Software News and Updates

SHARPEN—Systematic Hierarchical Algorithms for Rotamers and Proteins on an Extended Network

ILYA V. LOKSHA,[1] JAMES R. MAIOLO III,[2] CHENG W. HONG,[1] ALBERT NG,[1] CHRISTOPHER D. SNOW[2]
[1]*Division of Engineering and Applied Science, California Institute of Technology, 1200 E. California Blvd., MC 210-41, Pasadena, California 91125*
[2]*Division of Chemistry and Chemical Engineering, California Institute of Technology, 1200 E. California Blvd., MC 210-41, Pasadena, California 91125*

Received 13 October 2008; Accepted 5 December 2008
DOI 10.1002/jcc.21204
Published online in Wiley InterScience (www.interscience.wiley.com).

Abstract: Algorithms for discrete optimization of proteins play a central role in recent advances in protein structure prediction and design. We wish to improve the resources available for computational biologists to rapidly prototype such algorithms and to easily scale these algorithms to many processors. To that end, we describe the implementation and use of two new open source resources, citing potential benefits over existing software. We discuss CHOMP, a new object-oriented library for macromolecular optimization, and SHARPEN, a framework for scaling CHOMP scripts to many computers. These tools allow users to develop new algorithms for a variety of applications including protein repacking, protein–protein docking, loop rebuilding, or homology model remediation. Particular care was taken to allow modular energy function design; protein conformations may currently be scored using either the OPLSaa molecular mechanical energy function or an all-atom semiempirical energy function employed by Rosetta.

 J Comput Chem 00: 000–000, 2009

Key words: protein–structure prediction; object-oriented programming; rotamer optimization; distributed computing; Folding@Home

Introduction

The intrinsic properties of folded or misfolded proteins are largely derived from the final three-dimensional conformation of the chain. For molecular biology, bioengineering, and medical endeavors, the demand for detailed protein structural information continues to increase. Despite improvements arising from structural genomics efforts, the pace for experimental determination of detailed protein tertiary and quaternary structures is limited by automation and throughput challenges. X-ray crystallography can produce highly accurate protein structures (<1 Å RMSD), but crystallization and subsequent analysis of the X-ray diffraction patterns are time consuming and require significant manual intervention. Nuclear magnetic resonance (NMR) spectroscopy can be used to define the solution structure for proteins, but precise results require a large number of constraints, and standard methods are limited to proteins under 40 kDa.[1]

Computational prediction of protein structures can be used to circumvent traditional experimental methods or to augment speed and/or accuracy.[2] Furthermore, accurate computational models allow for *de novo* protein design[3] or guided directed evolution via library design,[4] with concomitant medical and engineering benefits.[5] To this end, a great deal of work has been done to produce computational methods of predicting folded conformation from the amino acid sequence.[6] Despite continuous progress in the algorithms and underlying computational hardware, available methods are often inaccurate or intractable for problems of interest. The computational intensity of the problem stems directly from the astronomical space of possible protein conformations. Despite the theoretical size of the search problem, many proteins need no assistance from molecular chaperones and require only time to consistently fold into the same native conformation.

Combinatorial Optimization

Algorithm development for computational structural biology has traditionally been a low throughput process because of the

Correspondence to: C. D. Snow; e-mail: csnow@alum.mit.edu

complexity and computational difficulty of implementing and comparing a large number of prospective algorithms. In this work, we introduce two software resources aimed to allow rapid prototyping and scaling of candidate computational structural biology algorithms that rely upon combinatorial optimization. Many problems at the forefront of computational structural biology, including homology model remediation, flexible protein-protein docking, and conformational change, can be addressed with combinatorial optimization. In particular, most popular structure prediction algorithms use combinatorial optimization of discrete side chain conformations (rotamers).

Energy Function Optimization

In addition to the difficulty of sampling massive search spaces for protein structure prediction, constructing or choosing an energy function with the correct properties is a major roadblock to obtain high-quality structure predictions.[7–9] Applicable energy functions range from molecular mechanical force fields, statistical energy functions, or hybrids, with increasing reliance on quantum chemical calculations for assigning parameter values.[10–14] The numbers of parameters in a typical energy function have prevented exhaustive parameter optimization. The new software was written with the goal of energy function optimization in mind. To address the challenge of mixing molecular mechanical energy functions with semiempirical energy functions like an all-atom scoring function from Rosetta, we implement energy functions as plug-in dynamic libraries.

