Heuristic Static Load-Balancing Algorithm Applied to the Fragment Molecular Orbital Method

Yuri Alexeev*, Ashutosh Mahajan*, Sven Leyffer, Graham Fletcher
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
{yuri,fletcher}@alcf.anl.gov
{mahajan,leyffer}@mcs.anl.gov

Dmitri G. Fedorov
National Institute of Advanced Industrial Science and Technology
Central 2, Umezono 1-1-1
Tsukuba 305-8568, Japan
d.g.fedorov@aist.go.jp

Abstract

In the era of petascale supercomputing, the importance of load balancing is crucial. Although dynamic load balancing is widespread, it is increasingly difficult to implement effectively with thousands of processors or more, prompting a second look at static load-balancing techniques even though the optimal allocation of tasks to processors is an NP-hard problem. We propose a heuristic static load-balancing algorithm, employing fitted benchmarking data, as an alternative to dynamic load balancing. The problem of allocating CPU cores to tasks is formulated as a mixed-integer nonlinear optimization problem, which is solved by using an optimization solver. On 163,840 cores of Blue Gene/P, we achieved a parallel efficiency of 80% for an execution of the fragment molecular orbital method applied to model protein-ligand complexes quantum-mechanically. The obtained allocation is shown to outperform dynamic load balancing by at least a factor of 2, thus motivating the use of this approach on other coarse-grained applications.

Keywords: Dynamic load balancing, static load balancing, heuristic algorithm, quantum chemistry, GAMESS, fragment molecular orbitals, FMO, optimization, MINLP, protein-ligand complex

I. INTRODUCTION

Achieving an even load balance is a key issue in parallel computing, and increasingly so as we enter the petascale supercomputing era. By Amdahl's law, the scalable component of the total wall time shrinks as the numbers of processors increases, while the load imbalance, together with the constant sequential component, acts to retard the scalability. Although parallelization of sequential code often requires rewriting the code, adopting an efficient load-balancing scheme can be a simple and effective way to boost scalability and performance.

Dynamic load balancing (DLB) and static load balancing (SLB) are two broad classes of load-balancing algorithms. Whereas SLB relies on previously obtained knowledge (for example benchmarking data), or consistent task sizes, DLB dynamically assigns jobs to processors during code execution. Many variations on SLB and DLB algorithms adapted for specific applications have been reported [1-4], using different techniques such as random stealing [5, 6], simulated annealing [7], recursive bisection methods [8-10], space-filling curve partitioning [11-14], and graph partitioning [15-21]. SLB is usually simple to implement and has negligible overhead, making it suitable for "fine-grained" parallelism consisting of many small tasks. However, if the application involves much larger tasks of diverse sizes, as is often the case with "coarse-grained" parallelism, DLB may be preferred. Since many applications naturally involve widely differing task sizes, DLB algorithms have become widespread. Indeed, as the number of available processors increases (for instance, when moving from a PC cluster environment to a large modern supercomputer), many applications find it advantageous to allocate work in larger chunks in the interest of reducing overhead.

In the shift from fine- to coarse-grained parallelism, DLB may seem to be the natural choice. However, the DLB schemes suitable for a PC cluster often perform poorly on many thousands of processors, prompting the search for load-balancing paradigms that can handle diverse task sizes with minimal overhead. One possibility is to adapt SLB techniques to pre-allocate tasks more effectively by drawing on a deeper understanding of the application at hand. However, the optimal static mapping of jobs to more than two processors is, in general, an NP-hard problem [22, 23]. Nevertheless, such SLB methods have been successfully applied to a large number of applications [1]. The success of applying SLB often relies on predictive models that can also depend on the accuracy of input data from a benchmarking study; both factors can be systematically improved. Furthermore, if the calculation is iterative, the lack of a dynamic means of allocating tasks can be accounted for in SLB schemes by redistributing work between iterations.

*YA and AM contributed equally to this work

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

In this paper we examine parallel load-balancing schemes applied to a quantum chemistry method - the fragment molecular orbital (FMO) method implemented in the quantum chemistry code GAMESS [24, 25] - on the Blue Gene/P [26] supercomputer at Argonne National Laboratory. While FMO has been shown before to achieve superior scalability for fine-grained systems such as water clusters [27], we aim to improve the scalability and efficiency of coarse-grained systems, such as proteins. We analyze why FMO's current DLB scheme is not optimal and propose an SLB alternative.

A key feature of our SLB method is the formulation of a mixed-integer nonlinear optimization (MINLP) problem to model the allocation of processing cores to tasks. The MINLP approach provides great flexibility in modeling the allocation problem realistically. Using nonlinear functions, we can capture complex relationships between running time and the number of processors. At the same time, we can impose integer restrictions on certain variables (e.g., number of processors). The solution to MINLP can then be directly used for load balancing in the GAMESS application. To solve the MINLP arising in our procedure, we use MINOTAUR [28], a freely available MINLP toolkit. It offers several algorithms for solving general MINLPs, and can be easily called from different interfaces. Our MINLP formulation requires a few parameters to accurately model the performance, obtained by collecting benchmarking data about the application and solving a fitting problem. We describe these methods in Section IV. Our experiments demonstrate that both the fitting problem and the MINLP problem can be solved quickly on a single core, and the resulting allocations lead to significant savings in the run time of the GAMESS application.

The DLB and SLB comparison is done on the receptor-ligand system – Aurora-A kinase and inhibitor shown in Fig. 1 (A). We demonstrate the performance of our method on a large protein system (see Fig. 1 (C)) using all 40 racks (163,840 cores) on Argonne's Blue Gene/P.

II. FRAGMENT MOLECULAR ORBITAL METHOD

Ab initio quantum chemistry methods are, in principle, applicable to any molecular system though the computational cost increases steeply with the system size. Even the simplest restricted Hartree-Fock (RHF) method scales approximately cubically with the system size. There are ongoing efforts to reduce the scaling of quantum-mechanical (QM) methods [29, 30] and parallelize them efficiently [31-38]. See, for example, the linearly scaling method developed by Challacombe and Schwegler [39], and the adaptive multiresolution method developed by Harrison, et al. [40]. Alternatively, fragment-based methods [41, 42] (which divide the system into fragments) can dramatically reduce computational cost, increase stability of calculations, and provide additional information on properties of fragments and their interactions. Algorithmically, fragmentation results in division of one large calculation into many small and nearly independent subtasks or loosely coupled ensemble calculations. As a result, fragmentation methods are efficient for performing quantum mechanical calculations on supercomputers.

One of the fragment-based methods is the FMO method [43], which has been interfaced with many QM methods and successfully applied to chemical systems such as proteins, DNA, silicon nanowires, and ionic liquids [44]. FMO has been implemented in GAMESS [45] and parallelized with the generalized distributed data interface (GDDI) [46-48].

In FMO, each fragment electronic state is computed in the potential exerted by all the others. Starting from an initial guess, fragment calculations that update the embedding potential are iterated until self-consistency is achieved. Subsequently, fragment pair calculations are performed in the embedding potential. The fragmentation, which is usually chemically motivated for rapid convergence, fixes the parallel domain decomposition at the outset.

The basic FMO equation has the form

$$E = \sum_{i=1}^{F} E_i + \sum_{\substack{(i,j):i=1\ldots F,\\ j=1\ldots F, i>j}} \left(E_{ij} - E_i - E_j\right), \qquad (1)$$

where F is the number of fragments and E_i, E_{ij} are the energies of fragment (monomer) i and fragment pair (dimer) ij, respectively. These energies are assembled according to Eq. (1) to give the total energy and other properties of the system.

