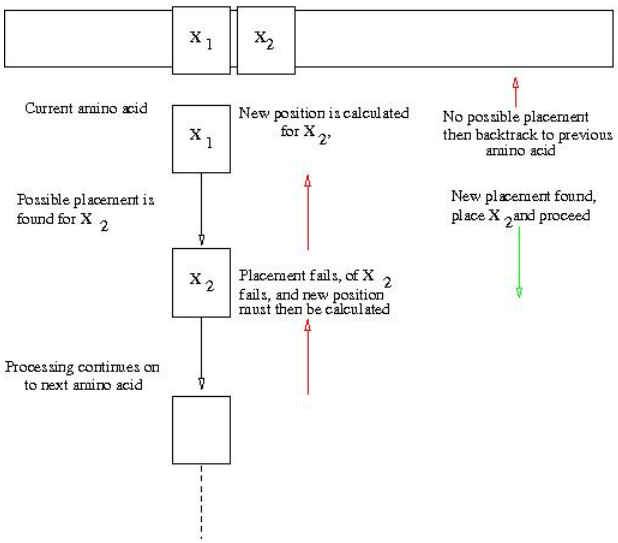

Figure 3: Amino acid placement during the search process

constrained by an upper limit of (126,127,126) and a lower limit of (124,125,124). An initial position will be set to the lower limit, (124,125,124). Assuming that the protein has not been labeled, and is being looked at for the first time, the placement process will try the first choice, (125,125,124) having incremented the x component for its first attempt and checks its validity. The check results in a valid choice, thus allowing the algorithm to proceed with the position placement of (125,125,124). Had the validity returned false, the next placement position would have been (126,125,124) again incrementing the x component, until the upper limit is reached. The placement process is discussed in more detail in Algorithm 4.

In Algorithm 4 line 1 the passed in parameter, `amino_acid_position`, tells the procedure the position in the amino acid chain to begin processing. Once the

search procedure starts, the amino acid is checked to see if it is a leaf (line 2). Leaves are finished sequences that have folding energy associated with them. The code that identifies the node as a leaf, works as the base case of the recursion for the `search_best` procedure. Line 3 calculates the energy of the folded amino acid sequence. Next, the calculated energy is checked against the current minimum energy value (line 4). If line 4 results in a true statement, the calculated energy is less than the current minimum, which results in positions of the current conformation being saved. Once the conformation has been saved, the state of the current fold is saved (line 6).

If the node is not a leaf, then the procedure jumps to line 10, and the amino acid is placed onto the search stack. The current amino acid is checked to see if it can be computed via fast indomain. The amino acid is marked for fast indomain if its two left amino acids have already been labeled (line 11) or if its two right amino acids have already been labeled (line 14). If the fast indomain flag is set (line 17) then the placement for this amino acid is done using the information of the surrounding amino acids (line 18).

When fast indomain is not selected, the normal indomain approach is followed (line 19). Normal indomain uses the information from the previous amino acid in its placement calculation. The first step in the search loop (lines 20-38) is to check for a valid position (line 21). If a valid position is found, then the loop is terminated (line 22). If the position is not valid, then the search for a valid

position continues (line 23). The search for a valid position is done by incrementing component x (line 24) until its upper limit is reached (line 25). Once the upper limit is reached, the component's value is reset to the lower limit (line 26) and the next component, in this case component y, is incremented. The same pattern is followed for component y (lines 29-31). In the case of component z, when the upper limit is reached we mark the current search as invalid (line 34) and set the loop variable to true in order to terminate the loop (line 35). As a result, the search is marked invalid, since there was no possible placement for the amino acid and the procedure continues in Algorithm 5.

Algorithm 5 continues the `search_best` algorithm. In this part of the `search_best` procedure, a while loop is used to process the search for placing an amino acid (line 1). The first check that is performed, is to determine whether the search will happen via normal indomain (line 2). If line 2 is a true statement, then normal indomain processing is performed (lines 3-16). If the flag in Algorithm 4 line 34, is set for an invalid position, then the loop terminates (line 4). Given a possible position from Algorithm 4 line 22, the position is saved in line 6 as a possible solution. The possible solution is checked. The code for checking the possible position occurs in lines 7-15.

If fast indomain is found in line 2, then the code jumps to line 18, and is processed with the fast indomain position found. The first step is to store the position found from Algorithm 4 line 18. Next, the variable is checked to determine

Algorithm 5 - search_best procedure pseudocode cont.

```
1: while go_on ≠ true do
2:   if normal_indomain = true then
3:     if position is invalid then
4:       go_on ← true
5:     else
6:       real_position ← check_position
7:       while done ≠ true do
8:         if check_position is valid then
9:           same as lines 22-37 in Algorithm 4
10:        else
11:          if check_position is valid then
12:            done ← true
13:          end if
14:        end if
15:      end while
16:    end if
17:  else
18:    real_position ← fast_position
19:    while done ≠ true do
20:      if checked_variables > 6 then
21:        go_on ← false
22:      else
23:        fast_position = adjoining amino acid (left or right as found previously)
24:        if position is within component limits then
25:          done ← true
26:        end if
27:      end if
28:    end while
29:  end if
30:  { ... continued in Algorithm 6 }
31:  backtrack to previous state
32: end while
33: end procedure
```

if more than six positions for fast indomain placement (line 20) have been searched. If this is the case, then the search loop (set in line 21) of Algorithm 6 lines 1-12 do not get executed, and the search backtracks to the previous amino acid. This is a result of having exhausted all possible positions. If possible positions still exist, then the algorithm proceeds to check for a possible position using the adjoining amino acids (line 23). If the position is within the component limits for x, y, and z, then the loop is terminated (line 25) with the position found on line 23. Algorithm 6 has been moved in order to better illustrate the entire `search_best` procedure from Algorithm 4. The last line in Algorithm 4 line 31 is used to backtrack to the previous state.