Code Availability

The core of the software is a new open-source library called CHOMP (Combinatorial Hierarchical Optimization for Macromolecular Problems). When packaged for scalable distribution, we refer to the program as SHARPEN (Systematic Hierarchical Algorithms for Rotamers and Proteins on an Extended Network). SHARPEN currently consists of the CHOMP library code, a script called the JobQueue, and an interface between the JobQueue and the Folding@Home (F@H) distribution platform.[15] The JobQueue is a python program that takes CHOMP scripts and splits them into self-contained "Jobs," which then bundles into multi-Job Work Units (or WUs). The operation of the JobQueue is self-contained, which will allow end-users to readily develop their own network distribution schemes. Source code, installation instructions, and examples for CHOMP and the JobQueue are available under the permissive MIT license from http://www.sharp-n.com/. We are currently using the new resource to prototype algorithms for loop rebuilding and homology model remediation. This work details the salient features of CHOMP and SHARPEN and describes the way they integrate with Folding@Home.

Scaling

The transition from single CPUs to multicore CPUs and the upcoming transition to many-core CPUs present an excellent opportunity to advance computationally intensive algorithm development. The JobQueue infrastructure allows users to deploy the same python script on a single processor, multiple processors on the same machine, a local heterogenous cluster, or a large-scale distributed computing network. This ease of scaling is critical for the rapid prototyping of algorithms that require substantial computational resources.

Because of the JobQueue's generality, a limiting factor to scaling distribution is serialization overhead. The JobQueue must package the intermediate steps in a computation for distribution over a network, which can become a bottleneck if the script is not well designed for parallel processing. Simply, if a server requires 1 s to serialize a 60-min Job, it can create new Jobs rapidly enough to support at most 3600 remote processors. Future JobQueue versions will take advantage of multiple server cores to increase the rate of Job serialization. Users can also use the SHARPEN architecture with their own domain-specific distribution frameworks, which will require additional design effort but can yield better scaling for specific problems.

CHOMP

CHOMP (Combinatorial Hierarchical Optimization for Macromolecular Problems) is a software library directed toward optimizing macromolecular structure. It provides data representations for atoms, amino acids, and sidechain rotamers, with pending support for nucleic acids and small molecules. CHOMP also offers modular combinatorial optimization methods. To address a problem such as side-chain repacking, one may select an optimization method from a growing list that includes basic enumeration, Monte Carlo, serial replica exchange, and simulated annealing.

Implementation

The core of CHOMP is written in C++ to provide a clean object-oriented interface coupled with good runtime performance. It is easiest to compile CHOMP using Boost.Jam and GCC under Linux and Mac OSX or Microsoft Visual Studio 9 under Windows. The Boost.Graph C++ library is used extensively for generic adjacency list-based graphs, and Blitz++, a C++ library for high-performance scientific computing, provides efficient implementations of basic mathematical data structures such as vectors and multidimensional arrays.[16,17] To provide an environment for rapid prototyping and testing, this core functionality is exposed to Python using the Boost.Python library and the Py++ code generator.[18] Thus, all of the objects composed in C++ can be instantiated in Python scripts, and their methods can be called as Python. Since both Python compatibility code generation and dependency detection are automated by Py++ and Boost.Jam, respectively, the build process is relatively simple. In recent years, the C++/Python combination has become popular for computational structural biology.[19] This choice allows rapid prototyping, ready code reuse, and integration with powerful high-level algorithms such as the JobQueue.

