## ABSTRACT

# PARALLEL EXACT ENUMERATION OF SELF-AVOIDING WALK ON CUBIC LATTICES AND ITS APPLICATIONS TO PROTEIN FOLDING STUDIES

by
Anek Vorapanya

Exact enumeration of self-avoiding walk on many lattices have been studied extensively recently. Even a short chain polymer (about 30 monomers) represented as a chain of cubic lattice sites requires a considerable amount of computer time to exhaustively search for all unique conformations. However, self-avoiding walk process can be modified such that it exhibits a high degree of independence among subprocesses. Parallel implementation of such subprocesses can reduce a great amount of enumeration time. Parallel enumeration makes longer chain enumeration possible.

Enumerating only unique conformations requires that all rotation and mirror conformations be removed. An algorithm to avoid generating such symmetrical conformations is presented. A set of parallel algorithms to solve exact enumeration of cubic lattice graphs subjected to various constraints (volume and/or contact constraints) is presented. The speed up and communication cost are analyzed. One of the most important application of lattice enumeration, enumerative kinetics of protein folding, is also discussed.

# PARALLEL EXACT ENUMERATION OF SELF-AVOIDING WALK ON CUBIC LATTICES AND ITS APPLICATIONS TO PROTEIN FOLDING STUDIES

by
Anek Vorapanya

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

May 1994

APPROVAL PAGE

# PARALLEL EXACT ENUMERATION OF SELF-AVOIDING WALK ON CUBIC LATTICES AND ITS APPLICATIONS TO PROTEIN FOLDING STUDIES

Anek Vorapanya

Dr. Lonnie Welch. Thesis Advisor	Date
Assistant Professor of Computer and Information Science, NJIT

Dr. James A. M. McHugh. Committee Member	Date
Professor of Computer and Information Science, NJIT

Dr. David Nassimi. Committee Member	Date
Associate Professor of Computer and Information Science, NJIT

## BIOGRAPHICAL SKETCH

**Author:** Anek Vorapanya

**Degree:** Master of Science in Computer Science

**Date:** May 1994

**Undergraduate and Graduate Education:**

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, USA, 1994
- Bachelor of Engineering in Electrical Engineering,
  Kasetsart University, Bangkhen, Bangkok, Thailand, 1990

**Major:** Computer Science

# ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to his advisor, Professor Lonnie Welch, for his guidance, patience, kindness and financial supports throughout this research.

Very special thanks to Dr.James McHugh and Dr.David Nassimi for serving as members of the committee.

Very special thanks to Dr.Peter Leopold for his friendships, moral support and for introducing this fascinating work to me.

Special thanks to Dr.Jeff Livingstone at Genetech,Inc., CA, for helping me running my programs on SGI parallel computers at Genetech.

And finally, thanks to Ganesh, and Jin for helping me with reading the manuscript.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

A protein is a linear chain of twenty amino acid groups and it performs crucial tasks in a living cell. It was discovered that a denatured, newly formed protein, called random coil, will fold up into a compact, unique shape, called native state, which is determined solely by its amino acid sequence. However, predicting the conformation of a protein native state based solely on the knowledge of its amino acid sequence is far from simple. This problem has been widely known as the protein folding problem [32].

There are many works, both theoretical and experimental, studying how a protein fold and what makes it fold to its native state. This thesis is on the theoretical aspect based on the 'thermodynamics hypothesis' [10]. The thermodynamics hypothesis of protein folding states that the native structure of the globular molecule is the conformation which has the lowest free energy, then the native structure could be identified in principle simply by systematic evaluation of the free energy of every possible conformation [10]. The problem is that calculating the free energy of every possible conformation of full protein is prohibitive using currently achievable computing power. Since high resolution simulation of protein folding is not possible, a simplified model of protein backbone (its primary structure) is needed. One of the most popular model of protein backbone representations used in protein folding studies is the cubic lattice model. The protein backbone is represented as a chain of cubic lattice sites. Using this simplified model, it is possible to study protein folding of longer chain of amino acids.

From amino acid sequence representation , we can study the thermodynamics hypothesis of protein folding by

1. Enumerate all unique compact conformations of an amino acid chain representation that lead to the folding pathways. A chain of lattice sites represents the protein backbone on its folding pathway.

2. Select a set of amino acid sequences that will be used in force-field simulation. Calculate free energy of each sequence based on all unique enumerated compact conformations from the first phase. For each particular amino acid sequence, the conformation that gives the lowest free energy should be the native state of that sequence according to the thermodynamics hypothesis of protein folding.

This thesis solved the first problem by applying parallel processing to exact enumeration problem. The second problem is being solved by the Harvard chemical physics polymer theory research group.

First, I will review previous literatures related to exact enumeration of cubic lattice and its application to heteropolymer studies. I will describe the cubic lattice model of protein backbone and show how to use it to represent protein backbone in enumeration problem. A set of graphs used to represent the model of computation will be introduced. Then I will present a set of parallel lattice enumeration algorithms subjected to volume and contact constraints. Experimental results are presented. Speed up and communication costs of parallel implementation will be discussed.

# CHAPTER 2

# LITERATURE SURVEY

Exact enumeration of self-avoiding walk has been studied intensively for the three decades. It is known to be a very important tool for polymer studies. The reference works on exact enumerations and self-avoiding walks for polymer physics studies are [2, 3, 4, 5, 6, 7, 8, 13, 14, 15, 16, 17, 20, 21, 22, 23, 26, 27, 28, 29, 30, 31, 34, 36, 37, 38, 39, 40, 41]

This thesis will focus on the work by Shakhnovich and Gutin [34]. In 1990, Shakhnovich and Gutin proposed a simplified model of heteropolymers with random sequence of links using a three-dimensional cubic lattice. They used this model to study the freezing transition of heteropolymer chains. The model and the enumeration algorithm used in that paper can be summarized as follow:

- Create a representation of heteropolymeric chains in cubic lattice. Each monomer is represented by lattice sites.

- Specify a volume constraint that will confine the enumeration process. In that paper, volume constraint lattice is used to represent the maximally compact conformations of heteropolymeric chains.

- From volume constraint graph, specify a set of starting chain conformations that are guaranteed to ramify to only unique structures when enumeration process is applied. Informally, the enumeration process is a process of counting all unique Hamiltonian path beginning from a specified sequence of lattice sites (monomer representation) on a specified lattice graph.

- Start the enumeration process from that set of initial conformations. The enumeration process is subjected to a volume constraint which means all monomer units in the specified volume graph must be presented in any completed chain computed by the enumerator and no lattice site which is not in that volume can be included in any chain conformation.

- Select a set of heteropolymeric chains with random sequence of links. Calculate the minimum free energies (corresponding to the frozen state of heteropolymeric chains) of each of these chains based on all enumerated maximally compact conformations. For each selected heteropolymeric chain, the maximally compact conformation that gives lowest free energies should be the frozen state of that heteropolymeric chain.

This paper also discussed the applications of exact enumeration of maximally compact conformation to investigation of thermodynamics of protein folding, especially for globular proteins which are very compact with density in the interior close to such of molecular crystals. Note that proteins are heteropolymers. The paper suggested the analogy of freezing transition of heteropolymeric chain to folding transition in proteins.

The enumeration algorithm used in that paper can be further improved as follow:

- The exact enumeration process is a blind exhaustive search with a time complexity of $O(C^N)$ where $N$ is the random monomer sequence length and $C$ is the degree of freedom or the number of ways that a search process can ramify the existing structure in the worst case. See figure 2.1. By modifying the initial conformation generator, I developed a set of parallel algorithms to enumerate all unique conformations. By observing the volume constraint in enumeration

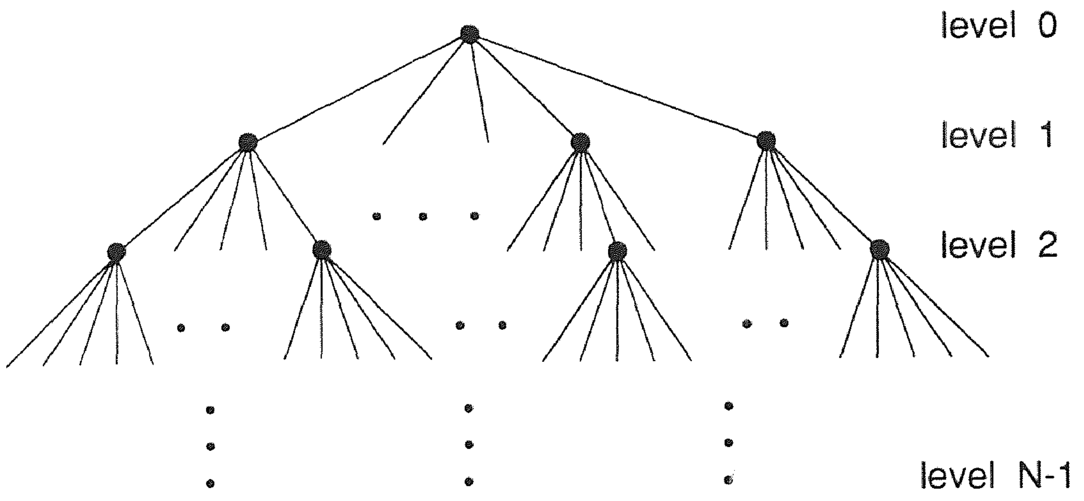


**Figure 2.1** A tree of N levels with branching factor of 5

problem, I developed an algorithm that can enhance the performance of any volume-constraint enumeration.

- Since self-avoiding walk (the enumeration subprocess) cannot avoid generating symmetry structures, therefore it requires that a user specifies an initial set of conformations which are guaranteed to ramify to only unique conformations when the walking process is applied. This process is being done by hand. This process becomes more complicated when a random monomer chain of interest is longer or when a new volume is needed. I developed an algorithm that will generate a set of initial conformations that will not ramify to any redundant structure when the self-avoiding walk procedure is applied.

- The algorithms presented in this thesis are also capable of enumerating any volumes of cubic lattice. This thesis also presents a parallel algorithm for contact-constraint enumeration problem. [10, 11].

# CHAPTER 3

# OBJECTIVE

As discussed in the previous chapter, the current approach used to solve enumeration problem requires human interactions during the initial phase. This process is necessary to remove a large number of initial conformations that will ramify to redundant structures during the enumeration process. If the enumerator cannot avoid generating redundant structures, the computation time used in generating and removing those redundant structures will account for most of the computation time of enumeration process.

It is observed that enumeration of structural representation of protein backbones in cubic lattice (with appropriated symmetry avoidance algorithm) exhibits a very high degree of parallelism. This is due to the fact that the tasks of finding all possible conformations (exact enumeration) from a set of initial conformations are totally independent. Therefore, parallel processing can be successfully applied to enumeration problems.

The objectives of this thesis are to devise an algorithm that will automate the process of removing initial redundancy in enumeration problem and to apply parallel processing to the exact enumeration problem to reduce the enumeration time which makes longer chain enumeration possible. This thesis also discussed the application of exact enumerations to protein folding studies.

In this thesis, I proposed a set of parallel algorithms to solve the exact enumeration problem on cubic lattice. The followings are important characteristics of proposed parallel exact enumeration algorithms:

- They are based on an exhaustive search. However, it is not a blind search. It is a heuristic guided algorithm that can avoid a large number of unsuccessful walks by using a graph connectivity checking algorithm.

- They automatically remove all initial conformations that will leads to redundant structures by using mathematics formulation of three-dimensional geometric transformations.

- They can enumerate any volume of cubic lattice structures, ie. not necessary to be a perfect cubic shape.

- These algorithms are relatively easy to parallelize. The resulting parallel algorithms are suitable for both shared and distributed memory parallel computers.

# CHAPTER 4

# THE PROTEIN FOLDING PROBLEM

In this chapter, I will discuss what is protein, what is the protein folding problem, why it is important to solve this problem and how exact enumeration of lattice helps solving the protein folding problem. Detailed discussion of protein folding problem can be found in [9, 10, 11, 12, 18, 19, 24, 25, 32, 33, 34, 35].

## 4.1 What are proteins?

Proteins are polymers. A protein is a linear polymer molecule, a chain of tens of thousands of monomer unit. The monomers are the twenty amino acids which consist of a central carbon atom – called the $\alpha$ - carbon – bound to an amino group ($NH_2$), a carboxyl group (COOH) and a side chain. The differences among amino acids lie in their side chains, namely in shape, size and its polarity. Shape and size affect the packing together of amino acids in the final molecule. Polarity (or lack of polarity) determines the nature and strength of interactions between amino acids in a protein and between the protein and water. Note that the interior of most cell is 70 to 90 percent water.

To understand how a protein functions, we must know its three-dimensional structure. From its three-dimensional structure, we can study its behaviors from the interactions of its molecules at the atomic level.

## 4.2 Protein folding

In the late 1950s, Christian B. Anfinsen [1] discovered that the forces most responsible for proper folding of the newly formed protein into a specific shaped could be derived from the basic principle of chemistry and physics. Only the amino acid sequence of the protein was fully sufficient to specify the molecule's ultimate three-dimensional shape and biological activity.

Unfolded or newly formed proteins are often called random coils, implying that no region of the protein backbone looks significantly different from any other region. For globular protein, its most important state known as its native or folded state, is extremely compact and unique. That is, a given globular protein folds to only one native state.

The goal of protein folding problem studies is to predict the compact three-dimensional structure only from the knowledge of the monomer unit (amino acid group) sequences. Solving the protein folding problem would unleash new power in biotechnology, ie. permitting the design of new proteins. Note that enzymes, which are the catalysts for virtually all biochemical reactions in living cell, are globular proteins.

The balance of forces that folds a protein into its unique, compact native structure is encoded within its amino acid sequence. This correspondence between sequence and structure is sometimes referred to as the "second genetic code." (The first genetic code is the correspondence between the base sequence of a DNA molecule and the amino acid sequence of the protein whose synthesis it controls).

## 4.3 Simplified models of protein backbones

To explore sequence-structure relationships, a class of model has emerged in which amino acid chains are represented as self-avoiding walk on lattices. Specific sequences of monomers are studied in chains short enough that the full conformational space can be enumerated exhaustively.

The simplified exact model of protein is necessary because even a short chain molecule of protein simulation requires a considerable amount of computer time to finish. This simplified exact model is helpful at the early stage of protein folding simulation. After the early stage, conformation tends to compact and have less variations in its three-dimensional structure, then one can employ a higher resolution exact model to achieve better result.

There are many simplified exact models of protein backbone that are of interest in protein folding studies. They are square lattice, cubic lattice, face-centered cubic (fcc) lattice, bcc lattice, diamond lattice, honey-comb lattice, chess-knight lattice and hybrid lattice.

Square and cubic lattices are the two most common types used in modeling protein backbones. This thesis will focus on cubic lattice based on the studies pioneered by Eugene Shakhnovich and Alexander Gutin [34]. Each protein backbone (the chain of its $\alpha$ carbon group, amino group and carbon group) will be represented by a single site of cubic lattice. Based on this model, we can exhaustively search all conformations of protein backbone subjected to various constraints (volume- and/or contact-constraint).

To find the stable native state of a protein, we should compute, for every possible conformations of the protein molecule chain, the sum of the free energies of the atomic interactions within the protein and with the solvent and then find the conformation with the lowest free energy.

Note that finding all possible conformations by exhaustive enumerating all possible self-avoiding walk on three-dimensional cubic lattice space is a time-consuming process because the number of conformation of a chain molecule grows exponentially with the chain length. It was observed that the amount of time to do exhaustive search is much larger than the time real protein uses to fold itself. This observation is known as Levinthal's paradox. However, enumerative kinetics of protein folding has contributed much to our understanding of protein folding.

# CHAPTER 5

# CUBIC LATTICES: PROPERTIES AND REPRESENTATIONS

In this chapter, properties of cubic lattice and its representation in our computational model are discussed. Symmetry properties of cubic lattice are discussed. An algorithm to avoid symmetry structures before starting the enumeration subprocess (the self-avoiding walk) in cubic lattice is presented.

## 5.1 Definitions and cubic lattice representations

To solve problems using computers, one needs some forms of problem representations which in turn be transformed to data structures in programming languages. Graph data model can be used to represent lattice enumeration problem subjected to various constraints very well. In this chapter, I will introduce terms and definitions used through out the thesis.

Monomers:

In our discussion, monomers (mer) are amino acid groups. These monomer units of protein primary structure (protein backbone) will be represented as lattice sites in cubic lattice which in turn represented by an undirect, connected graph. Since lattice sites represent monomers, the two terms will be used interchangeably in this thesis.

**Cubic lattices:**

Cubic lattice $C_L(M, L)$ is defined as an infinite three-dimensional rectangular grid in Cartesian coordinate system. At each intersection point of three orthogonal axes, there is a lattice site $M$. Each lattice site $M$ will have exactly six possible near-neighbor lattice sites at a unit distance along the three axes in the Cartesian

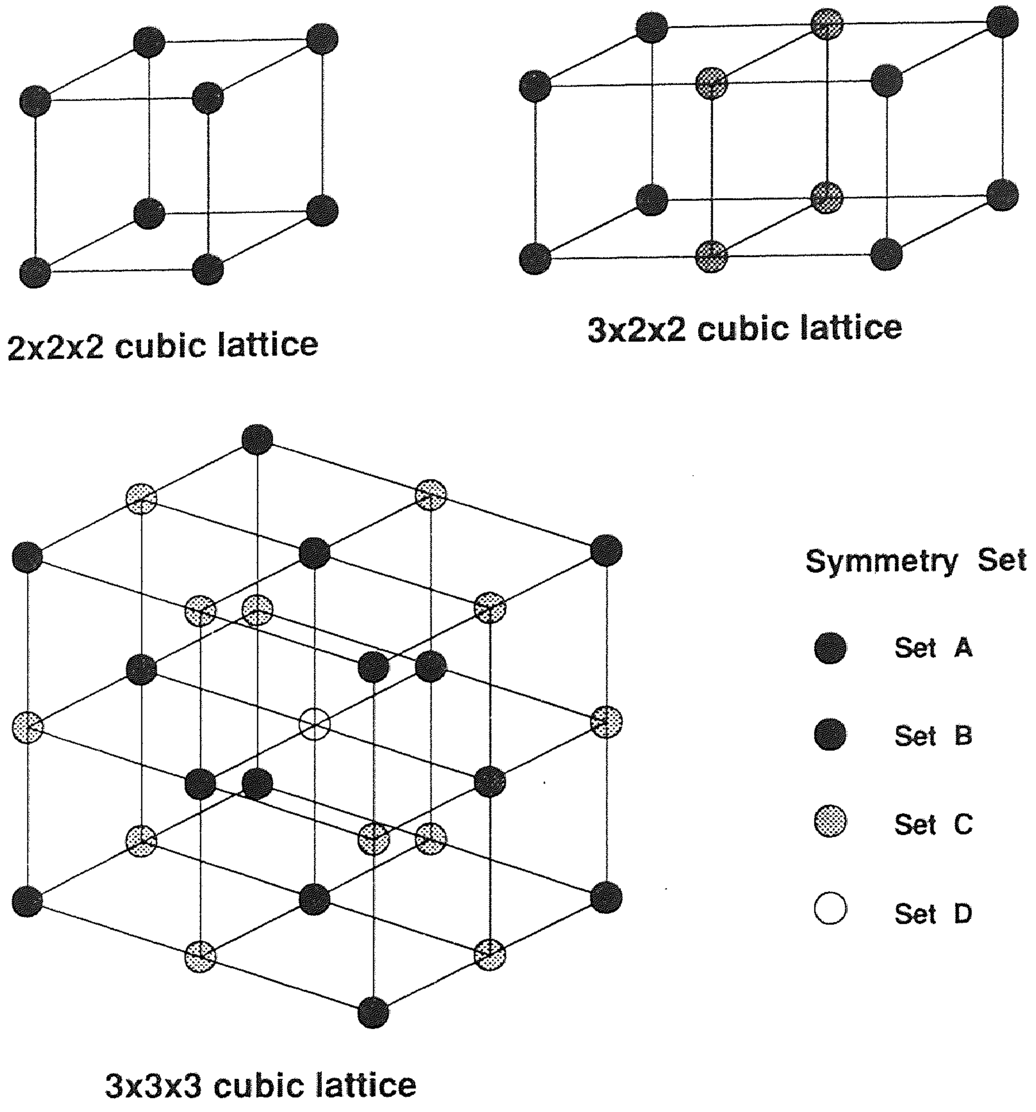


Figure 5.1 'Closed' cubic Lattices of various volumes

coordinate system, ie. $\langle \pm x, \pm y, \pm z \rangle$ directions. These lattice sites are connected together with links $L$. See figure 5.1.

Using graph data model, cubic lattice $C_L(M, L)$ can be represented as an undirected connected graph $G(V, E)$, *with* $M = V$ *and* $L = E$. Formally, $G(V, E)$, is cubic lattice graph corresponding to cubic lattice $C_L(M, L)$ with the following definition:

if $V_j$ *and* $V_k$ *in* $G(V, E)$ *are any two distinct vertices corresponding to* $M_j$ *and* $M_k$, *in* $C_L(M, L)$ *lattice sites respectively, and are near-neighbor of each other, then* $E_l$ *is an undirected edge, corresponding to* $L_{j,k}$ *link, connects* $V_j$ *and* $V_k$.