Algorithm 6 - search_best procedure pseudocode cont.

```
 1: if go_on ≠ true then
 2:    check current amino acid against constraints
 3:    if amino acid is valid then
 4:       set amino acid and add to backtracking stack
 5:       if rigid_block = true then
 6:          label and put down the amino acids for the block and move on
 7:          call search_best(current amino acid + block size)
 8:       else
 9:          call search_best(current amino acid + 1)
10:       end if
11:    end if
12: end if
```

Algorithm 6 completes the `search_best` algorithm. The first check that is performed, is to determine if the search process should continue (line 1). Next, the constraints for the amino acid are checked (line 2). The validity of the constraint

is then checked in line 3 for a valid position. The search places the amino acid and adds it to the backtracking stack (line 4). The algorithm then checks to see if the amino acid being placed is the start of a rigid amino acid block (line 5). Secondary structures, such as helixs and beta-sheets are considered rigid block , and must be placed as a whole. The placement of a rigid block is done with a recursive call to `search_best` with the block size plus current amino acid as a parameter (lines 6-7). The parameter to this recursive call will move past the position of the rigid block of amino acids that was found, and will continue the search process immediately following the rigid block of amino acids. If no rigid block is found, then the parameter to the recursive call `search_best` is the next amino acid in the sequence (line 9).

2.2 OVERALL APPROACH TO PARALLISATION

At first glance, parallelizing the code, could be done just by calling multiple instances of the code to achieve parallelism. However, this approach has many problems. The first being that nothing has been changed in the program to synchronize searching, other than having multiple searches executing. The other is a direct result of the first, there is no mechanism setup to prevent the duplication of work. A parallel implementation of the sequential protein folding program would take these issues into consideration.

The first problem that must be addressed is that the parallel implemen-

tation of the sequential program must have its processes synchronized. This is done to ensure that the same work is not given to more than one process. Another aspect which must be considered is, the data the processes are given can not overlap. The problems described above can be considered as tasks, where each task is responsible for a particular amount of processing. This task breakdown has similarities that can be found in a pattern for parallel programming, the task pattern. The task pattern is where tasks are either independent of each other or in a situation in which there are some dependencies among the tasks in the form of access to shared data. The task pattern can be implemented using different supporting structures.

There supporting structure designs are SIMD, loop parallelism, fork/join, and master/worker. SIMD (single instruction multiple data) is a program design where all of the processes execute the same program in parallel, but each has its own set of data. This supporting structure design will not work with the approach of creating multiple tasks for parallel computation. Loop parallelism is used when a runtime is dominated by set of compute-intensive loops where different iterations of the loop are executed in parallel. The protein folding problem has many iterations of the search loop, but the iterations of the search loop are based on the consecutive placement of the amino acid. As a result, loop iterations are dependent on each other. Fork/Join uses a main process, and forks off other processes. All the processes will continue in parallel to accomplish some portion

of the overall work that must be completed. When complete, the forked processes terminate, and then join back up. This structure design will not work either, due to the fact that there is no distinct way of separating the data that is searched by each fork. The remaining structure design to consider is, master/worker.

Master/Worker sets up a pool of worker processes with a bag of tasks that need to be performed. The workers execute concurrently, where each worker removes a task from the bag of tasks and processes it. This continues until all tasks have been processed or some other termination condition is reached. This structure design will work given the parallel processing needs. The master process is used to create worker processes and assign them tasks. By using a master process, it will ensure that no two tasks will process the same data, thereby avoiding work duplication. This structure fulfills the parallel processing needs identified earlier. Another advantage that the master/worker design provides is control, that no two processes work on the same position. The master process will search for possible positions and create a worker process for each possible position it finds. The master process' ability to delegate tasks will ensure that our second problem, data overlap, will be prevented. The master/worker process scheme can be seen in figure 4. This scheme gives us the benefit to create new processes in the most crucial portions of the protein folding program, searching the other possible placements.

The most crucial segments of the protein folding program for creating new

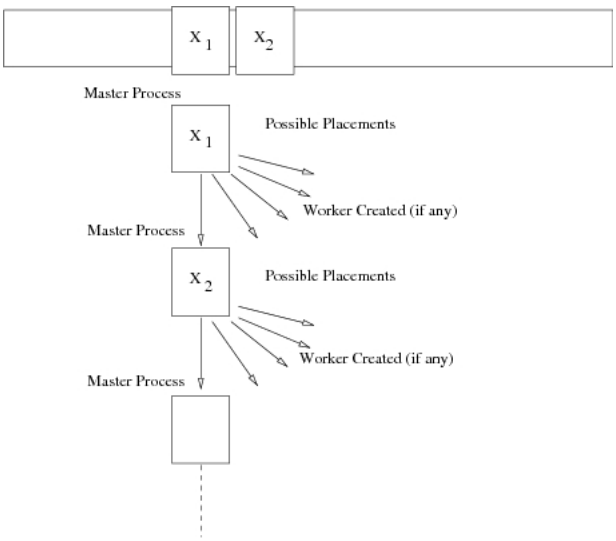

Figure 4: Master/Worker scheme during position search

processes is in the search algorithm. Given the arguments for a parallel approach figure 5 gives a modified flow diagram of the sequential code incorporates the changes necessary to parallise the program.

2.3 PARALLISATION IMPLEMENTATION

The following Algorithms, 7-11, illustrate the modifications (highlighted in gray) made to the sequential code in order to incorporate the parallel computing mechanism.

