ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-215

STALK Programmers Guide

by

David Levine,[*] *Michael Facello*,[†] *Philip Hallstrom*,[‡]
Greg Reeder,[‡] *Brian Walenz*,[‡] *and Fred Stevens*[§]

Mathematics and Computer Science Division

Technical Memorandum No. 215

July 1996

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Office of Health and Environmental Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[*]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439.

[†]Computer Science Department, University of Illinois, Champaign, Illinois.

[‡]Science and Engineering Research Semester Program, Argonne National Laboratory, Argonne, Illinois 60439.

[§]Center for Mechanistic Biology and Biotechnology, Argonne National Laboratory, Argonne, Illinois 60439.

Contents

STALK Programmers Guide

by

David Levine, Michael Facello, Philip Hallstrom,
Greg Reeder, Brian Walenz, and Fred Stevens

1 Introduction

STALK is a system that models molecular docking between two proteins. A problem is posed as an optimization problem where the objective is to minimize the intermolecular interaction energy between the two molecules. The possible number of conformations between the two molecules can be very large. A parallel genetic algorithm (GA) is used to explore the conformation space and identify the low-energy molecular configurations. The CAVE, a virtual reality environment, can be used to visualize and interact with the system while it is executing.

STALK consists of two programs: `stalk.ga`, the docking program that runs the GA, and `stalk.cave`, the visualization program. The visualization component is optional.

2 `stalk.ga`: Data Structures

2.1 General Atomic Input Information

Data about the atoms read from the .car files is stored in parallel arrays (i.e., arrays that are the same size and indexed similarly by j, an atom index) with `n_atoms`(i) elements for object (protein) i. The following arrays store atomic information in the order they occur in the .car file.

- `atom` – a character string array storing the type of atom (N, CA, O, HA, etc.). Field 1.
- `location` – the three coordinates of the atom. Fields 2–4.
- `res` – a character string array storing the residue type (SER, ASN, TYR, etc.). Field 5.
- `res_id` – a character string array storing the id of the residue. Field 6.
- `type` – atom type (h, hn, o, etc.). Field 7.
- `elem` – element code for the atom (C, H, N, O, etc.). Field 8.
- `charg` – atomic charge. Field 9.
- `n_atom` – number of the atom within the .car file (the index j)

Note that these are really 2-D arrays `a`(i,j), where $i = 1, 2$ is a protein index and $j = 1,$ `n_atoms`(i) is the number of atoms in protein i. Here $j = 1$ is the first atom in the .car file, $j = 2$, is the second atom in the .car file, and so on.

2.2 Sidechain Rotations

Translation and rotation are performed through linear transformations of the atom coordinates by using matrix operations. Sidechain rotations can be expressed as the rotation of a certain group of atoms around an axis. The important arrays are

- The array `location`(i, j) contains the x, y, and z coordinates of the atoms. The term i is a protein index, and j is an atom index. A third index, not shown in the figure, is for x, y, or z. The length of the array is `n_atoms`(i), the total number of atoms in protein i.

  Another array, `bew_location`(i, j), is calculated each time we evaluate a string that has the updated atom locations.

- The array `atom_CA`(i, s) contains the indices in the `location` array where alpha carbon (CA) atom s of protein i can be found. There are `k_CA`(i) alpha carbons for each protein i (this is the same as the number of amino acids).

- The array `sc_start`(i, s) contains the indices in the `location` array where the starting atom s (which is always N for nitrogen in .car file format) of protein i can be found. This array is also of length `k_CA`(i). (This array might be better named `aa_start`(i, s) because it is really where the amino acid starts, not just the sidechain.)

- The array `sc_angles`(i, k) contains the allele index in the GA string where the kth sidechain angle can be found. As shown in Figure 1, the sidechain has no chiral angles represented in the string. This situation can occur because the amino acid has no chiral angles or because an input error occurred and those chiral angles are not variables in the GA. In general, to find out how many chiral angles are in sidechain s, subtract the values for sidechain s from sidechain $s + 1$. If the result is 0, there are no chiral angles; if 1, there is one chiral angle; and so on. This array contains pointers for *both* proteins. The GA string is of length `num_angles`+6, where `num_angles` is the total number of sidechain angles in both proteins and the "6" represents three for translation and three for rotation.

As shown in Figure 1, several arrays store information about the sidechains. Specifically, `k_CA`(s) stores the index of the alpha carbon for the sth sidechain, and `sc_start`(s) stores the beginning of the sidechain. For each type of amino acid, up to four positions along the sidechain can be rotated. Given a type number t, the array `chi_angles`(t) stores the pointers to the key atoms along the sidechain for a sidechain of type t. (Note: the sidechain type is given by the function `sc_num`.) These key atoms are important for the rotations and for measuring chiral angles. These pointers assume that the sidechain atoms start at position 1; thus, the offset of the first atom of the sidechain in the `location` array needs to be added to the numbers in `chi_angles`. In addition, a chromosome in the GA stores the actual angle values, and the array `sc_angles`(s) stores a pointer to the first angle for sidechain s.