GDDI is a two-level parallelization scheme, which can be thought of as coarse-grained parallelism since all CPU cores are divided into a few groups. At the higher intergroup level the load balancing is accomplished by assigning fragments or fragment pairs to GDDI groups. At the lower intragroup level, the load balancing is accomplished by assigning some integral workload to individual CPU cores within a group. Various implementations of GDDI exist, of which the main ones are (1) UNIX socket-based, whereby each CPU core runs a GAMESS process and communicates over TCP/IP via sockets, and (2) MPI-based, where MPI communicators are created for groups. This two-level parallelization has been successful in obtaining up to about 90% of the perfect scalability (i.e., 90-fold speedup on a 100-fold increase in the number of cores) [48] on PC clusters with 128 CPUs connected by a low-end network (FastEthernet). FMO/GDDI has subsequently been used on larger computer systems such as the AIST supercluster [49]. More recently, FMO/GDDI has been successfully run for large water clusters on 131,072 CPU cores on Argonne's Blue Gene/P [27]. Our current MPI-based implementation on Blue Gene/P comprises compute- and data-server process pairs, so that half of all CPU cores are used for QM calculations, while the other half handle communications and distributed memory processing. We reported wall-clock timings for GAMESS runs on the total number of CPU cores.

In this paper we apply FMO to two protein-ligand systems. All benchmarking and tuning of both DLB and HSLB schemes have been done on Aurora-A kinase with inhibitor phthalazinone shown on Fig. 1 (A). Aurora kinases

are essential for cell proliferation and a major target in designing new anti-cancer drugs. The system is of moderate size: 155 fragments (154 amino acids and 1 ligand), with the total number of atoms equal to 2,604, computed at the RHF level of theory with the 6-31G* basis set. The production run was done for ovine COX-1 complexed with ibuprofen shown on Fig. 1 (B). The system consists of 17,767 atoms divided into 1,093 fragments. For this work, we used the distributed memory storage of fragment densities [27]. Various tasks, including the fragmentation of proteins, structure checking, the generation of GAMESS input for FMO calculations, and the visualization of results, were performed by the FMOtools suite of Python programs [50].

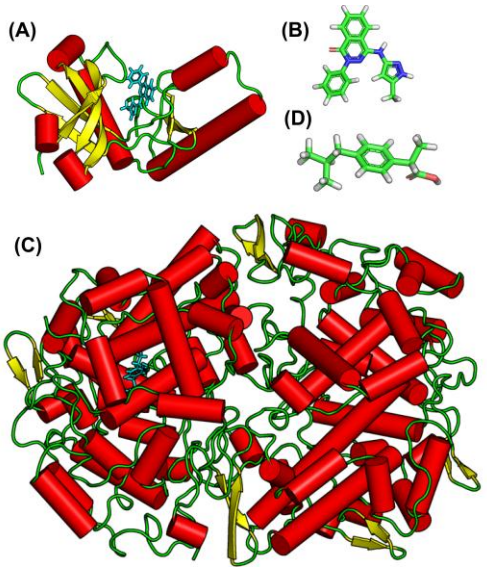

Figure 1: (A) A schematic view of the structure of Aurora-A kinase complexed with inhibitor – phthalazinone in cyan color (PDB code: 3P9J). (B) ball-and-stick representation of inhibitor. The system consists of 2,604 atoms divided into 155 fragments. (C) a schematic view of the structure of prostaglandin H(2) synthase-1 (COX-1) in a complex with ibuprofen in cyan color (PDB code: 1EQG). (D) ball-and-stick representation of ibuprofen. The system consists of 17,767 atoms divided into 1,093 fragments.

III. LOAD BALANCING IN FRAGMENT MOLECULAR ORBITAL METHOD

In the FMO method, a system is first subdivided into fragments. The proteins considered in this paper are divided naturally into amino acid residue fragments (at the Cα atoms) using the FMOgen tool in the FMOtools package [50]). We assumed that their standard protonation state lies at pH 7. The key issue for load balancing is that amino acid residues vary in size from the smallest with 7 atoms (Glycine) to the largest with 24 atoms (Tryptophan). The accuracy of FMO [44] is determined by the fragmentation, and hence fragments should not be very small. In other words, the number and the size of fragments are determined by the underlying chemistry and should not be modified merely to improve the efficiency of parallelization. The number of fragments and their sizes are therefore considered fixed for the purposes of parallelizing the calculations.

The size of a fragment greatly affects quantum chemistry calculations because the calculation cost tends to scale as a high power of the system size. For example, RHF scales as N^3 and coupled-cluster with perturbative triples (CCSD(T)) scales as N^7. The electronic state of some fragments can be frozen [51]. Still more variation in task sizes can arise from having different levels of theory and basis set for different regions of the system, as in the multilayer FMO method [52]. As the methodology of FMO becomes increasingly sophisticated, the time to solution and the scalability of individual fragment calculations become harder to model. An example of the variation in scalability of fragment calculations as a function of size is shown in Fig. 2.

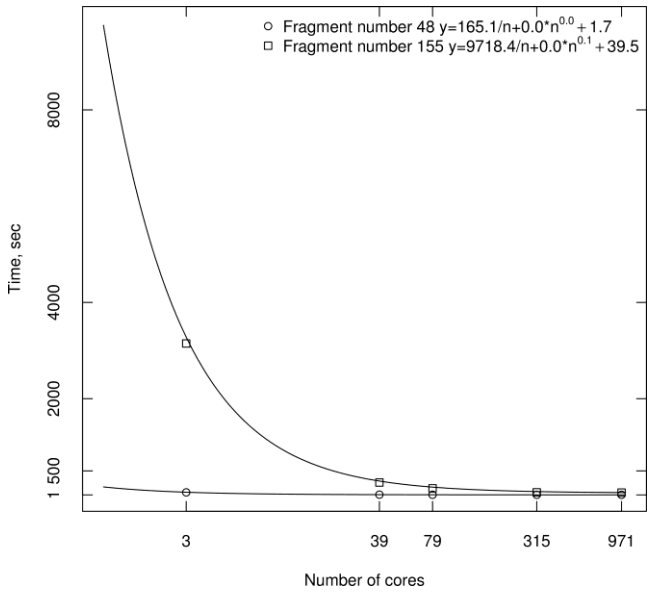

Figure 2: Scalability of FMO fragments in Aurora-A kinase and inhibitor system on Blue Gene/P. The smallest fragment (Gly amino acid residue) and the largest fragment (inhibitor) are represented by ○ and □, respectively. The data points were fitted and performance models for each fragment are shown. The cores represent the computational processes in GDDI. The scale of the x-axis is logarithmic.

A primary factor in the cost of quantum chemical calculations is the number of basis-functions (which is roughly proportional to the number of atoms). For FMO specifically (assuming the simple RHF level), important factors also include (1) the number of self-consistent field (SCF) iterations needed to achieve convergence; (2) the fragment packing density, namely, the number of fragments close to a given fragment, which strongly affects the computational time for the embedding potential; and (3) the fragment packing density. The latter has a large impact on dimer calculations owing to the use of electrostatic approximations (described elsewhere [44]). Furthermore, factors (1)-(3) strongly interact. For instance, the fragment packing density affects the SCF convergence, which, in turn, also depends on the charge, spin state, and the initial guess of the electron density. In addition, the scaling and parallel efficiency of the code are a complex function of these factors; for instance, the relative fraction of the number of sequential steps, such as matrix diagonalizations, is strongly affected by the choice of SCF convergence method, as well as by the number of SCF iterations (because the embedding potential is computed once before SCF). All these factors

make the modeling of the functional dependence of the timing upon the fragment size a formidable task.