Algorithm 7, shows the changes that are made to Algorithm 3. The first change needed is to create and initialize the shared memory variables that are going to be used (line 3). The master process has its process id number stored in line 4. The next difference is the semaphore that is created and initialized in

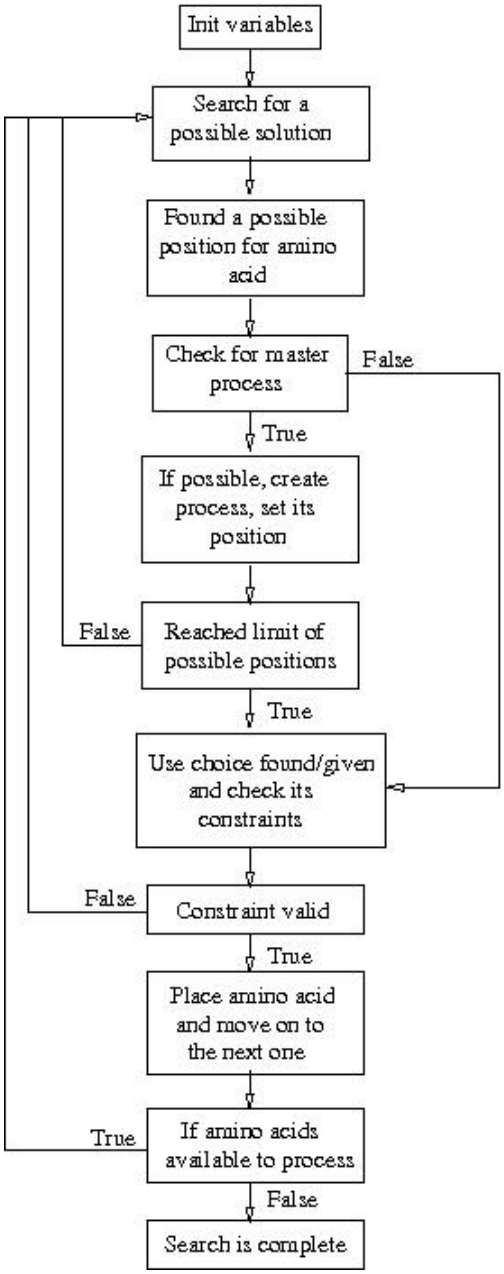

Figure 5: Overview of the parallel search algorithm

Algorithm 7 - start_search function - parallel pseudocode

```
1: procedure start_search( ):
2: initialize the start time
3: create and initialize shared memory for parallel processing
4: set process id of master
5: initialize the semaphore
6: call search_best(1)
7: get the end time
8: write output to the file
9: print summary information
10: remove the semaphore
11: end procedure
```

line 5. A semaphore is a shared memory programming convention that is used to control access to the segments of shared memory. One of the shared memory segments that is created, is a counter used for keeping track of available processes. This counter is incremented/decremented as child processes are created/destroyed. The other memory segment that is created, is a variable used to broadcast a found solution and a variable to identify the process that found the solution. Finally, the semaphore and memory segments that were created are removed on line 10.

Algorithm 8 illustrates the modifications made to Algorithm 4 needed to implement the parallel design. The new changes that were made, are highlighted to better illustrate the differences. The first difference, is that the algorithm checks to see if a possible solution has been found (line 2). If a solution has already been found, possibly be another process, then the search is terminated (line 3). If no solution was found, then the algorithm continues the search process. The next major difference in the algorithms can be found in lines 12-27.

Algorithm 8 - search_best procedure - parallel pseudocode

```
1: procedure search_best( amino_acid_position ):
2: if found_solution = true then
3:    return
4: end if
5: if amino_acid_position is a leaf then
6:    {...}
7: end if
8: {...}
9: if fast_indomain = true then
10:   search past labeled amino acids to place next protein
11: else
12:   if process_parameter > 1 & is_master_process = true then
13:      backup_position ← current_position
14:      if current_position is valid then
15:         master_position ← current_position
16:      else
17:         master_position ← 0
18:      end if
19:      while done ≠ true do
20:         if check_position is valid then
21:            if found_master_position ≠ true then
22:               master_position = check_position
23:            else
24:               if available_process < process_parameter then
25:                  get semaphore
26:                  available_process ← available_process − 1
27:                  release semaphore
28:               end if
29:            end if
30:         end if
31:         { ... continued in Algorithm 9 }
32:      end while
33:      { ... continued in Algorithm 10 }
34:   else
35:      { ... continued in Algorithm 11 }
36:   end if
37: end if
```

The first condition that is checked in line 12 is to determine if the program has been called with more than one process to run. This check is done to avoid unnecessary checks, when the program is called with only one process. The next condition checks to see if the current process is the master process, since the master process is the only process allowed to create new processes. If both of these conditions are met, then the following new algorithm sections are executed. The first is to backup the current position (line 13). This is done so that this position can be recovered if the placement algorithm fails. Next, the current position is validated (line 14). If valid, then the position is saved for the master to use (line 15). If not, then the position for the master is cleared (line 17). Once the position for the master has been set, the search for putting them into the task bag begins.

The position search is done by the master process, since it will be creating the worker processes. The search starts with checking and validating the current position (line 20). If the position is valid, and the master process has not been assigned a position (line 21), then master process is given this position (line 22). The master process is given the first valid position, since it must continue the search if no other possible positions, other than the current one, are found. If the master process has been assigned a position, then the master process checks for available process space (line 24). Available process space is determined by the shared memory variable that tracks the number of available slots for computa-

tion. If there is a slot available, the master process gets the semaphore (line 25). Once the master process has the semaphore, the available number of processes is decremented (line 26), and then releases the semaphore (line 27).

The remaining pieces of Algorithm 8 are split up into two. Algorithm 9 would be inserted on line 31 and Algorithm 11 would be inserted on line 34. This is done to demonstrate the placement of the remaining algorithm segments in the `search_best` procedure.