Figure 2 shows the contents of the `chi_angles`(t) (chiral angles) array. This array points at the key atoms (see the appendix in the users guide) in an amino acid that are used to define a chiral angle. For an amino acid with four chiral angles, we can get χ_1 using `chi_angles`(1)–`chi_angles`(4), χ_2 using `chi_angles`(2)–`chi_angles`(5), χ_3 using `chi_angles`(3)–`chi_angles`(6), and χ_4 using `chi_angles`(4)–`chi_angles`(7). In each case these four atoms define the respective chiral angle.

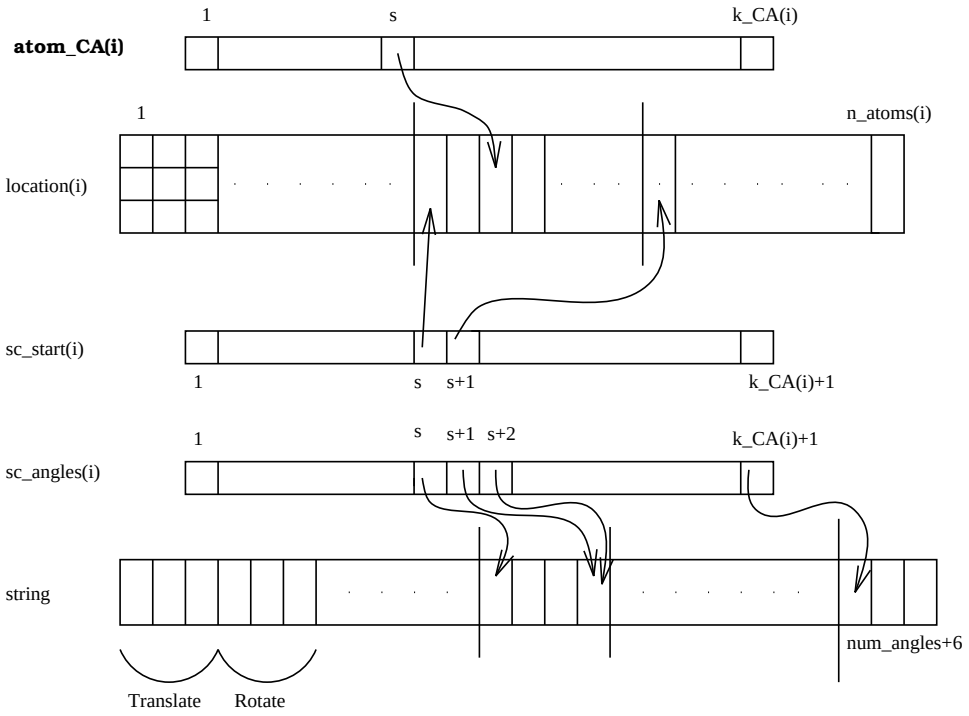

Figure 1: General Data Structures

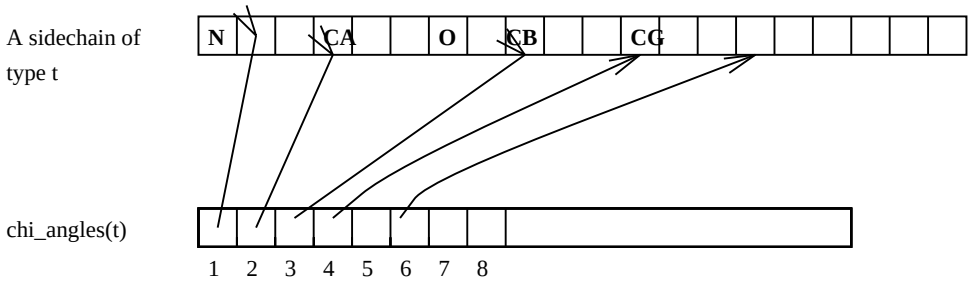

Figure 2: Chiral Angle Data Structures

The array `sc_num`(s) provides a mapping into `chi_angles`. To each amino acid s we assign a number from 1 to 20 (e.g., GLY→1, ALA→2,) and we get the t index for `chi_angles` via t = `sc_num`(s).

In order to use the above data structures to handle sidechain rotations, the program must assume a specific structure for the sidechains. If the input files do not conform to this structure, the software will give a warning. The program will still run, but the nonconforming sidechain will not be rotated. The specific structure expected for each amino acid is given in an appendix in the users guide.

2.3 Rotamer Initialization

Up to four positions along a sidechain can be rotated. A *rotamer* is a set of up to four angle values that uniquely specify the position of the sidechain. These angles are measured according to the standard convention for defining dihedral angles.

When rotamer initialization is performed, first a random rotamer is chosen for the particular type of sidechain being considered. Several rotamers can be specified in the input file, along with the probability that each will be chosen for initialization. After a rotamer is chosen, a rotation angle must to be computed for each of the rotation points that will transform the orientation given in the initial data set to that of the rotamer angles. The function `compute_chi_angle` computes the current angle at a bond in the sidechain. The difference between this and the rotamer angle is used to initialize the chromosome.

