A New Hierarchical Parallelization Scheme: Generalized Distributed Data Interface (GDDI), and an Application to the Fragment Molecular Orbital Method (FMO)

DMITRI G. FEDOROV,[1] RYAN M. OLSON,[2] KAZUO KITAURA,[1] MARK S. GORDON,[2] SHIRO KOSEKI[3]

[1]*National Institute of Advanced Industrial Science and Technology, 1-1-1 Umezono, Tsukuba, 305-6568 Ibaraki, Japan*

[2]*Ames Laboratory, US-DOE and Department of Chemistry, Iowa State University, Ames, Iowa 50011*

[3]*Department of Material Science, Osaka Prefecture University, 1-1 Gakuen-cho, Sakai, 599-8531 Osaka, Japan*

Received 9 September 2003; Accepted 18 December 2003

Abstract: A two-level hierarchical scheme, generalized distributed data interface (GDDI), implemented into GAMESS is presented. Parallelization is accomplished first at the upper level by assigning computational tasks to groups. Then each group does parallelization at the lower level, by dividing its task into smaller work loads. The types of computations that can be used with this scheme are limited to those for which nearly independent tasks and subtasks can be assigned. Typical examples implemented, tested, and analyzed in this work are numeric derivatives and the fragment molecular orbital method (FMO) that is used to compute large molecules quantum mechanically by dividing them into fragments. Numeric derivatives can be used for algorithms based on them, such as geometry optimizations, saddle-point searches, frequency analyses, etc. This new hierarchical scheme is found to be a flexible tool easily utilizing network topology and delivering excellent performance even on slow networks. In one of the typical tests, on 16 nodes the scalability of GDDI is 1.7 times better than that of the standard parallelization scheme DDI and on 128 nodes GDDI is 93 times faster than DDI (on a multihub Fast Ethernet network). FMO delivered scalability of 80–90% on 128 nodes, depending on the molecular system (water clusters and a protein). A numerical gradient calculation for a water cluster achieved a scalability of 70% on 128 nodes. It is expected that GDDI will become a preferred tool on massively parallel computers for appropriate computational tasks.

© 2004 Wiley Periodicals, Inc. J Comput Chem 25: 872–880, 2004

Key words: fragment molecular orbital (FMO); GAMESS; DDI; parallel

Introduction

Application of parallel programming has turned out to be a boon to quantum chemistry, greatly broadening the areas in which it can be practically applied. The early development of a distributed data approach for performing parallel electronic structure theory calculations with the GAMESS[1] (General Atomic and Molecular Electronic Structure System) code is based on the distributed data interface (DDI) developed by Fletcher et al.,[2] initially for second-order perturbation theory energies and gradients. More recent applications of DDI include the work of Umeda et al.[3] to parallelize multiconfigurational quasi-degenerate perturbation theory (MCQDPT),[4] the distributed data SCF by Alexeev et al.,[5] full configuration interaction (FCI),[6] multiconfigurational self-consistent field (MCSCF),[7] and perturbation theory energies and gradients for open shells.[8] Although the DDI interface remains a general tool applicable to any type of calculations GAMESS can perform, its efficiency is most useful for highly correlated calculations using large basis sets on hardware with very fast CPU interconnects. For the less efficient communication fabrics, such as Fast and Gigabit Ethernet, that are common on clusters, the efficacy of the current DDI implementation decreases.

In this work we propose a general hierarchical type of interface, presently with two levels, that works by dividing all nodes into groups. The two levels correspond to intergroup processing (higher level, coarse-grained parallel, when a group operates as one unit) and intragroup (lower level, finer grained when each group mem-

Correspondence to: D. G. Fedorov; e-mail: d.g.fedorov@aist.go.jp.

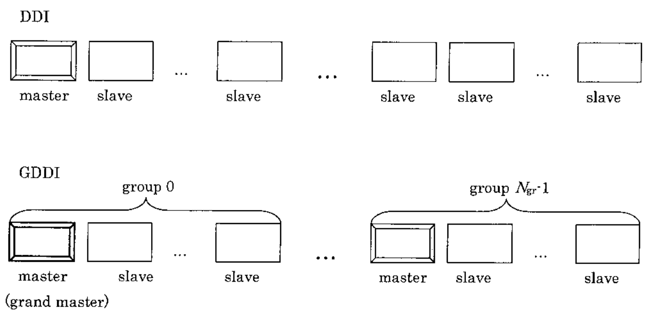

Figure 1. Comparison of notation for DDI and GDDI. Individual nodes are denoted by rectangular boxes, marked below as master and slaves.