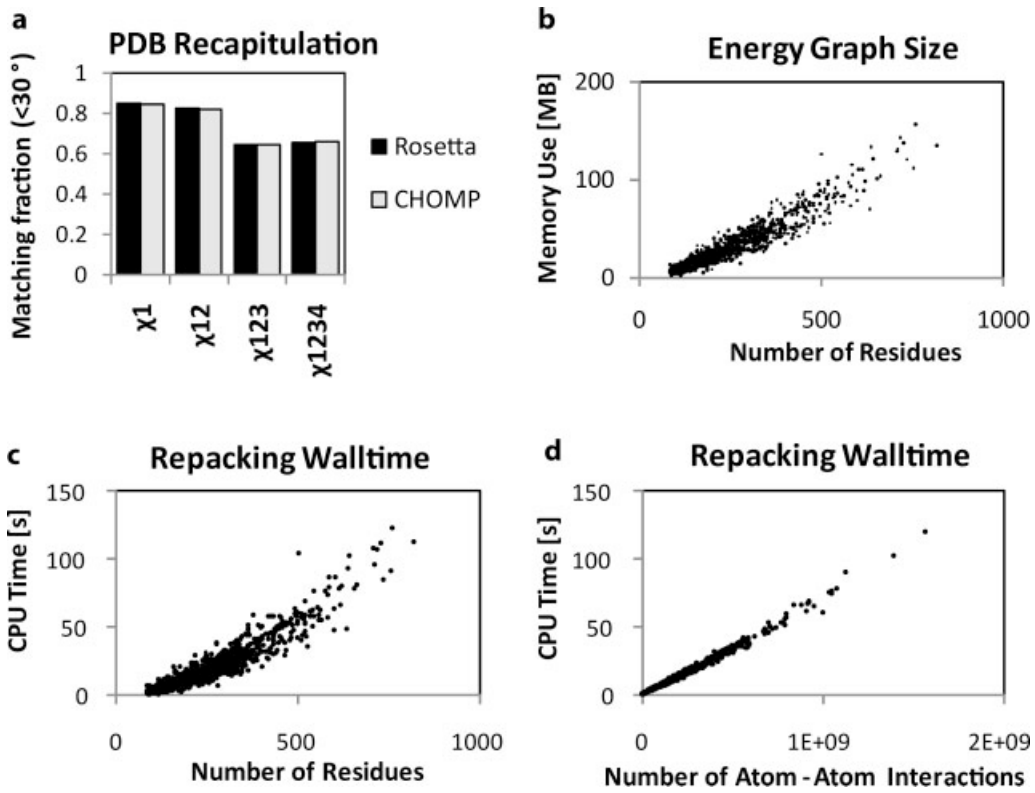

Figure 1. (a) PDB sidechain dihedral recapitulation (<30°) is similar to Rosetta and (b) the Energy-Graph data structure has relatively modest memory requirements. The CPU time needed to repack proteins (c) scales with protein size and (d) closely tracks the number of atom–atom energy evaluations.

Modular Energy Function

An energy function maps a protein conformation to an estimate of that conformation's energy. Energy function selection is an open problem and is of critical importance for the accuracy of the output protein structures that result from combinatorial optimization.[8] Two energy functions currently implemented in CHOMP are an all-atom energy function employed by the Rosetta software suite (version 2.1.0: scorefxn 12) and the OPL-Saa force field.[12] Most aspects of a CHOMP energy function can be modified by editing a set of precomputed tables of values, which are generated by external Python scripts. This approach offers the dual advantage of runtime efficiency and easy modification. The website (http://www.sharp-n.com/) provides example scripts for generating alternate treatments of repulsion and dispersion. The portions of the energy function that cannot be expressed in table form are compiled into a C++ shared library, which is loaded at runtime. Thus, we can swap and compare energy functions "on the fly." Computational biologists can therefore tailor energy functions for particular problems with a resultant increase in the accuracy. As a platform, SHARPEN is well suited to the kind of automated exploration and experimentation that will be necessary to gain greater insight into constructing tailor-made energy functions.