Algorithm 9 demonstrates the search process that the master process follows in order to find possible positions. The first step done, is to check if the last component modified was invalid (line 1). The components are searched in a circular list fashion, $x \rightarrow y \rightarrow z \rightarrow x$. A circular list is a list where the last element in the list refers to the first element as its next element. If no components have been modified (line 1), then the first component that will be modified is component x (line 2) and the last modified variable will be x (line 3). Similarly, the last modification, update next component, and save last component are done for components y (lines 7-9) and z (lines 10-12).

After the proper component has been modified and last modified is saved, the position is checked. The first step is to check if component x has exceeded its upper limit (line 14). If the upper limit of component x is exceeded, then its value is reset to the lower limit of component x (line 15), and then begins to modify component y (line 16), and saves the last modified as component y (line

Algorithm 9 - search_best procedure - parallel pseudocode cont.

```
1: if last_modified = invalid then
2:     component_x ← component_x + 1
3:     last_modified = x
4: else if last_modified = x then
5:     component_y ← component_y + 1
6:     last_modified = y
7: else if last_modified = y then
8:     component_z ← component_z + 1
9:     last_modified = z
10: else if last_modified = z then
11:     component_x ← component_x + 1
12:     last_modified = x
13: end if
14: if component_x > upper_limit then
15:     component_x ← component_x_lower_limit
16:     component_y ← component_y + 1
17:     last_modified = y
18: end if
19: if component_y > upper_limit then
20:     component_y ← component_y_lower_limit
21:     component_z ← component_z + 1
22:     last_modified = z
23: end if
24: if component_z > upper_limit then
25:     component_z ← component_z_upper_limit
26:     done ← true
27: end if
```

17). Components y (lines 19-22) and z (lines 24-26) are processed in a similar fashion. However, component z modifications differ slightly in that once its upper limit is reached (line 24), its value is reset to the lower limit (line 25), and then terminates the loop (line 26).

Algorithm 10 - search_best function - parallel pseudocode cont.

```
1: if master_process = true then
2:    if found_master_position ≠ true then
3:       check_position ← master_position
4:    else
5:       check_position ← backup_position
6:    end if
7: end if
```

The code in Algorithm 10 checks to see if the current process is the master process (line 1). The master process checks to see if it has already found a position to use (line 2). If no position has been found, then the previously found master position, in Algorithm 8 line 22, is used to proceed with the search. If a position has been found, then the master will restore its backup position to the check position, so that it can be processed later.

The final piece of the `search_best` procedure is shown in Algorithm 11. The most significant change that is made here, is to check if a solution has been found by another running process (lines 11-15). If this is true, then the process will exit. The semaphore is taken (line 12), and the flag to broadcast a found solution is set (line 13). Next, the semaphore is released and the process exits back to Algorithm 7 line 7. Otherwise, the process will continue the search for

Algorithm 11 - search_best function - parallel pseudocode cont.

```
1: while go_on ≠ true do
2:    if normal_indomain = true then
3:       {...}
4:    end if
5:    {...}
6:    if solution_found = true then
7:       return
8:    end if
9:    backtrack to previous state
10: end while
11: if amino_acid_position = 1 then
12:    get semaphore
13:    broadcast that a solution was found
14:    release semaphore
15: end if
16: end procedure
```

possible amino acid locations. The last code change is used to check for a possible solution. If a solution is found, then the search tree will have found all leaf nodes, placed the entire amino acid sequence, and have emptied the search stacks without a failure. Having removed the stack, the program will be at the first amino acid position, and the check will return a valid result.

3 EXPERIMENTS

The experiments were performed on a multi–user system with shared processes from other users. The experimental test setup was the same for both software versions (original and parallel code base).

3.1 EXPERIMENTAL SETTING

The hardware used was a Sun Fire T1000, with 8GB of main memory, and 32 core processors running at 1GHz each. The software was compiled using the Open Source GCC compiler (ver. 3.4.3) on Solaris Operating System (ver 5.10). Both programs were executed in the same manner from the command line. The only exception is that the parallel code requires an additional parameter to specify the total number of allowable processes that can be created by the master process at any moment during program execution.

3.2 BENCHMARK DESCRIPTION

In order to determine the increase in efficiency and speed up of the parallel program, a run of the original code is needed for comparison. The sequences used for testing are those included with the sequential protein folding program, 1KON and 1ZDD. An added benefit is that these proteins were tested with the sequential protein folding program. These two sequences represent good choices for amino acid sequence lengths, 1ZDD at 35 amino acids, and 1KON at 249 amino acids.

Each protein was run three consecutive times, in order to compute an average runtime. The runtime is determined internally be the program.

Table 1 list the runtimes that were found for the original program running the two proteins.

Table 1: Runtime of original code

Protein	System Time (seconds)
1ZDD	167.94
1KON	17430.02

3.3 EXPERIMENTAL RESULTS

The parallel code was run in a similar fashion to the sequential protein folding program. Each runtime was calculated as an average of three consecutive runs, for each parameter. For instance, parameter 4 (specifying the maximum number of processes), was run three times, an average was calculated, and recorded for this parameter.

The following tables list the runtimes for the parallel code, with respect to each protein. The first table, Table 2, lists the runtime of the parallel code when run for protein 1ZDD. This table shows the runtime for available processing slots, from 1 to 9 available slots for computation.

The next table, Table 3, displays the results of the parallel code on protein 1KON. This table displays the longer runtime that result from processing larger

Table 2: Runtime of parallel code (Protein 1ZDD)

Num of Processes	System Time (seconds)
1	169.32
2	166.79
3	36.11
4	36.23
5	36.82
6	37.12
7	37.75
8	39.03
9	39.15

amino acid chains; in this experiment, 1KON is used.