A cubic lattice $C_L(M, L)$ is called 'closed' if it contains a finite number of lattice sites and links. A cubic lattice $C_L(M, L)$ is called 'opened' if it has an infinite number of lattice sites and links. By definition, a cubic lattice is 'opened' if it is not explicitly declared as 'closed'.

## Sequences:

In our discussion, a sequence refers to a list of lattice sites (a chain conformation) on a lattice graph. There are two type of sequences, live and dead sequences.

## Live sequences:

A 'live' sequence $S_{live}$ is a sequence of lattice sites $M$ in a cubic lattice $C_L(M, L)$ which have been selected according to the required constraints (volume and/or contact) imposed to the walk process. It is named 'live' sequence because it is going to change until it has a required length.

## Active mers (active lattice sites):

At any instance of self-avoiding walk, there is only one lattice site (mer) that the self-avoiding walk can ramify from. This mer is called an 'active' mer. Active mers are always the last mer of any live sequence.

## Dead sequences:

When the self-avoiding walk found a complete sequence of mers (a sequence of mers of a predefined length) that satisfied the imposed constraint of self-avoiding walk, a 'dead' sequence $S_{dead}$ has been found. A dead sequence is a 'complete' sequence of mers $M$ of a given cubic lattice $C_L(M, L)$ that have been selected according to the required constraints (volume and/or contact) imposed to the self-avoiding walk. It is named a 'dead' sequence because it will never be changed again after it has been found. These 'dead' sequences are what we want to enumerate and record its three-dimensional structures. Note that there is no active mer on any dead sequence.

**Volume-constrained cubic lattice graphs:**

Volume-constraint is used to define a subspace or portion of opened cubic lattice which confines the self-avoiding walk into that region. Throughout this thesis, a volume-constrained cubic lattice is a special kind of 'closed' cubic lattice which has a symmetrical rectangular shape, ie. a $X \times Y \times Z$ volume. From the definition of cubic lattice, a volume-constrained cubic lattice graph is a closed cubic lattice with the following properties describes its vertices:

- $V_c$ are 'corner' vertices with out-degree of exactly three. Set $A$ vertices in figure 5.1 are examples of $V_c$.
- $V_e$ are 'cliffed-edge' vertices with out-degree of exactly four. Set $C$ vertices in figure 5.1 are examples of $V_e$.
- $V_f$ are 'faced-center' vertices with out-degree of exactly five. Set $B$ vertices in figure 5.1 are examples of $V_f$.
- $V_i$ are 'inner' vertices with out-degree of exactly six. Set $D$ vertices in figure 5.1 are examples of $V_i$.

It should be noted that in the case of perfect symmetry volume-confined shape (as in the case of $N \times N \times N$ volume), these vertices are symmetry to all other vertices in the same set. This is a very important characteristic that we can use to help avoiding a large number of rotation symmetry. Also note that any volume-constrained lattice graph needs not to have all these types of vertices. But all valid volume-constrained lattice graph must have $V_c$.

**Contact-constrained cubic lattice graphs:**

Contact constraint is used to define a set of required near-neighborness at some steps of self-avoiding walk. Figure 5.2 shows a contact-constrained sequence. The

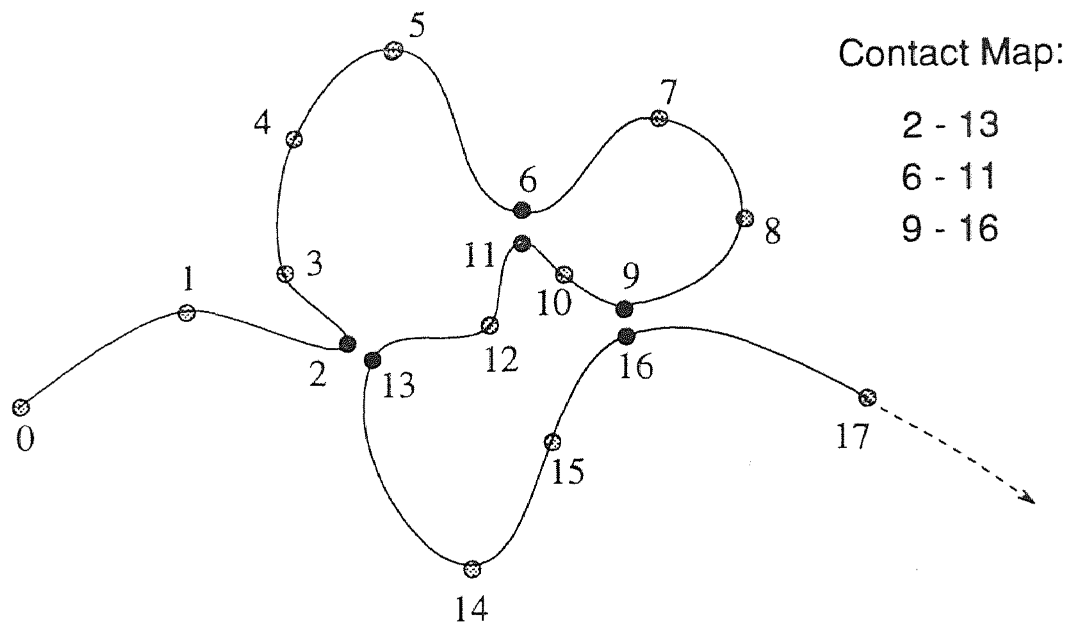


**Figure 5.2** A contact-constrained chains

numbers associated with the dots refer to step number of self-avoiding walk referenced to the first mer (step 0) of sequence. During the self-avoiding walk, the contact constraint map imposes a set of possible lattice sites that an active mer can ramify the self-avoiding walk. From the definition of cubic lattice, contact-constrained cubic lattice graph is an opened cubic lattice graph. The self-avoiding walk with contact constraint is free to ramify its active mer (the last lattice site in the sequence) to any unselected near-neighbor mer of the active mer as long as it satisfies the contact constraint map.

**Contact-constrained lists:**

A contact-constrained list $L_{CC}$ is a set of pairs $(a, b)$ where $a, b$ are the two different step number of self-avoiding walk. This contact constraint list $L_{CC}$ is used to enforce the required near-neighborness of lattice sites at step $a$ and $b$ of self-avoiding walk.

For any sequence $S$ to be qualified as a valid dead sequence in contact-constrained enumeration, it is required that the $a^{th}$ mer and $b^{th}$ mer in $S$ are near-neighbor of each other in a specified lattice graph.

**Self-avoiding walk:**

A self-avoiding walk is a process of finding a sequence $S$ of distinct mers from a specified lattice graph, volume-constrained or contact-constrained cubic lattice graph. For the purpose of our discussion in this thesis, there are only three possibilities in applying self-avoiding walk.

- If a self-avoiding walk is applied to a closed cubic lattice graph, a volume-constraint graph $G_{VC}$, then the self-avoiding walk is a process of finding Hamiltonian path on $G_{VC}$.
- If a self-avoiding walk is applied to an opened cubic lattice graph, ie. a contact-constraint graph , then a self-avoiding walk is a process of finding a sequence of distinct mers $S$, of some finite length $L$, that satisfies a list of required near-neighborness of mers on any sequence $S$, the contact-constraint list $L_{CC}$.
- If a self-avoiding walk is applied to a closed cubic lattice graph and it is also required that the self-avoiding walk process satisfies a contact-constrained list, then this is the case of both volume- and contact-constrained walk.

**Exact enumeration of self-avoiding walk on cubic lattice:**

An exact enumeration of self-avoiding walk on a cubic lattice graph is an exhaustive search procedure that counts (and records) all possible, unique conformations of a given cubic lattice graph (subjected to volume- and/or contact-constraint).

Uniqueness in enumeration process means we do not count any dead sequence of mers which is a mirror and/or rotation symmetry (to be defined later) of any previously recorded 'dead' sequence.

## 5.2 Symmetries in cubic lattices

In this section, I will discuss the symmetry properties of cubic lattice that effect enumeration problem. I will present an algorithm to remove such symmetry structures before starting the enumeration process.

In cubic lattice, there are at most six different directions to go from a lattice site to other near-neighbor lattice sites. They are $\langle \pm x, \pm y, \pm z \rangle$ directions in the three-dimensional Cartesian coordinate system. A self-avoiding walk on cubic lattices can be realized as a sequence of lattice site indices (a lattice site is indexed by a unique identifier) or as a sequence of directions that a walker walks on the associated sequence of lattice sites. The latter view is more suitable to the enumeration problem. Consider the self-avoiding walk on a cubic lattice as a 'direction' permutation. All one needs to compute exact enumeration of all conformations on a given lattice is to compute all permutations of six possible directions of desired length and check if each of such permutation sequence violates the enumeration constraint or not.

### 5.2.1 Mirror symmetry

Let $E_p(e_{p0}, e_{p1}, \cdots, e_{p(n-1)})$ be a 'direction' sequence of a self-avoiding walk on a undirect, connected graph $G(V, E)$ (it can be a volume-constrained and/or contact-constrained graph) and $E_q(e_{q0}, e_{q1}, \cdots, e_{q(n-1)})$ be another direction sequence on the same graph. $Q$ is said to be a mirror symmetry of P if and only if one of the following two conditions is satisfied:

- if $E_p \neq E_q$ and $\phi$ is any direction $\pm x$, $\pm y$ or $\pm z$ and $E_{pi}$ (the $i^{th}$ element of $E_p$) $\neq \phi$ then $E_{pi}$ must be equal to $E_{qi}$. If $E_{pi} = \phi$ then $E_{qi}$ must be equal to $-\phi$.
- If $E_p \neq E_q$ and $\delta, \lambda, \phi$ are any distinct $\pm$ of $x$, $y$ or $z$ direction and if the two direction sequences have three transformation pairs of this form: ($\delta$ in $E_p$ ,$\lambda$ in $E_q$), ($\lambda$ in $E_p$,$\delta$ in $E_q$) and ($\phi$ in $E_p$,$\phi$ in $E_q$), then $E_p$ and $E_q$ are mirror symmetry. This condition leads to mirror symmetry of the two sequences

because after transforming $\lambda$ in $E_q$ to match $\delta$ in $E_p$ and $\delta$ in $E_q$ to match $\lambda$ in $E_p$, all $\phi$ in $E_q$ will be transformed to $-\phi$ and the two sequences will now satisfy the first condition.

Figure 5.3 shows a set of lattice site chains that are mirror symmetry. The followings are 'direction' sequences of chains in figure 5.3:

$$\begin{array}{ll}(a) & \langle +x, +y, +z, +x, -z, +y, -x, +z, +x, +z\rangle, \\ (b) & \langle +y, +x, +z, +y, -z, +x, -y, +z, +y, +z\rangle, \\ (c) & \langle +z, +y, +x, +z, -x, +y, -z, +x, +z, +x\rangle.\end{array}$$

Sequence (a),(b) pairs satisfy the second condition with $\delta = +x$, $\lambda = +y$ and $\phi = +z$. Sequence (a),(c) pairs satisfy the second condition with $\delta = +x$, $\lambda = +z$ and $\phi = +y$.

Note that the sequence pairs (b),(c) are not mirror symmetry but rotation symmetry with the following three cyclic direction transformations

$$(+y \rightarrow +z), (+z \rightarrow +x), (+x \rightarrow +y)$$

(see the next subsection).

### 5.2.2 Rotation symmetry

Let's define a rotational transformation as a process of changing of a direction in three-dimensional Cartesian coordinate to another direction after a sequence of rotation steps.

Let $E_p(e_{p0}, e_{p1}, \cdots, e_{p(n-1)})$ be a 'direction' sequence of a self-avoiding walk on a undirect, connected graph $G(V, E)$ (it can be a volume-constrained and/or contact-constrained graph) and $E_q(e_{q0}, e_{q1}, \cdots, e_{q(n-1)})$ be another direction sequence on the same graph. $Q$ is said to be a rotation symmetry of P if and only if one of the following two conditions is satisfied:

- If $E_p = E_q$ then $E_p$ and $E_q$ is obviously rotation symmetry of each other.

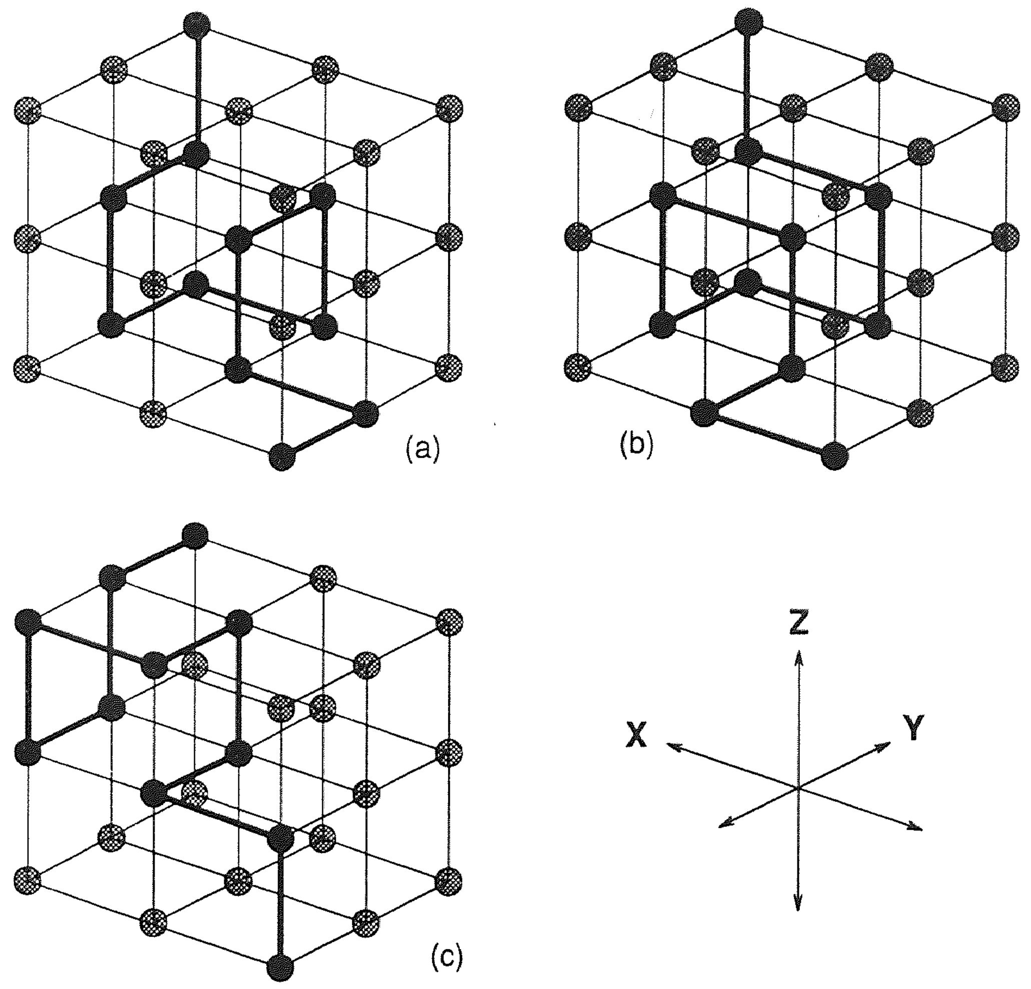


Figure 5.3 Mirror symmetry of chains in 3x3x3 cubic lattice

- If $E_p \neq E_q$ and $\delta, \lambda, \phi$ are any distinct $\pm$ of $x$, $y$ or $z$ direction, there are three cyclic direction transformations in the form $(\delta \rightarrow \lambda), (\lambda \rightarrow \phi)$, and $(\phi \rightarrow \delta)$ from $E_p$ sequence to $E_q$ sequence to make them satisfy the first condition. Note: $\rightarrow$ means 'transform to'.

Figure 5.4 shows a set of lattice site chains that are all rotation symmetry to each others. The followings are direction sequences of chains in figure 5.4:

$$(a) \quad \langle +x, +y, +z, +x, -z, +y, -x, +z, +x, +z \rangle,$$

$$(b) \quad \langle +y, +z, +x, +y, -x, +z, -y, +x, +y, +x \rangle,$$

$$(c) \quad \langle +z, +x, +y, +z, -y, +x, -z, +y, +z, +y \rangle.$$

The followings are cyclic direction transformations of pairs of sequences in figure 5.3:

$$(b) \quad to \quad (a): \quad ((x \rightarrow y), (y \rightarrow x), (z \rightarrow x)),$$

$$(c) \quad to \quad (a): \quad ((x \rightarrow z), (z \rightarrow y), (y \rightarrow x)),$$

$$(c) \quad to \quad (b): \quad ((y \rightarrow z), (z \rightarrow x), (x \rightarrow y)).$$

## 5.3 Avoiding redundant structures in self-avoiding walk

In this section, I will present an algorithm to avoid mirror and rotation symmetries before starting an enumeration procedure. Fortunately, avoid generating redundant structures is much easier than it seems to be as discussed in the previous section.

### 5.3.1 Avoiding rotation symmetry

To avoid rotation symmetry, I derived the method from the following facts about rotation symmetry of cubic lattice. First, any two three-dimensional structures will be rotation symmetry of each other if we can perfectly match them together by at most three different transformation steps in three-dimensional space. Second, any

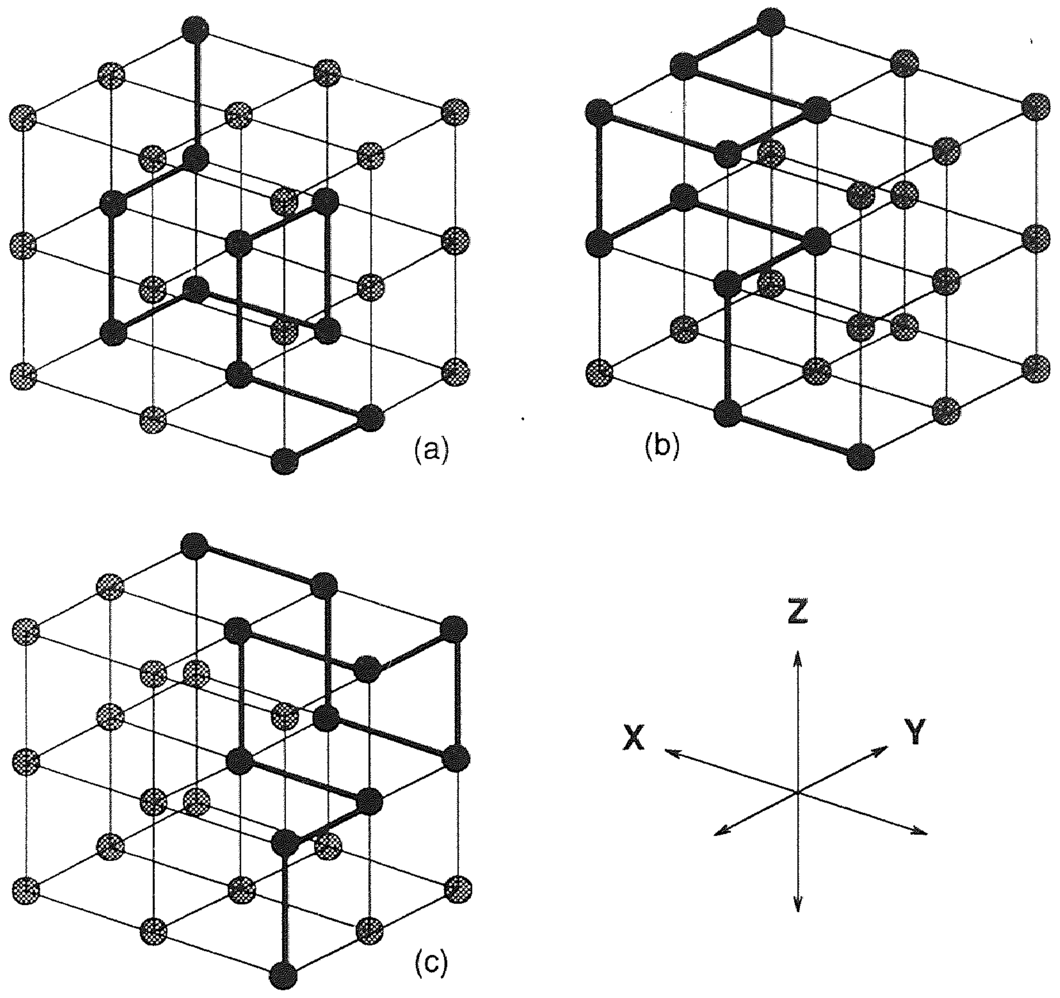


Figure 5.4 Rotation symmetry of chains in 3x3x3 cubic lattice

lattice site on cubic lattice which is the current point of ramification (the active mer) will not ramify to rotation symmetry structure if the sequence of selected mers is a three-dimensional structure. Third, a cubic lattice is a repetition of a fundamental unit, a lattice site with exactly six near-neighbor, which makes it perfectly symmetrical from any view.

