A new load-balancing strategy for the solution of dynamical large-tree-search problems using a hierarchical approach

S. Crivelli
T. Head-Gordon

We describe a new load-balancing strategy, applied here to the protein structure prediction problem, for improving the efficiency of the hierarchical approach when dealing with coarse-grained problems associated with large tree searches. Unlike other load-balancing strategies that reassign load from the heavily loaded processors to the lightly loaded or idle ones, the proposed strategy changes the virtual communication tree among the processors as the computational tree changes. The strategy incurs minimal overhead and is scalable.

1. Introduction

Many applications, such those found in combinatorial optimization graph searching [1], protein threading [2], and protein structure prediction [3–5], can be addressed by searching through a large tree of possible solutions. An often-cited example is the traveling salesman problem [6], in which a salesman must find the shortest path that connects a list of cities. The only way to solve the problem exactly is to try every possible itinerary. However, as the number of cities in the list increases, the size of the tree and consequently the computing time explode exponentially. In some cases, such problems are computationally intractable because the trees they generate become so large that their complete traversal is practically impossible.

In order to solve such problems, one can use techniques such as *branch and bound* that attempt to find the solution (or a good approximation) while minimizing the number of possible solutions investigated [7]. The technique reduces the search space by dynamically pruning those areas that are unlikely to generate a solution. For example, in the traveling salesman problem, the cost associated with visiting the cities decreases monotonically. If a partial solution (a leaf of the tree) has been found with a cost c, all of the partial itineraries with cost greater than c should not be pursued further.

The execution of a branch-and-bound algorithm in parallel further accelerates the search process by decomposing the tree (task) into different subtrees (subtasks) that are assigned to the processors for independent computation [8–12]. These computations are nonuniform because it is impossible to predict *a priori* the size and shape of the tree. An efficient approach for dealing with these problems must be dynamic in order to adapt to the changes occurring during computation. Also, it must be scalable in order to compute more subtrees in parallel when more processors are available. Finally, it must produce partial results quickly and spread those results to the system quickly so as to avoid unnecessary computations on more unproductive branches.

A number of strategies to deal with these programming issues have been proposed [13–24]. The centralized strategy is based on the master/slave approach, in which the master processor keeps the tasks in a queue and hands them to the slaves upon request [25]. Because the master processor takes care of the distribution and coordination of the tasks and gathering of the partial results, load balancing and pruning are straightforward. The strategy is easy to implement, but the master can become a bottleneck as the number of slaves increases. In the distributed approach, each processor is assigned a subtree and handles its own queue of tasks [26]. However, because the subtrees may have different sizes, a dynamic load-balancing strategy becomes necessary to distribute the load that can fluctuate with time across the processors. Also, exchange of the results must be asynchronous because of the uneven size of the subtrees. These

0018-8646/04/$5.00 © 2004 IBM

procedures incur substantial overhead and are not easy to implement [27, 28].

The *hierarchical approach* combines characteristics from the centralized and the distributed approaches [29, 30]. In the hierarchical approach, a master assigns tasks to a selected group of processors called the *supervisors*. Each supervisor further partitions the assigned task into subtasks and assigns the subtasks to its corresponding group of *workers*. The supervisors gather the results from the workers and send them to the *master*. To keep the load balanced, the master and the supervisors assign tasks upon request. The hierarchical approach presents a scalable alternative to the centralized one while keeping its simplicity. However, we discuss a type of problem in which the hierarchical approach requires additional load balancing in order to improve the efficiency of the parallel implementation. Such problems occur when the master runs out of tasks (because it reaches the last stage or because the algorithm requires synchronization every number of iterations) and it cannot assign work to the supervisors requesting more. Consequently, those supervisors and their corresponding workers remain idle waiting for the busy ones to finish their work. Because the size of the tasks is relatively large at the master–supervisor level since they are coarsely partitioned at the higher level of processors in the hierarchy, the strategy incurs a great deal of idle time.