Once a system is split into fragments FMO calculations can be performed by using the algorithm shown in Fig. 3. A detailed discussion of the algorithm is given elsewhere [48]. Here, we describe it briefly. At the coarse-grained DLB level, fragments are assigned to groups of CPU cores. In conjunction with MPI, GDDI can generate processor groups as shown in Fig. 3, line 3 (MPI_COMM_SPLIT function). Currently, the default option in GAMESS creates processor subgroups of uniform size. Each group performs single-point fragment calculations, assigned dynamically (see Fig. 3, line 7). Throughout this paper the theory is RHF. The output of such an RHF calculation is the fragment density in the Coulomb field of all fragments (Fig. 3, line 10). Since the new density changes the field, the process must be repeated until self-consistency is achieved. This process involves exchange of fragment densities among the groups by putting generated densities in DDI global array (DDI_put, Fig. 3, line 12). The fragment densities are accessed via DDI_get inside SCF(i) and SCF(i,j) in order to compute the embedding potential, lines 10 and 23, respectively. The iterative process is sometimes referred to as the "self-consistent charge" (SCC) or "monomer SCF" step, corresponding to the first term of the energy expansion in equation (1) with RHF theory. In the final step (Fig. 3, lines 17-26), fragment monomer densities are used to construct dimers from all pairs of monomers constituting a second round of larger RHF calculations. However, the "dimer" step is not iterated to self-consistency with respect to the embedding potential.

```
// Initialize variables
1:  number_of_fragments=input();
2:  number_of_groups=number_of_fragments/3;
3:  DDI_group=DDI_group_create(number_of_groups,DDI_world);
// Monomer loop
4:  do {
5:     for (i=1; i<number_of_fragments; i++) {
6:        DDI_scope(DDI_world);
7:        mytask=dynamic_load_balancing(DDI_world);
8:        if (mytask==i) {
9:           DDI_scope(DDI_group);
10:          fragment_density(i)=SCF(i);
11:          DDI_scope(DDI_world);
12:          DDI_put(fragment_density[i]);
13:       }
14:    }
15:    DDI_sync(DDI_world);
16: } while (fragment_density[]!=converged);
// Dimer loop
17: for (i=1; i<number_of_fragments; i++) {
18:    for (j=1; j<i; j++) {
19:       DDI_scope(DDI_world);
20:       mytask=dynamic_load_balancing(DDI_world);
21:       if (mytask==i,j) {
22:          DDI_scope(DDI_group);
23:          two_fragment_density(i,j)=SCF(i,j);
24:       }
25:    }
26: }
```

Figure 3: Pseudo-code of FMO calculations for dynamic load balancing.

For FMO, three types of load balancing have been attempted prior to this work, and we suggest an efficient modification of one of them, the static load balancing [48]. The alternative to SLB is DLB; in addition, there is semi-dynamic load balancing (SDLB) [49]. In DLB, an efficient means to improve efficiency is the large-jobs-first strategy [48]. This strategy considerably reduces the synchronization lag at the end of calculations because the smallest tasks are done last. DLB, in our experience, performs satisfactorily when the ratio of the total number of cores to the number of fragments is not very high (roughly 16 for our case, but it may vary considerably). Using this ratio and recalling that the number of fragments is fixed, DLB may be applied on a protein with 400 residues with good results on up to roughly 6,400 CPU cores. Adding more cores may result in a deterioration of the performance. The parallelization efficiency may drop because the calculations of small fragments cannot be efficiently parallelized on a large number of cores allocated under the equal partitioning scheme of DLB. An improvement of this DLB problem has been achieved with SDLB, in which a handful of the largest fragment calculations are performed using SLB, while the rest are done with DLB (after the CPU cores participating in SLB finish, they also join in the DLB calculations). However, such a strategy is useful mainly in cases where there are only a few large fragments and the total number of CPU cores is not high; otherwise the problems mentioned above cannot be avoided, an efficient solution is given by the heuristic static load balancing (HSLB) method proposed in Section IV. The main idea behind HSLB is to customize GDDI group sized to the fragment sizes. Since we solve an optimization problem heuristically, it can easily adapt to handle different numbers of CPUs and fragments.

The number of processor groups used in FMO calculations can vary from one to the number of fragments. Fig. 4 depicts the impact of the group count on the scalability of FMO for a single SCC iteration of a system with 155 fragments. In the case of a single GDDI group, each fragment calculation is executed on all CPU cores. Clearly, all but the largest fragments utilize the large processor count inefficiently, and the overall calculation has a low scalability. On the other hand, the 155-group calculation, in which there is a group for every fragment, exhibits improved scalability. The current default choice assumes three fragments per group, yielding 52 groups in this system. The difference in scalability and wall clock time for different group counts is explained in Figs 5 and 6. While the synchronization time shown is averaged over all GDDI groups, the efficiency is computed for each fragment separately and then averaged over all fragments. Thus, the efficiency, W_i, of fragment i, as a function of the number of CPU cores, is computed as

$$W_i(n) = \frac{T_i(n_0)/T_i(n)}{n/n_0}, \qquad (2)$$

where n_0 is the reference value of the number of CPU cores (n_0 =2 and $T_i(n_0)$ was obtained by extrapolation), n is the actual number of CPU cores, and $T_i(n_0)$ and $T_i(n)$ are the wall clock times to compute the energy of fragment i in FMO on n_0 and n CPU cores, respectively.

The data in Fig. 5 and Fig. 6 can be used to explain why the optimum group count with DLB is between 1 and 155. For example, the synchronization time tends to increase with the group count, starting at zero seconds in the case of a single group. However, computational efficiency also tends to increase with the group count as smaller groups encounter lower parallel overheads. Therefore, an optimal group-count can be obtained only by finding the right balance between the time spent in synchronization and that gained by parallelism. In addition, we must ensure that the variance in time taken by different fragments in minimized. These times in turn depend on hardware characteristics: the number of cores, CPU type, and the network type of the system.

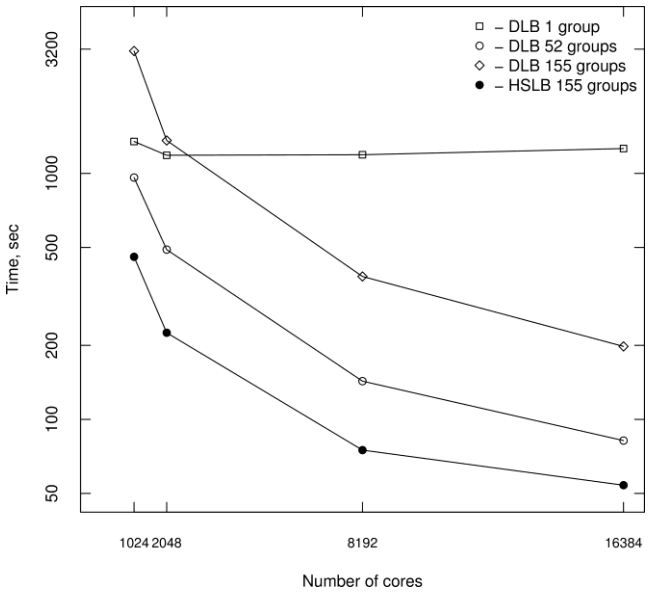

Figure 4: Wall-clock time to finish a single FMO SCC iteration with different load balancing schemes. The dataset is for Aurora-A kinase and inhibitor system. The calculations are done at the RHF-D level of theory and 6-31G* basis set on Blue Gene/P. The scale of the y-axis is logarithmic.

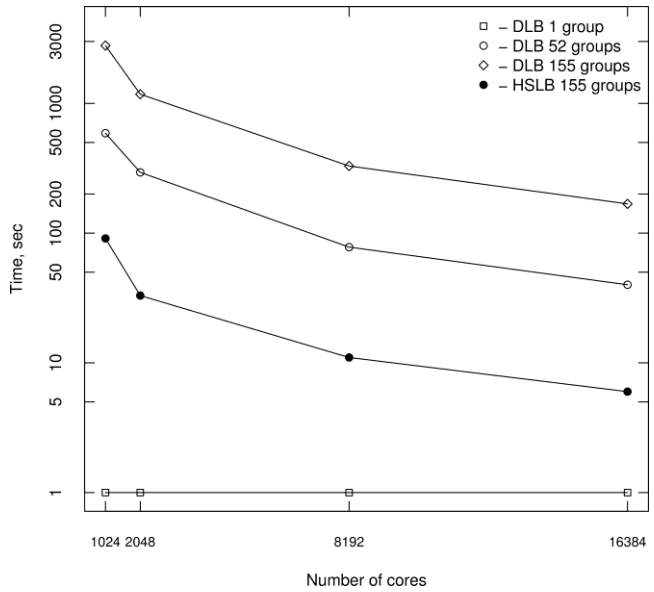

Figure 5: Average synchronization time among fragments accumulated during the first FMO SCC iteration. For DLB with one group, the synchronization time is equal to 0 seconds but because of the log scale it is shown as 1 second. The scale of the y-axis is logarithmic.