From these facts, we can see that to avoid rotation symmetry we have to find a set of starting mers on volume-confined structure that will ramify to only unique conformations. From this initial set of lattice sites, use it as the only possible starting points of walk. For each such mer, the self-avoiding walk can ramify one step at a time to its near-neighbors that are not rotation symmetry with respect to the current structure of walking path. Figure 5.5 shows three types of non-symmetry self-avoiding walk (avoiding both rotation and mirror symmetries). By following this guideline, the walker will never generate any redundant structure due to rotation symmetry.

### 5.3.2 Avoiding mirror symmetry

To avoid mirror symmetry, this requires knowledge of both rotation symmetry and mirror effect. It is observed that mirror and rotation symmetries usually present together during the self-avoiding walk. Fortunately, the mirror effect in cubic lattice conformations can be avoided by simply following the guideline used to avoid rotation symmetry discussed in section 4.3.1. This makes it a lot easier to remove mirror symmetry, otherwise it will be very difficult to remove all mirror structures.

### 5.3.3 An algorithm to avoid mirror and rotation symmetries

Formally, we define the following algorithm to avoid generating redundant structures in self-avoiding walk due to rotation and/or mirror symmetry properties of cubic lattices. This algorithm will generate a set of non-redundant initial conformations that will be used as an input to the enumeration procedure.

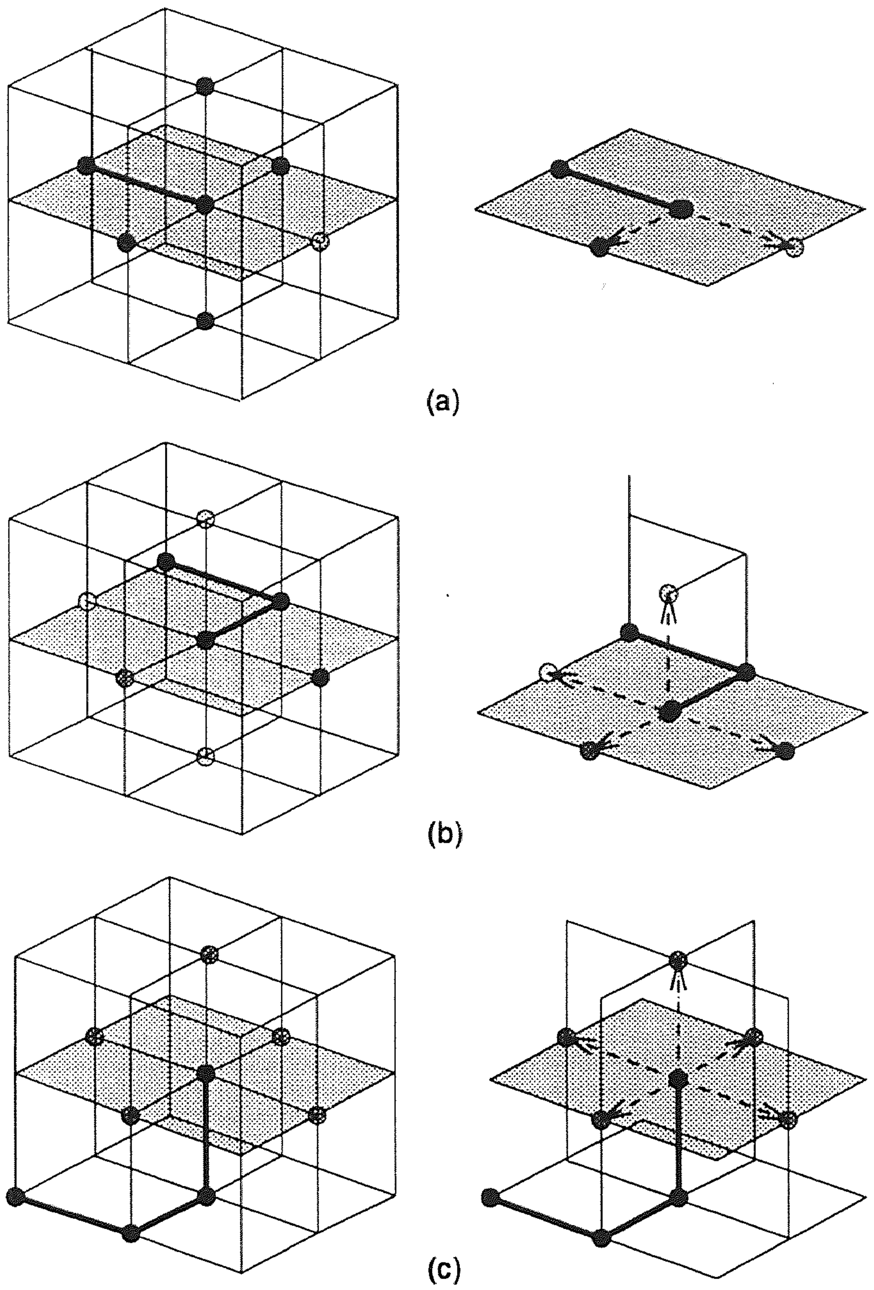


Figure 5.5 Non-symmetry self-avoiding walk: (a) line symmetry avoiding, (b) plane symmetry avoiding and (c) 3-D symmetry avoiding

**Algorithm 5.1: Rotation and Mirror Symmetry Avoidance**

[ *This algorithm will generate a set of initial conformations of a given volume– and/or contact–constraint lattice graph. When the walker ramifies these initial conformations, it will not generate any symmetry conformations.* ]

```
1. Let $Head(Q), Dequeue(Q), Enqueue(Q, M), Empty(Q)$ be generic queue
   functions that perform the following functions, return $Q$ head, dequeue $Q$,
   enqueue a new element $M$ to $Q$ and test emptiness of $Q$ respectively. Note
   that the $Q$ itself does not change when we apply functions $Head()$ and $Empty()$
   to it.

2. Create a queue of conformation $Q_R$ to keep redundant initial conformations.
   Reset this queue.

3. For each mer that is not rotation symmetry, create a conformation which has
   only one such mer and set the dimension of the conformation to zeroth (0D)
   and put it into $Q_R$.

4. Create a queue of conformation $Q_{NR}$ to keep non-redundant initial confor-
   mation. Reset this queue.

5. Let $P$ be a conformation.

6. while $Q_R$ is not empty do

   (a) assign $Head(Q_R)$ to $P$, and $Dequeue(Q_R)$

   (b) let $M$ be the active mer of conformation $P$ which is always the last mer
       of $P$.

   (c) repeat

       i. case 0D:

          ▷ Pick any near-neighbor $N$ of $M$ as the new active mer of $P$.
```

- ▷ Update the conformation dimension to the first dimension (1D) and $Enqueue(Q_R, P)$.

ii. **case 1D:** (figure 5.5(a)) Consider all near-neighbors of $M$:

- ▷ If a near-neighbor $N$ is in the same 'direction' as the first dimension of $P$, then pick $N$ as one of the next ramification. Create a new conformation $X$ which is the same as conformation $P$ and add mer $N$ to $X$ as the new active mer. Then $Enqueue(Q_R, X)$.
- ▷ Otherwise, pick any one of the rest near-neighbor of $M$ as the next possible ramification, called it $N$. Create a new conformation $X$ which is the same as conformation $P$ and add mer $N$ to $X$ as the new active mer. Update the dimension of conformation $X$ to the second dimension (2D). Then $Enqueue(Q_R, X)$.

iii. **case 2D:** (figure 5.5(b)) Consider all near-neighbors of $M$:

- ▷ For each near-neighbor $N$ which is in the plane of conformation $P$ (note that $P$ is currently a planar (2D) conformation), create a new conformation $X$ which is the same as conformation $P$ and add mer $N$ to $X$ as the new active mer. Then $Enqueue(Q_R, X)$.
- ▷ For the rest near-neighbor mers of $M$ that is not in the plane of conformation $P$, pick one of them as the new active mer. Create a new conformation $X$ which is the same as conformation $P$ and add mer $N$ as the new active mer of $X$. Update the dimension of conformation $X$ to the third dimension (3D). Then $Enqueue(Q_R, X)$.

iv. **case 3D:** (figure 5.5(c)) Record conformation $P$ as a new non-redundant structure by doing $Enqueue(Q_{NR}, P)$. When applying the

```
            self-avoiding walk procedure (in chapter 5) to conformation $P$, there
            will be no redundant structure.
        until current conformation $P$ is 3D.
    end while
End–Algorithm 5.1
```

The most interesting aspect of this algorithm is that this kind of prefix computation leads to the applicability of parallel processing of any volume. This initial set of prefix conformations can be realized as a set of independent parallel tasks that can be executed concurrently.

## 5.4 How to improve enumeration time

It is observed that volume constraint restricts the self-avoiding walk such that all nodes in the confined-volume must be used exactly once in any completed walk. If the self-avoiding walk is blocked before it found a complete conformation, the algorithm will backtrack to find another possibility to ramify from its current position

A blind exhaustive search will backtrack only when the search process is blocked. However, we can avoid a large number of unsuccessful walks (backtrack before the search is blocked) by checking that some portions of volume-confined structure will never be visited if we continue walking from the current active mer. See figure 5.6. By using a connectivity checking algorithm, we can avoid this unfruitful situation as soon as it occurs. The connectivity checker will check if the walk has divided the set of unvisited mers into two disjointed subsets or not. If that is the case, then one of the two unvisited mer groups will never be visited by the walker (without backtracking to the level of this active mer again). As soon as the set of unvisited mers is divided into two disjointed subsets, we can stop progressing

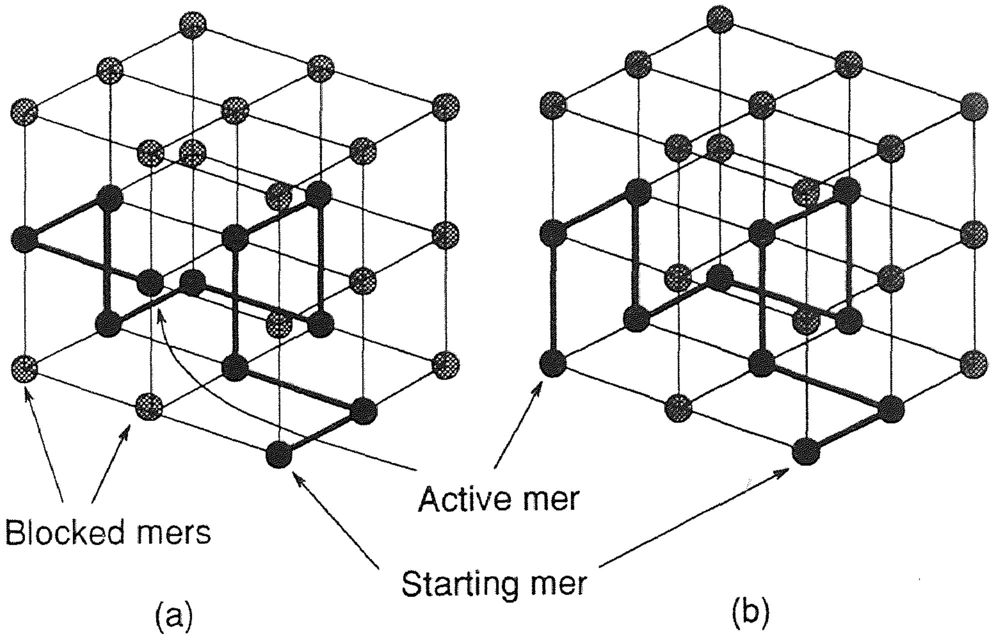


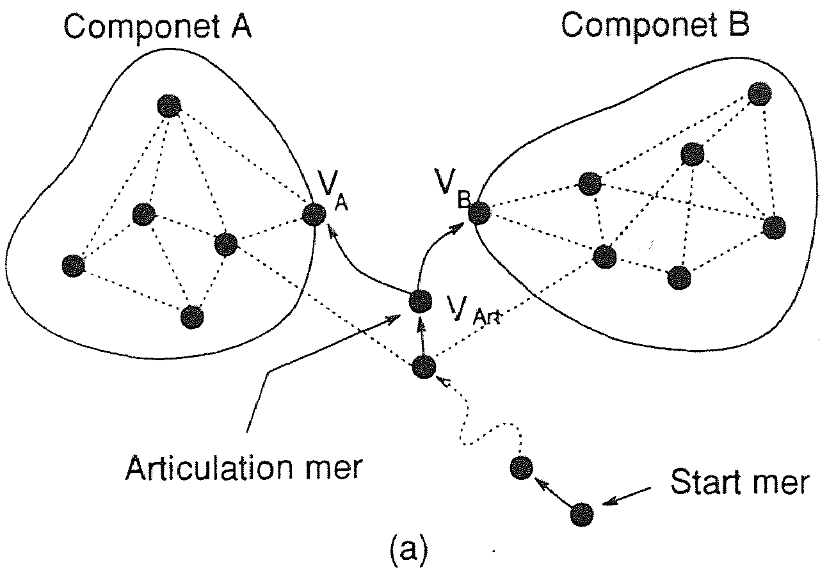
**Figure 5.6** Blocking in self-avoiding walk with volume-confined constraint

from the active mer of current conformation and try another near-neighbor which is available at that level of walk.

The connectivity checking algorithm is implemented as a depth first search procedure without backtracking capability. The time complexity of this connectivity checking algorithm is $O(N)$ which is less than the overall complexity of exhaustive enumeration algorithm.

The idea that makes this algorithm works is as follow. Let $VCG$ be a volume-constrained graph and $Q$ be a live sequence and $V_{art}$ be the active mer of $Q$. Now, supposed further that $V_{art}$ is the articulation point which connects the two subgraph $G_a$ and $G_b$ of unvisited mers together at mer $V_a$ and $V_b$,respectively. See figure 5.7.

At the next step of walk, we have to choose between $V_a$ or $V_b$. No matter which one we choose between these two mers, the volume graph of unvisited mers will be divided into two separated connect component. Once the walker decided to continue with one component, say $G_a$, by selecting $V_a$ as the next mer, then the other component $G_b$ will never be visited. This is because $V_{art}$ is the only mer that



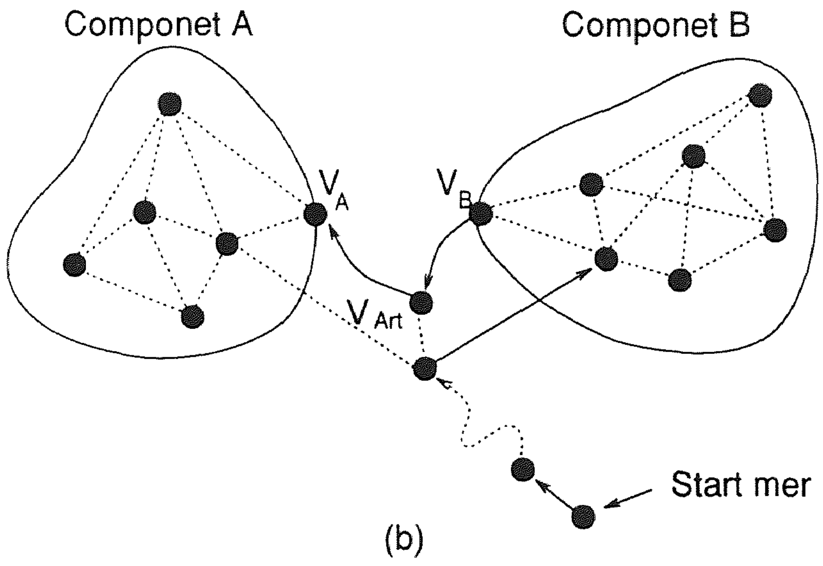


Figure 5.7 Components in self-avoiding walk with volume constraint

connects $G_a$ and $G_b$ together and the self-avoiding walk can use $V_{art}$ only once which means the self-avoiding walk cannot use $V_{art}$ to go from $G_a$ to $G_b$.

If the walking process keeps walking in one component, then it will eventually find that the search process is blocked and start to backtrack again and again until it comes back to $V_{art}$.

Therefore, as soon as the self-avoiding walk selects any mer which is a near-neighbor of $V_{art}$, the unvisited mer graph will be divided into two connected component graphs, then the search process can stop progressing the search with $V_{art}$ as the active mer since there will be no path that links $G_a$ and $G_b$ together in any ramification from $V_{art}$. If there exists such path that connects $G_a$ and $G_b$ together then $G_a$ and $G_b$ will not be two connected component graphs with articulation point $V_{art}$ which contradicts our assumption.

The arguments for the case of deciding to go in $G_b$ via $V_b$ first are similar.

# CHAPTER 6

# PARALLEL EXACT ENUMERATION OF LATTICE GRAPHS

In this chapter, a set of parallel exact enumeration algorithms are presented. Speed up and communication overheads of this parallel parallel algorithm are analyzed.

## 6.1 Master and workers programming model

In this thesis, I will present a parallel programming model called master and workers model. The master and workers paradigm is a very intuitive way of executing tasks in parallel. Imagine you are a software designer. You have a number of people do programming jobs for you. As a designer, you will first define problems that need to be solved, divide the problems into subproblems and so on. Note that during the problem division phase, you try to minimize dependency between these subproblems so that programmers need not to communicate or depend on other programmers too much. The more the subproblems are independent, the lower the communication cost. This is very important in parallel program design because communication and synchronizations are expensive operations in parallel computers. After you finish dividing the problems into subproblems, you will distribute the subproblems to each individual worker to perform these subproblems concurrently. When each individual worker finishes its assignment, it reports the result back to the master. The master will then combine all results received from workers together into the final solution.

It is observed that master and workers model can be scaled in a hierarchical way. Imagine a hierarchy of people in an organization. You may be the biggest boss of the company. When you want something to be done, you will ask your employees to do it. Those employees may ask other employees at the lower level in organization hierarchy to do the jobs (as long as the job can be divided) and so on. Therefore, if the problem size is bigger and the number of employees in that organization is larger, then the problem should be effectively solved as in the case of a smaller problem and a small number of employees.

## 6.2 Parallel exact enumeration algorithms

Fortunately, the algorithm to remove redundant structure (algorithm 5.1) is perfectly fit to the master-slave paradigm. The following is a parallel algorithm to do enumeration based on redundant structure avoidance algorithm presented in chapter 5.

The followings are available globally as pre-computed data or user input data:

- Volume-constraint lattice graph is an undirect, connected graph $VCG(M,L)$ of volume lattice with dimension $XDim \times YDim \times ZDim$. Each mer on lattice graph can be indexed by the index number computed by

  $$I(x,y,z) = z \times (XDim \times YDim) + y \times (XDim) + x$$

  For example, a lattice site at the Cartesian coordinate $(x,y,z)$ = (2,3,1) of volume lattice graph with dimension $(XDim, YDim, ZDim)$ = (3,3,2) is indexed as the lattice site number 20.

- The required monomer sequence length, ie. the number of element in a sequence that is considered a dead sequence is $L_{REQ}$.

- A contact constraint map $CC_{MAP}$ for the enumeration with contact constraint. This is a table with each entry represents a pairs of sequence positions that imposes near-neighborness of mers on that sequence position pairs.

Algorithm 6.1: **Parallel Enumeration of Self-Avoiding Walk on Cubic Lattice**

[ *Enumerate all possible geometrically distinct conformations of a given cubic lattice. The parallel implementation is based on a parallel programming model under UNIX environment, ie. using processes and inter-process communication provided by UNIX environment. The interprocess communication mechanism uses in this algorithm is a half- duplex 'pipe' communication channel. A set of multiple processes is created under UNIX environment to execute a set of parallel tasks, the enumeration processes.* ]

1. Let $Q_{nri}$ be a queue of initial conformations which is initialized to a empty queue.

2. Let $M_{nr}$ be a set of initial mers on a given cubic lattice in case of volume constraint enumeration which is not redundant (subjected to rotational symmetry), or $M_{nr}$ be a mer, usually with index 0, in case of contact constraint enumeration

3. While $M_{nr}$ is not empty do

   ▷ Pick a member of $M_{nr}$ called it $C_{mer}$ and update $M_{nr}$ to be $M_{nr} - C_{mer}$

   ▷ Generate non-redundant initial conformation path set from this $C_{mer}$ and record all such conformations in $Q_{nri}$ queue using algorithm 4.1.

4. Install a buffered communication channel, a *'pipe'* (an interprocess communication facility on UNIX), which allows interprocess communication between the master process and its slave processes.

5. If there is no initial conformation in $Q_{nri}$, ie. $Q_{nri}$ is empty then stop. (* we are done *)

6. Otherwise for each initial conformation path $P_{ic}$ in $Q_{nri}$ do the following:

▷ Create a parallel slave process to enumerate all possible distinct conformations of initial conformation $P_{ic}$ by ramifying $P_{ic}$ as an self-avoiding walk in depth-first search manner.

▷ This parallel process will enumerate a $P_{ic}$ using self-avoiding walk, ie. each parallel process will execute the algorithm 5.3, 5.4 or 5.5 (depending upon the enumeration constraint) with $P_{ic}$ as an input.