In this paper, we present a new load-balancing technique that improves the efficiency of the hierarchical approach when dealing with coarse-grained problems associated with large-tree searches. Unlike other load-balancing techniques that reassign work from the heavily loaded processors to the lightly loaded or idle ones, the proposed technique reassigns workers from the idle supervisors to the busy ones so that they can finish their job more rapidly. The technique incurs minimal overhead and it is scalable.

This paper is organized as follows. In Section 2 we present an example of the class of tree-search applications we want to address—the protein structure prediction problem. In Section 3 we describe the hierarchical approach for parallelization of such problems. In Section 4 we present a load-balancing strategy for improving the efficiency of the hierarchical approach when it is used in the context of coarse-grained problems, and discuss some performance results. Finally, in Section 6 we provide some conclusions.

2. A case study: The protein structure prediction problem

Proteins are molecules composed of a string of amino acids that fold into complex three-dimensional topologies (tertiary structure). The amino acid sequence and its aqueous environment give rise to energetic forces which guide the formation of the tertiary structure of the protein that in turn determines its function. Assuming that the tertiary structure occurs at the global minimum of the free-energy surface, the problem can be formulated as a global minimization problem. However, finding the global minimum of the energy surface is a challenging task, because the energy surface presents many low-lying local minima whose number is assumed to grow exponentially with the number of amino acids in the sequence [31, 32].

The potential energy function used is the AMBER (Assisted Model Building with Energy Refinement) molecular mechanical force field [33], which is based upon the Cartesian coordinates of the n atoms of the protein. The position of the atoms may be described by parameters b_i, the distance between the ith atom and a designated neighboring atom, θ_i, the bond angle formed by a sequence of three bonded atoms, and χ, a dihedral angle formed by a sequence of four bonded atoms. Typically, force fields represent bonds and angles as harmonic distortions, dihedrals by a truncated Fourier series, and nonbonded interactions via Lennard-Jones and Coulomb's law for electrostatic interactions between point charges. The AMBER function is defined as

$$E_{\mathrm{AMBER}} = \sum_{i}^{\#bonds} k_b(b_i - b_0)^2 + \sum_{i}^{\#angles} k_\theta(\theta_i - \theta_0)^2 + \sum_{i}^{\#dihedrals} k_\chi[1 + \cos(n\chi + \delta)] + \sum_{i}^{\#atoms} \sum_{i<j}^{\#atoms} \left\{ \frac{q_i q_j}{r_{ij}} + \varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - 2\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \right\}.$$

The first three terms represent the bonded interactions, and the final terms represent the nonbonded interactions. The nonbonded interactions occur between every pair (i, j) of atoms and depend on the distance r_{ij} between the pair of atoms, and on their charges q_i and q_j. The terms with zero subscripts pertain to equilibrium values.

To introduce a stabilizing force for forming hydrophobic cores, a solvation term, $E_{\mathrm{solvation}}$, has been added that accounts for hydrophobic effects [34]. It is represented by a sum of M Gaussians, viz.,

$$E_{\mathrm{solvation}} = \sum_{i}^{N_c} \sum_{j}^{N_c} \sum_{k}^{M} h_k \exp\left[-\left(\frac{r_{ij} - c_k}{w_k} \right)^2 \right],$$

where the sums over i and j are over the aliphatic carbon centers, and each of the Gaussians is parameterized by position c_k, depth h_k, and width w_k.

Our protein structure prediction method consists of two phases [3]. Phase 1 generates good initial configurations that are low-energy local minima. Phase 2 improves the initial structures by performing small-scale global minimizations in various subspaces of the space of torsion angles of the protein. The small-scale global optimizations use the stochastic method of Rinnooy-Kan and Timmer [35], which is general in the sense that subspaces of arbitrary dimension can be explored. However, in practice, the amount of work required to reach the theoretical guarantee is prohibitive for large subspaces. Thus, we usually select a subspace size of 6 to 12 variables. Our method selects a local minimizer from a list of minimizers and a subset of coordinates and performs a small-scale global minimization on that subset using the selected coordinates as variables while keeping the remaining ones temporarily fixed at their current values. The small-scale optimization produces a number of local minimizers in the subspace of chosen coordinates. Configurations are ranked in terms of their energy value, and the best refers to the lowest energy value. A number of those conformations with low energy values are selected for local minimizations on the full variable space. The new minimizers obtained from the local minimizations are merged into the current list ordered by energy value. The process repeats iteratively, as follows:

(1) For some number of iterations:
   (a) Select a configuration and small subset of parameters to improve:
   Select a local minimizer to improve from the list of full-dimensional local minimizers. Select a subset of variables.
   (b) Small subproblem global optimization:
   Apply a global optimization algorithm to the energy of the selected configuration with only the selected parameters as variables.
   (c) Full-dimensional local minimization:
   Apply a local minimization procedure, with all of the parameters as variables, to the lowest-energy configurations that resulted from step (b), and merge the new local minimizers into the list of local minimizers.

(2) After a number of iterations:
   Cluster local minima and test for convergence:
   Cluster the list of local minimizers via pairwise RMSD, and if the stopping criteria have not been met, repeat the iterative procedure with a new list of local minimizers containing the lowest-energy minimizer from each cluster.

After a number of iterations, the list of local minimizers are clustered via pairwise root mean squared deviation (RMSD) evaluations and ordered by energy value within each cluster. A new list is formed with the lowest-energy minimizer from each cluster, and the iterative procedure starts again with this new list. The algorithm is considered to have converged when it finds no change in cluster number and no further lowering of energy in the lowest-energy cluster.

Computationally, the problem can be viewed as a search through a huge tree of possible solutions for which the root of the tree corresponds to the primary sequence of amino acids; a subtree consists of an initial minimizer and all of the minimizers generated from it by applying a global optimization on a small subspace followed by a local minimization over the entire space; and the leaves correspond to the local minima. Clustering corresponds to a synchronization point, and it has the effect of pruning the tree of possible configurations. Researchers have estimated that to solve the protein structure prediction problem by traversing the entire tree would take 10^{27} years for an average-sized protein of 100 amino acids [36]!

In order to make the search through this tree possible, our method creates configurations that are *low-energy local minima* and uses them as the *roots* of the tree. Thus, the tree that must be traversed is significantly smaller than the one that begins with the extended sequence of amino acids. However, since the amount of computational time needed for proteins of realistic size remains high, the use of parallel computers becomes a necessity. In fact, our current runs on the IBM SP* computer at the NERSC (National Energy Research Scientific Computing Center) facility at Berkeley, California, require the use of many processors for many hours and even weeks to converge. Our method is highly but not straightforwardly parallelizable. The main problems are how to partition the load and keep it balanced, considering that the work is dynamically generated and its computational time unknown, and how to efficiently gather partial results generated without jeopardizing the scalability of the code. In the next sections we describe an approach to deal with these issues.

3. The hierarchical approach

We assume a task-parallel model of computation in which a task corresponds to the execution of a node of the tree. Each task may generate other tasks. Because the tree is usually very large and traversing it completely is computationally impossible, it is important to use partial results generated to guide the search to the most promising areas.

A hierarchical approach provides such complex applications as tree searches with a natural programming paradigm for their implementation on large systems. It assigns subtrees to different groups of processors, allowing one to maintain the information updated without incurring

significant communication overhead. The system is divided into three different categories of processors and two levels of work, each dealing with different types of tasks and granularities. In the first category, a *master processor* keeps the global information updated, assigns coarsely divided tasks to a selected group of processors called the *supervisors*, and collects the partial results. In addressing the protein structure prediction problem, the master has a list of initial configurations created in Phase 1 and assigns one configuration to each supervisor for global minimization in a subspace. The master maintains the list of updated local minimizers and distributes them to the supervisors upon request. Each supervisor partitions the assigned task into subtasks and assigns the subtasks to its corresponding group of *workers*. In the protein example, the global optimization task involves different subtasks such as sampling over the parameter space, selecting sample points to be starting points for local minimizations, and performing the local minimizations themselves. The supervisors assign those subtasks to the workers and then gather the results. Also, the supervisors can participate in the execution of the subtasks. When the assigned task is complete, the supervisors send a number of the best energy structures found to the master. The supervisors control the work of their workers and manage the local information in their group but have no knowledge about the computation in other groups. Finally, in the third category are the workers that perform smaller subtasks assigned by the supervisors and report the results to their supervisors. To keep the load balanced, the master and the supervisors assign tasks upon request. Synchronization at the end of each task is avoided because the sizes of the tasks vary.