Energy Graphs and Hypergraphs

To perform combinatorial optimization, it is necessary to compute the energy of interaction between pairs of candidate rotamers. The memory required to store such a data set is quadratic in the number of rotamers and can quickly enter the multi-gigabyte range for even modestly sized problems. To reduce memory usage, we store our data in an adjacency list-based graph using Boost.Graph and omit energies that are sufficiently close to zero. The resulting memory consumption scales reasonably with protein size; well under 100 MB for typical rotamer repacking problems (Fig. 1a). Boost.Graph offers simple and powerful data structures "out-of-the-box," which we have not tuned for our specific problem domain. Other rotamer optimization software packages use more complex data structures that are designed to be more cache-friendly, but we have not observed significant performance deficits. Specifically, time trial-simulated annealing repacking of 1100 diverse proteins indicated that CHOMP was comparable with Rosetta (6.6 CPU hours rather than 7.0 CPU hours using 1.66 GHz Core Duo processors). CHOMP benefits from shared initialization overhead, but Rosetta currently has superior scaling with problem size because of the trie pruning algorithms that are not yet implemented in CHOMP.[20]

For more complex optimization problems, CHOMP provides a hypergraph data structure that can store multibody effects. A hypergraph can be used in place of a normal energy graph to represent problems in which the energy is not strictly decomposable into pairwise terms. This additional generality comes at a performance penalty—the memory usage is approximately doubled and the simulated annealing performance is degraded by a factor proportional to the average number of edges per vertex in the graph. However, the gain in algorithmic flexibility from the hypergraph abstraction is enormous. Any type of higher order terms can be stored in the hypergraph. Potential higher order correction terms include effects that are not pairwise decomposable such as protein polarization, surface area effects, or Poisson-Boltzmann solvation.[21–23]

Chain Rebuilding CCD Variant

To use CHOMP for backbone search applications, we also implemented a form of loop rebuilding. The current loop rebuilding algorithm is a variant of cyclic coordinate descent (CCD), which will restore proper chain connectivity at breaks between poses.[24] Our algorithm alternates between updating the N- and C-terminal branches bracketing a discontinuity. Torsion step sizes are reduced farther from the discontinuity to counteract the increased lever arm. To reconnect the backbone, we randomly choose φ and ψ for the mobile residues and attempt to reconnect them with CCD, repeating this process several hundred times to find the most favorable φ and ψ. Random φ and ψ values are drawn from three 30° bin Ramachandran histograms (Pro, Gly, or others) derived from 994 high-resolution PDB models.[25]

SHARPEN

SHARPEN (Systematic Hierarchical Algorithms for Rotamers and Proteins on an Expansive Network) describes any packaging of the CHOMP library into a form readily used for distributed CHOMP calculations (Fig. 2). The key features needed for SHARPEN are the JobQueue, which packages calculations into Jobs and Work Units, and an interface to a distribution system (Distributor) that sends work units to clients and receives finished work units from clients. Although a more general client-server distribution scheme could readily be developed and integrated with the SHARPEN JobQueue, we were able to make use of the existing F@H architecture run by the Pande group at Stanford for our distribution scheme. Future versions of SHARPEN may include Distributor and Client programs for distribution of CHOMP calculations outside of the F@H network (for example on a local cluster).

The JobQueue

The JobQueue is a Python script that sits between a "master script" written to perform a calculation using the CHOMP library and a client-server communication system (Distributor) that distributes and retrieves computational units (Fig. 2). The

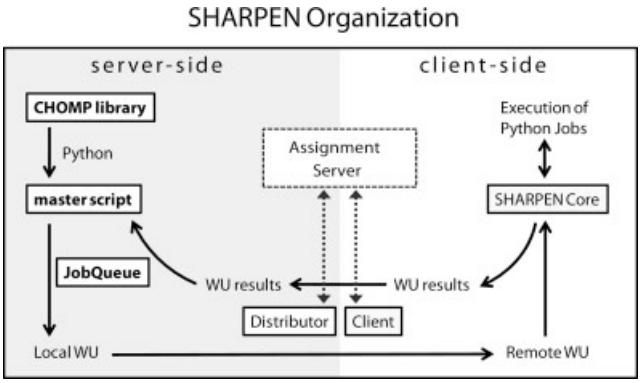

Figure 2. Calling routines from the CHOMP library, a master script is written in a functional style. The script includes calls to functions intended for remote execution (Jobs). The JobQueue packages these Jobs into work units (WU). If a Client program is assigned to this server (by an Assignment server), the Client downloads a WU and passes control to the Core, which executes the Python Jobs within the WU. Once results are returned, the JobQueue resolves any Job dependencies.