▷ When a slave process finishes the enumeration on a given $P_{ic}$, it sends the enumeration result to the master process by putting it into the communication *'pipe'* and terminates itself.

7. The master process reads the communication *'pipe'* and sums up all the results which is the total number of geometrically distinct conformation of a given constrained cubic lattice.

End–Algorithm 6.1

### Algorithm 6.2: Disconnected Volume Graph Detection

[ *Check to see whether the self-avoiding walk process reached the state where the specified volume lattice graph has been divided into two disconnected components. This means we want to check the connectivity of unvisited mers in the specified volume lattice graph. It returns true if input lattice volume constraint graph is a single connected component, otherwise return false.* ]

1. **parameter:** $VG_{uv}$ be a set of unvisited mers in a volume constraint graph.

2. Let $M_{curr}$ be any member of set $VG_{uv}$.

3. Let $Q_M$ be a queue of mers with $M_{curr}$ as the only element in it.

4. Let $Head(Q), Dequeue(Q), Enqueue(Q, M), Empty(Q)$ be generic queue functions that perform the following functions, return $Q$ head, dequeue $Q$,

enqueue a new element $M$ to $Q$ and test emptiness of $Q$ respectively. Note that the $Q$ itself does not change when we apply functions $Head()$ and $Empty()$ to it.

5. Let $N$ be a set of all near-neighbor mers of a mer $X$.

6. While $Empty(Q_M)$ is false do the following:

   ▷ Set $N$ to all near-neighbors of $Head(Q_M)$ which are not already in $Q_M$.

   ▷ If $N$ is empty then $Dequeue(Q_M)$ and discard the element.

   ▷ Otherwise, enqueue all members of set $N$ to $Q_M$ and set $VG_{uv}$ to $VG_{uv} - N$.

7. If $VG_{uv}$ is empty then return true. Otherwise return false.

End–Algorithm 6.2

**Algorithm 6.3: Volume Constraint Enumeration**

[ *Enumerate all geometrically distinct sequence conformations which ramify from a given initial geometrically distinct conformation and subject to a volume constraint $VCG(M, L)$. The ramification of a given initial conformation is a self-avoiding walk based on the depth first search (DFS) algorithm.* ]

1. **parameter:** $S_{mmc}$ is a sequence of lattice site index number defined on volume lattice graph $VCG(M, L)$ and represent an initial conformation mer sequence we want to ramify to find all completed conformations (dead sequence) based on this initial sequence.

2. Let $NNB(M)$ be a mapping set contains all unvisited near-neighbor of mer $M$. This mapping set can be computed from the volume constraint graph $VCG(M, L)$ and the initial conformation sequence $S_{mmc}$ as follows:

- ▷ For each mer $M$ do the following:
  - o Determine all the near-neighbors of $M$, record this in $NNB(M)$.
  - o Set $NNB(M)$ to $NNB(M) - S_{mmc}$

3. Let $C_{ds}$ be a counter of found 'dead sequences' (a geometrically distinct conformation from a given initial conformation sequence $S_{mmc}$) and initialize it to 0 (zero).

4. Let $L_{curr}$ and $L_{init}$ be the lengths of initial input sequence $S_{mmc}$.

5. Connect the communication '*pipe*' to the master process.

6. If $L_{init}$ is equal to the length of required monomer sequence $L_{REQ}$ then send 1 (one) as the number of found 'dead sequence' through the communication '*pipe*' channel to the master process. Terminate the process. (* we are done *)

7. Otherwise, do the following:

   - ▷ Let $M_{curr}$ be the last mer of the conformation at any stage of ramification. Initialize this variable to the last mer of $S_{mmc}$.
   - ▷ Let $M_{next}$ be any unvisited near-neighbor mer of $M_{curr}$. There is no need to initialize this variable at this time.
   - ▷ While $L_{curr}$ is greater than or equal to $L_{init}$ do the following:
     - o Set $M_{next}$ to one of unvisited near-neighbor mers of $M_{curr}$ (any mer in $NNB(M_{curr})$) and update $NNB(M_{curr})$ to $NNB(M_{curr}) - M_{next}$.
     - o If there is no $M_{next}$ or there is $M_{next}$ but if we select that $M_{next}$, the lattice volume graph $VCG(M, L)$ will be divided into two connected components according to the algorithm 5.2 (check connectivity), then backtrack the self-avoiding walk by decreasing $L_{curr}$ by 1 and set $M_{curr}$ to the mer at the position $L_{curr}$ of $S_{mmc}$ sequence.

- Otherwise, set $M_{curr}$ to $M_{next}$, increase $L_{curr}$ by 1 and update sequence $S_{mmc}$ at the position $L_{curr}$ to $M_{curr}$.
- If the sequence length $L_{curr}$ is equal to the required monomer sequence length $L_{REQ}$, then increment the number of 'dead sequence' $Cds$ by 1.

▷ send the number of 'dead sequence' $C_{ds}$ to the master process through the communication '*pipe*' and terminate the process.

End–Algorithm 6.3

Algorithm 6.4: **Volume and Contact Constraint Enumeration**

[ *Enumerate all geometrically distinct sequence conformations which ramify from a given initial conformation and subject to both a volume constraint $VCG(M, L)$ and a contact constraint $CC_{MAP}$. The ramification of a given initial conformation is a self-avoiding walk process based on the depth first search (DFS) algorithm.* ]

1. **parameter:** $S_{mmc}$ is a sequence of lattice site index number defined on volume lattice graph $VCG(M, L)$ and represent an initial conformation mer sequence we want to ramify to find all completed conformations (dead sequence) based on this initial sequence.

2. Let $NNB(M)$ be a mapping set contains all unvisited near-neighbor of mer $M$. This mapping set can be computed from the volume constraint graph $VCG(M, L)$ and the initial conformation sequence $S_{mmc}$ as in algorithm 5.3.

3. Let $C_{ds}$ be a counter of found 'dead sequences' (a geometrically distinct conformation from a given initial conformation sequence $S_{mmc}$) and initialize it to 0 (zero).

4. Let $L_{curr}$ and $L_{init}$ be the lengths of initial input sequence $S_{mmc}$.

5. Connect the communication *'pipe'* to the master process.

6. Verify that $S_{mmc}$ satisfies the contact constraint by checking whether there is any pairs of mer in $S_{mmc}$ violates the contact requirement according to the contact constraint map $CC_{MAP}$. If it does not, send 0 (zero) through the communication *'pipe'* to the master process as no 'dead sequence' has been found. Terminate the process. (we are done).

7. If $L_{init}$ is equal to the length of required monomer sequence $L_{REQ}$ then send 1 (one) as the number of found 'dead sequence' through the *'pipe'* to the master process. Terminate the process. (* we are done *)

8. Otherwise, do the following:

   ▹ Let $M_{curr}$ be the last mer of the conformation at any stage of ramification. Initialize this variable to the last mer of $S_{mmc}$.

   ▹ Let $M_{next}$ be any unvisited near-neighbor mer of $M_{curr}$. There is no need to initialize this variable at this time.

   ▹ While $L_{curr}$ is greater than or equal to $L_{init}$ do the following:

     o Set $M_{next}$ to one of unvisited near-neighbor mers of $M_{curr}$ and update $NNB(M_{curr})$ to $NNB(M_{curr}) - M_{next}$. Verify that this $M_{next}$ does not violate the contact constraint requirement. If it does, we have to pick another mer that doesn't by repeating this step until we find it or there is no more unvisited near-neighbor mers of $M_{curr}$ left.

     o If there is no such $M_{next}$ or if there is such $M_{next}$ but if we choose that $M_{next}$, then it will divide the lattice volume graph $VCG(M, L)$ in to two disconnected components according to the algorithm 5.2 (check connectivity), then backtrack the self-avoiding walk by decreasing

$L_{curr}$ by 1 and set $M_{curr}$ to the mer at the position $L_{curr}$ of $S_{mmc}$ sequence.

- o Otherwise, set $M_{curr}$ to $M_{next}$, increase $L_{curr}$ by 1 and update sequence $S_{mmc}$ at the position $L_{curr}$ to $M_{curr}$.
- o If the sequence length $L_{curr}$ is equal to the required monomer sequence length $L_{REQ}$, then increment the number of 'dead sequence' $C_{ds}$ by 1.

▹ Send the number of 'dead sequence' $C_{ds}$ to the master process through the communication '*pipe*' and terminate the process.

End–Algorithm 6.4

**Algorithm 6.5: Contact Constraint Enumeration**

[ *Enumerate all geometrically distinct sequence conformations which ramify from a given initial conformation and subject to a contact constraint $CC_{MAP}$. The ramification of a given initial conformation is a self-avoiding walk process based on the depth first search (DFS) algorithm.* ]

1. **parameter:** $S_{mmc}$ is a sequence of lattice site index number defined on volume lattice graph $VCG(M, L)$ and represent an initial conformation mer sequence we want to ramify to find all completed conformations (dead sequence) based on this initial sequence.

2. Let $NNB(M)$ be a mapping set contains all unvisited near-neighbor of mer $M$. This mapping set can be computed from the volume constraint graph $V\check{C}G(M, L)$ and the initial conformation sequence $S_{mmc}$ as in algorithm 5.3.

3. Let $C_{ds}$ be a counter of found 'dead sequences' (a geometrically distinct conformation from a given initial conformation sequence $S_{mmc}$) and initialize it to 0 (zero).

4. Let $L_{curr}$ and $L_{init}$ be the lengths of initial input sequence $S_{mmc}$.

5. Connect the communication *'pipe'* to the master process.

6. Verify that $S_{mmc}$ satisfies the contact constraint by checking whether there is any pairs of mer in $S_{mmc}$ violates the contact requirement according to the contact constraint map $CC_{MAP}$. If it does not, send 0 (zero) through the communication *'pipe'* to the master process as no 'dead sequence' has been found. Terminate the process. (we are done).

7. If $L_{init}$ is equal to the length of required monomer sequence $L_{REQ}$ then send 1 (one) as the number of found 'dead sequence' through the *'pipe'* to the master process. terminate the process. (* we are done. *)

8. Otherwise, do the following:

   ▷ Let $M_{curr}$ be the last mer of the conformation at any stage of ramification. Initialize this variable to the last mer of $S_{mmc}$.

   ▷ Let $M_{next}$ be any unvisited near-neighbor mer of $M_{curr}$. There is no need to initialize this variable at this time.

   ▷ Repeat

   - o Set $M_{next}$ to one of unvisited near-neighbor mers of $M_{curr}$ and update $NNB(M_{curr})$ to $NNB(M_{curr}) - M_{next}$. Verify that this $M_{next}$ does not violate the contact constraint requirement. If it does, we have to pick another one that doesn't by repeating this step until we find it or there is no more unvisited near-neighbor mers of $M_{curr}$ left.

   - o If there is no such $M_{next}$ then backtrack the self-avoiding walk by decreasing $L_{curr}$ by 1 and set $M_{curr}$ to the mer at the position $L_{curr}$ of $S_{mmc}$ sequence.

```
        o Otherwise, set $M_{curr}$ to $M_{next}$, increase $L_{curr}$ by 1 and update
          sequence $S_{mmc}$ at the position $L_{curr}$ to $M_{curr}$
        o If the sequence length $L_{curr}$ is equal to the required monomer sequence
          length $L_{REQ}$, then increment the number of 'dead sequence' $C_{ds}$ by
          1.
      until $L_{curr}$ is smaller than $L_{init}$
    ▷ send the number of 'dead sequence' $C_{ds}$ to the master process through the
      communication 'pipe' and terminate the process.
End–Algorithm 6.5
```

## 6.3 Implementation

The sequential enumeration algorithm was originally written in the UNIX environment. Then the parallel algorithm has been developed using multiple concurrent processes on a parallel computer running UNIX. There is a parent process acts as the master running algorithm 5.1 which creates a set of child processes (workers) to enumerate all conformations based on a set of initial conformations. When the child processes finish their jobs, they will report the results back to the master (parent) process.

The parallel implementation was done on a Silicon Graphics Iris parallel computer at Harvard Chemistry Department. All codes are conformed to POSIX.1 standard and should be portable to any POSIX.1 compliant system.

# CHAPTER 7

# RESULTS AND DISCUSSIONS

In this chapter, some enumeration results will be presented. The speed up of these results as well as communication cost will also be discussed.

## 7.1 Exact enumeration results with various constraints

The followings are results of some volume and/or contact constraint enumerations. Table 7.1 shows volume constraint enumerations. In all tables, initial conformation is the total number of non-redundant conformation of a given lattice graph as well as the number of independent parallel tasks. The unique conformation is the total number of geometrically distinct conformations. Table 7.2 is the sample of volume and contact-constraint enumeration on a $3 \times 3 \times 3$ cubic lattice with various contact-constraint lists. Table 7.3 is the sample of contact-constraint enumeration of a chain of 30 monomers.

## 7.2 Discussions

The following three tables show the running time of all algorithms on a Silicon Graphics Iris parallel computer with 4 processors. Table 7.4 shows speed up of volume constraint enumeration. Table 7.5 shows speed up of volume and contact constraint enumeration. Table 7.5 shows speed up of contact constraint enumeration.

From table 7.4, 7.5, 7.6, the speed up are almost linear from one to four processors. The communication cost between the master and workers is very low in all algorithms. Actually, there is no synchronization at all between worker processes. This is a very desirable characteristic to achieve an almost linear speedup since

Table 7.1 Volume constraint enumeration of various volumes

| Volume | Chain Length | Number of Conformations | | Time |
|---|---|---|---|---|
| | | Initial | Unique | (seconds) |
| $2 \times 2 \times 2$ | 8 | 2 | 3 | 0.77 |
| $2 \times 2 \times 3$ | 12 | 24 | 73 | 1.96 |
| $2 \times 3 \times 3$ | 18 | 116 | 2110 | 6.81 |
| $3 \times 3 \times 3$ | 27 | 152 | 103346 | 65.49 |

Table 7.2 Volume and contact constraint enumerations of a volume $3 \times 3 \times 3$

| Contact Constraint List | Number of Conformations | | Time |
|---|---|---|---|
| | Initial | Unique | (seconds) |
| 21-12, 23-16 | 152 | 1594 | 22.27 |
| 15-4, 11-26 | 152 | 3482 | 11.6 |
| 18-11, 25-8 | 152 | 2475 | 17.6 |
| 8-17, 21-2 | 152 | 2562 | 13.41 |

Table 7.3 Contact constraint enumeration of a chain of 30 monomers

| Contact Constraint List | Number of Conformations | | Time |
|---|---|---|---|
| | Initial | Unique | (seconds) |
| 2-11 29-10 28-3 13-0 18-7 5-16 | 513 | 22540648 | 40961.66 |
| 12-5 2-29 20-15 3-14 8-1 17-24 | 64 | 40407229 | 17317.71 |
| 2-11 19-2 22-15 25-4 14-7 25-8 | 619 | 382716 | 55.13 |
| 2-17 13-4 24-5 9-2 28-9 7-20 | 178 | 435183 | 261.91 |

Table 7.4 Speed up of a volume constraint enumeration of volume $3 \times 3 \times 3$

| Number of PEs | Time (sec) | Speed Up |
|---|---|---|
| 1 | 62.21 | 1 |
| 2 | 32.09 | 1.94 |
| 3 | 21.57 | 2.88 |
| 4 | 16.1 | 3.86 |

Table 7.5 Speed up of a volume and contact constraint enumerations of volume $3 \times 3 \times 3$

| Number of PEs | Contact Constraint List | Time (sec) | Speed Up |
|---|---|---|---|
| 1 | 21-12, 23-16 | 23.59 | 1 |
| 2 | | 12.35 | 1.91 |
| 3 | | 8.25 | 2.86 |
| 4 | | 6.10 | 3.87 |

Table 7.6 Speed up of contact constraint enumeration of a chain of 30 monomers

| Number of PEs | Contact Constraint List | Time (sec) | Speed Up |
|---|---|---|---|
| 1 | 2-17 13-4 24-5 9-2 28-9 7-20 | 259.22 | 1 |
| 2 | | 138.62 | 1.87 |
| 3 | | 88.77 | 2.92 |
| 4 | | 67.68 | 3.83 |

process communication and synchronization are very expensive operations in any parallel computer system.

# CHAPTER 8

# CONCLUSIONS

This thesis presented a set of parallel algorithms to solve exact enumeration problems. It discussed the application of exact enumerations to the protein folding problem. The experimental results justified the success of applying parallel processing to a class of lattice graph enumeration problems. The speed up obtained by these algorithms are almost linear in all algorithm which are due to the fact that all subtasks are highly independent. The communication and synchronization between subtasks are very low.

There are also a number of enumeration problems that need to be solved and require even more computing power than problems discussed in this thesis, for example, a binomial contact constraint exact enumeration.