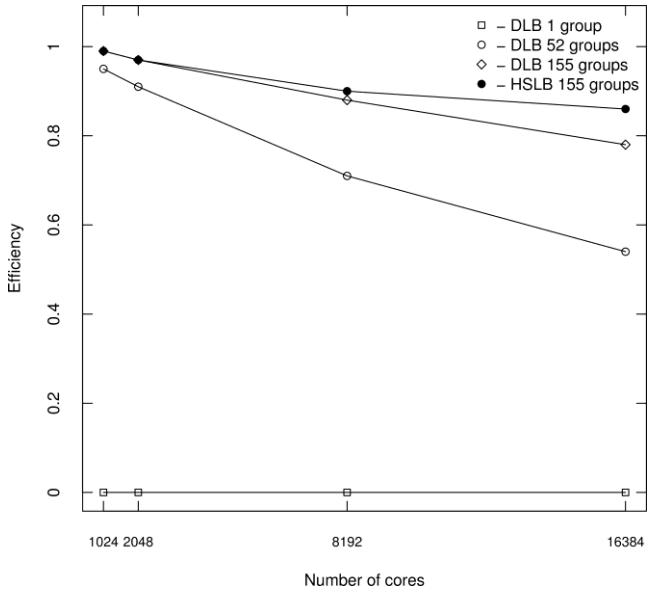

Figure 6: Parallel efficiency averaged over fragments during the first FMO SCC iteration for different load balancing schemes on Blue Gene/P. The dataset is for Aurora-A kinase and inhibitor system.

IV. HEURISTIC STATIC LOAD-BALANCING ALGORITHM

Our heuristic static load-balancing method consists of four steps. First, we collect benchmarking data related to the compute time of fragments. Second, we solve for the optimal parameters by a least-squares method based on our chosen scalability model. Third, we solve an integer optimization problem in order to obtain an optimal allocation of cores. Fourth, we allocate the optimal number of cores obtained from the optimization to run FMO in static load-balancing mode.

With a suitable model for the compute time, one can apply this four-step procedure to any other coarse-grained application. Before describing each of these steps for our application, we list in Table I the notation used to denote variables and parameters in our models.

Table I. List of variables and parameters used in models described in Section IV.

Symbol	Description
$\Re_+$	Set of positive real numbers.
F	Total number of tasks (fragments) among which we want to allocate available cores.
N	Total number of cores available for allocation.
n_i	Number of cores allocated for processing task-i.
$T_i(n_i)$	Performance function that models the time taken to process task-i by using n_i number of cores.
$T_i^{scal}(n_i)$	Scalable component of the function $T_i(n_i)$.
$T_i^{serial}(n_i)$	"Serial" component of the function $T_i(n_i)$.

$T_i^{nonlin}(n_i)$	Component of the function $T_i(n_i)$ other than $T_i^{scal}(n_i)$ and $T_i^{serial}(n_i)$.
D_i	Total number of data points available for creating the performance function model for fragment i.
a_i, b_i, c_i, d_i	Parameters associated with the performance function, $T_i(n_i)$, of task-i.
τ	Wall-clock time obtained from solving the allocation problem.
y_{ij}	Observed wall-clock time in the $j-$th run of fragment i, $j = 1,\ldots,D_i$, in the benchmarking stage.
n_{ij}	Number of cores allocated in the $j-$th run of fragment i, $j = 1,\ldots,D_i$, in the benchmarking stage.

A. Performance Model

Choosing an appropriate performance model is one of the most important steps in designing a successful SLB algorithm. Over the years many performance models have been developed [53]. Many of parallel performance models begin by identifying sequential and parallel components of the execution time in accordance with Amdahl's law. They try to capture the salient features of the calculation in terms of the key parameters of the problem. For the FMO application considered here, the key feature is the coarse-grained parallelism, which can be captured by selecting mathematical models for the run time of each fragment independently. In this work, we use the nonlinear model

$$T_i(n_i) = T_i^{scal}(n_i) + T_i^{nonlin}(n_i) + T_i^{serial} = \frac{a_i}{n_i} + b_i n_i^{c_i} + d_i,\ i = 1,\ldots,F, \quad (3)$$

where $T_i(n_i)$ represents the wall-clock time to compute the i^{th} fragment as a function of n_i the number of processor cores allocated to process it. The three components of $T_i(n_i)$ are described next.

The quantity $T_i^{scal}(n_i)$ represents the component of the wall-clock time with perfect (or *linear*) scalability. It is a monotonically decreasing function that asymptotically approaches zero. The quantity $T_i^{serial}(n_i)$, on the other hand, represents the time spent in the nonparallelized component of the application. It is independent of the number of cores n_i and includes any purely serial part of code. From the mathematical point of view it is a constant that defines the minimum value of $T_i(n_i)$ (ignoring $T_i^{nonlin}(n_i)$). As n_i increases, T_i^{serial} is expected to dominate $T_i(n_i)$.

The quantity $T_i^{nonlin}(n_i)$ represents the component of the wall-clock time that is not described by either $T_i^{scal}(n_i)$ or $T_i^{serial}(n_i)$. It represents the time spent in code that is only partially parallelized or depends on n_i in a way more complicated than the other two components. An example of a partially parallel component of our application is the diagonalization of the Fock matrix in the self-consistent field (SCF) method. Generally, $T_i^{nonlin}(n_i)$ may include time spent in activities such as initialization, communication, and synchronization. Our choice of the form of $T_i^{nonlin}(n_i)$ gives our model the ability to account for all these components without constraining it to be an increasing or decreasing function. The sign of the parameters b_i and c_i determines the shape of the function, and consequently every fragment may have a different shape of $T_i^{nonlin}(n_i)$.

The functional form of $T_i(n_i)$ seems to make sense both mathematically and from the viewpoint of Amdahl's law. From the mathematical perspective, one component of $T_i(n_i)$ decreases, while another increases with n_i. The function may increase or decrease for different values of n_i depending on the dominating component for that number of cores. Two real examples of $T_i(n_i)$ are illustrated in Fig. 2, where the probed range of the number of cores is not large enough to observe a complex behavior and $T_i(n_i)$ is a smoothly decreasing function. From the perspective of Amdahl's law, in the absence of the complicating component $T_i^{nonlin}(n_i)$, $T_i^{scal}(n_i)$ accounts for largest contribution when n_i is small, while T_i^{serial} is the largest contribution to for large n_i.

B. Fitting Data

We estimate the parameters $a_i, b_i, c_i,$ and d_i used in Eq. (3) by fitting the values of wall-clock time of each fragment over the first SCC iteration for different CPU core groupings. In other words, we perform calculations of each fragment in the embedding potential, varying the number of cores per GDDI group. The timings are collected as a function of the number of cores per group, and we fit the coefficients. In the future we plan to examine the possibilities of using several SCC iterations for the fitting.

For the i^{th} fragment, we obtain the best fit by solving the least squares problem

$$\min_{a_i, b_i, c_i, d_i} \sum_{j=1}^{D_i} \left(y_{ij} - \frac{a_i}{n_{ij}} - b_i n_{ij}^{c_i} - d_i \right)^2, \quad (4)$$
$$\text{subject to } a_i, b_i, c_i, d_i \in \Re_+,$$

where y_{ij} is the observed value of time taken in solving for fragment j when n_{ij} cores are allocated to it. D_i is the number of different GDDI groups sizes tried in the fitting procedure (in this paper, D_i varied from 3 to 7, depending on the system). The objective function of the optimization problem (4) is in general not convex, and there may be several locally optimal solutions of the problem. Since nonlinear optimization algorithms are iterative, selecting a different starting point may lead the solver to a different local solution. We experimented with different starting solutions and observed that even though the parameter values may

differ, the solution value of problem (4) did not vary significantly. More important was the observation that the differences in parameter values did not translate into significant differences in the optimal allocation of cores that we calculate in the next step.