ber constitutes one unit). This approach allows individual tasks to be assigned to groups that do them independently (or nearly so) of each other. This greatly increases efficiency, as the scalability at the group level is nearly linear. This new interface can be applied to those types of computations where it is feasible to divide the work into smaller units. A commonly used parallel library MPI[9] also has the capability to divide nodes into groups; however, it does not provide the means to divide the work load between groups dynamically. That is, MPI is also a hierarchical approach but is static in nature. At present, two types of calculations can be performed with generalized DDI (GDDI): numerical derivatives and the fragment molecular orbital method (FMO) developed by Kitaura et al.[10,11] An implementation of the FMO method based on the MPI static two-level hierarchy is also available in the ABINIT-MP program.[12] FMO is a powerful method enabling quantum-mechanical calculations of large molecules, and it has been applied to molecules containing up to 4000 atoms.[13] However, the ability to handle such large systems incurs high computational costs and requires massively parallel computers and programs utilizing them with high efficiency. The hierarchical scheme proposed in this work may be easily applied to the FMO method.

Numeric derivatives, while potentially subject to numerical inaccuracy, remain an important tool in quantum chemistry, as it is commonplace to develop energy functions significantly ahead of the implementation of analytic gradients. Especially for the most sophisticated computational methods, the derivation and coding of analytic derivatives is a challenge that can take years to accomplish. Therefore, the new GDDI implementation presented here provides an efficient way to improve scalability and to flexibly use the network topology. GDDI can also be used with clusters of computer clusters.

Methodology and Implementation

The basic idea is illustrated in Figure 1. The implementation is based upon a socket library written in C for UNIX systems; there is also a MPI version with slightly reduced capabilities. In the original DDI, a given task (e.g., energy plus gradient) is divided as efficiently as possible among all available processors. In GDDI, each group is assigned a similar independent task. The task assigned to each such group may then be treated as in DDI.

In parallel computations using MPI libraries, the concept of communicator is introduced. A communicator is defined as "an MPI object that describes the communication context and an associated group of processes."[9] In other words, a subset of all nodes is assigned to a group that is labeled by an integer index (a communicator). Thereafter, one can do global sums or broadcasts within the nodes of a group by specifying its communicator. When the job starts, there is only one communicator defined that includes all available nodes and more communicators can easily be created when needed.

The MPI version of GDDI was very straightforward to construct using the MPI_COMM_SPLIT subroutine, by creating three levels of communicators: WORLD, GROUP, and MASTER. The first of these includes all compute nodes, the second includes all compute nodes in a group, and the third includes all master compute nodes. Note that WORLD as described above is not the default MPI communicator MPI_COMM_WORLD, as all three communicators are limited to compute nodes; data servers are not included (see the original DDI reference for the concepts of compute and data server processes[2]). In addition, a communicator that includes both compute and data server nodes in a group is also created. The switching from one communicator to another is accomplished by assigning one of the three communicator values obtained from calling MPI_COMM_SPLIT to a common block variable that is then used as the current communicator in all MPI calls. The current communicator environment (that is, WORLD, GROUP, or MASTER) is called the *scope.*

For the UNIX socket-based library the underlying MPI structure is emulated by storing the node IDs into arrays used in all communications, in analogy with the MPI communicators. For sockets, some work is needed to switch scopes. This includes changing the way communications that are restricted to a given scope occur, whereas for MPI communications can, in principle, occur freely by simply using a different index (communicator). Either way, all nodes within a scope must perform the same parallel communication synchronously lest a deadlock should occur; thus, there is no practical difference.

Regardless of which library is used, the outer wrapper subroutines in the library-dependent source file DDI.SRC free the quantum-mechanical part of the program from knowing all of these unnecessary details. In fact, even most of DDI.SRC does not know if DDI or GDDI is used as the changes required to the parallel routines were minimal, excluding the core part that creates and switches communicators. It is possible and useful, as shown below, to change the number of groups on the fly, to accommodate the needs of a particular step of some calculation. The implementation of this is straightforward: a global wait to synchronize groups, followed by updating node arrays that contain the group division (sockets). A similar approach to create new communicators can be followed for MPI.

After all nodes switch to the group scope, parallel communications occur independently of each other. Each group has its own synchronization points, global sums distributed memory operations, etc. This effectively localizes communications and greatly speeds up calculations on massively parallel computers. Although the original MPI provides group division as well, it is a very basic and static interface, which also localizes communications within groups. Furthermore, the available distributed memory implementation in GAMESS cannot use the original MPI group-divided nodes, whereas GDDI supports distributed memory operations localized within groups.