JobQueue runs the master script, executing all the code it encounters locally until it finds a call to a function in a module that has been designated as remote. Each call to a function in a remote module is meant to be run as a single computation on a client machine, although exactly the same script may be run entirely locally for testing and debugging purposes. Instead of executing immediately, remote functions are packaged into Jobs and Work Units, which are sent over the network for remote execution. For good scalability, the CPU-time to complete a Job should greatly exceed the overhead CPU-time needed to serialize its inputs for sending over the network. At a minimum, Jobs should correspond to several minutes of computation.

Whenever the JobQueue receives a request for WUs from the Distributor (e.g., F@H server process), it packages a configurable number of Jobs into each WU by serializing both the Python script representing the Job and its inputs. For nontrivial master scripts, it will often be the case that the input to some Jobs will be the output from other Jobs. To manage these dependencies, the JobQueue maintains a data structure of dependencies between Jobs, and it will wait for a Job's dependencies to be met before sending it out for execution. To make it possible for the JobQueue to reliably calculate the dependency graph of Jobs, it is necessary that master scripts be written in somewhat "functional" style of programming (see website, for example master scripts). In particular, remote functions may not have any side effects and may not reference or change any global variables (although global constants are permitted). In this way, the JobQueue can analyze the data flow of the computation and properly distribute Jobs as WUs when their inputs are available. Once the WU is produced, it is written to disk at a configurable location, and when all the requested WUs have been produced, the Distributor is notified.

To maintain the reliability and consistency of the JobQueue, all critical data structures are stored in a MySQL database (http://www.mysql.com/). This allows the JobQueue to resume

execution mid-script in the event of a crash. To achieve this level of robustness, the JobQueue stores a complete record of a script's computation, including all intermediate values. As a convenient side effect, this allows users to examine each step in their computation for debugging or analysis purposes. In principle, one can substitute a different value for the result of a Job in the middle of a finished computation, and then re-run only the Jobs affected by this change—essentially "going back in time" to try a different execution path. This is a valuable feature for scripts that might take CPU-centuries to re-run in their entirety. When it is unnecessary or impractical to maintain a complete record, the JobQueue can be set to automatically "garbage collect" data that is no longer needed as an input for further Jobs.

Custom Distributor: Folding@Home

The Distributor is any process that handles server–client communication, including the transfer of work units to remote machines. For a local resource, this role could be filled by any number of cluster management software packages. To capitalize on volunteer computing efforts, we used the existing F@H server infrastructure. At the time of writing, the F@H network had a computational capacity of more than 3 petaflops, making it one of the most powerful computing resources on the planet (http://www.top500.org/). In this F@H architecture, client computers first contact an assignment server (AS), which maintains a list of all active data servers (Fig. 2). The AS responds to the client machine with the IP address of an active data server. The client then sends a request to the data server for a WU. The data server responds with a WU, which the client downloads and processes. Currently, WUs are designed to take $\sim$2 days on most machines so that server load is reduced. When the client has finished processing the WU, it returns the result, and the process begins anew.

Only small modifications to the existing F@H data server were necessary to create a SHARPEN Distributor. The Distributor sends a message to the JobQueue at startup, asking for a configurable number of initial WU. The JobQueue packages these WUs and then sends a reply message to the server indicating that they are ready. As the server sends out work, it requests additional batches of WU from the JobQueue whenever WU are depleted below a specified threshold. When the server receives a completed WU, it writes it to disk. The JobQueue has a thread that periodically checks if outstanding WU have been returned. When it encounters a completed WU, it reads the results into the database and updates the dependency graph. When all Jobs in a master script are completed, the master script returns, completing the communication loop from master script to client code execution (Fig. 2).