The array `chi_angles` defines the key atoms along the sidechain used in computing the chiral angles. In some cases, it is not clear which atoms should be used. These cases are cited in the appendix. Should the information in the current `chi_angles` array be incorrect, it is easy to change the initialization of this array in the file `transform.F`.

Figure 3 shows three arrays used in STALK. The first, `rot_prob`, contains cumulative probabilities for the occurrence of the specified rotamer. In the figure, the probabilities of each rotamers occurring are .2, .3, .3, and .1, respectively, for the cumulative probabilities shown. The probability of randomly generating the angles is given by 1.0 minus the final cumulative probability (.1 in this example).

The array `rot_angles` specifies the possible rotamer angles. If the angle is in the range $[\pi, \pi]$, we interpet that to mean that the user specified the angle; otherwise we generate it randomly. As a reminder, all angles in STALK are in radians.

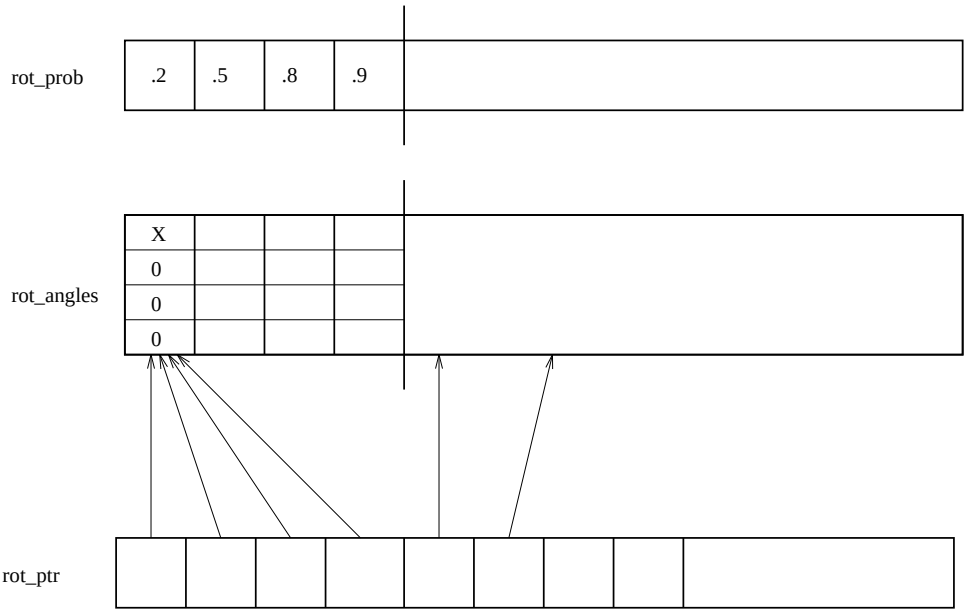

Figure 3: Rotamer Data Structures

2.4 Energy Computation

The energy function is used to rank the GA strings according to which is the best solution to the problem at any iteration. The energy computation used computes both the coulombic and Van der Waals energy. In the absence of sidechain rotations, only the energy between the two molecules is computed, since the energy within a protein is constant. This is not true if sidechain rotations are allowed.

2.4.1 Basic Energy Function

The basic energy function computes the intermolecular energy. Let P_i be the set of atoms in protein i, with $|P_i| = n_i$. Let a_{ij} be the jth atom of P_i. Then, let

E_C^{inter}	...	Coulombic energy
E_V^{inter}	...	Van der Waals energy
q_{ij}	...	charge of atom j in protein i
$d(a_{ij},\, a_{i'j'})$	...	Euclidean distance between the two atoms
D	...	dielectric constant
A_{ij}, B_{ij}	...	Van der Waals constants depending totally on the type of atom