The hierarchical approach presents a scalable alternative to the centralized approach while keeping its simplicity. In fact, as the number of processors grows, new layers can be added to the hierarchy to prevent the supervisors and/or the master from becoming communication bottlenecks. However, some synchronization may be necessary during the computation, either to gather the final results at the end of the computation or to gather partial results for every number of iterations. In those cases, upon reaching a specific number of iterations, the master stops assigning work. Therefore, because the task times are unpredictable and usually long, some supervisors and their workers may be idle until other supervisors and their workers complete their jobs. The idle time the supervisors and their workers accumulate after running a number of such synchronization points becomes a considerable percentage of the total execution time. In the next section, we present a load-balancing strategy that minimizes this idle time.

4. A new load-balancing strategy

We propose a load-balancing strategy that reassigns workers to new supervisors as they become idle. Thus, instead of reassigning tasks, the proposed strategy reassigns workers from the idle supervisors to the busy ones. Our approach changes dependencies in the hierarchy rather than transferring load, so that busy supervisors will have a larger number of workers assigned to them to complete their tasks in less time. The hierarchical approach makes it easy and cost-efficient to reassign processors to new supervisors by letting the idle supervisors reassign their workers.

When the master reaches a synchronization point, it sends a message to the supervisors that contact it in search of more work. The message that the master sends to the idle supervisors contains a list of the supervisors that are still busy. Upon receiving this message, the idle supervisor splits its workers among the busy supervisors. The busy supervisors may accept the offer or not, depending on the number of workers they already have and the amount of work to do. To avoid a situation in which a worker is assigned to a supervisor that becomes idle before accepting it, the former supervisor waits for an acknowledgment message from its worker before deleting it from its list. Another undesirable situation may occur when only one supervisor remains busy. Assigning all of the workers to the busy supervisor will have the same undesirable consequences as using the master/slave approach, with the busy supervisor becoming a bottleneck. To avoid such a situation, a maximum limit is set on the number of idle workers a busy supervisor can accept. This number usually depends on the computer and the total number of processors used. If the busy supervisor is offered more processors than the maximum number it can handle, the supervisor measures the amount of work to be done (this may be measured by the number of tasks in the queue). If this amount appears to be relatively large, the busy supervisor may split its task into two subtasks and assign one subtask to the central scheduler so that this task can be subdivided among the idle supervisors. At this point, the supervisor may or may not release some of its extra workers to their previous supervisors depending on whether or not they have been assigned jobs. **Figure 1** outlines the hierarchical approach using the proposed load-balancing strategy.

We ran two experiments on the IBM SP computer at NERSC in order to examine the effectiveness of the load-balancing strategy. In the first experiment, we compared the hierarchical approach without and with the new load-balancing strategy using 19, 34, 68, and 127 processors. We designated the approach that does not use the new load-balancing strategy as the *static* approach, while the one that uses the new load-balancing strategy was designated as the *dynamic* approach. In all cases, we kept

```
The master:
    while (iteration does not reach synchronization point)
        Select a task from the queue of tasks.
        Assign the task to an idle supervisor.
        Put the new tasks generated by the supervisors in the queue.
        Update the queue of tasks according to the results found so far (pruning).
    end while
    if (synchronization point is reached and all supervisors are finished)
        Send STOP message to all supervisors.
    else
        Send IDLE message to each supervisor that requests work.
    end if

The supervisors:
    while (receiving new task from master)
        Perform the task.
        Divide the task into smaller subtasks.
        Schedule the workers to perform the subtasks.
        Gather results from workers.
        Send results to the master.
    end while
    if (IDLE message is received from the master)
        Reassign workers.
        Update list of workers.
    else if (STOP message is received from the master)
        Send STOP message to workers.
    end if

The workers:
    while (receiving new tasks from supervisor)
        Perform subtask.
        Send results to supervisor.
        if (new supervisor is assigned)
            Update supervisor ID.
    end while
```