WU Validation

In a trusted environment, such as a local cluster, data integrity is not usually a major concern. However, in the F@H system, the work is being performed by a heterogeneous set of client machines that are not under control of the script writers. Therefore, it is generally necessary to check the validity of the returned work. While checksums are used to guard against transmission errors, it is conceivable that by error or manipulation, certain clients could return corrupted work. There is a fundamental tradeoff between guaranteeing data integrity and efficient performance. In the case of SETI@Home, WU were redundantly processed several times by different clients to guarantee the result.[26] Likewise, SHARPEN can easily duplicate entire WU, at a cost of multiplying the computational work. Fortunately, WU that represent a collection of short Jobs can be validated more efficiently. CHOMP Jobs are deterministic and will produce the same results on any machine. Therefore, we can repeat a small number of randomly selected Jobs within a WU to probabilistically ensure the absence of WU corruption. If some Jobs are particularly critical or if their results fall outside the expected range, we can package these Jobs into another WU and send it to another volunteer and compare the results. Reinsertion of Jobs into the queue is also a mechanism to overcome bottlenecks in master script completion that result from clients that return work late or not at all.

User Security

Our top priority when running SHARPEN on the F@H network is the security of the client machines. Scientific code run by the F@H clients comes with a cryptographic signature to ensure security. We do not allow deployed scripts to interact with the client's file system, so no scripts coming from our servers will damage a user's machine. However, for absolute safety, we must also guard against attacks on our network to prevent distribution of malicious code to client machines. To prevent such an attack, we cryptographically sign each of our work units on a dedicated machine that accepts no incoming network connections. To deploy a script onto the network, one must be physically sitting at the terminal of this machine. The F@H client software then uses the cryptographic signature on each work unit to ensure that the work came from our secure server. These stringent security measures ensure that F@H users undertake no additional risk when running SHARPEN work units.

Client-Side Implementation Details

When using SHARPEN with a homogeneous set of machines, assumptions about the operating system, installed Python version, and available system libraries can vastly simplify code distribution. However, on the F@H network, all three of these factors can vary widely. Therefore, it was necessary to package a standalone Python interpreter with our "scientific core" (the code that F@H distributes to allow clients to process our WUs). The scientific core includes CHOMP as well as a main Python script that has been processed by cx_Freeze, a program that transforms Python scripts into standalone executables by packaging the necessary interpreter components with the script (http://python.net/crew/atuining/cx_Freeze/). This allows our code to execute reliably on Linux or Windows systems. We are currently developing solutions along similar lines for Mac OS X machines. The scientific core also includes code specific to

F@H for communication between the F@H Client program (which we use unaltered) and the SHARPEN core (Fig. 2).

In addition to handling client machine inhomogeneity, the scientific core must also contend with unexpected shutdowns. The F@H client is designed to run in the background, so user shutdowns or restarts are expected to occur frequently and without warning. To circumvent this issue, all F@H scientific cores implement "checkpointing" —saving intermediate computation status. With SHARPEN WUs, checkpointing is implemented by saving the results of each Job as it is completed. Upon restarting, the core is then able to read and verify the results from disk, before skipping ahead to the next Job.

Finally, F@H rewards volunteers by distributing points for work done. Many users monitor these points, so it is vital that they be accumulated fairly. In most existing F@H WUs, the amount of computational work is fixed across every WU of a given type, so a representative WU can be run on a benchmark machine, and its point value determined. For certain applications, master scripts could yield SHARPEN WU of fixed length. More generally, it is convenient to group arbitrary Jobs of variable length. In the latter case, we must monitor the computational work expended with each Job to ensure an equitable aggregate work for each WU. Each function in CHOMP that is expected to perform significant computational work accepts a RuntimeStats object as one of its arguments. This object tracks the number of time-consuming operations performed (e.g., atom–atom pairwise energy evaluations, simulated annealing iterations) during the course of the computation. As expected, the number of atom–atom interaction calculations is highly predictive of the total repacking time required by a benchmark processor ($R^2 = 0.99$). Thus, while executing on a client machine, the scientific core can track the total amount of work done after each Job and return the WU after reaching a given threshold. This allows us to ensure that each WU of a given type has the same amount of computational work, and that the points awarded will be fair. Any Jobs left undone by a returning WU are reinserted into the JobQueue.