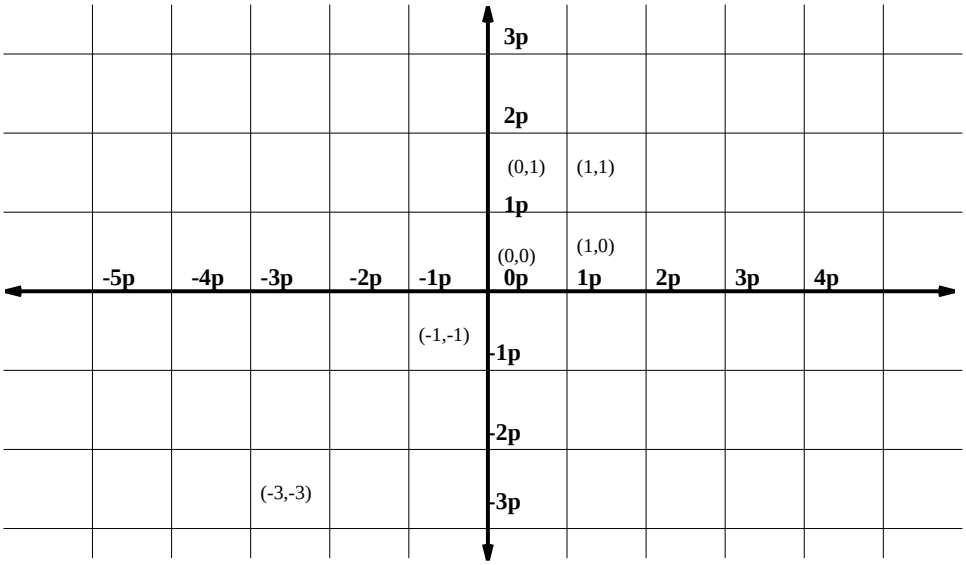

Figure 4: Energy Function

$$
\begin{aligned}
E_C^{\text{inter}} &= \sum_{j=1}^{n_1} \sum_{j'=1}^{n_2} 0.322\ q_{1i}\ q_{2i'}\ /\ (D\ d(a_{1j},\ a_{2j'})) \\
E_V^{\text{inter}} &= \sum_{j=1}^{n_1} \sum_{j'=1}^{n_2} A_{1j}\ A_{2j'}\ /\ d(a_{1j},\ a_{2j'})^6 - B_{1j}\ B_{2j'}\ /\ d(a_{1j},\ a_{2j'})^{12} \\
E_{tot}^{\text{inter}} &= E_C^{\text{inter}} + E_V^{\text{inter}}
\end{aligned}
$$

This formula implies an $O(n_1 n_2)$ time algorithm for computing the energy, which is quite large. Section 2.4.2 presents an approximation technique that requires less computation.

2.4.2 Partitioning

To reduce the computation time, we use a three-dimensional subdivision of the space, with the size of a cell of the subdivision specified by a parameter. The cell containing an atom is computed for each atom. When the energy is being computed, only pairs of atoms that lie in the same cell or immediately adjacent cells contribute to the energy sum.

As an example, a two-dimensional partition of partition size p is shown in Figure 4. The x coordinate of a box is from the grid point to the left. The y coordinate of a box is from the grid point below. To calculate the energy, we consider only atoms in our nearest neighbor (27-point stencil) boxes. With the above data structures we can answer queries such as, What atoms lie in box(i, j, k)?

The energy function using this partitioning scheme is defined as follows. Given a parameter d_{min} (indicating that if two atoms are within distance d_{min} of each other, they should be included in this sum), then

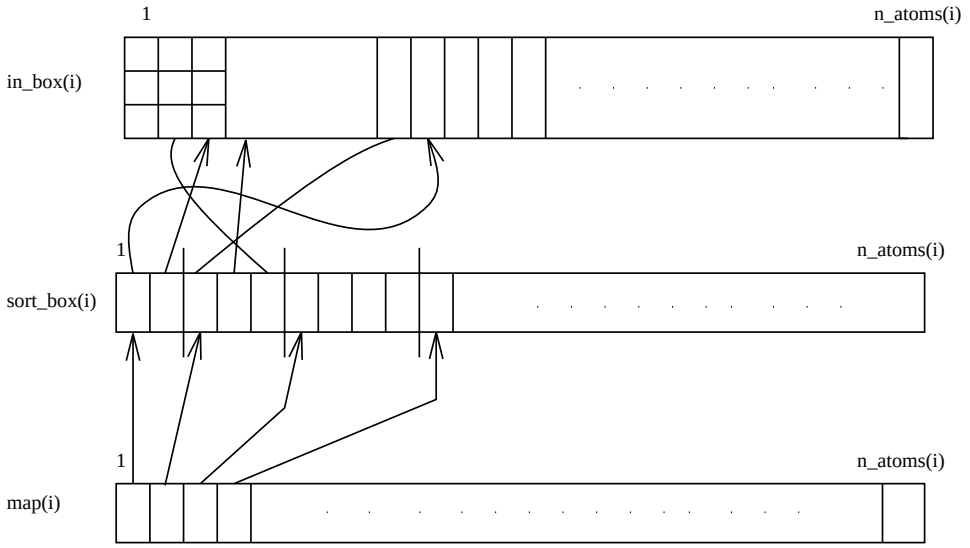

Figure 5: Energy Function

$$C_{ij} \quad \ldots \quad \text{the set of atoms not in protein } i \text{ within distance } d_{min} \text{ of atom } a_{ij}$$

$$E_C^{\text{inter}} = \sum_{j=1}^{n_1} \sum_{a_{2j'} \in C_{1j}} 0.322\ , and q_{1j}\ q_{2j'}\ /\ (D\ d(a_{1j},\ a_{2j'}))$$

$$E_V^{\text{inter}} = \sum_{i=1}^{n_1} \sum_{a_{2j'} \in C_{1j}} A_{1j}\ A_{2j'}\ /\ d(a_{1j},\ a_{2j'})^6 + B_{1j}\ B_{2j'}\ /\ d(a_{1j},\ a_{2j'})^{12}.$$

Figure 5 shows the arrays involved in holding the partitioned information. The array `in_box`(i) contains the coordinates of which *box* atom i is in. The array `sort_box`(i) contains pointers to the atoms in `in_box`. These have been sorted lexiographically (we say $(x_1, y_1, z_1) < (x_2, y_2, z_2)$ if (1) $x_1 < x_2$, $y_1 < y_2$, and $z_1 < z_2$, or $x_1 = x_2$, $y_1 < y_2$, and $z_1 < z_2$, or $x_1 = x_2$, $y_1 = y_2$, and $z_1 < z_2$) according to which box they are in. The array `map`(i)contains the number of atoms in each box (it is of length n_atoms, since potentially each atom could be in its own box). In Figure 5 there are 2 atoms in the first box, 3 atoms in the next box, and so on.