Table 3: Runtime of parallel code (Protein 1KON)

Num of Processes	System Time (seconds)
1	17510.65
2	17392.74
3	73.13
4	74.67
5	80.68
6	86.72
7	105.50
8	106.57
9	105.40

The last table, Table 4, gives a comparison of the runtimes of the original code, and the parallel code on both proteins. Speed up, and efficiency are included in the table for both versions of the code.

Table 4: Comparison of runtimes

Protein-Code Base	Original	Parallel[1]	Speed up[2]	Efficiency[2]
1ZDD	167.94	36.11	4.65	1.55
1KON	17430.02	73.13	238.34	79.44

The following figures 6 - 11, display the resulting structures. Figure 6, displays the protein 1ZDD structure that results from the sequential code, and figure 7 displays the results from the parallel code. Figure 8 is an overlay of figures 6 and 7, which illustrates that both versions of the code result in the same structure.

Figure 9, displays the protein 1KON structure that results from the sequential code, and figure 10 displays the results from the parallel code. Figure 11 is an overlay of figures 9 and 10, which illustrates the different structures produced. The structure in white is the sequentail code and the pink is the parallel code. The difference in structures can be explained in how the program model determines a valid structure. The model creates a search tree that contains many possible solutions. In the parallel code with multiple processes, each process will search one branch until it finds a solutions or no solution is found. The search tree has varying depths to it. It is this attribute of the search tree that lead to the possibility of a 3D protein structure solution being on a shallow branch in the search tree (as seen in protein 1KON).

[1]number of processes 3 used for comparison

[2]defined on p.160[17]

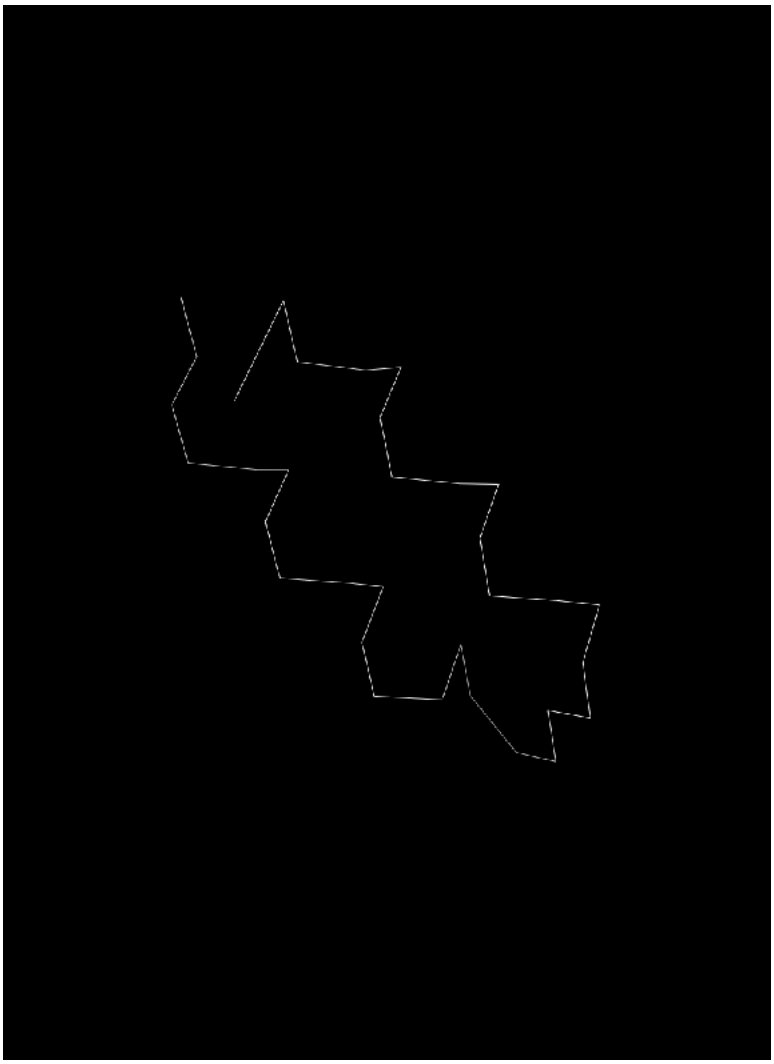

Figure 6: 3D view of 1ZDD (sequential)