Results

Tutorial scripts available online demonstrate various applications for CHOMP including pdb scoring, energy decomposition, sidechain repacking, mutation, repacking selected residues, loop rebuilding, and protein–protein docking. To demonstrate both the rapid prototyping and scaling allowed by CHOMP and SHARPEN, respectively, we implemented a simplistic backbone search algorithm over a weekend. Simply, our script specified 200 short Monte Carlo trajectories, where each step consisted of a small ($<10^\circ$) random perturbation to a randomly selected backbone φ or ψ dihedral followed by sidechain optimization. We limited the length of each trajectory with a cutoff for computational work. The script required 27 h to run on a single core of a 2-GHz Xeon. By installing client software on a heterogeneous set of local machines (32 cores), the same script was completed 70 min after deploying the script with the JobQueue. The increased speed and the ease of modifying a python master script make it practical to look for improved performance by varying the algorithm, even for relatively inexperienced programmers.

To benchmark the underlying CHOMP library, we compared rotamer repacking via simulated annealing to optimization in Rosetta (Fig. 1a). Unsurprisingly, we had comparable rotamer repacking accuracy when using the same energy function and rotamer generation scheme: extra χ_1 values for buried residues ($\pm 2\times$ the standard deviation from the Dunbrack rotamer library) and similar extra values for the χ_2 dihedrals of buried aromatic residues.[27] The test set consisted of 1100 nonredundant (20%) high-resolution ($\leq$1.6 Å) protein structures culled from the PDB by the PISCES server.[28] Repacking these targets required 6.3 CPU hours for repeated calls to Rosetta (on a 1.66-GHz Core Duo). When distributed to eight cores via SHARPEN, the aggregate repacking CPU time on the remote cores was 5.6 CPU hours and the entire process required 1.0 CPU hours. Notably, Rosetta repacked the largest proteins twice as rapidly. We expect CHOMP repacking speed to improve when we implement trie pruning.[20]

Discussion

Although no one software package can fill all needs of the computational structural biology community, SHARPEN aims to fulfill several core needs including rapid prototyping and scalable distribution. The underlying library, CHOMP, was designed to encourage increasing automation of algorithm development and parameter tuning. One particularly exciting example of this flexibility is the EnergyHyperGraph structure. This component will allow convenient testing of new algorithms that incorporate nonpairwise effects. One such algorithm is the cluster expansion method described by Keating and coworkers.[29]

One additional benefit of Python integration is the possibility of interactive calculations. First, we use the Remote Python Call (RPyC) library to start a server process that runs CHOMP. Second, we open a socket connection from PyMOL. Because sidechain repacking can be made to be quite rapid, there is potential for a responsive loop-connecting manual manipulation of a protein model in PyMOL with automated optimization and scoring on the back end.

SHARPEN is under active development, with pending support for nonprotein elements, such as nucleic acids and small molecules. Also, future versions of the JobQueue will likely include many enhancements when compared with the initial release. Planned enhancements include prioritization of the Jobs, optimized packaging of Jobs into WU to maximize the overlap for any shared dependencies, and speed up and parallelization of Job serialization. Finally, we aim to provide Distributor programs for non-F@H scenarios including a Distributor for multiple cores on one machine, and an ssh-based Distributor for local Linux clusters.

Conclusions

We have presented a novel software for developing new protein optimization algorithms. CHOMP provides the critical elements

(energy functions and sampling algorithms) needed to prototype applications competitive with leading tools for protein structure prediction and design. The benefits of SHARPEN are simple—developing and debugging a python script on one processor, and then deploying the same script to many computers. The resulting platform provides an example for future Python-based distributed computing efforts, with Folding@Home or stated otherwise.

Acknowledgments

The authors thank Frances H. Arnold for supporting the project; the Jane Coffin Childs foundation and KAUST for postdoctoral support of C.D.S; Mani Chandy for teaching the Distributed Systems course; Jason Paryani for porting the CCD code; Ram Kandasamy for implementing amino acid variants; Phillip Romero and Ben Allen for helpful discussions; Vijay Pande for access to the Folding@Home code; and David Baker for access to the Rosetta code.