2.5 Program Structure

The program is broken down into nine source files:

aux.F This file contains auxiliary functions, including several sorting and searching routines.

energy.F This file contains functions for computing energy. There is an implementation of both the basic energy computation (`evaluate2`) and the energy computation using the partitioning (`evaluate`).

io.F All input and output related functions.

main.F All Main driver program. Sets up and initializes MPI communicators. Determines if `stalk.cave` is to be used.

matrix.F This file contains the matrix and vector operations used in `transform.F`.

partition.F This file contains the functions for computing the subdivision of the spaced filled by the molecule.

runga.F Calls the initialization routines, calls the GA driver routine, and then calls the clean up routine.

talktocave.F Handles all communication with `stalk.cave` program.

transform.F All procedures for computing the coordinates of a protein after the translation and rotation of both the molecule and sidechains as specified by a chromosome.

Header files used are

energy.inc This file contains global variable declarations. This file must be included in every function that accesses any of these variables.

tags.h Defines parameters used for communication between `stalk.ga` and `stalk.cave`.

The relevant files for compiling and building the program are

compile A Unix shell script that calls the `Makefile` with the proper variables.

Makefile Called by `compile` to actually compile the above files.

In addition the user will need the appropriate MPI and PGAPack header (`mpif.h` and `pgapackf.h`) and library files (`libmpi.a` and `libpga0.a`).

2.6 Compilation

This section discusses compiling `stalk.ga`. For execution information see the STALK users guide [LeFaHaReWaSt95].

The source files, header files, and compilation files for `stalk.ga` are in the directory `/home/STALK/source.ga`. The subdirectory `RCS` there is a symbolic link to `/home/STALK/source.RCS`. This directory contains the source code in "revision control system" format. These files must to be checked out before compilation can occur. (They must also be checked out, in *lock* mode, before any editing can be done.)

In this directory, if one types `compile` with no parameters you will see

```
usage: compile <Architecture> <Device> <Visualization>
       Architecture  = { sp2, rs6000, IRIX, sun4 }
       Device        = { ccomm, nexus-tcp, nexus-mpl, p4, mpl }
       Visualization = { vis, novis }
```

This says that `compile` must be called with three parameters—the machine it is being built for (IBM SP, IBM RS6000 workstations, Silicon Graphics workstations, or Sun 4), the communication mechanism (`mpl` is the only mechanism known to work in the recent past), and whether or not `stalk.cave` will be used. For example, to build `stalk.ga` for the IBM SP without a corresponding visualization program use `compile sp2 mpl novis`.

Here is a complete example where the user edits a file (`runga.F`) and rebuilds `stalk.ga`. First, (`runga.F`) is checked out of RCS in locked mode, edited, and replaced. Next, all files needed for compilation are checked out (not in lock mode) and `compile` is used to build an executable.

In this case, building for the IBM SP, this compilation must be done on either `bonnie.mcs.anl.gov` or `clyde.mcs.anl.gov`. The `compile` script will automatically copy the executable (`stalk.ga.sp2.mpl`) to the correct directory (`/home/STALK/bin`).

```
cd /home/STALK/source.ga
co -l runga.F
vi runga.F
ci runga.F
co *.F energy.inc tags.h
compile sp2 mpl novis
```

3 stalk.cave: Data Structures

The docking package is written in C and runs on an SGI Onyx with three reality engines. (The Onyx drives the CAVE virtual environment.) The STALK package is written in Fortran and runs on multiple nodes of a 128-node IBM Scalable POWERparallel system. Because these platforms have separate file systems, a p4 secure server is started on the machine driving the CAVE. The programs can then communicate via MPI (Message Passing Interface) calls.

The docking program communicates with the STALK program that drives it, using a version of MPI. In addition to the MPI library, docking also uses the OpenGL library for the graphics display, and the CAVElib library to drive the CAVE virtual environment.

3.1 Program Datatypes

Three declared datatypes are used in this program.

The structure *atom* is a link in a linked list. Its elements are the x, y, and z coordinates of an atom; the coordinates of the atom preceding it in the .car file that it connects to; the element symbol; and a pointer to the next link. There is a separate linked list of atoms for each chain in each molecule, which are linked by the datatype chain.