We have constrained the variables in our fitting problem Eq. (4) to be nonnegative even though doing so is not necessary mathematically. It makes sense for parameters *a*, *b*, and *d* to be positive because they represent values of time. It is less obvious what the constraints for *c* should be. In general, $T_i^{nonlin}(n_i)$ can be increasing or decreasing, but we prefer a positive *c* because our application is highly scalable. The total time does not increase even when the number of cores used in production runs is much larger than that in trial runs for gathering data. Thus, a positive value of *c* ensures that our model has a better fit even when we extrapolate it to a large number of cores.

The examples of fitted $a_i, b_i, c_i,$ and d_i can be seen for the smallest and the largest fragments in Fig. 2. Since the values y_{ij} are gathered from actual runs on the system, it is important to judiciously choose trial values of n_{ij} in the data-gathering stage. There is an obvious trade-off between the time taken to obtain y_{ij} and the quality of the model. Since the solution procedure in GAMESS is iterative and the nature of work is similar for all iterations, we can model the functions using time observations for a single iteration only. It helps us save time without sacrificing accuracy. To obtain good estimates of $a_i, b_i, c_i,$ and d_i, we recommend sampling n_{ij} from a large range of core counts: from a few to thousands for each fragment. In order to avoid *over-fitting*, the number of samples should be at least greater than four for each fragment. We used eight samples in our experiments. The number of samples should obviously increase with the level of noise in the application and the number of parameters to be estimated.

In general, one should judiciously pick samples based on *a priori* knowledge of the tasks. Lacking such knowledge, we began by dividing the available cores equally among all groups. This approach proved satisfactory for systems with similar-sized fragments (along with a consistent theory and basis set). In the cases when, for example, a ligand is much larger than the largest amino acid, a more sophisticated allocation for sampling may needed. We also note that the recorded times do not include FMO initialization and intergroup synchronization time, but they do include all intragroup computation and communication including synchronization.

Our procedure of first collecting data and then rerunning the full application from scratch can be improved. We use our simple procedure to demonstrate the effectiveness of using an optimal allocation of cores. Our procedure can be modified with little effort to reuse more information from the data collection stage for the solving stage.

C. Formulating the Optimization Problem

Once we have identified an appropriate performance model and obtained values of all parameters from the previous steps, we can formulate an optimization problem to find the optimal allocation of cores. The *decision variables* that we seek to optimize are the number of processors, n_i, to be allocated to each fragment $i \in \{1,..F\}$. The choice of objective that we seek to minimize or maximize depends on the preference of the user. One can minimize the total wall-clock time of the application the following *min-max* function can be used

$$\min_{n} \max_{i \in \{1,\ldots,F\}} T_i(n_i). \tag{5}$$

Alternatively, the *objective function* is just the sum of times used by each task,

$$\min_{n} \sum_{i=1}^{F} T_i(n_i). \tag{6}$$

One can also seek to maximize the minimum time used by a task. Like the min-max criterion, the max-min criterion also seeks to obtain a fair distribution of cores by taking away allocations from the fastest tasks. It is written as

$$\max_{n} \min_{i} T_i(n_i). \tag{7}$$

The physical restrictions of the system can be modeled by adding constraints to the optimization problem; for example, the number of cores used in calculations cannot exceed the total number of available cores, *N*,

$$\sum_{i=1}^{F} n_i \le N. \tag{8}$$

We can also have constraints based on user's preferences, e.g. the user may wish to minimize the wall clock time with an additional constraint that the total core time must be below a threshold *T*:

$$\sum_{i=1}^{F} T_i(n_i) \le T. \tag{9}$$

Some constraints may be needed to make the model amenable for the solver. In particular, most solvers require the derivatives of objective and constraints to be continuous. The min-max objective function should be therefore be replaced by an objective of minimizing a new variable, say η, and additional constraints must be introduced to ensure η is no less than each $f(n_i)$. The full model is

$$\begin{aligned} &\min_{n_i,\tau} \tau \text{ subject to } \sum_{i=1}^{F} n_i \le N \\ &\frac{a_i}{n_i} + b_i n_i^{c_i} + d_i \le \tau, i = 1,\ldots,F, \\ &n_i \ge 0, \text{integer}, i = 1,\ldots,F. \end{aligned} \tag{10}$$

We considered the three objective functions described above, together with constraint (8) in our models. We observed in our experiments that the min-max function (5) outperforms the other objectives, which makes sense from the viewpoint of minimizing the overall wall-clock time. Minimizing total time, at the other extreme, may lead to a solution where one fragment is solved in exceptionally large time (Fig. 7), thus keeping the other processors waiting.

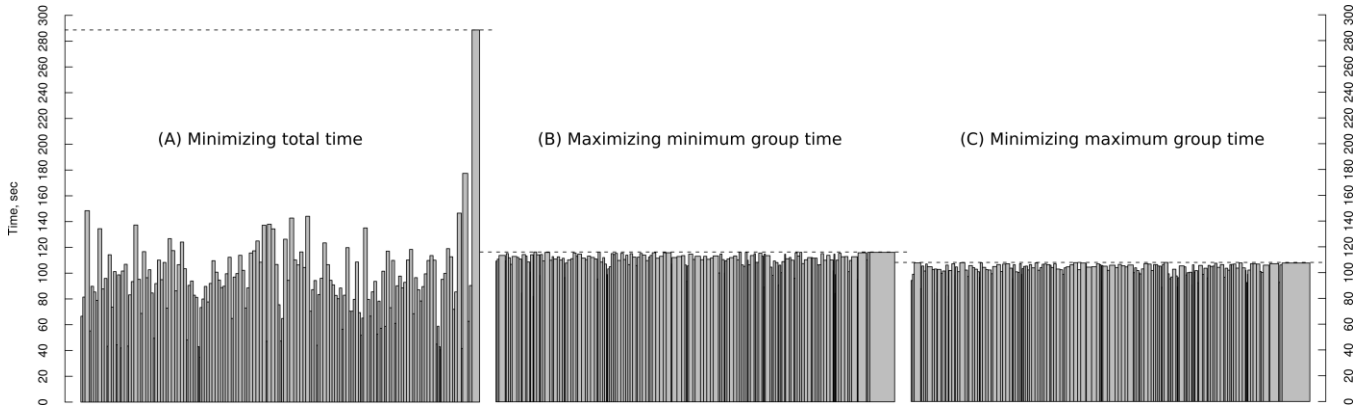

Figure 7: Allocation of different solvers: (A) minimizing total time, (B) maximizing the minimum group time, and (C) minimizing the maximum group time. The height of each column represents time to compute one fragment, and the width of each column represents how many cores were assigned. The dataset is for the complex of Aurora-A kinase and its inhibitor, which was collected on 1024 cores of Blue Gene/P for FMO at the RHF/6-31G* level.

D. Solving the MINLP Model

MINLP problems, of which the optimization problem Eq. (10) is a special case, are NP-hard in general. Certain specific classes of MINLP, such as the single constraint resource constrained problems with nonincreasing objective functions can be solved in polynomial time [54]. But they require customized algorithms. Hence we consider algorithms for general MINLPs only. The algorithms to solve general MINLPs are usually based on the branch-and-bound method [55]. These methods are guaranteed to provide an optimal solution or show that none exists. In addition to the number of variables and constraints, the time required to solve these problems depends on the type of functions used in the objective and constraints. For instance, if all the nonlinear functions are convex, then a local solution of the continuous relaxation is also its global solution. Several specialized algorithms exploit this fact and other useful properties of convex functions [55-60]. On the other hand, if any function is not convex, then the continuous relaxation does not give a bound on the objective value. In this case, one needs to further relax the continuous problem by introducing new variables and modifying the constraints [61, 62].

We wrote our optimization problem in the AMPL [63] modeling language. AMPL enables users to write optimization model using simple mathematical notation. It also provides derivatives of nonlinear functions automatically, and it can be used with several different solvers. To solve the problem, we used the open-source solver toolkit MINOTAUR [28]. MINOTAUR offers different solvers based on the algorithms mentioned above and also offers advanced routines to reformulate MINLPs. It provides libraries that can be called from other C++ and FORTRAN codes and hence can be directly called without requiring AMPL. For solving our problem, we use the LP/NLP [56] solver implemented in MINOTAUR. Since the coefficients a_i, b_i, c_i are positive, the nonlinear functions are convex, and this algorithm finds a global solution of the problem. We briefly describe this algorithm next.