Figure 1

Hierarchical algorithm using a dynamic approach.

the amount of work fixed to a certain number of iterations (i.e., the number of nodes of the tree that were expanded was fixed). For those runs, we kept the starting number of workers per supervisor fixed to eight and varied the number of supervisors. Thus, as a function of processors, we used 2, 4, 7, and 14 supervisors, respectively. Note that if the dynamic algorithm is used, the number of workers varies dynamically because they are reassigned during the computation to keep the work balanced.

Figure 2 compares the time required to complete 14 iterations using the static and dynamic algorithms for the analysis of 1P0U, a protein with 70 amino acids. Although the times for ten processors, i.e., eight workers, one supervisor, and one master, were the same (dynamic reallocation of workers is not possible with a single supervisor), the dynamic approach shows significant gains if larger numbers of processors are used. Note that synchronization points for these tests occurred only at the end of the computation. Thus, more significant differences between the static and dynamic approaches should be expected when further synchronization points are included in the computation. This experiment also indicated good scalability of the dynamic approach, with execution times decreasing with the number of processors until reaching a certain limit that could not be improved without further refinement of the granularity of tasks. Also, note that Figure 2 compares execution times obtained by using an already efficient hierarchical approach that keeps the processors busy *most* of the time with an even more efficient approach that keeps processors busy *all* of the time. Thus, although the overall speedup shown does not appear to be impressive, the actual gains in the total execution times were significant.

The second experiment further explored the scalability of our algorithm at the supervisor level by increasing the number of supervisors as well as the number of minimizers expanded for each run while keeping the initial number of workers per supervisor fixed at eight. Thus, unlike the previous experiment, for every set of one supervisor and eight workers added, we increased the size of the tree to

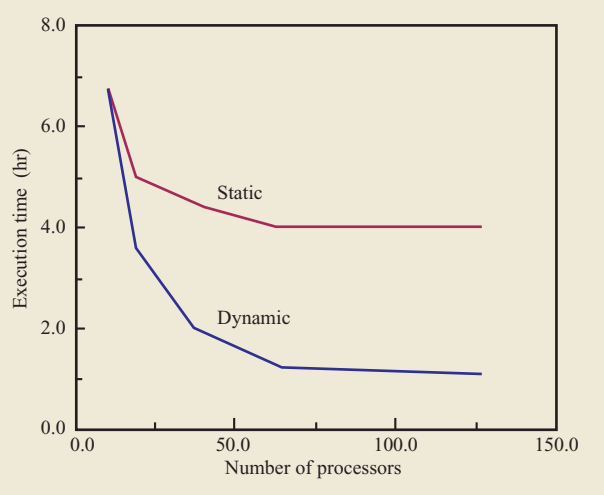

Figure 2

Execution times for the static and dynamic approaches using different numbers of processors. Adapted from [5] with permission.

Table 1 Execution times for increasing numbers of processors and nodes.

No. of supervisors	*No. of processors*	*No. of nodes expanded*	*Execution time* (hr)
14	127	27	3:52
18	163	35	3:53
22	199	43	3:51
26	235	51	3:54
30	271	59	3:53
34	307	67	3:56
38	343	75	3:55
42	379	83	3:57

search. Our goal was to show that the algorithm allows one to explore more of the tree by adding more processors without losing performance.

We started with 14 supervisors with eight workers each (i.e., 127 processors). For each run, we incremented the number of supervisors by 4. Thus, the next run consisted of 18 supervisors (i.e., 163 processors); the following run consisted of 22 supervisors (i.e., 199 processors), etc., until we reached a total of 42 supervisors (three times the original number of supervisors used in this test). The execution times for all of the runs are shown in **Table 1**. The times in all cases remained within five minutes of the initial time.