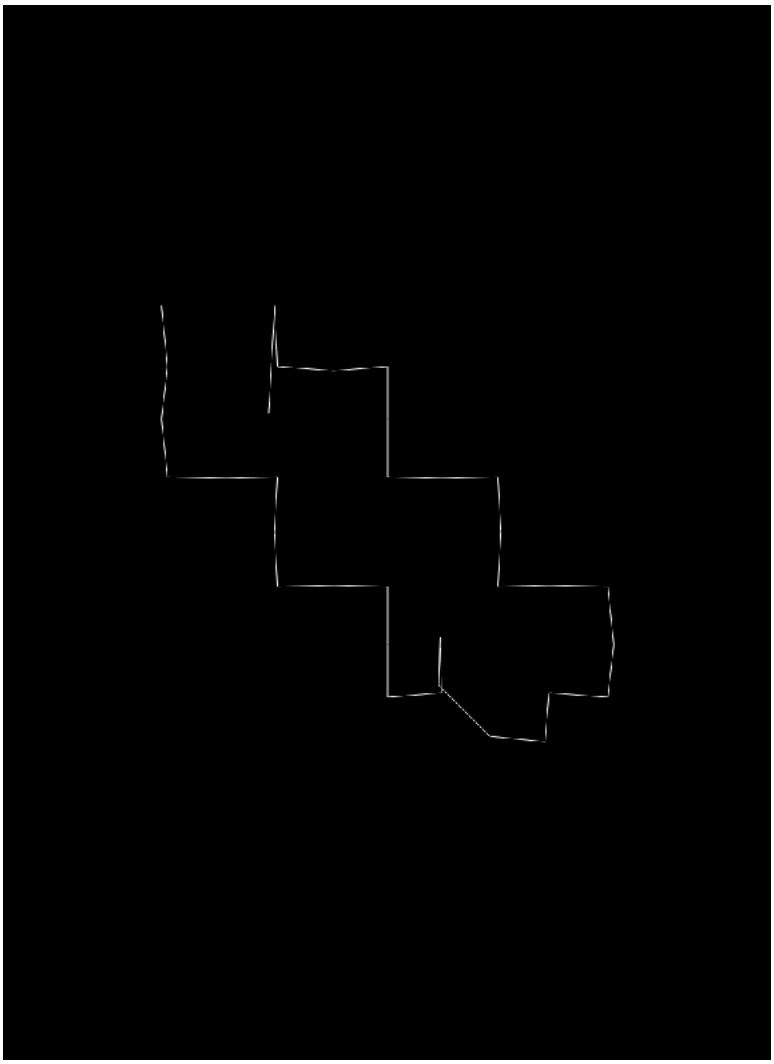

Figure 7: 3D view of 1ZDD (parallel)

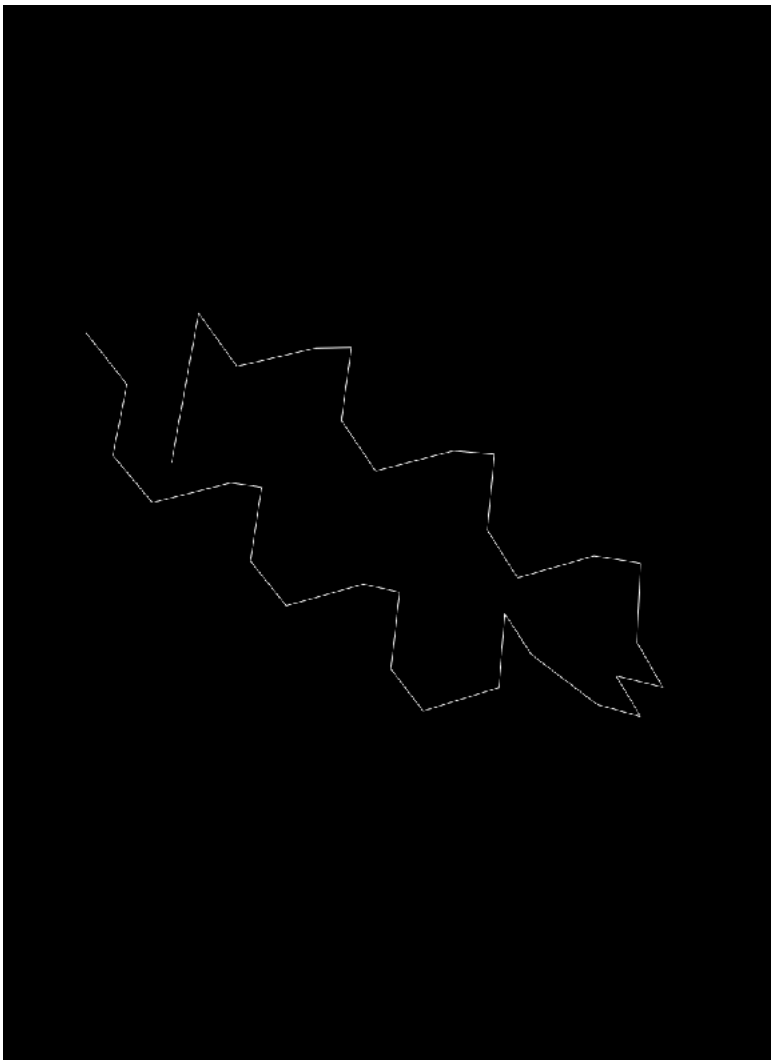

Figure 8: 3D view of 1KON (overlay)