The LP/NLP algorithm is initialized by first creating a linear relaxation of the MINLP. Suppose we have a nonlinear constraint of the form $f(x) \le 0$, where f is a continuously differentiable convex function. A linear relaxation of the constraint is obtained by the linearization around any point x^k,

$$\nabla f(x^k)^T \left(x - x^k\right) + f(x^k) \le 0. \qquad (11)$$

In general, the more the number of linearization constraints obtained from distinct points, the closer is the relaxation to the original problem. However, a large number of constraints can slow the solver. In order to mitigate this problem, linearization-constraints derived from only a single point are added initially. This point is the obtained by solving the continuous nonlinear programming (NLP) problem. We later add linearization constraints for only those nonlinear constraints that are violated significantly by the solution.

After the initial linear programming (LP) relaxation is created, it is added to a list of unsolved sub-problems. The value of the incumbent solution of MINLP is initialized to infinity. In each step of the algorithm, we remove a sub-problem from the list and solve the linear relaxation using an LP solver. If the solution value is greater than the incumbent, we discard this sub-problem because it does not contain any solution better than the incumbent. If the solution $(\hat{x})$ of the LP problem has fractional values, we create two new sub-problems by branching. We choose an integer variable i for which $\hat{x}_i$ is fractional. In one sub-problem we add the constraint $x_i \ge \lceil \hat{x}_i \rceil$. In the other, we add the constraint $x_i \le \lfloor \hat{x}_i \rfloor$. These two sub-problems are added to

the list of unsolved sub-problems. If $\hat{x}$ satisfies integer constraints, we check whether it satisfies all the nonlinear constraints as well. If it is feasible, then we have an incumbent solution. Otherwise, we add more linearization constraints around $\hat{x}$ of the form shown in Eq. (10), and continue. The algorithm terminates when the list is empty.

In MINOTAUR, the LP problems are solved by using the CLP solver [64], and the NLP problems use filterSQP [65]. In the worst case, the algorithm may require solving an exponential (in the number of integer variables) number of LP and NLP problems, but in practice it takes much fewer. For example, the MINLP for 4096 cores took < 180 seconds on one core to solve, and made 12863 calls to the LP solver and 2 calls to the NLP solver. For 16384 cores, these numbers were 165 seconds, 9883 and 2, respectively.

E. Summary of HSLB Algorithm

Before presenting the results of our experiments, we summarize the four-step HSLB algorithm and discuss some ways of further improving it.

(1) **Gather Data:** Perform a single SCC iteration for the given molecular system (protein-ligand complex) with FMO by executing GAMESS D times using a different total numbers of cores, with suitable choices for D_i. Collect the running times y_{ij} for each fragment i.

(2) **Fit:** Next, solve F different least squares problems (4) to determine the coefficients a_i, b_i, c_i, and d_i in Eq. (3) for each fragment i.

(3) **Solve:** Determine the best allocation by solving the MINLP (10), and obtain the optimal values of size n_i for each fragment i.

(4) **Execute:** Execute the complete FMO run with GAMESS, using the determined group sizes in step (3).

This algorithm, being of a general nature, can be improved in several ways for a given application. The data gathering step (1) can be avoided altogether if reliable benchmarks are already available, for example, from previous experiments. Steps (2) and (3) can be solved by calling a MINLP solver directly from the application, thus avoiding the use of AMPL. The least-squares problem can be solved with a MINLP solver by just calling its nonlinear solver once. After it is solved, the MINLP solver can then solve the MINLP of step (3). More improvements are possible if the HSLB procedure is called more than once to reallocate the cores after a few iterations of the complete run. The running time of all iterations can be stored, a better fit be obtained, and the MINLP re-solved to obtained better allocation based on the new data.

In this work, we applied HSLB only to the monomer step in FMO, which is iterative and requires running each monomer calculation typically 20-25 times. We used DLB for the dimer step, which involves computing each dimer once; and thus the benefits of an optimized allocation in HSLB do not merit its application given the need to do preliminary data gathering. However, in the future it is conceivable to construct a good guess for an optimum node allocation in dimers based on the monomer data, which would accelerate the dimer step as well. The load balancing in dimers is also less severe than in monomers, because the number of dimers, for which quantum-mechanical calculations are performed, is typically 3-4 times the number of fragments F (dimers that are spatially well separated are computed with a very fast electrostatic approximation) [44].

V. RESULTS AND DISCUSSION

The performance of HSLB is compared to that of DLB with different numbers of groups for the system of Aurora kinase and inhibitor phthalazinone (see Fig. 1 (A) and (B)). This system has 155 fragments. Fig. 4 shows that the HSLB scheme outperforms the DLB schemes by at least a factor of two in the wall-clock time. We also found that some DLB schemes have scalability similar to HSLB. We also make other observations about the performance of HSLB. Fig. 5 shows that HSLB has the lowest synchronization time even on thousands of processors. Since the synchronization time becomes important when a large number of CPU cores are used, HSLB should be preferred for such systems. The HSLB algorithm also shows excellent efficiency, greater than 90% on large numbers of cores, as shown in Fig. 6. As the number of cores increases, we anticipated that the scalability and efficiency of HSLB might deteriorate. To quantify this deterioration, we tested the performance of HSLB for larger processor counts using a larger problem: COX-1 complexed with ibuprofen (see Fig. 1 (C) and (D)); a total of 1093 fragments and 17,767 atoms. Fig. 8 shows that the COX-1 calculation achieves 80% efficiency averaged over all fragments for the SCC iterations in FMO on 163,840 cores at the RHF, 6-31G* level of theory. The single-point

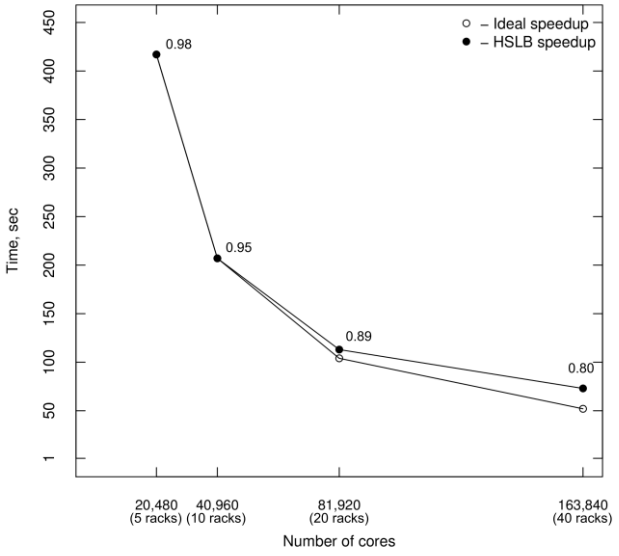

Figure 8: Ideal and observed scalability curves based on wall-clock time for the first FMO SCC iteration on Blue Gene/P for COX-1 complexed with ibuprofen. All calculations are done in a "dual" mode that restricts processes to 2 MPI tasks per node. Efficiency averaged over all the fragments is shown for each run.

energy calculation takes only ~54 minutes (6+ years on a single core). The results obtained at this computational level strongly suggest that significantly higher processor counts can be efficiently utilized for larger problems.

While HSLB outperforms DLB, it still exhibits a small decline in scalability and efficiency for high processor counts for both the Aurora kinase and COX-1 calculations. This decline may be due to sequential steps in the fragment SCF and the fluctuations in the synchronization time caused by runtime operating system tasks, shared network issues, hardware failure, deficiencies of performance model or benchmarking data, and so forth. It is commonly understood that for these reasons, synchronization becomes more problematic as the number of processors increases. Although these fluctuations do not appear in Fig. 5 (only averaged values are shown), use of a low level of theory (RHF, 6-31G*) here has helped uncover the limitations of the HSLB approach by raising the significance of the synchronization time (for density functional theory (DFT) one can expect a better parallel efficiency because of a high scaling of the DFT specific grid integration). From the data, the operating limits of HSLB on Blue Gene/P would appear to be anywhere from three cores per task up to the point where random computational noise (>100 thousands cores) hampers the ability to predict the time to solution for tasks.