# APPENDIX A

# AN IMPLEMENTATION OF PARALLEL VOLUME AND CONTACT CONSTRAINT EXACT ENUMERATION

In followings are the implementation of parallel exact enumerations wi th volume, and volume and contact constraint, ie. algorithms 4.1, 5.1, 5. 2, 5.3 and 5.4.

```
/*
   Cubic lattice enumeration with volume, and volume and contact const
raint

   by: Anek Vorapanya
   last modified: summer of 1993
   known bugs: none

   portability: all environments conform to the POSIX.1 standard
   language standard: ANSI C

   compiler: GNU compiler (gcc)
   compiler options: -ansi -O2

   usage: gwalk -h

   tested environment:
       - ULTRIX 4.3A (Rev. 146) (DEC 5900)
       - SUN OS 4.1.3 (Sun)
       - AIX 3.1.2 (IBM RS/6000)
       - IRIX 5.1 (SGI)
*/

#include <string.h>
#include <malloc.h>
#include <math.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <stdarg.h>

#define MIN_ARGUMENT        4        /* min. argument required */
#define MAX_PROCESS            1        /* default max. processes */
#define DEF_CNT_LV_CHK       2        /* default connect level check,
 min.=2 */

#define MAX_DIR                6
#define MAX_NEIGHBOR        6
#define INVALID_NODE        (-1)
#define INVALID_NNBNO        (-1)
#define OPPOSITE_DIR_DIFF    (abs(PLUS_X-MINUS_X))

#define DISPCNT_MOD            1000L
#define MIN_DISPCNT_MOD        100L
#define MAX_PIPL            50    /* max. % of initial path length */

typedef enum { false, true } boolean;

typedef enum {
    PLUS_X='0',
    MINUS_X='1',
    PLUS_Y='2',
    MINUS_Y='3',
    PLUS_Z='4',
    MINUS_Z='5',
    INVALID_DIR='#'
    } DIRECTIONS;

typedef struct {
    int node;
    char dir;
    } neighbor;

typedef struct {
    int x;    /* first node of contact */
    int y;    /* second node of contact */
    } contact;

typedef neighbor neighborlist[MAX_NEIGHBOR];

typedef struct {
    int *node;    /* currently visited node no. at this level in path
*/
    int *nnbno;    /* currently use of child no of this node */
    DIRECTIONS *dir;
    } path;

typedef struct {
    boolean *symnodeflag;
    } symnodeseq;

typedef enum { _0Dpath, _1Dpath, _2Dpath, _3Dpath } pathtype;

typedef enum { invalid_planetype, xyplane, yzplane, xzplane } planetyp
e;

typedef struct pathlist_s {
    pathtype ptype;
    planetype pltype;
    int last;
    int *node;
    DIRECTIONS *dir;
    boolean *usedflag;
    struct pathlist_s *next;
    } pathlist;


int xsize;
int ysize;
int zsize;
long sys_max_child;
long max_process;
int min_level;
int dispcnt_mod;
enum { gen_initial_path, count_path, gen_path } option;
boolean verbose=false;
boolean resource_usage=false;
int totalnode;
neighborlist *nearneighbor;
path *p;
boolean *usedflag;
long pathcount;
long total_nonsym_initial=0L;
int *num_nearneighbor;
symnodeseq *symnodetable;
int *nonsymnodelist;
int totalnonsymnode;
pathlist *stpl;
pathlist *nspl;
pathlist *last_stpl;
pathlist *last_nspl;
pathlist head_stpl;
pathlist head_nspl;
FILE *gwalk_log;
FILE *fp;
char filename[256];
char initialnodes[1024];
long clktck;
boolean *tmp_used;    /* temporary 'usedflag' */
int    *queue;
contact *contact_set;
int total_contact=0;
FILE *fout;


void error (char *fmt, ...)
{
    va_list ap;
    char buf[512];

    va_start(ap,fmt);
    vsprintf (buf,fmt,ap);
    va_end(ap);
    write (fileno(stderr),buf,strlen(buf));
    exit (1);
}

void usage (void)
{
    char tmp[128];

    fprintf (stderr,"Enumerate geometrically-distinct walk for a X*Y*Z
 volume\n");
    fprintf (stderr,"usage: gwalk <xsize> <ysize> <zsize> [options]\n"
);
    fprintf (stderr,"options:\n");
    fprintf (stderr,"  -i : to generate initial paths only (save in lo
g file)\n");
    fprintf (stderr,"  -c : to count all possible paths\n");
    fprintf (stderr,"  -w : to save generated paths in file (save in p
ath.xsize.ysize.zsize)\n");

    if (sys_max_child==0)
        strcpy (tmp,"unknown");
    else
        sprintf (tmp,"%ld",sys_max_child-1L);

    fprintf (stderr,"  -pX : specify no. of processes to run simultane
ously (your system max %s)\n",tmp);
    fprintf (stderr,"  -lX : specify percentage of initial path length
 (recommend 15-25,max %d)\n",MAX_PIPL);
    fprintf (stderr,"  -tX <contact set>: specify contact set where X
is number of contacts\n");
    fprintf (stderr,"      and <contact set> is a contact set in the f
orm 'a-b c-d e-f ...' \n");
    fprintf (stderr,"  -dX : specify counter display modulo value (min
 %d)\n",MIN_DISPCNT_MOD);
    fprintf (stderr,"  -v : show the enumeration counting\n");
    fprintf (stderr,"  -r : show resource usage\n");
    fprintf (stderr,"  -h  : print this help screen\n");
    exit (0);
}

void cmdline_parse (int argc,char *argv[])
{
    char *s;
    char tmp[1024];
    char c;
    int i,j,p;
    int contact_no=(-1);
    int tmptotalnode;

    xsize=0;
    ysize=0;
    zsize=0;
    option=count_path;
    min_level=0;
    dispcnt_mod=DISPCNT_MOD;

    if ((sys_max_child=sysconf(_SC_CHILD_MAX))==0) {
        fprintf (stderr,"* can't determine 'max. processes for user'\n
");
        fprintf (stderr,"  use %d as a default value\n",MAX_PROCESS);
        sys_max_child=MAX_PROCESS;
        }

    /* set default max. process */
    max_process=MAX_PROCESS;

    if (argc<MIN_ARGUMENT)
        usage();

    for (i=1;i<argc;i++) {
        if (i<MIN_ARGUMENT) {
            switch (i) {
                case 1:
                    if ((xsize=atoi(argv[1]))==0 || xsize<2)
                        usage();
                    break;
                case 2:
                    if ((ysize=atoi(argv[2]))==0 || ysize<2)
                        usage();
                    break;
                case 3:
                    if ((zsize=atoi(argv[3]))==0 || zsize<2)
                        usage();
                    break;
                }
            }
        else {
            if (contact_no==(-1)) {
                s=argv[i];
                if (s[0]!='-')
                    usage();
                s++;
                c=tolower(s[0]);
                }
            else
                c='t'; /* continue scanning contact set */

            switch (c) {
                case 'c': /* count path only */
                    option=count_path;
                    break;
                case 'i': /* generate only initial path and save in lo
g file */
                    option=gen_initial_path;
                    break;
                case 'w': /* generate path file */
                    option=gen_path;
                    /* prepare output file */
                    strcpy (filename,"path");
                    for (j=1;j<MIN_ARGUMENT;j++) {
                        strcat(filename,".");
                        strcat(filename,argv[j]);
                        }
                    if ((fp=fopen(filename,"wb"))==NULL)
                        error ("error: can't open output file (%s)\n",
filename);
                    break;
                case 'l': /* minimum start level */
                    if ((p=atoi(++s))==0 || p>MAX_PIPL)
                        usage();
                    min_level=xsize*ysize*zsize*p/100;
                    fprintf (stderr,"* use min. initial level = %d\n",
min_level);
                    break;
                case 'p': /* maximum processes to run simultaneous */
                    /* we have to subtract 1 from sys_max_child
                       based on assumption that we will have one
                       and only one process running (parent).
                    */
                    if ((max_process=atol(++s))==0 || max_process>(sys
_max_child-1L))
                        usage();
                    break;
                case 'd': /* counter display modulo value */
                    if ((dispcnt_mod=atoi(++s))==0 || dispcnt_mod<MIN_
DISPCNT_MOD)
                        usage();
                    break;
                case 'v': /* verbose */
                    if (s[1]=='-')
                        verbose=true;
                    break;
                case 't': /* contact constraint set */
                    if (contact_no!=-1) {
                        if ((s=strchr(argv[i],'-'))!=NULL) {
                            s[0]='\0';
                            if (((contact_set[contact_no].x=atoi(argv[
i]))==0 && argv[i][0]!='0') || contact_set[contact_no].x>=xsize*ysize*
zsize)
```

```
                                error("error: invalid contact set no.
(%s)\n",argv[i]);
                            s++;
                            if (s[0]=='\0')
                                error("error: incomplete contact set (
contact no. %d)\n",contact_no);
                            if (((contact_set[contact_no].y=atoi(s))==
0 && s[0]!='0') || contact_set[contact_no].y>=xsize*ysize*zsize)
                                error("error: invalid contact set no.
(%s)\n",s);
                            }
                        else
                            error("error: contact set requires 'x-y' f
ormat where x and y are nodes no. to form the contact.\n");

                        if (++contact_no==total_contact)
                            contact_no=(-1);
                        }
                    else {
                        if ((total_contact=atoi(++s))!=0) {
                            /* allocate contact set data */
                            contact_set=(contact *)malloc(total_contac
t*sizeof(contact));
                            contact_no=0;
                            }
                        else
                            usage();
                        }
                    break;
                case 'r':
                    resource_usage=true;
                    break;
                default:
                    fprintf (stderr,"* error: unknown options (%c)\n",
s[0]);
                case 'h':
                    usage();
                    break;
                }
            }
        }

    /* check if we got all contact as specify with -tX option or not *
/
```

```
    if (contact_no!=-1)
        error("error: number of contacts mismatched\n");
}

void reset_usedflag (void)
{
    int i;

    for (i=0;i<totalnode;i++)
        usedflag[i]=false;
}

void create_near_neighbor_table (void)
{
    int node,count;
    void sort_nearneighbor (void);

    for (node=0;node<totalnode;node++) {
        count=0;
        if ((node%xsize)<(xsize-1)) {
            nearneighbor[node][count].node=node+1;
            nearneighbor[node][count].dir=PLUS_X;
            count++;
            }
        if ((node%xsize)>0) {
            nearneighbor[node][count].node=node-1;
            nearneighbor[node][count].dir=MINUS_X;
            count++;
            }
        if ((node%(xsize*ysize))<(xsize*ysize-xsize)) {
            nearneighbor[node][count].node=node+xsize;
            nearneighbor[node][count].dir=PLUS_Y;
            count++;
            }
        if ((node%(xsize*ysize))>=xsize) {
            nearneighbor[node][count].node=node-xsize;
            nearneighbor[node][count].dir=MINUS_Y;
            count++;
            }
        if ((node+(xsize*ysize))<totalnode) {
            nearneighbor[node][count].node=node+(xsize*ysize);
            nearneighbor[node][count].dir=PLUS_Z;
            count++;
            }
        if ((node-(xsize*ysize))>=0) {
            nearneighbor[node][count].node=node-(xsize*ysize);
            nearneighbor[node][count].dir=MINUS_Z;
            count++;
            }
        num_nearneighbor[node]=count;
        }

    sort_nearneighbor ();
}

void sort_nearneighbor (void)
{
    boolean changed;
    int i,j;
    neighbor tmp;

    /* sort neighbor list using bubble sort */
    for (i=0;i<totalnode;i++) {
        do {
            changed=false;
            for (j=0;j<num_nearneighbor[i]-1;j++) {
                if (nearneighbor[i][j].node>nearneighbor[i][j+1].node)
 {
                    tmp=nearneighbor[i][j];
                    nearneighbor[i][j]=nearneighbor[i][j+1];
                    nearneighbor[i][j+1]=tmp;
                    changed=true;
                    }
                }
            } while (changed==true);
        }
}

boolean is_neighbor (int x,int y) {
    int i;

    for (i=0;i<num_nearneighbor[x];i++) {
        if (nearneighbor[x][i].node==y)
            return true;
        }

    return false;
}
```

```
void set_contact_constraint (void) {
    int i,j;
    contact tmp;

    /* switch position of x and y such that x is higher than y */
    for (i=0;i<total_contact;i++) {
        if (contact_set[i].x<contact_set[i].y) {
            /* swap */
            j=contact_set[i].x;
            contact_set[i].x=contact_set[i].y;
            contact_set[i].y=j;
            }
        /*
        printf ("switch contact %d (%d-%d)\n",i,contact_set[i].x,conta
ct_set[i].y);
        */
        }

    /* we will sort it using bubble sort */
    for (i=0;i<total_contact;i++) {
        for (j=total_contact-1;j>i;j--) {
            if (contact_set[j].x<contact_set[j-1].x) {
                /* swap */
                tmp=contact_set[j];
                contact_set[j]=contact_set[j-1];
                contact_set[j-1]=tmp;
                }
            }
        fprintf (stderr,"sort contact %d (%d-%d)\n",i,contact_set[i].x
,contact_set[i].y);
        }

}

DIRECTIONS getdir (int cnode,int nnode)
{
    int i;

    /* find direction from this nextnode to destination */
    for (i=0;i<num_nearneighbor[cnode] && nearneighbor[cnode][i].node<
=nnode;i++) {
        if (nearneighbor[cnode][i].node==nnode) {
            return nearneighbor[cnode][i].dir;
            }
        }
    return INVALID_DIR;
}

int getnextnode (path *p,int level)
{
    int i,validnnbno;
    int node;

    node=p->node[level];
    validnnbno=0;
    for (i=0;i<num_nearneighbor[node];i++) {
        if (!usedflag[nearneighbor[node][i].node] && validnnbno++==p->
nnbno[level]) {
            p->dir[level]=nearneighbor[node][i].dir;
            return nearneighbor[node][i].node;
            }
        }
    return INVALID_NODE;
}

void list_nearneighbor (path *p)
{
    int l,tmp,j;

    for (l=0;l<totalnode;l++) {
        fprintf (fout,"node %d (#nb %d): ",l,num_nearneighbor[l]);
        p->node[0]=l;
        usedflag[l]=true;
        for (j=0;j<num_nearneighbor[l];j++) {
            p->nnbno[0]=j;
            if ((tmp=getnextnode (p,0))!=INVALID_NODE) {
                fprintf (fout,"%d ",tmp);
                usedflag[tmp]=false;
                }
            }
        usedflag[l]=false;
        fprintf (fout,"\n");
        }
}

void print_symnodetable (FILE *stream)
{
    int i,j;

    for (i=0;i<totalnode;i++) {
        fprintf (stream,"node %d: ",i);
        for (j=0;j<totalnode;j++)
            if (symnodetable[i].symnodeflag[j]==true)
                fprintf (stream,"%d ",j);
        fprintf (stream,"\n");
        }
}

int point (int x,int y,int z)
{
    if (x>=xsize || y>=ysize || z>=zsize)
        error("internal error: invalid x,y,z (%d,%d,%d), max size (%d,
%d,%d)\n",x,y,z,xsize,ysize,zsize);
    return (int)((int)(z*ysize*xsize)+(int)(y*xsize)+x);
}

void gen_non_symmetry_node (void)
{
    int i,j;
    boolean *symnodemark;

    symnodemark=(boolean *)malloc(sizeof(boolean)*totalnode);
    if (symnodemark==NULL)
        error ("error: memory allocation (symnodemark)\n");

    for (i=0;i<totalnode;i++)
        symnodemark[i]=false;

    totalnonsymnode=totalnode;

    for (i=0;i<totalnode;i++) {
        if (symnodemark[i]==true)
            continue;
        for (j=0;j<totalnode;j++) {
            if (symnodetable[i].symnodeflag[j]==true && i!=j) {
                symnodemark[j]=true;
                totalnonsymnode--;
                }
            }
        }
    nonsymnodelist=(int *)malloc(sizeof(int)*totalnonsymnode);
    if (nonsymnodelist==NULL)
        error ("error: memory allocation (nonsymnodelist)\n");

    j=0;
    for (i=0;i<totalnode;i++) {
        if (symnodemark[i]==false)
            nonsymnodelist[j++]=i;
        }

    fprintf (fout,"non-symmetrical node (%d): ",totalnonsymnode);
    for (i=0;i<totalnonsymnode;i++)
        fprintf (fout,"%d ",nonsymnodelist[i]);
    fprintf (fout,"\n");

    fprintf (gwalk_log,"non-symmetrical node (%d): ",totalnonsymnode);

    for (i=0;i<totalnonsymnode;i++)
        fprintf (gwalk_log,"%d ",nonsymnodelist[i]);
    fprintf (gwalk_log,"\n");
}

void INCLUSIVE_OR_sym_nodeseq (int snode,int dnode)
{
    int i;

    for (i=0;i<totalnode;i++) {
        if (symnodetable[snode].symnodeflag[i]==true)
            symnodetable[dnode].symnodeflag[i]=true;
        else if (symnodetable[dnode].symnodeflag[i]==true)
            symnodetable[snode].symnodeflag[i]=true;
        }
}

void addsymnode (int x,int y)
{
    symnodetable[x].symnodeflag[y]=true;
    symnodetable[y].symnodeflag[x]=true;
}

void gen_sym_node_table (void)
{
    int i,j;
    int x,y,z,xx,yy,zz;
    int a,b,c,d;
    int czsize,cysize,cxsize;

    symnodetable=(symnodeseq *)malloc(sizeof(symnodeseq)*totalnode);
    for (i=0;i<totalnode;i++) {
        symnodetable[i].symnodeflag=(boolean *)malloc(sizeof(boolean)*
totalnode);
        for (j=0;j<totalnode;j++)
            symnodetable[i].symnodeflag[j]=false;
        }

    /* find symmetry node when rotate cube around z direction */

    x=0;
    y=0;
    z=0;

    for (czsize=zsize,cysize=ysize,cxsize=xsize;czsize>0;czsize-=2,cys
ize-=2,cxsize-=2) {

        for (zz=z;zz<czsize;zz++) {

            a=point(x,y,zz);
            c=point(x+cxsize-1,y+cysize-1,zz);

            b=point(x+cxsize-1,y,zz);
            d=point(x,y+cysize-1,zz);

            if (cxsize==cysize) {

                while (a<point(x+cxsize-1,y,zz)) {
                    addsymnode(a,b);
                    addsymnode(a,d);
                    addsymnode(c,b);
                    addsymnode(c,d);

                    a++;
                    b+=cxsize;
                    c--;
                    d-=cxsize;
                    }
                }
            else {
                /* corners will always symmertry to other corners */
                addsymnode(a,b);
                addsymnode(a,d);
                addsymnode(c,b);
                addsymnode(c,d);

                while (a<point(x+cxsize-1,y,zz)) {
                    addsymnode(a,c);
                    a++;
                    c--;
                    }

                while (b<point(x+cxsize-1,y+cysize-1,zz)) {
                    addsymnode(b,d);
                    b+=cxsize;
                    d-=cxsize;
                    }
                }
            }

        x++;
        y++;
        z++;
        }

    /* find symmetry node when rotate cube around x direction */

    x=0;
    y=0;
    z=0;

    for (czsize=zsize,cysize=ysize,cxsize=xsize;cxsize>0;czsize-=2,cys
ize-=2,cxsize-=2) {

        for (xx=x;xx<cxsize;xx++) {

            a=point(xx,y,z);
            c=point(xx,y+cysize-1,z+czsize-1);

            b=point(xx,y+cysize-1,z);
            d=point(xx,y,z+czsize-1);

            if (cysize==czsize) {

                while (a<point(xx,y+cysize-1,z)) {
```

```
                addsymnode(a,b);
                addsymnode(a,d);
                addsymnode(c,b);
                addsymnode(c,d);

                a+=cxsize;
                b+=cxsize*cysize;
                c-=cxsize;
                d-=cxsize*cysize;
                }
            }
        else {
            /* corners will always symmertry to other corners */
            addsymnode(a,b);
            addsymnode(a,d);
            addsymnode(c,b);
            addsymnode(c,d);

            while (a<point(xx,y+cysize-1,z)) {
                addsymnode(a,c);
                a+=cxsize;
                c-=cxsize;
                }

            while (b<point(xx,y+cysize-1,z+czsize-1)) {
                addsymnode(b,d);
                b+=cxsize*cysize;
                d-=cxsize*cysize;
                }
            }
        }

    x++;
    y++;
    z++;
    }

/* find symmetry node when rotate cube around y direction */

x=0;
y=0;
z=0;

for (czsize=zsize,cysize=ysize,cxsize=xsize;cysize>0;czsize-=2,cys
ize-=2,cxsize-=2) {

        for (yy=y;yy<cysize;yy++) {

            a=point(x,yy,z);
            c=point(x+cxsize-1,yy,z+czsize-1);

            b=point(x+cxsize-1,yy,z);
            d=point(x,yy,z+czsize-1);

            if (cxsize==czsize) {

                while (a<point(x+cxsize-1,yy,z)) {
                    addsymnode(a,b);
                    addsymnode(a,d);
                    addsymnode(c,b);
                    addsymnode(c,d);

                    a++;
                    b+=cxsize*cysize;
                    c--;
                    d-=cxsize*cysize;
                    }
                }
            else {
                /* corners will always symmertry to other corners */
                addsymnode(a,b);
                addsymnode(a,d);
                addsymnode(c,b);
                addsymnode(c,d);

                while (a<point(x+cxsize-1,yy,z)) {
                    addsymnode(a,c);
                    a++;
                    c--;
                    }

                while (b<point(x+cxsize-1,yy,z+czsize-1)) {
                    addsymnode(b,d);
                    b+=cxsize*cysize;
                    d-=cxsize*cysize;
                    }
                }
            }
        x++;
        y++;
        z++;
        }

    /* retrieve all symmetry node seq. */
    for (i=0;i<totalnode;i++) {
        for (j=0;j<totalnode;j++) {
            if (symnodetable[i].symnodeflag[j]==true)
                INCLUSIVE_OR_sym_nodeseq (i,j);
            else if (i==j)
                symnodetable[i].symnodeflag[j]=true;
            }
        }

    print_symnodetable (gwalk_log);
    fflush (gwalk_log);
}

boolean issymnode (int na,int nb)
{
    if (nb==INVALID_NODE)
        return false;
    else if (symnodetable[na].symnodeflag[nb]==true)
        return true;
    else
        return false;
}

void get_all_available_nearneighbor (boolean *usedflag,int cnode,neigh
bor *ann,int *cnt)
{
    int i;

    *cnt=0;
    for (i=0;i<num_nearneighbor[cnode];i++) {