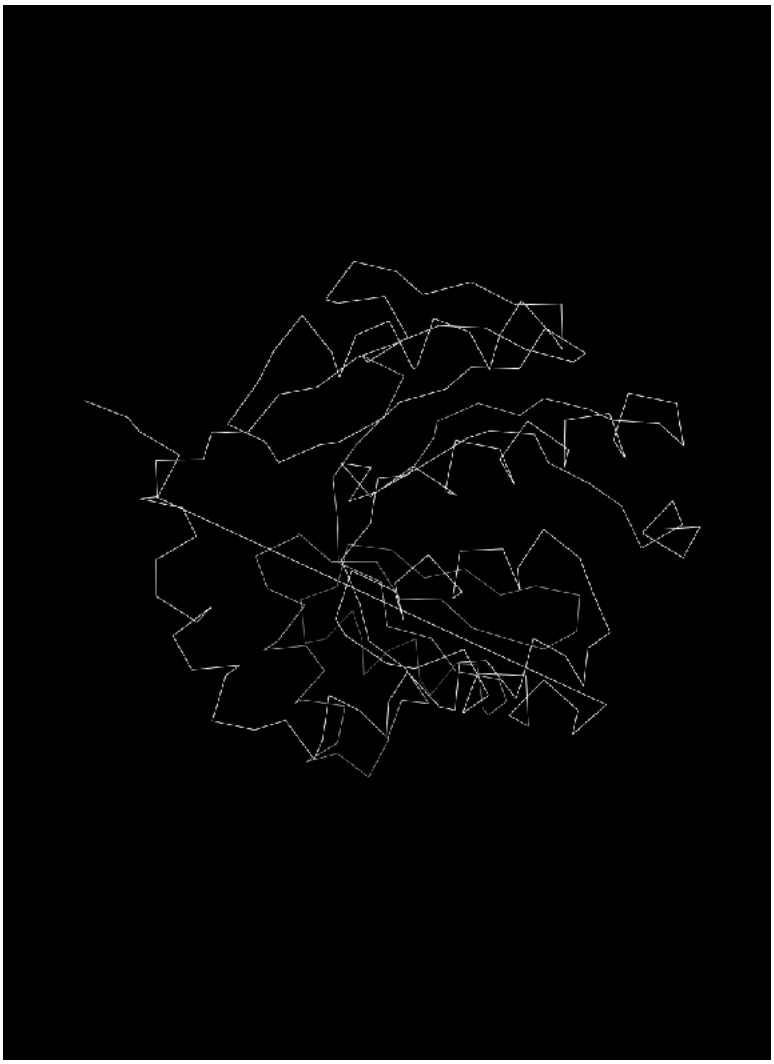

Figure 9: 3D view of 1KON (sequential)

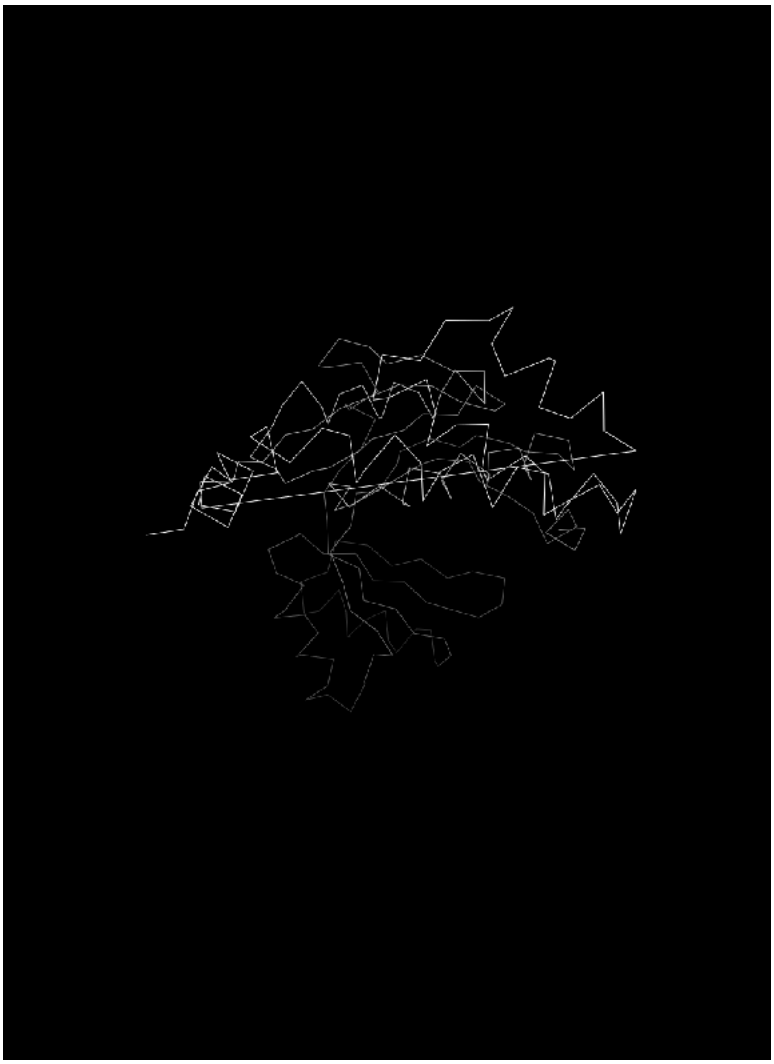

Figure 10: 3D view of 1KON (parallel)

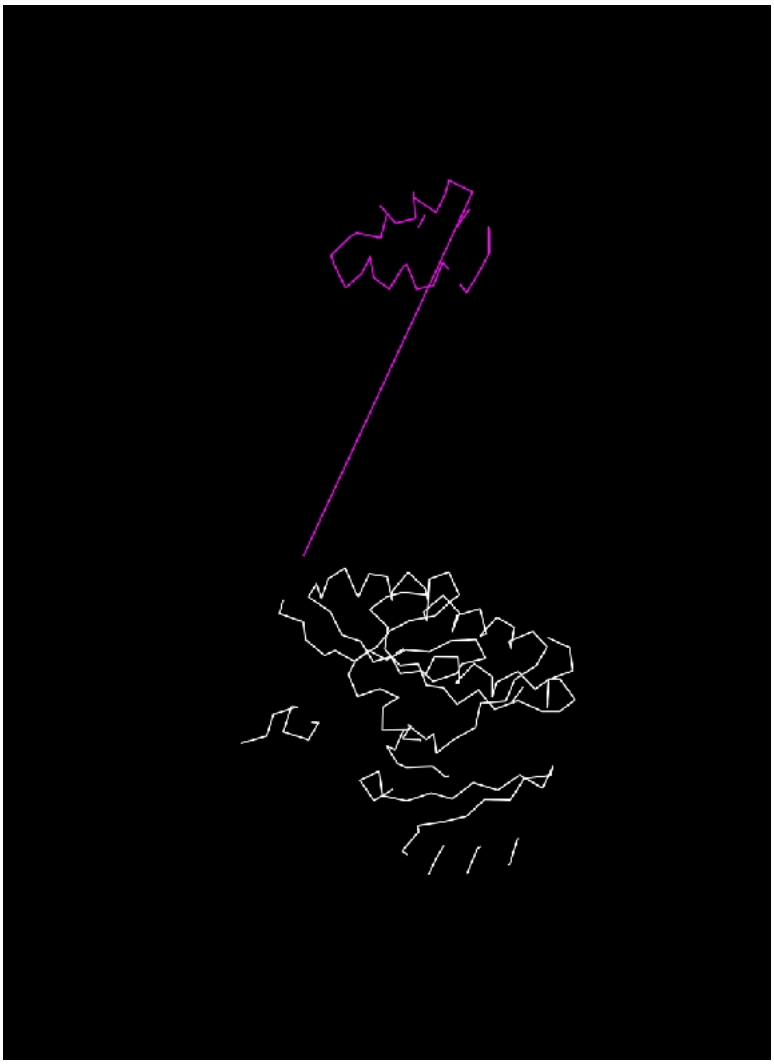

Figure 11: 3D view of 1KON (overlay)