We have identified directions for further improving our load-balancing approach. We observed that the iteration time in our application is not a constant but tends to decrease because successive SCC iterations typically require fewer micro-iterations to converge the density. Moreover, this behavior is not uniform over different fragments because they converge at different rates. We propose to apply HSLB adaptively. We can fit scalability curves, obtain the nonlinear equations and solve for the optimal allocation for all SCC iterations, as described in Section IV. To this end, we have interfaced MINOTAUR directly with GAMESS on Blue Gene/P. It enables us to directly optimize without making system calls to execute the AMPL model. We have not included results for adaptive HSLB here because our goal is to present the fundamental HSLB concept. That said, adaptive HSLB offers a promising direction for future development because it combines the efficiency of HSLB with the adaptability of DLB.

VI. Conclusions

The method development in this paper is an evolution of the parallelization of a complex quantum-mechanical program GAMESS [24, 25] over dozens of CPU cores in DDI introduced in 2000 [46], extended to hundreds with GDDI in 2004 [48] as demonstrated on a powerful supercomputer of that time (in 2005 [49]). The manual variation of the group size in GDDI to optimize its performance used in a Supercomputing-2005 paper [49] inspired the present work, which we have conducted based on advanced mathematical methods guaranteeing the best allocation for a given number of cores and a molecular system. Although for fine-grained systems (water clusters) the previously developed load balancing has performed well up to about 130,000 CPU cores [27], coarse-grained systems (proteins) cannot be treated with high efficiency on modern petascale computers in the same way.

We have shown that the present HSLB approach is twice as fast as the previous DLB method and achieves a parallel efficiency of about 80% on petascale core-counts (hundreds of thousands of cores). Thus, from the user-perspective, HSLB is enabling FMO to handle automatically very large problems with diverse fragment sizes. Many interesting cases fall into the latter category. For example, in the study of photosynthesis, the reaction center [49, 66] features the chlorophyll special pair, which is large and difficult to fragment for the chemical reasons (significant electron delocalization across the planar system). Another common situation is found in drug design, where the drug molecules often have 50-100 atoms with extended conjugation. Such large fragments typically coexist with many small ones, such as explicit water molecules having only three atoms per fragment. Where DLB based on uniform group sizes would be unable to utilize many cores effectively for such systems, by fitting the GDDI group sizes to the fragments HSLB can efficiently utilize CPU core counts in the 100,000-range with negligible overhead. In this sense, HSLB is similar in spirit to the use of "preliminary" benchmarks in previous work to guess the optimum group sizes [49].

Our current era of petascale computing already has an eye on the coming exascale era, and the development of software capable of efficiently utilizing many thousands or millions of CPU cores is a topic of great interest. FMO accelerated by HSLB on petascale and exascale computers can become a powerful tool for drug and material design [44], realizing the high potential held by quantum-mechanical methods on massively parallel computers.

The present coarse-grained optimization algorithm is not limited to FMO. Many coarse-grained applications can benefit from the present approach. For instance, many other fragment-based methods can be similarly parallelized. As the number of cores increases, the issues of minimizing the synchronization time while retaining a high efficiency will put load balancing schemes to a highly stressful test. We believe that for coarse-grained applications our HSLB algorithm is a promising and general approach.

Acknowledgment

We thank Dr. R. Loy and ALCF team members for discussions and help related to the paper. We thank Dr. M. Mazanetz from Evotec for providing the PDB structure of Aurora-A kinase system used in our calculations. DGF thanks the Next Generation Super Computing Project, Nanoscience Program (MEXT, Japan) and Computational Materials Science Initiative (CMSI, Japan) for financial support and Prof. K. Kitaura for fruitful discussions.

The submitted manuscript has been created by the UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne") under Contract No. DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. This work was also supported by the U.S. Department of Energy through grant DE-FG02-05ER25694.

References

[1]C. Xu and F. C. M. Lau, *Load balancing in parallel computers: theory and practice*. Norwell, MA. Kluwer Academic Publishers, 1997.

[2]K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Applied Numerical Mathematics*, vol. 52, pp. 133-152, 2005.

[3]M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, pp. 979-993, 1993.

[4]Y. Bejerano, S. J. Han, and L. E. Li, "Fairness and load balancing in wireless LANs using association control," in *Proceedings of the 10th annual international conference on mobile computing and networking*, New York, NY, 2004, pp. 315-329.

[5]B. Y. Zhang, Z. Y. Mo, G. W. Yang, and W. M. Zheng, "An efficient dynamic load-balancing algorithm in a large-scale cluster," *Distributed and Parallel Computing*, pp. 174-183, 2005.

[6]M. J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations," in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, 1996, pp. 282-291.

[7]R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and experience*, vol. 3, pp. 457-481, 1991.

[8]M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *Computers, IEEE Transactions on*, vol. 100, pp. 570-580, 1987.

[9]H. D. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, vol. 2, pp. 135-148, 1991.

[10]V. E. Taylor and B. Nour-Omid, "A study of the factorization fill-in for a parallel implementation of the finite element method," *International journal for numerical methods in engineering*, vol. 37, pp. 3809-3823, 1994.

[11]M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the ACM/IEEE Supercomputing 1993 Conference*, Portland, 1993, pp. 12-21.

[12]J. R. Pilkington and S. B. Baden, "Partitioning with spacefilling curves, CSE Technical Report CS94-349," Dept. of Computer Science Engineering, University of California, San Diego, CA1994.

[13]A. Patra and J. T. Oden, "Problem decomposition for adaptive hp finite element methods," *Computing Systems in Engineering*, vol. 6, pp. 97-109, 1995.

[14]J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws," *Journal of Parallel and Distributed Computing*, vol. 47, pp. 139-152, 1997.

[15]A. Pothen, H. D. Simon, and K. P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal on Matrix Analysis and Applications* vol. 11, pp. 430-452, 1990.

[16]E. Leiss and H. Reddy, "Distributed load balancing: design and performance analysis," *WM Keck Research Computation Laboratory*, vol. 5, pp. 205-270, 1989.

[17]G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, p. 359, 1999.

[18]Y. F. Hu and R. J. Blake, "An optimal dynamic load balancing algorithm, Technical Report DL-P-95-011," Daresbury Laboratory, Warrington, WA4 4AD, UK1995.

[19]B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the ACM Supercomputing 1995 Conference*, New York, 1995, pp. 28-42.

[20]G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, pp. 279-301, 1989.

[21]T. Bui and C. Jones, "A heuristic for reducing fill in sparse matrix factorization," in *SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, PA, 1993, pp. 445-452.

[22]S. H. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. 100, pp. 207-214, 1981.

[23]S. H. Bokhari, *Assignment problems in parallel and distributed computing* vol. 32. New York, NY. Springer-Verlag, 1987.

[24]M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. S., T. L. Windus, M. Dupuis, and J. A. J. Montgomery, "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, pp. 1347-1363, 1993.

[25]M. S. Gordon and M. W. Schmidt, "Advances in electronic structure theory: GAMESS a decade later," in *Theory and Applications of Computational Chemistry: The First Forty Years*, C. Dykstra, G. Frenking, K. Kim, and G. Scuseria, Eds., ed. Elsevier Science, 2005, pp. 1167-1189.

[26]*Argonne National Laboratory: Argonne Leadership Computing Facility*. Available: http://www.alcf.anl.gov/,

[27]G. D. Fletcher, D. G. Fedorov, S. R. Pruitt, T. L. Windus, and M. S. Gordon, "Large-scale MP2 calculations on the Blue Gene architecture using the Fragment Molecular Orbital method," *Journal of Chemical Theory and Computation*, vol. 8, pp. 75-79, 2012.