5. Conclusions

We have discussed the parallel implementation of a type of tree-search problem that is extremely difficult to solve. Its solution requires a branch and bound algorithm to prune the unproductive branches of the tree. The hierarchical approach assigns subtrees to different groups of processors, allowing the work to be partitioned efficiently and information to be updated without incurring significant communication overhead. The system can be divided initially into three different categories of processors and two levels of work, each dealing with different types of tasks and granularities. The approach is scalable. In fact, as the number of processors increases, new categories can be added to the hierarchy to avoid communication bottlenecks.

We have implemented an efficient load-balancing strategy that reassigns idle workers to busy supervisors, thus changing the virtual communication tree among the processors as the computational tree changes. This is highly efficient because rather than moving tasks and incurring significant bookkeeping overhead, the idle supervisors communicate only their workers' IDs to the busy ones. The busy supervisors, in turn, need only update their table of workers. This is an important feature because tree searches usually require efficient strategies that allow the code to quickly finish the calculation of the nodes of the tree in order to more rapidly prune the unproductive branches—to avoid unnecessary computation and to focus on the most promising areas of the tree.

In order for this load-balancing strategy to be highly effective, the granularity of the tasks assigned to the supervisors must be large. In our protein structure prediction application, a task consisted of taking a configuration and a subspace and performing a global optimization in that subspace using the configuration as a starting point. The global optimization was followed by full-dimensional local minimizations performed on a number of the minimizers found. Depending on the size of the protein, we adjusted some parameters, such as the size of the subspace and the number of iterations of the global optimization algorithm, so that the tasks required hours to complete. Because the completion times for such tasks may differ substantially, the chances of having all of the supervisors complete their tasks concurrently are small, and the gains for using the load-balancing strategy may be very high.

Acknowledgments

One of us (S.C.) gratefully acknowledges support through the NSF/ITR program, Grant No. NSF6022533. T.H.-G. acknowledges support through the NSF/ITR program, Grant No. NSF6022533, and the U.S. Department of Energy, Contract No. DE-AC-03-76SF00098. We thank the National Energy Research Scientific Computing Center (NERSC) for significant computer resources. T.H.-G. also thanks the IBM SUR grant program.