The structure *chain* is a link in a linked list. Its elements are the x, y, and z coordinates of the alpha carbon that it connects to; the name of the residue it belongs to; a pointer to a list of atoms; and a pointer to the next link. The first link points to the molecule's backbone, the next points to the first sidechain, and so forth. The coordinates of all the atoms in a chain are relative to coordinates of its alpha carbon. For the backbone, these coordinates are (0, 0, 0).

Table 1: `stalk.cave` Global Variables

Type	*Name*	*Description*
chain	*protein	linked list of protein chains
chain	*drug	linked list of drug chains
scinfo	*protein_info	array of info on protein sidechains
scinfo	*drug_info	array of info on drug sidechains
int	nsc_protein	number of protein chains (including backbone)
int	nsc_drug	number of drug chains (including backbone)
int	max_menu_item_length[ ]	length of the longest item in menu i
int	num_menu_items[ ]	number of items in menu i
GLuint	PROTEIN_LIST	base number for protein display lists
GLuint	DRUG_LIST	base number for drug display lists
(the following variables are stored in shared memory)		
double	*PPos	CAVE coordinates of protein position
double	*PAngle	protein rotations about x, y, and z axes
double	*DPos	coordinates of drug relative to protein
double	*DAngle	drug rotations relative to protein
double	*energy	total energy of system
short	*menu	current menu to be displayed (0 = no menu)
short	*item	menu item highlighted
short	*action	action to take when restarting genetic algorithm
short	*stop	program terminates when *stop = 1
int	*sc	selected sidechain
int	*gen	genetic algorithm generation
short	*state	state variable array for parameters (see below)
void	*Arena	shared memory arena
int	*ProIndex	sidechain angle index array for protein
int	*DrugIndex	sidechain angle index array for drug
int	chromlen	length of the genetic algorithm chromosome

The structure *scinfo* is an element in an array that has information on each sidechain. Its elements are the x, y, and z coordinates of the alpha carbon the sidechain connects to; the coordinates of the first atom in the chain; the name of the residue to which the sidechain belongs; the angle that the sidechain is rotated; and the display mode of the sidechain. The display mode is 0 for normal display, 1 for highlighted, or 2 for not displayed.

3.2 Global Variables

Table 1 is a list of the global variables in `stalk.cave`.

Table 2: `stalk.cave` Program States

Number	*Mnemonic*	*States*	*Description*
0	GARUN	TRUE/FALSE	whether genetic algorithm is running
1	INFO_DISPLAY	ON/OFF	whether drug info is displayed
2	PROTEIN_SURFACE	ON/OFF	whether protein surface is displayed
3	DRUG_COLOR	PURPLE/FULL_COLOR	how drug is displayed
4	SC_DISPLAY	ON/OFF	whether sidechains are displayed
5	WANDMOVE	DRUG/BOTH	what the wand moves
6	WHICH_SC_SELECTED	DRUG/PROTEIN	which sidechains are selected
7	SC_SELECT_MODE	TRUE/FALSE	if in select sidechain mode
8	SC_ROTATE_MODE	TRUE/FALSE	if in rotate sidechain mode
9	ENERGY_DISPLAY	ON/OFF	if energy shown

3.3 Program States

Table 2 is a list of the program states in `stalk.cave`.

3.4 Top-Level Functions

void preinit(int argc, char **argv) initializes MPI, input data, allocate shared memory, initialize shared data (arc and argv passed to MPI_Init)

void init(void) sets up OpenGL lighting model and display lists

void display(void) controls the CAVE display function

void navigate(void) handles molecule and menu manipulation

void Data_Transfer(void) handles MPI communication with STALK program

cleanup(void) stops the STALK program and MPI

3.5 Program Flow

The program follows the basic format for a CAVE program, as described in the CAVE users guide. However, this program forks an additional process for the navigate function, which handles all wand transformations and menu control. The computation loop is used for MPI communication. Thus, three processes are running: the display routine, the navigation routine, and the computation loop.

From main, the preinit function is called. This function initializes MPI, inputs the molecular data for the protein and drug molecules, allocates shared memory, and initializes shared data. The CAVE is then initialized; and the init function, which sets up the OpenGL lighting model and display lists is passed to CAVEInitApplication. From there, the program starts three separate processes. The display function is passed to CAVEDisplay, for the first process. The second process is forked manually and runs the navigate function. The third process is the CAVE computation loop, from which is run

the Data_Transfer function, which handles communication with the STALK program. When "End Simulation" is selected from the main menu, the cleanup function and CAVEExit are called, and the program ends.

3.5.1 Input

The input function is called from the preinit function first for the protein molecule, then for the drug molecule. For its input, it takes a pointer to the chain data type. It returns an integer representing the number of chains (backbone and sidechains) in the molecule.

For each molecule, the docking program receives five sets of data from the STALK program. The first is the number of atoms in the molecule; the second is an array of character strings for the atom types; the last three are arrays of doubles for the x, y, and z coordinates.

The information for each atom is then placed in the chain linked lists, with the atom types determining which atoms are connected and into which chain the atom is placed. The coordinates are scaled for the CAVE.