3.4 EVALUATION & DISCUSSION

Table 1 illustrates how the differences in runtime for the proteins vary. It also demonstrates how different length amino acids effect the runtime of the sequential code.

Tables 2 and 3, list the runtimes of the parallel code for 1ZDD and 1KON respectively. When comparing the runtime of 1ZDD from Table 1 to the results from Table 2, the runtime results are similar in the case where the number of processes is 1 and 2. Similar results are found for protein 1KON in Tables 1 and 3. The similarities of runtimes changes when the number of processes is increased to 3.

After the number of processes is increased to 3, the runtime of the program drops dramatically. Comparing the runtime of 1ZDD in Table 1 to Table 2, a speed up of 4.6 occurs in the parallel implementation. The same occurs for protein 1KON. Comparing the runtime of 1KON in both Tables 1 and 3, a speed up of 238.3 occurs in the parallel implementation. This decrease in the runtimes, demonstrates that the parallel implementation of the program is finding a solution for the tertiary structure of the protein. An interesting thing to note is that, after parameter 3, the runtimes begin to increase in Tables 2 and 3. This trend can be a result of an increase in tracking processes as the number of slots for computation increases. Despite the increase in runtimes, they are still significantly less then

the runtimes found in Table 1.

Table 4 summaries the comparison of the sequential protein folding code, and the parallel version. The runtimes for the sequential protein folding code are taken from Table 1 and the runtimes for the parallel version are taken from Table 2 and 3. For the parallel version, the runtime listed for number of processes 3, is used. The speed up and efficiency are listed to shown how much of an improvement the parallel version is over the sequential protein folding code.

4 CONCLUSION & FUTURE WORK

In conclusion, this thesis presented a parallel version of a sequential program for solving the protein folding problem in a crystal lattice representation of the 3D space. The parallel version has been implemented in the search algorithm to maximize the potential for parallelism. The parallel version has demonstrated an increase in speed up and efficiency, given varying length amino acids. Given the performance increases in the parallel code, the program can handle longer amino acid sequences with respectable runtime. The current trend in computer hardware is heading towards using multi-core computing systems. The parallel version presented in this thesis can begin to take advantage of this trend.

For future work, one proposed method would be to explore a threading model implementation. This model would allow for better control of the each individual process, thus providing a possible performance increase [16].

REFERENCES

[1] A. Dovier A. Dal Palù and F. Fogolari. Constraint Logic Programming approach to protein structure prediction. *BMC Informatics*, 2004.

[2] A. Dovier A. Dal Palù and E. Pontelli. Heuristics, optimizations, and parallelism for protein structure prediction in clp(fd). In *PPDP*, 2005.

[3] A. Dovier A. Dal Palù and E. Pontelli. A new constraint solver for 3D lattices and its application to the protein folding problem. In *LPAR*, pages 48–63, 2005.

[4] K.R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[5] R. Backofen. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints*, 6(2–3):223–255, 2001.

[6] Lesk A Bashford D, Chothia C. Determinants of a protein fold. unique features of the goblin amino acid sequences. *Journal of Molecular Biology*, 196:199–216, 1987.

[7] Feng Z Gilliland G Bhat TN Weissig H Shindyalov IN Berman HM, Westbrook J and Bourne PE. The protein data bank. *Nucleic Acdis Res*, 28:235–242, 2000.

[8] Naval Research Labs Center for Computational Materials Science. Crystal lattice structures.

[9] P. Clote and R. Backofen. Computational molecular biology. *John Wiley & Sons*, 2001.

[10] C.C. Huang G.S. Couch D.M. Greenblatt E.C. Meng E.F. Pettersen, T.D. Goddard and T.E Ferrin. Ucsf chimera – a visualization system for exploratory research and analysis. *J. Comput. Chem. 25*, 25(13):1605–1612, 2004.

[11] J. Skolnick et al. Reduced models of proteins and applications. *Polymer*, 45:511–524, 2004.

[12] R. Agarwala et al. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the hp model. *Journal of Computational Biology*, pages 275–296, 1997.

[13] Skolnick J Godzik A, Kolinski A. Lattice representation of globular proteins: how good are they? *Journal of Computational Chemistry*, 14:1194–1202, 1993.

[14] W. Hart and A. Newman. The computational complexity of protein structure prediction in simple lattice models. *CRC Press*, 2003.

[15] H. Molinari M. Berrera and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Informatics*, 4:8, 2003.

[16] Proceedings of the 2008 conference on Computing frontiers. *Improving single–thread performance with fine-grain state maintenance*, 2008.

[17] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004. pages 160.

[18] Kolinski A Skolnick J. Reduced models of proteins and their applications. *Polymer*, 45:511–524, 2004.

[19] L. Toma and S. Toma. Folding simulation of protein models on the structure on the structure–based cubo–octahedral lattice with contact interations algorithm. *Protein Science*, 8:196–202, 1999.

[20] Y. Zhang. Progress and challenges in protein structure prediction. *Curr Opin Struct Biol 18*, 3:342–348, 2008.