References

1. Yu, H. Proc Natl Acad Sci USA 1999, 96, 332.
2. Shen, Y.; Lange, O.; Delaglio, F.; Rossi, P.; Aramini, J. M.; Liu, G.; Eletsky, A.; Wu, Y.; Singarapu, K. K.; Lemak, A.; Ignatchenko, A.; Arrowsmith, C. H.; Szyperski, T.; Montelione, G. T.; Baker, D.; Bax, A. Proc Natl Acad Sci USA 2008, 105, 4685.
3. Kuhlman, B.; Dantas, G.; Ireton, G. C.; Varani, G.; Stoddard, B. L.; Baker, D. Science 2003, 302, 1364.
4. Rothlisberger, D.; Khersonsky, O.; Wollacott, A. M.; Jiang, L.; Dechancie, J.; Betker, J.; Gallaher, J. L.; Althoff, E. A.; Zanghellini, A.; Dym, O.; Albeck, S.; Houk, K. N.; Tawfik, D. S.; Baker, D. Nature 2008, 453, 190.
5. Jiang, L.; Althoff, E. A.; Clemente, F. R.; Doyle, L.; Rothlisberger, D.; Zanghellini, A.; Gallaher, J. L.; Betker, J. L.; Tanaka, F.; Barbas, C. F.; Hilvert, D.; Houk, K. N.; Stoddard, B. L.; Baker, D. Science 2008, 319, 1387.
6. Service, R. F. Science 2008, 321, 784.
7. Samudrala, R.; Levitt, M. Protein Sci 2000, 9, 1399.
8. Zhu, J.; Fan, H.; Periole, X.; Honig, B.; Mark, A. E. Proteins 2008, 72, 1171.
9. Stone, A. J. Science 2008, 321, 787.
10. Schueler-Furman, O. W.; Bradley, C.; Misura, K. M.; Baker, D. Science 2005, 310, 638.
11. Misura, K. M.; Baker, D. Proteins 2005, 59, 15.
12. Jorgensen, W. L.; Maxwell, D. S.; Tirado-Rives, J. J Am Chem Soc 1996, 118, 11225.
13. MacKerell, A. D.; Bashford, D.; Bellott, M.; Dunbrack, R. L.; Evanseck, J. D.; Field, M. J.; Fischer, S.; Gao, J.; Guo, H.; Ha, S. J Phys Chem B 1998, 102, 3586.
14. Wang, J.; Kollman, P. J Comput Chem 2000, 21, 1049.
15. Shirts, M.; Pande, V. S. Science 2000, 290, 1903.
16. Lie-Quan, L.; Jeremy, G. S.; Andrew, L.In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Denver, Colorado, 1999.
17. Veldhuizen, T. L. International Scientific Computing in Object-Oriented Parallel Environments; Santa Fe: New Mexico, 1998.
18. Yakovenko, R. 2008. http://www.language-binding.net/pyplusplus/pyplusplus.html
19. Chowdry, A. B.; Reynolds, K. A.; Hanes, M. S.; Voorhies, M.; Pokala, N.; Handel, T. M. J Comput Chem 2007, 28, 2378.
20. Leaver-Fay, A.; Kuhlman, B.; Snoeyink, J. Rotamer-Pair Energy Calculations Using a Trie Data Structure; Springer Berlin: Heidelberg, 2005.
21. Schnieders, M. J.; Baker, N. A.; Ren, P.; Ponder, J. W. J Chem Phys 2007, 126, 124114.
22. Marshall, S. A.; Vizcarra, C. L.; Mayo, S. L. Protein Sci 2005, 14, 1293.
23. Vizcarra, C. L.; Mayo, S. L. Curr Opin Chem Biol 2005, 9, 622.
24. Canutescu, A. A.; Dunbrack, R. L. Protein Sci 2003, 12, 963.
25. Yuret, D., 2003. http://www.denizyuret.com/bio/ramachandran2/
26. Anderson, D. P.; Cobb, J.; Korpela, E.; Lebofsky, M.; Wethimer, D. Commun ACM 2002, 45, 56.
27. Dunbrack, R. L.; Cohen, F. E. Protein Sci 1997, 6, 1661.
28. Wang, G.; Dunbrack, R. L. Bioinformatics 2003, 19, 1589.
29. Grigoryan, G.; Zhou, F.; Lustig, S. R.; Ceder, G.; Morgan, D.; Keating, A. E. PLoS Comput Biol 2006, 2, 63.