3.5.2 Display Lists

Four display lists are created for the backbone and each sidechain of each molecule. This configuration allows for separate and quick manipulation of individual sidechains. The first set of display lists is for the normal, full-color display mode. The second is for a special mode, which is with a surface for the protein and ia in magenta for the drug. The third set is for highlighted sidechains in the normal mode. The last set is for highlighted sidechains in the special mode, which is redundant in the case of the drug molecule, because the atom sizes are the same as in the normal mode. These sets are created by calling the function Atom_List with the proper input values.

The Atom_List function creates a separate display list for each chain (including the backbone) of a molecule, according to the input parameters. Its inputs are a pointer to the molecule's chain linked-list, a display list base number, a floating-point number to scale the atom sizes, and color vectors for each atom and for the molecular bonds. For each chain, a new display list is created. For each atom in the chain, a sphere is drawn at the coordinates stored in its link, with the size determined by the atom type and scaled by the input scale, and its color determined by the input color for that atom type. A line is then drawn in its input color, from the atom's coordinates to the coordinates of the atom it connects to (cx, cy, and cz in the link).

After the molecule display lists are created, the chain data structures for the molecules are no longer needed, so the sidechain information is transferred to the scinfo arrays, and the memory for the linked lists is freed up.

To access a particular display list, three pieces of information are needed. The first is the display list base, which is either PROTEIN_LIST or DRUG_LIST. Second is the mode, which is selected by multiplying zero, one, two, or three by the number of sidechains in the molecule (`nsc_protein` or `nsc_drug`) and adding this to the display list base. Third is the particular sidechain, which is selected by adding its number to the result above.

A display list is also created for each character in the bitmap font, with a call to the function makeRasterFont, which is taken from the OpenGL sample program "font.c".

3.5.3 Display Loop

The display function is called twice per frame (once for each eye) for each wall. It first clears the viewing screens to black, then draws the view according the program states. If a menu is selected, it calls display_menu with the current menu and item numbers. The function display_menu draws the menu with the entries given by the function menu_entry and highlights the input item. If the STALK program has sent the initial drug position, the display function calls the function display_info. The function display_info displays the generation number at the top center of the front wall, and, if selected, the drug position and total energy. Finally, the molecules are displayed by calls to the function display_molecule.

The function display_molecule displays a molecule according to the input parameters. Its inputs are the position, angle, and sidechain information arrays for the molecule; the number of chains in the molecule; the display list base number; a number signifying which display mode the molecule is to be drawn in (normal or special); and the sidechain selected (or 0, if no sidechain is selected). First, the backbone is drawn at its location and orientation, according to the display mode. Then, each sidechain is drawn at its relative position and orientation, according to its individual display mode, the overall display mode, and whether or not the sidechain is currently selected.

3.5.4 Navigation Loop

The navigate function runs on a separate process and exits, without returning, when "End Simulation" is selected from the main menu. It is used to manipulate the program by using the wand. What the wand manipulates is based on the program states. The possibilities, in order of preference, are sidechain selection, sidechain rotation, menu control, and movement of molecules.

In the sidechain selection mode, the navigate function calls the nav_select function with the number of protein chains if protein sidechains is selected, or the number of drug sidechains otherwise. The other input to this function is a pointer to the selected sidechain number. If button 1 on the wand is pressed, the function increments this number. If button 2 is pressed, the number is decremented. If button 3 is pressed, the sidechain selection mode is exited.

In the sidechain rotation mode, the navigate function calls the nav_rotate function with the protein `scinfo` array if protein sidechains is selected, or the drug `scinfo` array otherwise. The other input to this function is the selected sidechain number. While button 1 is pressed, the function increases the sidechain angle, rotating it counterclockwise. While button 2 is pressed, the angle decreases, rotating the sidechain clockwise. If button 3 is pressed, the sidechain rotation mode is exited.

If neither of the above modes is in effect and a menu is selected, the function nav_menu is called, with pointers to the menu and item numbers. If button 1 is pressed, the item number is incremented. If button 2 is pressed, the item number is decremented. If button 3 is pressed, the menu action from the function menu_action is taken for the selected item.

If neither of the above modes is in effect and no menu is selected, the function nav_molecule is called, with a number signifying whether to move just the drug molecule or both molecules. While button 1 is pressed, the molecules are translated by the motion of the wand. While button 2 is pressed, the molecules are rotated by the orientation of the wand. If button 3 is pressed, the main menu is selected. If the drug molecule is moved by itself, the energy is blanked until a new energy evaluation

is done.

3.5.5 Computation Loop

The function Data_Transfer is called by the CAVE computation loop. The loop exits when "End Simulation" is selected from the main menu.

The Data_Transfer function handles all MPI communication with the STALK program, after the initial input. In the normal running mode, on each iteration the function sends a signal to the STALK program, requesting that it send the best chromosome and its energy evaluation. The function then receives these values and extracts the drug position and orientation from the chromosome, scaling the coordinates for the CAVE. The coordinates and angles are also added to the protein's, so that the drug's position and orientation are relative to the protein. The individual sidechain rotations are also extracted.