References

1. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, New York, 1979.
2. R. H. Lathrop, "The Protein Threading Problem with Sequence Amino Acid Interaction Preferences Is NP-Complete," *Protein Eng.* **7,** 9–14 (1994).
3. S. Crivelli, E. Eskow, B. Bader, V. Lamberti, R. Byrd, R. Schnabel, and T. Head-Gordon, "A Physical Approach to Protein Structure Prediction," *Biophys. J.* **82,** 36–49 (2002).
4. S. Crivelli, T. M. Philip, R. Byrd, E. Eskow, R. Schnabel, R. C. Yu, and T. Head-Gordon, "A Global Optimization Strategy for Predicting Protein Tertiary Structure: α-Helical Proteins," *Comput. & Chem.* **24,** 489–497 (2000).
5. S. Crivelli, T. Head-Gordon, R. Byrd, E. Eskow, and R. Schnabel, "A Hierarchical Approach for Parallelization of a Global Optimization Method for Protein Structure Prediction," *Proceedings of EuroPar'99*, P. Amestoy, P. Berger, M. Dayde, I. Duff, V. Fraysse, L. Giraud, and D. Ruiz, Eds., Lecture Notes in Computer Science Series, Springer-Verlag, Berlin, 1999, pp. 578–585.
6. M. Bellmore and G. Nemhauser, "The Traveling Salesman Problem: A Survey," *Oper. Res.* **16,** 538–558 (1968).
7. L. G. Mitten, "Branch-and-Bound Methods: General Formulation and Properties," *Oper. Res.* **18,** 24–34 (1970).
8. B. Bourbeau, T. G. Crainic, and B. Gendron, "Branch-and-Bound Parallelization Strategies Applied to a Depot Location and Container Fleet Management Problem," *Parallel Comput.* **26,** 27–46 (2000).
9. S. Tschoeke, R. Lueling, and B. Monien, "Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a 1024-Processor Network," *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, IEEE Computer Society, 1999.
10. V. Rao and V. Kumar, "Parallel Depth First Search. Part I. Implementation," *Int. J. Parallel Program.* **16,** No. 6, 479–499 (1987).
11. V. Kumar and V. Rao, "Parallel Depth First Search. Part II. Analysis," *Int. J. Parallel Program.* **16,** No. 6, 501–519 (1987).
12. B. Gendron and T. G. Crainic, "Parallel Branch-and-Bound Algorithms: Survey and Synthesis," *Oper. Res.* **42,** No. 6, 1042–1066 (1994).
13. Z. Hafidi, E.-G. Talbi, and G. Goncalves, "Load Balancing and Parallel Tree Search: The MPIDA* Algorithm," *Proceedings of ParCo'95, Parallel Computing: State-of-the-Art and Perspectives*, E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, Eds., Elsevier Science, St. Louis, MO, 1996, pp. 93–100.
14. C.-Z. Xu, B. Monien, R. Luling, and F. C. M. Lau, "An Analytical Comparison of Nearest Neighbor Algorithms for Load Balancing in Parallel Computers," *Proceedings of the International Parallel Processing Symposium*, *IPPS95*, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 472–479.
15. M. A. Bhandarkar, R. K. Brunner, and L. V. Kale, "*Run-Time Support for Adaptive Load Balancing*"; see *http://charm.cs.uiuc.edu/*.
16. S. Chakrabarti, A. Ranade, and K. Yelick, "Randomized Load Balancing for Tree-Structured Computation," *Proceedings of the IEEE Scalable High-Performance Computing Conference*, Knoxville, TN, IEEE Computer Society, 1994, pp. 666–673.
17. A. Y. Grama, V. Kumar, and P. Pardalos, "Parallel Processing of Discrete Optimization Problems: A Survey," *Encyclopedia of Microcomputers*, Vol. 13, pp. 129–147, John Wiley and Sons, Hoboken, NJ, 1993.
18. A. B. Sinha and L. V. Kale, "A Load Balancing Strategy for Prioritized Execution of Tasks," *Proceedings of the 7th International Parallel Processing Symposium*, Newport Beach, CA, 1993, pp. 230–237.
19. W. Shu and L. V. Kale, "Chare–Kernel—A Runtime Support System for Parallel Computations," *J. Parallel & Distr. Comput.* **11,** 198–211 (1991).
20. G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel & Distr. Comput.* **7,** 279–301 (1989).
21. S. Arvindam, V. Kumar, and V. Rao, "Floorplan Optimization on Multiprocessors," *Proceedings of the International Conference on Computer Design*, IEEE Computer Society, 1989, pp. 109–114.
22. T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.* **14,** No. 2, 141–154 (1998).
23. R. A. Finkel, "Large-Grain Parallelism—Three Case Studies," *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds., MIT Press, Cambridge, MA, 1987, pp. 21–63.
24. R. Finkel and U. Manber, "DIB—A Distributed Implementation of Backtracking," *ACM Trans. Program. Lang. & Syst.* **9,** No. 2, 235–256 (1987).
25. H.-C. Lin and C. S. Raghavendra, "A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC)," *IEEE Trans. Software Eng.* **18,** No. 2, 148–158 (1992).
26. D. L. Eager, E. D. Lasowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Perform. Eval.* **6,** 53–68 (1986).
27. S. A. Crivelli and E. R. Jessup, "The PMESC Programming Library for Distributed-Memory MIMD Computers," *J. Parallel & Distr. Comput.* **57,** No. 3, 295–321 (1999).
28. S. A. Crivelli and E. R. Jessup, "Task Parallelism: What a Tool Can Provide and What Should Be Left to the User," *Proceedings of EuroPar'96*, P. Amestoy, P. Berger, M. Dayde, I. Duff, V. Fraysse, L. Giraud, and D. Ruiz, Eds., Lecture Notes in Computer Science Series, Springer-Verlag, Heidelberg, 1999, pp. 151–154.
29. R. Pollak, "A Hierarchical Load Balancing Environment for Parallel and Distributed Supercomputer," *Proceedings of the International Symposium on Parallel and Distributed Supercomputing (PDSC'95)*, Fukuoka, Japan, 1995; see *www.informatik.uni-stuttgart.de/ipvr/as/projekte/palaber/Palaber.html#HDR1/*.
30. I. Ahmad, A. Ghafoor, and G. Fox, "Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers," *J. Parallel & Distr. Comput.* **20,** 317–329 (1994).
31. F. Eisenhaber, B. Persson, and P. Argos, "Protein Structure Prediction: Recognition of Primary, Secondary, and Tertiary Structural Features from Amino Acid Sequence," *Crit. Rev. Biochem. Mol. Biol.* **30,** 1–94 (1995).
32. M. Vasquez, G. Nemethy, and H. Scheraga, "Conformational Energy Calculations on Polypeptides and Proteins," *Chem. Rev.* **94,** 2183–2239 (1994).
33. W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz, D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman, "A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules," *J. Amer. Chem. Soc.* **117,** 5179–5197 (1995).
34. G. Hura, J. M. Sorenson, R. M. Glaeser, and T. Head-Gordon, "Solution X-Ray Scattering as a Probe of Hydration-Dependent Structuring of Aqueous Solutions," *Perspect. Drug Discov. Des.* **17,** 97–118 (1999).