[28]A. Mahajan, S. Leyffer, J. Linderoth, J. Luedtke, and T. Munson. *MINOTAUR wiki*. Available: http://www.mcs.anl.gov/minotaur, (January 16, 2012)

[29]R. Zalesny, M. G. Papadopoulos, P. G. Mezey, and J. Leszczynski, *Linear-Scaling Techniques in Computational Chemistry and Physics*. New York, NY. Springer, 2011.
[30]J. R. Reimers, *Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology*. Singapore. Wiley, 2011.
[31]E. Apra, R. J. Harrison, W. Shelton, V. Tipparaju, and A. Vázquez-Mayagoitia, "Computational chemistry at the petascale: Are we there yet?," in *Journal of Physics: Conference Series*, 2009, p. 012027.
[32]Y. Hasegawa, J. I. Iwata, M. Tsuji, D. Takahashi, A. Oshiyama, K. Minami, T. Boku, F. Shoji, A. Uno, and M. Kurokawa, "First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer," in *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, Seattle, 2011, pp. 1-11.
[33]E. Apra, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid water: obtaining the right answer for the right reasons," in *Proceedings of the ACM/IEEE Supercomputing 2009 Conference*, Portland, 2009, p. 66.
[34]K. Kowalski, S. Krishnamoorthy, R. M. Olson, V. Tipparaju, and E. Aprà, "Scalable implementations of accurate excited-state coupled cluster theories: Application of high-level methods to porphyrin-based systems," in *Proceedings of the ACM/IEEE Supercomputing 2011 Conference*, Seattle, 2011, pp. 1-10.
[35]Y. Alexeev, R. A. Kendall, and M. S. Gordon, "The distributed data SCF," *Computer Physics Communications*, vol. 143, pp. 69-82, 2002.
[36]Y. Alexeev, M. W. Schmidt, T. L. Windus, and M. S. Gordon, "A parallel distributed data CPHF algorithm for analytic Hessians," *Journal of Computational Chemistry*, vol. 28, pp. 1685-1694, 2007.
[37]M. Krishnan, Y. Alexeev, T. L. Windus, and J. Nieplocha, "Multilevel parallelism in computational chemistry using Common Component Architecture and Global Arrays," in *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, Seattle, 2005, pp. 23-23.
[38]G. Fletcher, "A parallel multi-configuration self-consistent field algorithm," *Molecular Physics*, vol. 105, pp. 2971-2976, 2007.
[39]M. Challacombe and E. Schwegler, "Linear scaling computation of the Fock matrix," *Journal of Chemical Physics*, vol. 106, pp. 5526-5536, 1997.
[40]R. J. Harrison, G. I. Fann, T. Yanai, Z. Gan, and G. Beylkin, "Multiresolution quantum chemistry: Basic theory and initial applications," *Journal of Chemical Physics*, vol. 121, p. 11587, 2004.
[41]M. S. Gordon, S. R. Pruitt, D. G. Fedorov, and L. V. Slipchenko, "Fragmentation methods: a route to accurate calculations on large systems," *Chemical Reviews*, vol. 112, pp. 632-672, 2012.
[42]S. Hirata, M. Valiev, M. Dupuis, S. S. Xantheas, S. Sugiki, and H. Sekino, "Fast electron correlation methods for molecular clusters in the ground and excited states," *Molecular Physics*, vol. 103, pp. 2255-2265, 2005.
[43]K. Kitaura, E. Ikeo, T. Asada, T. Nakano, and M. Uebayasi, "Fragment molecular orbital method: an approximate computational method for large molecules," *Chemical Physics Letters*, vol. 313, pp. 701-706, 1999.
[44]D. G. Fedorov, T. Nagata, and K. Kitaura, "Exploring chemistry with the Fragment Molecular Orbital method," *Physical Chemistry Chemical Physics*, vol. 14, pp. 7562-7577, 2012.
[45]D. G. Fedorov and K. Kitaura, "The importance of three-body terms in the fragment molecular orbital method," *Journal of Chemical Physics*, vol. 120, pp. 6832-6840, 2004.
[46]G. D. Fletcher, M. W. Schmidt, B. M. Bode, and M. S. Gordon, "The distributed data interface in GAMESS," *Computer Physics Communications*, vol. 128, pp. 190-200, 2000.
[47]J. L. Bentz, R. M. Olson, M. S. Gordon, M. W. Schmidt, and R. A. Kendall, "Coupled cluster algorithms for networks of shared memory parallel processors," *Computer Physics Communications*, vol. 176, pp. 589-600, 2007.
[48]D. G. Fedorov, R. M. Olson, K. Kitaura, M. S. Gordon, and S. Koseki, "A new hierarchical parallelization scheme: Generalized distributed data interface (GDDI), and an application to the fragment molecular orbital method (FMO)," *Journal of Computational Chemistry*, vol. 25, pp. 872-880, 2004.
[49]T. Ikegami, T. Ishida, D. G. Fedorov, K. Kitaura, Y. Inadomi, H. Umeda, M. Yokokawa, and S. Sekiguchi, "Full electron calculation beyond 20,000 atoms: Ground electronic state of photosynthetic proteins," in *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, Seattle, pp. 10-10.
[50]Y. Alexeev. *FMO portal: Web interface for FMOtools*. Available: http://www.fmo-portal.info, (January 16, 2012)
[51]D. G. Fedorov, Y. Alexeev, and K. Kitaura, "Geometry optimization of the active site of a large system with the fragment molecular orbital method," *Journal of Physical Chemistry Letters*, vol. 2, pp. 282-288, 2011.
[52]D. G. Fedorov, T. Ishida, and K. Kitaura, "Multilayer formulation of the fragment molecular orbital method (FMO)," *The Journal of Physical Chemistry A*, vol. 109, pp. 2638-2646, 2005.
[53]C. L. Janssen and I. M. B. Nielsen, *Parallel computing in quantum chemistry*. CRC Press, 2008.
[54]T. Ibaraki and N. Katoh, *Resource allocation problems: algorithmic approaches*. Cambridge, MA. The MIT Press, 1988.

[55]R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *The Computer Journal*, vol. 8, pp. 250-255, 1965.
[56]I. Quesada and I. E. Grossmann, "An LP/NLP based branch and bound algorithm for convex MINLP optimization problems," *Computers & Chemical Engineering*, vol. 16, pp. 937-947, 1992.
[57]M. A. Duran and I. E. Grossmann, "An outer-approximation algorithm for a class of mixed-integer nonlinear programs," *Mathematical Programming*, vol. 36, pp. 307-339, 1986.
[58]R. Fletcher and S. Leyffer, "Solving mixed integer nonlinear programs by outer approximation," *Mathematical Programming*, vol. 66, pp. 327-349, 1994.
[59]T. Westerlund and F. Pettersson, "An extended cutting plane method for solving convex MINLP problems," *Computers & Chemical Engineering*, vol. 19, pp. 131-136, 1995.
[60]A. Mahajan, S. Leyffer, and C. Kirches, "Solving mixed-integer nonlinear programs by QP-diving," Argonne National Laboratory ANL/MCS-P2071-0312, 2012
[61]R. Horst and T. Hoang, *Global Optimization: Deterministic Approaches*. Berlin. Springer-Verlag, 1996.
[62]M. Tawarmalani and N. V. Sahinidis, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications* vol. 65. Dordrecht. Kluwer Academic Publishers, 2002.
[63]R. Fourer, D. M. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming, 2nd Edition* Independence, KY. Cengage Learning, 2002.
[64]J. J. Forrest. *Clp project*. Available: http://projects.coin-or.org/Clp, (January 16, 2012)
[65]R. Fletcher and S. Leyffer, "Nonlinear programming without a penalty function," *Mathematical Programming*, vol. 91, pp. 239-269, 2002.
[66]T. Ikegami, T. Ishida, D. G. Fedorov, K. Kitaura, Y. Inadomi, H. Umeda, M. Yokokawa, and S. Sekiguchi, "Fragment molecular orbital study of the electronic excitations in the photosynthetic reaction center of Blastochloris viridis," *Journal of Computational Chemistry*, vol. 31, pp. 447-454, 2010.