        if (usedflag[nearneighbor[cnode][i].node]==false) {
            ann[*cnt].node=nearneighbor[cnode][i].node;
            ann[*cnt].dir=nearneighbor[cnode][i].dir;
            (*cnt)++;
            }
        }
}
```

```
void create_newpath (pathlist *orgpath,pathlist **newpath,int newlastn
ode,DIRECTIONS dirlastnode)
{
    int i;

    *newpath=(pathlist *)malloc(sizeof(pathlist));
    (*newpath)->node=(int *)malloc(sizeof(int)*totalnode);
    (*newpath)->dir=(DIRECTIONS *)malloc(sizeof(DIRECTIONS)*totalnode)
;
    (*newpath)->usedflag=(boolean *)malloc(sizeof(boolean)*totalnode);

    for (i=0;i<totalnode;i++) {
        (*newpath)->node[i]=INVALID_NODE;
        (*newpath)->dir[i]=INVALID_DIR;
        (*newpath)->usedflag[i]=false;
        }

    if (orgpath!=NULL) {
        (*newpath)->last=orgpath->last;
        (*newpath)->ptype=orgpath->ptype;
        (*newpath)->pltype=orgpath->pltype;
        memcpy((*newpath)->node,orgpath->node,orgpath->last*sizeof(int
));
        memcpy((*newpath)->dir,orgpath->dir,(orgpath->last-1)*sizeof(D
IRECTIONS));
        memcpy((*newpath)->usedflag,orgpath->usedflag,totalnode*sizeof
(boolean));
        (*newpath)->dir[(*newpath)->last-1]=dirlastnode;
        }
    else {
        (*newpath)->last=0;
        (*newpath)->ptype=_ODpath;
        (*newpath)->pltype=invalid_planetype;
        }

    (*newpath)->node[(*newpath)->last]=newlastnode;
    (*newpath)->usedflag[newlastnode]=true;
    (*newpath)->last++;
    (*newpath)->next=NULL;
}

void add_to_stpl (pathlist *p)
{
    last_stpl->next=p;
    last_stpl=p;
}

void add_to_nspl (pathlist *p)
{
    last_nspl->next=p;
    last_nspl=p;
}

void del_head_stpl (void)
{
    pathlist *p;

    if (stpl->next!=NULL) {
        p=stpl->next;
        stpl->next=stpl->next->next;
        /* throw it away */
        free (p);
        }
}

pathlist *del_head_nspl (void)
{
    pathlist *p;

    if (nspl->next!=NULL) {
        p=nspl->next;
        nspl->next=nspl->next->next;
        return p;
        }
    else
        return NULL;
}

planetype det_planetype (DIRECTIONS cdir,DIRECTIONS ndir)
{
    switch (cdir) {
        case PLUS_X:
        case MINUS_X:
            if (ndir==PLUS_Y || ndir==MINUS_Y)
                return xyplane;
            else
                return xzplane;
            break;
        case PLUS_Y:
        case MINUS_Y:
            if (ndir==PLUS_X || ndir==MINUS_X)
                return xyplane;
            else
                return yzplane;
            break;
        case PLUS_Z:
        case MINUS_Z:
            if (ndir==PLUS_Y || ndir==MINUS_Y)
                return yzplane;
            else
                return xzplane;
            break;
        }
}

void remove_ann (neighbor *ann,int idx,int *cnt)
{
    int i;

    if (*cnt!=1) {
        /* fill this 'hole' in 'ann' with last 'ann' */
        ann[idx].node=ann[*cnt-1].node;
        ann[idx].dir=ann[*cnt-1].dir;
        (*cnt)--;
        }
}

void remove_symnode (neighbor *nn,int *cnt)
{
    int i,j;

    for (i=0;i<*cnt;i++) {
        for (j=i+1;j<*cnt;) {
            if (issymnode(nn[i].node,nn[j].node))
                remove_ann (nn,j,cnt);
            else
                j++;
            }
        }
}
```

```
void gen_non_symmetry_path (void)
{
    pathlist *np;
    pathlist *cp;
    int i,nsc,cnt;
    neighbor ann[MAX_NEIGHBOR];
    boolean add;

    /* init STPL (symmetry test path list) and NSPL (non-symmetry init
ial path list */
    head_stpl.next=NULL;
    head_nspl.next=NULL;
    stpl=(pathlist *)&head_stpl;
    nspl=(pathlist *)&head_nspl;
    last_stpl=(pathlist *)&head_stpl;
    last_nspl=(pathlist *)&head_nspl;

    /* add 'non-symmetry node' to initial stpl */
    for (nsc=0;nsc<totalnonsymnode;nsc++) {
        create_newpath (NULL,&np,nonsymnodelist[nsc],INVALID_DIR);
        add_to_stpl (np);
        }

    while ((cp=stpl->next)!=NULL) {
        get_all_available_nearneighbor (cp->usedflag,cp->node[cp->last
-1],ann,&cnt);
        switch (cp->ptype) {
            case _0Dpath:
                /* we just start walking so we choose to walk
                   to only node that are non-symmetrical */
                remove_symnode (ann,&cnt);
                for (i=0;i<cnt;i++) {
                    create_newpath (cp,&np,ann[i].node,ann[i].dir);
                    np->ptype=_1Dpath;
                    np->usedflag[ann[i].node]=true;
                    add_to_stpl (np);
                    }
                break;
            case _1Dpath :
                /* find 'stright path' , this path will not
                   symmetry to other pathes */
                for (i=0;i<cnt;i++) {
                    if (ann[i].dir==cp->dir[0]) {
                        create_newpath (cp,&np,ann[i].node,ann[i].dir)
;
                        np->usedflag[ann[i].node]=true;
                        add_to_stpl (np);
                        /* remove this from available near neighbors *
/
                        remove_ann (ann,i,&cnt);
                        break;
                        }
                    }

                /* add only non-symmetry neighbor */
                /* and it will be 'plane path' from now on */
                remove_symnode (ann,&cnt);
                for (i=0;i<cnt;i++) {
                    create_newpath (cp,&np,ann[i].node,ann[i].dir);
                    np->usedflag[ann[i].node]=true;
                    np->ptype=_2Dpath;
                    np->pltype=det_planetype (cp->dir[0],ann[i].dir);
                    add_to_stpl (np);
                    }
                break;
            case _2Dpath :
                /* find 'ann' that is in plane */
                /* add it to 'stpl' (it will not symmetry) */
                for (i=0;i<cnt;) {
                    if ((cp->pltype==xyplane && ann[i].dir!=PLUS_Z &&
ann[i].dir!=MINUS_Z) || (cp->pltype==yzplane && ann[i].dir!=PLUS_X &&
ann[i].dir!=MINUS_X) || (cp->pltype==xzplane && ann[i].dir!=PLUS_Y &&
ann[i].dir!=MINUS_Y)) {
                        create_newpath (cp,&np,ann[i].node,ann[i].dir)
;
                        np->usedflag[ann[i].node]=true;
                        remove_ann (ann,i,&cnt);
                        add_to_stpl (np);
                        }
                    else
                        i++;
                    }

                /* remove symmetry node and add the rest to 'NSPL' */
                remove_symnode (ann,&cnt);
                for (i=0;i<cnt;i++) {
                    create_newpath (cp,&np,ann[i].node,ann[i].dir);
                    np->usedflag[ann[i].node]=true;
                    /* now we have non-symmetrical initial path
                       list if user didn't specify 'min. level of
                       initial path', we will use this as NSPL
                    */
                    if (np->last>=min_level) {
                        add_to_nspl (np);
                        total_nonsym_initial++;
                        }
                    else {
                        np->ptype=_3Dpath;
                        add_to_stpl (np);
                        }
                    }
                break;
            case _3Dpath :
                for (i=0;i<cnt;i++) {
                    create_newpath (cp,&np,ann[i].node,ann[i].dir);
                    np->usedflag[ann[i].node]=true;
                    if (np->last<min_level)
                        add_to_stpl (np);
                    else {
                        add_to_nspl (np);
                        total_nonsym_initial++;
                        }
                    }
                break;
            }

        del_head_stpl ();
        }
}

int satisfy_cc (int contact_no,int level,int node) {
    /* if no contact constrints, or contact constraint was already sat
isfy then return true */
    if (total_contact==0 || contact_no>=total_contact) return true;

    /* check if it satisfy contact constraints */
    /*
    fprintf (stderr,"check cc contact_no %d,level %d,node %d\n",contac
t_no,level,node);
    */
    while (level==contact_set[contact_no].x) {
        if (is_neighbor(p->node[contact_set[contact_no].y],node)==fals
e)
            return -1;    /* if it false, we can stop immediately */
        /* if it true, we have to check that it true in all contact */

        contact_no++;
        }

    /*
    fprintf (stderr,"* check ok, new contact no. %d\n",contact_no);
    */
    return contact_no;
}

boolean connect (int level,int node) {
    int i,cur,last,cnode;

    if (level>=totalnode-DEF_CNT_LV_CHK)
        return true;    /* assume default 'connect' */

    /* use BFS to check connectivity of non-planar graph */

    memcpy (tmp_used,usedflag,sizeof(boolean)*totalnode);
    tmp_used[node]=true;
    for (i=0;i<totalnode && tmp_used[i]==true;i++);
    tmp_used[i]=true;

    cur=0;
    last=0;
    queue[0]=i;

    do {
        cnode=queue[cur];
        for (i=0;i<num_nearneighbor[cnode];i++) {
            if (tmp_used[nearneighbor[cnode][i].node]==false) {
                queue[++last]=nearneighbor[cnode][i].node;
                tmp_used[nearneighbor[cnode][i].node]=true;
                }
            }
        } while (++cur<=last);

    if ((last+1)==totalnode-(level+2))
        return true;
    else
```

```
        return false;
}

void walk (path *p,int level)
{
    int i,nextnode;
    int startlevel;
    char line[256];
    int contact_no=0;
    int newctno;

    startlevel=level;

    /* check contact constraint */
    if (total_contact!=0 && level>=contact_set[contact_no].x) {
        i=contact_set[contact_no].x;    /* initial level */
        do {
            if (i==contact_set[contact_no].x) {
                if ((newctno=satisfy_cc(contact_no,i,p->node[contact_s
et[contact_no].x]))==-1)
                    return; /* this initial path didn't satisfy contac
t constraints */
                contact_no=newctno;
                }
            } while (++i<=level);
        }

    /* start walking (exhaustive search) */
    do {
        if ((nextnode=getnextnode(p,level))!=INVALID_NODE && connect(l
evel,nextnode) && (newctno=satisfy_cc(contact_no,level+1,nextnode))!=-
1) {

            p->node[++level]=nextnode;
            contact_no=newctno;

            if (level==totalnode-1) {
                ++pathcount;
                if (verbose) {
                    if (!(pathcount%dispcnt_mod)) {
                        sprintf (line,"   %s [%ld]\n",initialnodes,pat
hcount);

                        write (fileno(fout),line,strlen(line));
                        }
                    }
                if (option==gen_path) {
                    p->dir[totalnode-1]='\n';
                    write (fileno(fp),p->dir,sizeof(DIRECTIONS)*totaln
ode);
                    }

                /* backtrack to search new path by (3) step */
                for (i=0;i<3;i++)
                    usedflag[p->node[level--]]=false;

                /* 'rewind' contact constraint checking level */
                if (total_contact!=0) {
                    while (contact_no!=0 && contact_set[contact_no-1].
x>level)
                        contact_no--;
                    }
                /*
                fprintf (stderr,"- (fin) backtrack at level %d new con
tact no. %d\n",level,contact_no);
                */
                }
            else {
                p->nnbno[level]=(-1);
                usedflag[nextnode]=true;
                }
            }

        /* change to new nnbno; if nnbno is above limit, change level
*/
        while (++p->nnbno[level]>=num_nearneighbor[p->node[level]] &&
level>startlevel) {
            usedflag[p->node[level]]=false;
            level--;

            /* 'rewind' contact constraint checking level */
            if (total_contact!=0) {
                while (contact_no!=0 && contact_set[contact_no-1].x>le
vel)
                    contact_no--;
                }
            /*
            fprintf (stderr,"- backtrack at level %d new contact no. %
d\n",level,contact_no);
```

```
            */
            }

        } while (level>startlevel || (level==startlevel && p->nnbno[le
vel]<num_nearneighbor[p->node[level]]));
}

void print_resource_usage (void) {
    struct rusage prusage;
    struct rusage crusage;
    double cputime[2][2];
    double total_cputime;
    int i;

    if (resource_usage) {
        getrusage (RUSAGE_SELF,&prusage);
        getrusage (RUSAGE_CHILDREN,&crusage);

        fprintf (stderr,"resource usage:\n");
        fprintf (stderr,"              parent       child\n");
        fprintf (stderr,"  user:     %7.3f      %7.3f  [user time use
d]\n",cputime[0][0]=(double)prusage.ru_utime.tv_sec+(double)((double)p
rusage.ru_utime.tv_usec/1000000.0),cputime[0][1]=(double)crusage.ru_ut
ime.tv_sec+(double)((double)crusage.ru_utime.tv_usec/1000000.0));
        fprintf (stderr,"  sys:      %7.3f      %7.3f  [system time u
sed]\n",cputime[1][0]=(double)prusage.ru_stime.tv_sec+(double)((double
)prusage.ru_stime.tv_usec/1000000.0),cputime[1][1]=(double)crusage.ru_
stime.tv_sec+(double)((double)crusage.ru_stime.tv_usec/1000000.0));
/*
        fprintf (stderr,"  maxrss:    %6d       %6d  [maximum residen
t set size]\n",prusage.ru_maxrss,crusage.ru_maxrss);
        fprintf (stderr,"  ixrss:     %6d       %6d  [integral shared
 text size]\n",prusage.ru_ixrss,crusage.ru_ixrss);
        fprintf (stderr,"  idrss:     %6d       %6d  [integral data r
esident set size]\n",prusage.ru_idrss,crusage.ru_idrss);
        fprintf (stderr,"  isrss:     %6d       %6d  [integral stack
resident set size]\n",prusage.ru_isrss,crusage.ru_isrss);
        fprintf (stderr,"  minflt:    %6d       %6d  [page faults not
 requiring physical I/O]\n",prusage.ru_minflt,crusage.ru_minflt);
        fprintf (stderr,"  majflt:    %6d       %6d  [page faults req
uiring physical I/O]\n",prusage.ru_majflt,crusage.ru_majflt);
        fprintf (stderr,"  nswap:     %6d       %6d  [swaps]\n",prusa
ge.ru_nswap,crusage.ru_nswap);
        fprintf (stderr,"  inblock:   %6d       %6d  [block input ope
rations]\n",prusage.ru_inblock,crusage.ru_inblock);
        fprintf (stderr,"  oublock:    %6d       %6d  [block output op
erations]\n",prusage.ru_oublock,crusage.ru_oublock);
        fprintf (stderr,"  msgsnd:     %6d       %6d  [messages sent]\
n",prusage.ru_msgsnd,crusage.ru_msgsnd);
        fprintf (stderr,"  msgrcv:     %6d       %6d  [messages receiv
ed]\n",prusage.ru_msgrcv,crusage.ru_msgrcv);
        fprintf (stderr,"  nsignals:   %6d       %6d  [signals receive
d]\n",prusage.ru_nsignals,crusage.ru_nsignals);
        fprintf (stderr,"  nvcsw:      %6d       %6d  [voluntary conte
xt switches]\n",prusage.ru_nvcsw,crusage.ru_nvcsw);
        fprintf (stderr,"  nivcsw:     %6d       %6d  [involuntary con
text switches]\n",prusage.ru_nivcsw,crusage.ru_nivcsw);
*/

        total_cputime=0;
        for (i=0;i<2;i++) {
            total_cputime+=cputime[i][0];
            total_cputime+=cputime[i][1];
            }
        fprintf (stderr,"\n* total CPU time: %.3f\n\n",total_cputime);

        }
}

void print_time (clock_t real,struct tms *tmsstart, struct tms *tmsend
)
{
    double cputime[2][2];
    double total_cputime;
    int i;

    fprintf (fout,"CPU time usage (seconds):\n");
    fprintf (fout,"  parent:\n");
    fprintf (fout,"    user: %.3f\n",cputime[0][0]=(tmsend->tms_utime-
tmsstart->tms_utime)/(double)clktck);
    fprintf (fout,"    sys:  %.3f\n",cputime[0][1]=(tmsend->tms_stime-
tmsstart->tms_stime)/(double)clktck);
    fprintf (fout,"  child:\n");
    fprintf (fout,"    user: %.3f\n",cputime[1][0]=(tmsend->tms_cutime
-tmsstart->tms_cutime)/(double)clktck);
    fprintf (fout,"    sys:  %.3f\n",cputime[1][1]=(tmsend->tms_cstime
-tmsstart->tms_cstime)/(double)clktck);
```

```
    total_cputime=0;
    for (i=0;i<2;i++) {
        total_cputime+=cputime[i][0];
        total_cputime+=cputime[i][1];
        }
    fprintf (fout,"\n* total cpu time: %.3f\n",total_cputime);
    fprintf (fout,"* wall clk time:  %.3f (note: on SUN, this value wi
ll be 0)\n\n",(double)real/(double)clktck);
}

void main (int argc,char *argv[])
{
    int node,j,l,i;
    char filename [64];
    pathlist *nsp;
    char tmp[128];

    int pid;
    char line[1024];
    int fd[2];
    long sum;
    long cnt;
    long waitrun,currun;
    int r,x,y;

    struct tms tmsstart,tmsend;
    clock_t start,end;

    /* to solve problem of logging output in shell script */
    fout=stdout;
    setbuf(fout,NULL);    /* set it to unbuffered mode */

    /* get system 'clock tick value' */
    if ((clktck=sysconf(_SC_CLK_TCK))==0)
        error("error: can't fetch 'clk tick value'");

    cmdline_parse(argc,argv);
    totalnode=zsize*ysize*xsize;

    sprintf (line,"date >> date.%d.%d.%d",xsize,ysize,zsize);
    system(line);

    sprintf (filename,"gwalk.%d.%d.%d.log",xsize,ysize,zsize);
    if ((gwalk_log=fopen (filename,"wb"))==NULL)
```

```
        error ("error: can't open log file (%s)\n",filename);

    /* allocate data */
    p=(path *)malloc(sizeof(path));
    p->node=(int *)malloc(sizeof(int)*totalnode);
    p->nnbno=(int *)malloc(sizeof(int)*totalnode);
    p->dir=(DIRECTIONS *)malloc(sizeof(DIRECTIONS)*totalnode);
    nearneighbor=(neighborlist *)malloc(sizeof(neighbor)*MAX_NEIGHBOR*
totalnode);
    usedflag=(boolean *)malloc(sizeof(boolean)*totalnode);
    num_nearneighbor=(int *)malloc(sizeof(int)*totalnode);
    tmp_used=(boolean *)malloc(sizeof(boolean)*totalnode);
    queue=(int *)malloc(sizeof(int)*totalnode);

    if (p==NULL || p->node==NULL || p->nnbno==NULL || p->dir==NULL ||
nearneighbor==NULL || usedflag==NULL || num_nearneighbor==NULL || tmp_
used==NULL || queue==NULL)
        error ("(001) Memory allocation error.\n");

    fprintf (fout,"xsize %d, ysize %d, zsize %d, totalnode %d\n",xsize
,ysize,zsize,totalnode);
    fprintf (gwalk_log,"xsize %d, ysize %d, zsize %d, totalnode %d\n",
xsize,ysize,zsize,totalnode);

    create_near_neighbor_table ();
    set_contact_constraint ();

    gen_sym_node_table ();
    gen_non_symmetry_node ();
    gen_non_symmetry_path ();

    fprintf (fout,"total 'initial' non-symmetrical path: %d\n",total_n
onsym_initial);
    fprintf (gwalk_log,"total 'initial' non-symmetrical path: %d\n",to
tal_nonsym_initial);

    /* create 'pipe' for inter-process communicatation */
    if (pipe(fd)<0)
        error ("error create pipe");

    cnt=0L;
    currun=0L;
    waitrun=total_nonsym_initial;
    sum=0L;
    do {

        while ((currun<max_process || option==gen_initial_path) && wai
trun>0L) {

            nsp=del_head_nspl();
            if (nsp==NULL)
                break;

            waitrun--;

            if (xsize==3 && ysize==3 && zsize==3 && (nsp->node[0]==1 |
| nsp->node[0]==13))
                continue;

            sprintf (initialnodes,"(%ld) enum ",cnt++);
            reset_usedflag ();

            /* prepare initial path */
            for (i=0;i<nsp->last;i++) {
                sprintf (tmp,"%d ",nsp->node[i]);
                strcat (initialnodes,tmp);
                if (option!=gen_initial_path) {
                    p->node[i]=nsp->node[i];
                    p->dir[i]=nsp->dir[i];
                    p->nnbno[i]=0;
                    usedflag[nsp->node[i]]=true;
                    }
                }

            /* we have to flush fout, log file here, otherwise
               child processes will do this and we will get a lot
               of duplicate data. (stderr is, by default,
               unbuffered)
            */
            fflush (fout);
            fflush (gwalk_log);

            if (option!=gen_initial_path) {
                if ((pid=fork())<0) {
                    fprintf (stderr,"fork error (Ctrl-C to stop all ch
ildren)\n");
                    getchar ();
                    exit (1);
                    }
                else if (pid==0) { /* child process */
                    /* get start time */
                    start=times(&tmsstart);

                    /* start walking from initial path */
                    pathcount=0L;
                    walk (p,nsp->last-1);

                    /* get stop time */
                    end=times(&tmsend);

                    if (verbose) {
                        /* record total walks, time usage (only 'paren
t') */
                        sprintf (line," * %s: [%ld] [user %.2g,sys %.2
g]\n",initialnodes,pathcount,(tmsend.tms_utime-tmsstart.tms_utime)/(do
uble)clktck,(tmsend.tms_stime-tmsstart.tms_stime)/(double)clktck);
                        write (fileno(fout),line,strlen(line));
                        write (fileno(gwalk_log),line,strlen(line));
                        }

                    if (write (fd[1],&pathcount,sizeof(long))!=sizeof(
long))
                        error ("write to pipe failed\n");

                    exit (0);     /* child terminate normally */
                    }
                else /* parent process */
                    currun++;
                }
            else {
                strcat (initialnodes,"\n");
                write(fileno(gwalk_log),initialnodes,strlen(initialnod
es));
                }

            }

        /* if total running process is more than limit, wait for some
of them to
           finished, then create more .. */
        while ((currun>=max_process || (waitrun==0L && currun>0L)) &&
option!=gen_initial_path) {
            r=wait(NULL);
            if (r<0)
                error ("wait error");
            else if (r>0) {
                if (read (fd[0],&pathcount,sizeof(long))!=sizeof(long)
)
                    error ("read pipe error");

                sum+=pathcount;
                currun--;
                }
            }
        } while (currun>0L || waitrun>0L);

    fprintf (fout,"* Enumeration ended\n");
    if (option!=gen_initial_path) {
        fprintf (fout,"\n* total geometrically-distinct walks: %ld\n",
sum);
        fprintf (gwalk_log,"\n* total geometrically-distinct walks: %l
d\n",sum);
        }

    sprintf (line,"date >> date.%d.%d.%d",xsize,ysize,zsize);
    system (line);

    print_resource_usage();
}
```