35. A. Rinnooy-Kan and G. Timmer, "Stochastic Methods for Global Optimization," *Amer. J. Math. & Manage. Sci.* **4,** 7–40 (1984).
36. C. Levinthal, "Are There Pathways to Protein Folding?," *J. Chem. Phys.* **65,** 44–45 (1968).

Received September 12, 2003; accepted for publication November 17, 2003

Silvia Crivelli *Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, California 94720 (SNCrivelli@lbl.gov).* Dr. Crivelli received a B.S. degree in applied mathematics in 1979 from the National University in Argentina, and M.S. and Ph.D. degrees in computer science from the University of Colorado, Boulder, in 1991 and 1995, respectively. From 1997 to 2001, she was a Postdoctoral Fellow in the Physical Biosciences Division of the Lawrence Berkeley National Laboratory (LBNL). She was a Computing System Engineer III in the Computational Research Division until 2002. Dr. Crivelli is currently a Staff Scientist at LBNL. During all of her years at LBNL, she has worked on the development of computational methodologies for protein structure prediction. She is the leader of the group that developed ProteinShop, an interactive visualization and manipulation tool for protein structure prediction and modeling. Dr. Crivelli has published more than twenty papers in the areas of parallel and scientific computing, and protein structure prediction. One of her recent publications won the Best Paper Award at the IEEE Visualization 2003 Conference in Seattle. Dr. Crivelli is a member of SIAM and ACM.

Teresa Head-Gordon *Department of Bioengineering, University of California, Berkeley, California 94720 (TLHead-Gordon@lbl.gov).* Professor Head-Gordon received a B.S. degree in chemistry from Case Western Reserve University in 1983 and a Ph.D. degree in theoretical chemistry from Carnegie Mellon University in 1989. She was a Postdoctoral Member of the Technical Staff at AT&T Bell Laboratories from 1990 to 1992 and a Staff Scientist at the Lawrence Berkeley National Laboratory (LBNL) from 1992 to 2001. She is currently a professor in the Department of Bioengineering at the University of California at Berkeley, a faculty staff scientist and Department Head of the Computational Structure Group in the Physical Biosciences Division at LBNL, Editor of the *Biophysical Journal*, and an Editorial Advisory Board member of the *Journal of Computational Chemistry*. Her research program encompasses the development of general computational methodologies and modeling applied to biology in the areas of water and aqueous hydration, protein folding, and structure prediction; and there is a growing experimental component to her laboratory in these areas. Professor Head-Gordon has published more than fifty papers on theoretical, computational, and experimental biology and chemistry.