When the user suspends the algorithm, a signal is sent to the STALK program telling it to wait for the drug's new position and orientation to be sent back. These will be sent back with the protein information subtracted out and the coordinates unscaled. The information will be sent with a tag telling the STALK program what action to take. The possible actions are as follows:

Energy evaluation Evaluate the drug's energy at this position, send the evaluation back, and wait to receive the information again.

Replace worst member Restart the algorithm, replacing the worst member with the returned chromosome, and wait for a request.

Reseed population Restart the algorithm, reseeding the population with the returned chromosome, and wait for a request.

Restart algorithm Restart the algorithm without changing anything, and wait for a request.

If a request for an energy evaluation was sent, the function receives this information and sends a new chromosome, when the time comes. Otherwise, it returns to the normal running mode.

3.5.6 Program Menus

The program menus allow the user to change certain parameters of the program. The number of menus, number of items in each menu and maximum length of a menu entry for each menu, are defined in the global variables max_menu_item_length[ ] and num_menu_items[ ]. Max_menu_item_length[i] is the maximum length of an entry for menu i. Num_menu_items[i] is the number of items in menu i. Max_menu_item_length[0] and num_menu_items[0] are always 0. The features of the individual menu items are defined in the functions menu_entry and menu_action.

The function menu_entry is called for each item in the current menu, from the display_menu function in the display loop. Its inputs are the current menu number and an item number. The menu number is used by a switch statement to choose a nested switch statement for that menu. The item number is used by that switch statement to choose what to display for that entry.

The function menu_action is called from the nav_menu function in the navigation loop when button 3 of the wand is pressed. Pointers to the numbers of the selected menu and item are its inputs. The menu number is used by a switch statement to choose a nested switch statement for that menu. The item number is used by that switch statement to choose the action to take for the item chosen.

Currently three menus are used by the program. Menu 1 (the main menu) is brought up by pressing button 3 of the wand when no menu is displayed; it controls overall program parameters. Menu 2 (sidechain operations) is brought up by selecting "Sidechain Operations" from the main menu; it controls the parameters of individual sidechains. Menu 3 (restart options) is brought up when the genetic algorithm is restarted from the main menu; it controls how the algorithm is to be restarted. Menus 1 and 2 are exited by selecting "Exit Menu"; menu 3 is exited by selecting any item.

To add an item to an existing menu, one simply adds a case statement for the item in the appropriate switch statement for the menu to which it will be added, in both the `menu_entry` and `menu_action functions`. Then, the entry is updated in `num_menu_items` for the appropriate menu. If the item added has a longer entry than any existing items in the menu, the appropriate entry of `max_menu_item_length` must be updated.

To add a new menu, a switch statement for the menu is added in both the `menu_entry` and `menu_action` functions. Then entries for the menu are added in `num_menu_items` and `max_menu_item_length`.

3.6 Program Files

The project directory for the docking program is `/afs/fl/home/levine/STALK`. There is also a test directory at `/afs/fl/home/levine/STALK/util/CAVE2`. The program header files are in the `source.cave` directory. These are listed below:

stalk.cave.h program includes, defines, and typedefs

globals.h all global variables

fonts.h bitmap definition of the font used for menu and data display

The program source code is in the `source.cave` directory. The functions in each file are listed below:

main.c main program loop

preinit.c preinit, cleanup

init.c init, Atom_List, SolidSphere, Square

display.c display, display_menu, display_info, display_molecule

nav.c navigate, nav_select, nav_rotate, nav_menu, nav_molecule

comp.c Data_Transfer

input.c input

menu.c menu_entry, menu_action

font.c makeRasterFont, printString

3.7 Compilation

This section discusses compiling `stalk.cave`. For information on running it, refer to the users guide.

If one types `compile` with no parameters in the directory `/afs/fl/home/levine/STALK/source.cave` you will see

```
usage: compile <Architecture> <Device> <Visualization>
       Architecture  = { sgi }
       Device        = { ccomm, nexus, p4 }
       Visualization = { sim, cave }
```

This says that `compile` must be called with three parameters—the machine it is being built for (The Silicon Graphics Onyx is the only choice currently), the communication mechanism (`nexus` is the only mechanism that works with `stalk.ga`'s `mpl`), and whether to use the CAVE (`cave`) or CAVE simulator (`sim`).

References

[Le95a] D. Levine. *Users Guide to the PGAPack Parallel Genetic Algorithm Library.* Argonne National Laboratory, ANL-95/18, January 1996.

[Le95b] D. Levine. A public-domain parallel genetic algorithm library. Available by anonymous `ftp` from `info.mcs.anl.gov` in the directory `pub/pgapack`, file `pgapack.tar.Z`

[LeFaHaReWaSt95] D. Levine, M. Facello, P. Hallstrom, G. Reeder, B. Walenz, and F. Stevens. *STALK Users Guide.* ANL/MCS-TM-214, Argonne National Laboratory, 1996.