# APPENDIX B

# AN IMPLEMENTATION OF PARALLEL CONTACT CONSTRAINT EXACT ENUMERATION

In following are the implementation of parallel exact enumeration with contact constraint, ie. algorithm 4.1 and 5.5.

```
/*
   Cubic lattice enumeration with contact constraint

   by: Anek Vorapanya
   last modified: summer of 1993
   known bugs: none

   portability: all environments conform to the POSIX.1 standard
   language standard: ANSI C

   compiler: GNU compiler (gcc)
   compiler options: -ansi -O2

   usage: cwalk -h

   tested environment:
       - ULTRIX 4.3A (Rev. 146) (DEC 5900)
       - SUN OS 4.1.3 (Sun)
       - AIX 3.1.2 (IBM RS/6000)
       - IRIX 5.1 (SGI)
*/

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <stdarg.h>

#define MIN_ARGUMENT        4        /* min. argument required */
#define MAX_PROCESS            1        /* default max. processes */
#define DEF_CNT_LV_CHK        2        /* default connect level check,
 min.=2 */

#define MAX_DIR                6
#define MAX_NEIGHBOR        6
#define INVALID_NODE        (-1)
#define INVALID_NNBNO        (-1)
#define OPPOSITE_DIR_DIFF    (abs(PLUS_X-MINUS_X))

#define DISPCNT_MOD            1000L
#define MIN_DISPCNT_MOD        100L
#define MAX_PIPL            50    /* max. % of initial path length */

/* maximum dimension */
#define MAX_X                10
#define MAX_Y                10
#define MAX_Z                10

#define plus_dir(dir)        (((dir%2)==0)?dir:(dir-1))
#define minus_dir(dir)        (((dir%2)==0)?(dir+1):dir)
#define inttodir(i)            ((DIRECTIONS)(i+'0'))

typedef enum { false, true } boolean;

typedef enum {
    PLUS_X='0',
    MINUS_X=minus_dir(PLUS_X),
    PLUS_Y='2',
    MINUS_Y=minus_dir(PLUS_Y),
    PLUS_Z='4',
    MINUS_Z=minus_dir(PLUS_Z),
    INVALID_DIR='#'
    } DIRECTIONS;

typedef struct {
    int x;
    int y;
    int z;
    } NODE;
```

```
typedef struct {
    int ca;    /* first node of contact */
    int cb;    /* second node of contact */
    } contact;

typedef struct {
    NODE *node;            /* currently visited node no. at this level
 */
    DIRECTIONS *dir;    /* direction of walk */
    int *nnbno;            /* currently used child no of this node */
    } path;

typedef enum { _0Dpath, _1Dpath, _2Dpath, _3Dpath } pathtype;

typedef enum { invalid_planetype, xyplane, yzplane, xzplane } planetyp
e;

typedef struct pathinfo_s {
    pathtype ptype;
    planetype pltype;
    int last;
    NODE *node;
    DIRECTIONS *dir;
    int *nnbno;
    struct pathinfo_s *next;
    } pathinfo;

long sys_max_child;
long max_process;
int min_level;
int dispcnt_mod;
enum { gen_initial_path, count_path, gen_path } option;
boolean verbose=false;
boolean resource_usage=false;
int totalnode;
long pathcount;
long total_nonsym_initial=0L;
int totalnonsymnode;
pathinfo *stpl;
pathinfo *nspl;
pathinfo *last_stpl;
pathinfo *last_nspl;
pathinfo head_stpl;
pathinfo head_nspl;
FILE *cwalk_log;
FILE *fp;
char filename[256];
char initialnodes[1024];
long clktck;
contact *contact_set;
int total_contact=0;
FILE *fout;

int nsplcnt=0;
int tmpnsplcnt=0;

void error (char *fmt, ...)
{
    va_list ap;
    char buf[512];

    va_start(ap,fmt);
    vsprintf (buf,fmt,ap);
    va_end(ap);
    write (fileno(stderr),buf,strlen(buf));
    exit (1);
}

void usage (void)
{
    char tmp[128];

    fprintf (stderr,"Enumerate geometrically-distinct walk with contac
t constraints\n");
    fprintf (stderr,"usage: cwalk <totalnode> [options]\n");
    fprintf (stderr,"options:\n");
    fprintf (stderr,"  -i : to generate initial paths only (save in lo
g file)\n");
    fprintf (stderr,"  -c : to count all possible paths\n");
    fprintf (stderr,"  -w : to save generated paths in file (save in p
ath.<totalnode>)\n");

    if (sys_max_child==0)
        strcpy (tmp,"unknown");
    else
        sprintf (tmp,"%ld",sys_max_child-1L);
    fprintf (stderr,"  -pX : specify no. of processes to run simultane
ously (your system max %s)\n",tmp);
    fprintf (stderr,"  -lX : specify percentage of initial path length
 (recommend 15-25,max %d)\n",MAX_PIPL);
    fprintf (stderr,"  -tX <contact set>: specify contact set where X
is number of contacts\n");
    fprintf (stderr,"      and <contact set> is a contact set in the f
orm 'a-b c-d e-f ...' \n");
    fprintf (stderr,"  -dX : specify counter display modulo value (min
 %d)\n",MIN_DISPCNT_MOD);
    fprintf (stderr,"  -v : show the enumeration counting\n");
    fprintf (stderr,"  -r : show resource usage\n");
    fprintf (stderr,"  -h  : print this help screen\n");
    exit (0);
}

void cmdline_parse (int argc,char *argv[])
{
    char *s;
    char tmp[1024];
    char c;
    int i,j,p;
    int contact_no=(-1);
    int tmptotalnode;

    option=count_path;
    min_level=0;
    dispcnt_mod=DISPCNT_MOD;

    if ((sys_max_child=sysconf(_SC_CHILD_MAX))==0) {
        fprintf (stderr,"* can't determine 'max. processes for user'\n
");
        fprintf (stderr,"  use %d as a default value\n",MAX_PROCESS);
        sys_max_child=MAX_PROCESS;
        }

    /* set default max. process */
    max_process=MAX_PROCESS;

    if (argc<MIN_ARGUMENT)
        usage();

    for (i=1;i<argc;i++) {
        if (i==1) {
            /* get total node */
            if ((totalnode=atoi(argv[i]))==0)
                usage();
            }
        else {
            if (contact_no==(-1)) {
                s=argv[i];
                if (s[0]!='-')
                    usage();
                s++;
                c=tolower(s[0]);
                }
            else
                c='t'; /* continue scanning contact set */

            switch (c) {
                case 'c': /* count path only */
                    option=count_path;
                    break;
                case 'i': /* generate only initial path and save in lo
g file */
                    option=gen_initial_path;
                    break;
                case 'w': /* generate path file */
                    option=gen_path;
                    /* prepare output file */
                    strcpy (filename,"path");
                    strcat(filename,".");
                    strcat(filename,argv[1]);
                    if ((fp=fopen(filename,"wb"))==NULL)
                        error ("error: can't open output file (%s)\n",
filename);
                    break;
                case 'l': /* minimum start level */
                    if ((p=atoi(++s))==0 || p>MAX_PIPL)
                        usage();
                    min_level=totalnode*p/100;
                    fprintf (stderr,"* use min. initial level = %d\n",
min_level);
                    break;
                case 'p': /* maximum processes to run simultaneous */
                    /* we have to subtract 1 from sys_max_child based
on assumption that we will have one and only one running (parent proce
ss).
```

```
                    */
                    if ((max_process=atol(++s))==0 || max_process>(sys
_max_child-1L))
                        usage();
                    break;
                case 'd': /* counter display modulo value */
                    if ((dispcnt_mod=atoi(++s))==0 || dispcnt_mod<MIN_
DISPCNT_MOD)
                        usage();
                    break;
                case 'v': /* verbose */
                    if (s[1]=='-')
                        verbose=true;
                    break;
                case 't': /* contact constraint set */
                    if (contact_no!=-1) {
                        if ((s=strchr(argv[i],'-'))!=NULL) {
                            s[0]='\0';
                            if (((contact_set[contact_no].ca=atoi(argv
[i]))==0 && argv[i][0]!='0') || contact_set[contact_no].ca>=totalnode)

                                error("error: invalid contact set no.
(%s)\n",argv[i]);
                            s++;
                            if (s[0]=='\0')
                                error("error: incomplete contact set (
contact no. %d)\n",contact_no);
                            if (((contact_set[contact_no].cb=atoi(s))=
=0 && s[0]!='0') || contact_set[contact_no].cb>=totalnode)
                                error("error: invalid contact set no.
(%s)\n",s);
                        }
                        else
                            error("error: contact set requires 'x-y' f
ormat where x and y are nodes no. to form the contact.\n");

                        if (++contact_no==total_contact)
                            contact_no=(-1);
                    }
                    else {
                        if ((total_contact=atoi(++s))!=0) {
                            /* allocate contact set data */
                            contact_set=(contact *)malloc(total_contac
t*sizeof(contact));
```

```
                            contact_no=0;
                            }
                        else
                            usage();
                        }
                    break;
                case 'r':
                    resource_usage=true;
                    break;
                default:
                    fprintf (stderr,"* error: unknown options (%c)\n",
s[0]);
                case 'h':
                    usage();
                    break;
                }
            }
        }

    /* check if we had all contact as specify with -tX option or not *
/
    if (contact_no!=-1)
        error("error: number of contacts mismatched\n");
}

boolean usednode (NODE *nl,int last,NODE *n)
{
    int i;

    for (i=0;i<=last;i++) {
    /*
        printf ("node list [%d] (%d,%d,%d), node (%d,%d,%d)\n",i,nl[i]
.x,nl[i].y,nl[i].z,n->x,n->y,n->z);
    */
        if (nl[i].x==n->x && nl[i].y==n->y && nl[i].z==n->z) {
            /*
            printf ("* used\n");
            */
            return true;
            }
        }

    /*
    printf ("* UNused\n");
    */
    return false;
}

boolean is_neighbor (NODE *n,NODE *m)
{
    if ((abs(n->x-m->x)==1 && n->y==m->y && n->z==m->z) ||
        (n->x==m->x && abs(n->y-m->y)==1 && n->z==m->z) ||
        (n->x==m->x && n->y==m->y && abs(n->z-m->z)==1))
        return true;
    else
        return false;
}

void set_contact_constraint (void)
{
    int i,j;
    contact tmp;

    /* switch position of x and y such that x is lower than y */
    for (i=0;i<total_contact;i++) {
        if (contact_set[i].ca>contact_set[i].cb) {
            /* swap */
            j=contact_set[i].ca;
            contact_set[i].ca=contact_set[i].cb;
            contact_set[i].cb=j;
            }
        /*
        printf ("switch contact %d (%d-%d)\n",i,contact_set[i].ca,cont
act_set[i].cb);
        */
        }

    /* we will sort it using bubble sort (straight forward) */
    for (i=0;i<total_contact;i++) {
        for (j=total_contact-1;j>i;j--) {
            if (contact_set[j].ca<contact_set[j-1].ca) {
                /* swap */
                tmp=contact_set[j];
                contact_set[j]=contact_set[j-1];
                contact_set[j-1]=tmp;
                }
            }
/*
```

```
        fprintf (stderr,"sort contact %d (%d-%d)\n",i,contact_set[i].c
a,contact_set[i].cb);
*/
        }
}

boolean getnextnode (path *p,int level)
{
    int i,validnnbno;
    NODE node;

    validnnbno=0;
    for (i=0;i<MAX_NEIGHBOR;i++) {
        node=p->node[level];
        switch (inttodir(i)) {
            case PLUS_X :
                node.x+=1;
                break;
            case MINUS_X :
                node.x-=1;
                break;
            case PLUS_Y :
                node.y+=1;
                break;
            case MINUS_Y :
                node.y-=1;
                break;
            case PLUS_Z :
                node.z+=1;
                break;
            case MINUS_Z :
                node.z-=1;
                break;
            }
        if (!usednode(p->node,level,&node)) {
            if (validnnbno==p->nnbno[level]) {
                p->dir[level]=inttodir(i);
                p->node[++level]=node;
                return true;
                }
            else
                validnnbno++;
            }
        }
    return false;
}

int get_available_neighbor (NODE *cp,int level,NODE *ann,DIRECTIONS *d
ir)
{
    int i,cnt;
    NODE na,nb;

    cnt=0;
    na=cp[level];
    for (i=0;i<MAX_NEIGHBOR;i++) {
        nb=cp[level];
        switch (inttodir(i)) {
            case PLUS_X :
                nb.x+=1;
                break;
            case MINUS_X :
                nb.x-=1;
                break;
            case PLUS_Y :
                nb.y+=1;
                break;
            case MINUS_Y :
                nb.y-=1;
                break;
            case PLUS_Z :
                nb.z+=1;
                break;
        case MINUS_Z :
                nb.z-=1;
                break;
            }
        if (usednode(cp,level,&nb)==false) {
            ann[cnt]=nb;
            dir[cnt]=inttodir(i);
            cnt++;
            }
        }

    return cnt;
}
```

```
void create_newpath (pathinfo *orgpath,pathinfo **newpath,NODE *newlas
tnode,DIRECTIONS dirlastnode)
{
    int i;

    *newpath=(pathinfo *)malloc(sizeof(pathinfo));
    (*newpath)->node=(NODE *)malloc(sizeof(NODE)*totalnode);
    (*newpath)->dir=(DIRECTIONS *)malloc(sizeof(DIRECTIONS)*totalnode)
;
    /* for 'nnbno', we have to use 'calloc' to clear to zero */
    (*newpath)->nnbno=(int *)calloc(totalnode,sizeof(int));

    if (orgpath!=NULL) {
        (*newpath)->last=orgpath->last;
        (*newpath)->ptype=orgpath->ptype;
        (*newpath)->pltype=orgpath->pltype;
        memcpy((*newpath)->node,orgpath->node,orgpath->last*sizeof(NOD
E));
        memcpy((*newpath)->dir,orgpath->dir,(orgpath->last-1)*sizeof(D
IRECTIONS));
        memcpy((*newpath)->nnbno,orgpath->nnbno,(orgpath->last-1)*size
of(int));
        (*newpath)->dir[(*newpath)->last-1]=dirlastnode;
        }
    else {
        (*newpath)->last=0;
        (*newpath)->ptype=_ODpath;
        }

    (*newpath)->node[(*newpath)->last]=(*newlastnode);
    (*newpath)->last++;
    (*newpath)->next=NULL;
}

void add_to_stpl (pathinfo *p)
{
    last_stpl->next=p;
    last_stpl=p;
    nsplcnt++;
}

void add_to_nspl (pathinfo *p)
{
    last_nspl->next=p;
    last_nspl=p;
}

void del_head_stpl (void)
{
    pathinfo *p;

    if (stpl->next!=NULL) {
        p=stpl->next;
        stpl->next=stpl->next->next;
        /* throw it away */
        free (p);
        }
}

pathinfo *del_head_nspl (void)
{
    pathinfo *p;

    if (nspl->next!=NULL) {
        p=nspl->next;
        nspl->next=nspl->next->next;
        return p;
        }
    else
        return NULL;
}

planetype det_planetype (DIRECTIONS cdir,DIRECTIONS ndir)
{
    switch (cdir) {
        case PLUS_X:
        case MINUS_X:
            if (ndir==PLUS_Y || ndir==MINUS_Y)
                return xyplane;
            else
                return xzplane;
            break;
        case PLUS_Y:
        case MINUS_Y:
            if (ndir==PLUS_X || ndir==MINUS_X)
                return xyplane;
            else
                return yzplane;
                break;
            case PLUS_Z:
            case MINUS_Z:
                if (ndir==PLUS_Y || ndir==MINUS_Y)
                    return yzplane;
                else
                    return xzplane;
                break;
            }
}

int print_pathinfo (pathinfo *pi,char *name)
{
    pathinfo *tmp;
    int i,j;

    tmp=pi->next;
    j=0;
    while (tmp!=NULL && j<tmpnsplcnt) {
        tmp=tmp->next;
        j++;
        }
    printf ("%s:\n",name);
    while (tmp!=NULL) {
        printf ("   (%d) ptype %d, pltype %d, level %d\n",j++,tmp->pty
pe,tmp->pltype,tmp->last-1);
        for (i=0;i<tmp->last;i++)
                printf ("       node (%d,%d,%d)\n",tmp->node[i].x,tmp-
>node[i].y,tmp->node[i].z);
        tmp=tmp->next;
        }
    return j;
}

boolean satisfy_cc (int level,NODE *nodelist,NODE *nextnode) {
    static NODE ann[MAX_NEIGHBOR];
    static DIRECTIONS dir[MAX_NEIGHBOR];
    int j,i,mrs;
    int cnt;
    boolean satisfy;

    satisfy=true;
    for (i=0;i<total_contact;i++) {
        /* ''CHECK'' contact-constraint satisfaction */
        if (level+1==contact_set[i].cb) {
            if (is_neighbor(&nodelist[contact_set[i].ca],nextnode)==fa
lse) {
                /* this path (with 'nextnode') does not satisfy at lea
st one contact-constraint.
                */
                return false;
                }
            }

        /* ''PREDICT'' contact-constraint satisfaction */
        if (level+1>contact_set[i].ca && level+1<contact_set[i].cb) {
            satisfy=false;
            cnt=get_available_neighbor (nodelist,contact_set[i].ca,ann
,dir);
            for (j=0;j<cnt;j++) {
                mrs=abs(ann[j].x-nextnode->x)+
                    abs(ann[j].y-nextnode->y)+
                    abs(ann[j].z-nextnode->z);
                if (mrs<=contact_set[i].cb-(level+1)) {
                    /* this contact constraint has been satisfied by o
ne of neighbor of 'lower' contact node
                    */
                    satisfy=true;
                    break;
                    }
                }
            }
        if (satisfy==false)
            /* none of neighbor of 'lower' contact node satisfy this c
ontact constraint, ie. fails in all contact-constraint test too.
            */
            return false;
        }

    return satisfy;
}

void gen_non_symmetry_path (void)
{
    pathinfo *np;
    pathinfo *cp;
    NODE *ann;            /* available near neighbor */
    DIRECTIONS *dir;    /* 'direction' of 'ann' */
    int idx,total_nn;
    boolean _1Dto_2D,_2Dto_3D;

    /* initialize STPL (symmetry test path list) and
       NSPL (non-symmetry initial path list)
    */
    head_stpl.next=NULL;
    head_nspl.next=NULL;
    stpl=(pathinfo *)&head_stpl;
    nspl=(pathinfo *)&head_nspl;
    last_stpl=(pathinfo *)&head_stpl;
    last_nspl=(pathinfo *)&head_nspl;

    ann=(NODE *)malloc(sizeof(NODE)*MAX_NEIGHBOR);
    dir=(DIRECTIONS *)malloc(sizeof(DIRECTIONS)*MAX_NEIGHBOR);
    if (ann==NULL || dir==NULL)
        error ("(002) Memory allocation error.\n");

    /* add '(0,0,0)' to initial stpl, this node is the only one to bec
ome a starting point for contact-constraint enumeration.
    */
    ann[0].x=0;
    ann[0].y=0;
    ann[0].z=0;

    create_newpath (NULL,&np,&ann[0],INVALID_DIR);
    add_to_stpl (np);

    _1Dto_2D=false;
    _2Dto_3D=false;

    while ((cp=stpl->next)!=NULL) {
        /*
        print_pathinfo (stpl,"STPL");
        if (nsplcnt!=tmpnsplcnt)
            tmpnsplcnt=print_pathinfo (nspl,"NSPL");
        */

        total_nn=get_available_neighbor(cp->node,cp->last-1,ann,dir);
        for (idx=0;idx<total_nn;) {
            /*
            printf ("cnt %d total %d\n",cnt,total_nn);
            */
            switch (cp->ptype)     {
                case _0Dpath:
                    /* we just start walking so we choose to walk
                       to only node that are non-symmetry. There
                       is only of them in case of cubic lattice.
                       we don't need to check contact-constraint
                       here, because it should satisfy, otherwise
                       contact-constraint set is wrong
                    */
                    create_newpath (cp,&np,&ann[idx],dir[idx]);
                    np->ptype=_1Dpath;
                    add_to_stpl (np);
                    /* we need no more neighbor for this path */
                    idx=total_nn;
                    break;
                case _1Dpath :
                    /* find 'stright path' , this path will not
                       symmetry to other path
                    */
                    if (dir[idx]==cp->dir[0]) {
                        if (satisfy_cc(cp->last-1,cp->node,&ann[idx]))
 {
                            create_newpath (cp,&np,&ann[idx],dir[idx])
;
                            add_to_stpl (np);
                            }
                        }
                    else if (_1Dto_2D==false) {
                        /* this next non-symmetry neighbor will
                           change this path to 'plane path' type
                           from now on. There is only such neighbor
                           in cubic lattice.
                         */
                        if (satisfy_cc(cp->last-1,cp->node,&ann[idx]))
 {
                            create_newpath (cp,&np,&ann[idx],dir[idx])
;
                            np->ptype=_2Dpath;
                            np->pltype=det_planetype (cp->dir[0],dir[i
dx]);
                            add_to_stpl (np);
                            }
                        /* set flag to indicate that we had
                           neighbor that change 1D path to 2D path
                            (we need only of this
                              in case of cubic lattice.
                           */
                           _1Dto_2D=true;
                           }
                       idx++;    /* fetch next neighbor */
                       break;
                   case _2Dpath :
                       /* find 'ann' that is in plane add it to
                          'stpl' (it will not symmetry)
                        */
                       if ((cp->pltype==xyplane && dir[idx]!=PLUS_Z && di
r[idx]!=MINUS_Z) ||
                               (cp->pltype==yzplane && dir[idx]!=PLUS_X &
& dir[idx]!=MINUS_X) ||
                               (cp->pltype==xzplane && dir[idx]!=PLUS_Y &
& dir[idx]!=MINUS_Y)) {
                           if (satisfy_cc(cp->last-1,cp->node,&ann[idx]))
 {
                               create_newpath (cp,&np,&ann[idx],dir[idx])
;

                               add_to_stpl (np);
                               }
                           }
                       else if (_2Dto_3D==false) {
                           /* Now path will not symmetry and there is
                              only such neighbor in cubic lattice.
                              Add it to 'NSPL'
                           */
                           if (satisfy_cc(cp->last-1,cp->node,&ann[idx]))
 {
                               create_newpath (cp,&np,&ann[idx],dir[idx])
;


                               /* now we have non-symmetrical initial
                                  path list if user didn't specify 'min.
                                  level of initial path', we will use
                                  this as NSPL, otherwise we continue
                                  to expand it as 3D segment.
                               */
                               if (np->last>=min_level) {
                                   add_to_nspl (np);
                                   total_nonsym_initial++;
                                   }
                               else {
                                np->ptype=_3Dpath;
                                add_to_stpl (np);
                                }
                            }
                        /* set flag to indicate that we had
                           neighbor that change 2D path to 3D path
                           (we need only of this
                           in case of cubic lattice.
                         */
                        _2Dto_3D=true;
                        }
                    idx++;    /* fetch next neighbor */
                    break;
                case _3Dpath :
                    /* for 3D segment, all neighbor will form
                       up a unique pattern in 3D space
                    */
                    if (satisfy_cc(cp->last-1,cp->node,&ann[idx])) {
                        create_newpath (cp,&np,&ann[idx],dir[idx]);
                        if (np->last<min_level)
                            add_to_stpl (np);
                        else {
                            add_to_nspl (np);
                            total_nonsym_initial++;
                            }
                        }
                    idx++;    /* fetch next neighbor */
                    break;
                }
            }

        del_head_stpl();
        /* reinitialize path transformation flag */
        _1Dto_2D=false;
        _2Dto_3D=false;
        }

    free (ann);
    free (dir);
}

void walk (pathinfo *pi)
{
    char line[256];
    int level;
    path p;

    p.node=pi->node;
    p.dir=pi->dir;
    p.nnbno=pi->nnbno;
    level=pi->last-1;

    /* start walking (exhaustive search) */
    do {
        if (getnextnode(&p,level)==true && satisfy_cc(level,p.node,&p.
node[level+1])) {
            /* update 'level' ('node' and 'dir' filled in 'getnextnode
' fn) */
            ++level;

            if (level==totalnode-1) {
                ++pathcount;
                if (verbose) {
                    if (!(pathcount%dispcnt_mod)) {
                        sprintf (line,"   %s [%ld]\n",initialnodes,pat
hcount);

                        write (fileno(fout),line,strlen(line));
                        }
                    }
                if (option==gen_path) {
                    p.dir[totalnode-1]='\n';
                    write (fileno(fp),p.dir,sizeof(DIRECTIONS)*totalno
de);

                    }

                /* backtrack to search new path by (1) step,
                   (contact constraint, we can backtrack at most
                   1 step
                */
                level--;
                }
            else
                p.nnbno[level]=(-1);
            }

        /* change to new nnbno; if nnbno is above limit, change level
*/
        while (++p.nnbno[level]>=MAX_NEIGHBOR && level>pi->last-1)
```

```
            level--;

        } while (level>pi->last-1 || (level==pi->last-1 && p.nnbno[lev
el]<MAX_NEIGHBOR));
}

void print_resource_usage (void) {
    struct rusage prusage;
    struct rusage crusage;
    double cputime[2][2];
    double total_cputime;
    int i;

    if (resource_usage) {
        getrusage (RUSAGE_SELF,&prusage);
        getrusage (RUSAGE_CHILDREN,&crusage);

        fprintf (stderr,"resource usage:\n");
        fprintf (stderr,"             parent       child\n");
        fprintf (stderr,"  user:      %7.3f      %7.3f  [user time use
d]\n",cputime[0][0]=(double)prusage.ru_utime.tv_sec+(double)((double)p
rusage.ru_utime.tv_usec/1000000.0),cputime[0][1]=(double)crusage.ru_ut
ime.tv_sec+(double)((double)crusage.ru_utime.tv_usec/1000000.0));
        fprintf (stderr,"  sys:       %7.3f      %7.3f  [system time u
sed]\n",cputime[1][0]=(double)prusage.ru_stime.tv_sec+(double)((double
)prusage.ru_stime.tv_usec/1000000.0),cputime[1][1]=(double)crusage.ru_
stime.tv_sec+(double)((double)crusage.ru_stime.tv_usec/1000000.0));
/*
        fprintf (stderr,"  maxrss:     %6d       %6d  [maximum residen
t set size]\n",prusage.ru_maxrss,crusage.ru_maxrss);
        fprintf (stderr,"  ixrss:      %6d       %6d  [integral shared
 text size]\n",prusage.ru_ixrss,crusage.ru_ixrss);
        fprintf (stderr,"  idrss:      %6d       %6d  [integral data r
esident set size]\n",prusage.ru_idrss,crusage.ru_idrss);
        fprintf (stderr,"  isrss:      %6d       %6d  [integral stack
resident set size]\n",prusage.ru_isrss,crusage.ru_isrss);
        fprintf (stderr,"  minflt:     %6d       %6d  [page faults not
 requiring physical I/O]\n",prusage.ru_minflt,crusage.ru_minflt);
        fprintf (stderr,"  majflt:     %6d       %6d  [page faults req
uiring physical I/O]\n",prusage.ru_majflt,crusage.ru_majflt);
        fprintf (stderr,"  nswap:      %6d       %6d  [swaps]\n",prusa
ge.ru_nswap,crusage.ru_nswap);
        fprintf (stderr,"  inblock:    %6d       %6d  [block input ope
rations]\n",prusage.ru_inblock,crusage.ru_inblock);
```

```
        fprintf (stderr,"  oublock:    %6d       %6d  [block output op
erations]\n",prusage.ru_oublock,crusage.ru_oublock);
        fprintf (stderr,"  msgsnd:     %6d       %6d  [messages sent]\
n",prusage.ru_msgsnd,crusage.ru_msgsnd);
        fprintf (stderr,"  msgrcv:     %6d       %6d  [messages receiv
ed]\n",prusage.ru_msgrcv,crusage.ru_msgrcv);
        fprintf (stderr,"  nsignals:   %6d       %6d  [signals receive
d]\n",prusage.ru_nsignals,crusage.ru_nsignals);
        fprintf (stderr,"  nvcsw:      %6d       %6d  [voluntary conte
xt switches]\n",prusage.ru_nvcsw,crusage.ru_nvcsw);
        fprintf (stderr,"  nivcsw:     %6d       %6d  [involuntary con
text switches]\n",prusage.ru_nivcsw,crusage.ru_nivcsw);
*/

        total_cputime=0;
        for (i=0;i<2;i++) {
            total_cputime+=cputime[i][0];
            total_cputime+=cputime[i][1];
            }
        fprintf (stderr,"\n* total CPU time: %.3f\n\n",total_cputime);

        }
}

void print_time (clock_t real,struct tms *tmsstart, struct tms *tmsend
)
{
    double cputime[2][2];
    double total_cputime;
    int i;

    fprintf (fout,"CPU time usage (seconds):\n");
    fprintf (fout,"  parent:\n");
    fprintf (fout,"    user: %.3f\n",cputime[0][0]=(tmsend->tms_utime-
tmsstart->tms_utime)/(double)clktck);
    fprintf (fout,"    sys:  %.3f\n",cputime[0][1]=(tmsend->tms_stime-
tmsstart->tms_stime)/(double)clktck);
    fprintf (fout,"  child:\n");
    fprintf (fout,"    user: %.3f\n",cputime[1][0]=(tmsend->tms_cutime
-tmsstart->tms_cutime)/(double)clktck);
    fprintf (fout,"    sys:  %.3f\n",cputime[1][1]=(tmsend->tms_cstime
-tmsstart->tms_cstime)/(double)clktck);

    total_cputime=0;
    for (i=0;i<2;i++) {
        total_cputime+=cputime[i][0];
        total_cputime+=cputime[i][1];
        }
    fprintf (fout,"\n* total cpu time: %.3f\n",total_cputime);
    fprintf (fout,"* wall clk time:  %.3f (note: on SUN, this value wi
ll be 0)\n\n",(double)real/(double)clktck);
}

void main (int argc,char *argv[])
{
    int j,l,i,r;
    char filename [64];
    pathinfo *nsp;
    char tmp[128];
    char line[1024];
    int pid;
    int fd[2];
    long sum,cnt;
    long waitrun,currun;

    struct tms tmsstart,tmsend;
    clock_t start,end;

    /* to solve problem of logging output in shell script */
    fout=stdout;
    /*
    setbuf(fout,NULL);
    */

    /* get system 'clock tick value' */
    if ((clktck=sysconf(_SC_CLK_TCK))==0)
        error("error: can't fetch 'clk tick value'");

    /* command line parsing */
    cmdline_parse(argc,argv);

    sprintf (filename,"cwalk.%d.log",totalnode);
    if ((cwalk_log=fopen (filename,"wb"))==NULL)
        error ("error: can't open log file (%s)\n",filename);

    /* set up contact constraint list */
    set_contact_constraint ();
    /* generate all non-redundant initial conformations */
    gen_non_symmetry_path ();

    fprintf (fout,"total 'initial' non-symmetrical path: %d\n",total_n
onsym_initial);
    fprintf (cwalk_log,"total 'initial' non-symmetrical path: %d\n",to
tal_nonsym_initial);

    /* create 'pipe' for inter-process communicatation */
    if (pipe(fd)<0)
        error ("error create pipe");

    cnt=0L;
    currun=0L;
    waitrun=total_nonsym_initial;
    sum=0L;

    /* create slaves to do enumeration */
    do {

        while ((currun<max_process || option==gen_initial_path) && wai
trun>0L) {

            nsp=del_head_nspl();
            if (nsp==NULL)
                break;

            waitrun--;

            sprintf (initialnodes,"(%ld) enum ",cnt++);

            for (i=0;i<nsp->last;i++) {
                sprintf (tmp,"(%d,%d,%d) ",nsp->node[i].x,nsp->node[i]
.y,nsp->node[i].z);
                strcat (initialnodes,tmp);
                }

            /* we have to flush fout and log file here, otherwise
               child processes will do this and we will get a lot of
               duplicate data. (stderr is, by default, unbuffered)
            */
            fflush (fout);
            fflush (cwalk_log);
            if (option!=gen_initial_path) {
                if ((pid=fork())<0) {
                    fprintf (stderr,"fork error (Ctrl-C to stop all ch
ildren)\n");
                    getchar ();
                    exit (1);
                    }
                else if (pid==0) { /* child process */
                    /* get start time */
                    start=times(&tmsstart);

                    /* start walking from initial path */
                    pathcount=0L;
                    walk (nsp);

                    /* get stop time */
                    end=times(&tmsend);

                    if (verbose) {
                        /* record total walks, time usage (only 'paren
t') */
                        sprintf (line," * %s: [%ld] [user %.3f,sys %.3
f]\n",initialnodes,pathcount,(tmsend.tms_utime-tmsstart.tms_utime)/(do
uble)clktck,(tmsend.tms_stime-tmsstart.tms_stime)/(double)clktck);
                        write (fileno(fout),line,strlen(line));
                        write (fileno(cwalk_log),line,strlen(line));
                        }

                    if (write (fd[1],&pathcount,sizeof(long))!=sizeof(
long))
                        error ("write to pipe failed\n");

                    exit (0);     /* child terminate normally */
                    }
                else /* parent process */
                    currun++;
                }
            else {
                strcat (initialnodes,"\n");
                write(fileno(cwalk_log),initialnodes,strlen(initialnod
es));
                }
            }
```

```
        /* if total running process is more than limit, wait for some
of them
   to finished, then create more ..
*/

        while ((currun>=max_process || (waitrun==0L && currun>0L)) &&
option!=gen_initial_path) {
            r=wait(NULL);
            if (r<0)
                error ("wait error");
            else if (r>0) {
                if (read (fd[0],&pathcount,sizeof(long))!=sizeof(long)
)
                    error ("read pipe error");

                sum+=pathcount;
                currun--;
                }
            }
        } while (currun>0L || waitrun>0L);

    fprintf (fout,"* Enumeration ended\n");
    if (option!=gen_initial_path) {
        fprintf (fout,"\n* total geometrically-distinct walks: %ld\n",
sum);
        fprintf (cwalk_log,"\n* total geometrically-distinct walks: %l
d\n",sum);
        }

    /* display resource usage */
    print_resource_usage();
}
```

## REFERENCES

1. Anfinsen,C.B., "Principles that govern the folding of protein chains," *Science*, vol. 181, pp. 223–230, 1973.

2. Barat,K., Karmakar,S.N., and Chakrabarti,B.K., "Self-avoiding walk connectivity constant and theta-point on percolating lattices," *Journal of Physics A - Mathematical and General*, vol. 24, pp. 851–860, February 1991.

3. Barber,M.N. and Ninham,B.W., *Random and Restricted Walks: Theory and Applications*, Gordon and Breach, New York, 1970.

4. Berretti,A. and Sokal,A.D., "Vectorized program for Monte-Carlo simulation of self-avoiding walks," *Computer Physics Communications*, vol. 58, pp. 1–16, February 1990.

5. Bishop,M. and Clarke,J.H.R., "Investigation of the end-to-end distance distribution function for random and self-avoiding walks in 2 and 3 dimensions," *Journal of Chemical Physics*, vol. 94, pp. 3936–3942, March 1991.

6. Bradley,R.M., Debierre,J.M., and Strenski,P.N., "A novel growing self-avoiding walk in 3-dimensions," *Journal of Physics A - Mathematical and General*, vol. 25, pp. 1541–1548, May 1992.

7. Bradley,R.M., Strenski,P.N., and Debierre,J.M., "A growing self-avoiding walk in 3 dimensions and its relation to percolation," *Physical Review A*, vol. 45, pp. 8513–8524, June 1992.

8. Caracciolo,S., Pelissetto,A., and Sokal,A.D., "Nonlocal Monte-Carlo algorithm for self-avoiding walks with fixed endpoints," *Journal of Statistical Physics*, vol. 60, pp. 1–53, July 1990.

9. Chan,H.S. and Dill,K.A., "Compact polymers," *Macromolecules*, vol. 22, pp. 4559–4573, December 1989.

10. Chan,H.S. and Dill,K.A., "Intrachain loops in polymers: Effects of excluded volume," *Journal of Chemical Physics*, vol. 90, pp. 492–509, January 1989.

11. Chan,H.S. and Dill,K.A., "The effects of internal constraints on the configurations of chain molecules," *Journal of Chemical Physics*, vol. 92, pp. 3118–3135, March 1990.

12. Chan,H.S. and Dill,K.A., "The Protein Folding Problem," *Physics Today*, pp. 24–32, February 1993.

13. Conway,A.R., Enting,I.G., and Guttmann,A.J., "Algebraic techniques for enumerating self-avoiding walks on the square lattice," *Journal of Physics A - Mathematical and General*, vol. 26, pp. 1519–1534, April 1993.

14. Conway,A.R. and Guttmann,A.J., "Enumeration of self-avoiding trails on a square lattice using a transfer matrix technique," *Journal of Physics A - Mathematical and General*, vol. 26, pp. 1535–1552, April 1993.

15. Dayantis,J. and Palierne,J.F., "Monte-Carlo precise determination of the end-to-end distribution function of self-avoiding walks on the simple-cubic lattice," *Journal of Chemical Physics*, vol. 95, pp. 6088–6099, October 1991.

16. Eizenberg,N. and Klafter,J., "Self-avoiding walks on a simple cubic lattice," *Journal of Chemical Physics*, vol. 99, pp. 3976–3982, September 1993.

17. Englisch,H., Wang,J.F., and Yao,K.L., "Directed true self-avoiding levy flights," *Journal of Physics A - Mathematical and General*, vol. 24, pp. 4843–4851, October 1991.

18. Gibson,K.D. and Scheraga,H.A., *The multiple-minima problem in protein folding*, pp. 67–94, Adenine Press, Guilderland, New York, 1988.

19. Godzik,A., Kolinski,A., and Skolnick,J., "De novo and inverse folding predictions of protein structure and dynamics," *Journal of Computer-Aided Molecular Design*, vol. 7, pp. 397–438, 1992.

20. Hammersley,J.M., "Self-avoiding walk," *Physica A*, vol. 177, pp. 51–57, September 1991.

21. Kim,I.M., Kim,A.S., and Kim,S.W., "An exactly solvable self-avoiding walks model," *Journal of Physics A - Mathematical and General*, vol. 22, pp. 2533–2537, July 1989.

22. Kim,I.M., Kim,J.E., and Kim,S.W., "An exactly solvable self-avoiding walks model 2: Triangular and honeycomb lattices," *Journal of Physics A - Mathematical and General*, vol. 24, pp. 1903–1914, April 1991.

23. Klein,D.J. and Schmalz,T.G., "Exact enumeration of long-range-ordered dimer coverings on the square-planar lattice," *Physical Review B - Condensed Matter*, vol. 41, pp. 2244–2248, February 1990.

24. Lau,K.F. and Dill,K.A., "A lattice statistical mechanics model of the conformational and sequence spaces of proteins," *Macromolecules*, vol. 22, pp. 3986–3997, October 1989.

25. Leopold,P., Montal,M., and Onuchic,J.N., "Protein folding funnels: A kinetic approach to the sequence-structure relationship," *Proceeding of National Academy of Science, USA*, vol. 89, pp. 8721–8725, September 1992.

26. Macdonald,D., Hunter,D.L.,, Kelly,K., and Jan,N., "Self-avoiding walks in 2 to 5 dimensions - exact enumerations and series study," *Journal of Physics A - Mathematical and General*, vol. 25, pp. 1429–1440, March 1992.

27. Machta,J., "The computational complexity of the self-avoiding walk on random lattices," *Journal of Physics A - Mathematical and General*, vol. 25, pp. 521–527, February 1992.

28. Masand,B., Wilensky,U., Massar,J.P., and Redner,S., "An extension of the 2-dimensional self-avoiding walk series on the square lattice," *Journal of Physics A - Mathematical and General*, vol. 25, pp. 1365–1369, April 1992.

29. Mayer,J.M., Guez,C., and Dayantis,J., "Exact computer enumeration of the number of Hamiltonian paths in small plane square lattices," *Physical Review B - Condensed Matter*, vol. 42, pp. 660–664, July 1990.

30. Moon,J. and Nakanishi,H., "Self-avoiding levy walk - a model for very stiff polymers," *Physical Review A*, vol. 42, pp. 3221–3224, September 1990.

31. Nemirovsky,A.M., Dudowicz,J., and Freed,K.F., "Thermodynamics of a dense self-avoiding walk with contact interactions," *Journal of Statistical Physics*, vol. 67, pp. 395–412, April 1992.

32. Richards,F.M., "The Protein Folding Problem," *Scientific American*, pp. 54–63, January 1991.

33. Ripoll,D.R. and Thomas,S.J., "A parallel Monte-Carlo search algorithm for the conformational analysis of polypeptides," *Journal of Supercomputing*, vol. 6, pp. 163–185, 1992.

34. Shakhnovich,E. and Gutin,A., "Enumeration of all compact confomrations of copolymers with random sequence of links," *Journal of Chemical Physics*, vol. 93, pp. 5967–5971, October 1990.

35. Skolnick,J. and Kolinski,A., "Simulations of the folding of a globular protein," *Science*, vol. 250, pp. 1121–1125, November 1990.

36. Smailer,I., Machta,J., and Redner,S., "Exact enumeration of self-avoiding walks on lattices with random site energies," *Physical Review E*, vol. 47, pp. 262–266, January 1993.

37. Sumners,D.W. and Whittington,S.G., "Detecting knots in self-avoiding walks," *Journal of Physics A - Mathematical and General*, vol. 23, pp. 1471–1472, April 1990.

38. Vilgis,T.A., "Generalized screening of excluded-volume interactions in levy walks and self-avoiding walks," *Physical Review A*, vol. 43, pp. 4156–4158, April 1991.

39. Wang,J., "A new algorithm to enumerate the self-avoiding random walk," *Journal of Physics A - Mathematical and General*, vol. 22, pp. 1969–1971, October 1989.

40. Windwer,S., "Knottedness in self-avoiding walks," *Journal of Physics A - Mathematical and General*, vol. 22, pp. 1605–1608, July 1989.

41. Windwer,S., "On the topology of loop-erased self-avoiding random walks," *Journal of Chemical Physics*, vol. 93, pp. 765–766, July 1990.