The basic usage of the three scopes is as follows. The program begins by running in the WORLD scope, then each group is assigned a task and the scope is switched to GROUP. After the group finishes its task, it gets another one until all tasks are completed. Next, the scope is switched either to MASTER or WORLD, and the nodes in the new scope exchange results (the choice between the two depends on the type of computation), accomplished by global sums and broadcasts. Finally, the scope is switched back to WORLD, so the whole computation may be repeated (e.g., during a geometry optimization based upon numeric gradients).

The load balancing can be chosen to be either static or dynamic. The former is a simple fixed division of work according to the work index, whereas the latter is a flexible scheme in which the computational units request the next work index upon completion of the current work. Load balancing is done at all hierarchical levels; that is, at the intergroup and intragroup levels. In the former case, the grand master keeps track of the job indices given out to groups, while in the latter case, it is the group master that does the same within its group.

It is important to stress that dynamic load balancing has overwhelming importance for parallel computations, unless only a few nodes are used. In practice, clusters can be expected to be heterogeneous, with nodes having varying capabilities. This means some nodes will finish their tasks ahead of others. Even for clusters with identical nodes, the mathematical nature of the tasks strongly affects the computational time they require, even if tasks are formally of the same size (due to, for instance, integral screening).

In the present work, a modified version of GAMESS was used in which the option to store two-electron integrals in memory was added. In most of the systems considered below the amount of memory present was sufficient to store all two-electron integrals, greatly speeding up the calculations. There was only one system (C, see below) where 0–20% (depending on the group size) of the total number of two-electron integrals were stored on disk. With this exception the amount of disk I/O was really small and limited mostly to storing one-electron integrals and the monomer densities for all fragments.

General first and second numeric derivative codes were added to GAMESS, both based on single-point energy runs with double differencing. These codes can be used to do geometry optimizations, saddle point searches, and Hessians.

All applications below are of the restricted Hartree–Fock (RHF) type. Although GDDI can be used for any type of wave function that GAMESS supports, the parallel strategy that is making the decision on how to divide nodes into groups and how to set up other options, depends upon the wave function type. Some electron correlation methods, such as MP2, have substantial memory requirements that are divided among nodes in a group using the DDI interface. This places a constraint on the minimum number of nodes per group that is generally not an issue for RHF, for instance. In addition, parallel communication overhead is often larger if electron correlation is included. Nevertheless, general trends that are easily illuminated with RHF runs are applicable to other wave function types. Because RHF does not use distributed arrays, parallelization is based on manual memory division between nodes, using global sums and broadcast routines not involving DDI specific operations, such as data distribution.

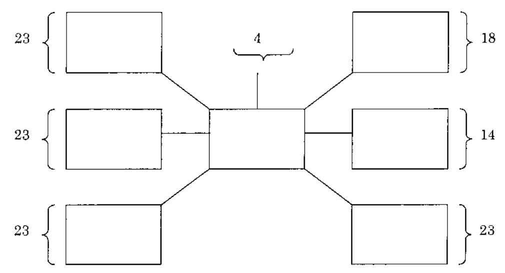

Figure 2. One hundred twenty-eight node PC cluster with star topology, connected by Fast Ethernet. Each rectangular block denotes a 24-port Fast Ethernet hub. The numbers near brackets show the number of nodes connected to the particular hub. The reason the nodes are divided in this way is that the 128 node cluster used for the tests is part of a 400-node cluster, and some ports on the above hubs are used by other clusters.

The applications discussed below use group sizes that are as close to uniform as possible, although nonuniform distributions could conceivably be more efficient in some cases. It is hard or perhaps impossible to suggest a general rule of granulating group sizes to achieve maximum performance. The actual network topology and the details of the system affect scalability in a complex way as elucidated below. It is likely that granulating group sizes may become more important issues if the number of available nodes grows beyond 1024.

Network topology can be very simply utilized with GDDI. It seems best to localize all group members within one hub (or its equivalent if a different type of network is used). Because hubs can have different numbers of nodes attached, it is often not possible to maintain a constant group size. Therefore, both automatic and manual node division have been implemented. In the former case, the user only specifies the constant group size, while in the latter case each node is manually assigned to a group. Due to global broadcasts, communication between nodes does take place and hinders performance to some extent. Finally, for SMP machines all of the CPUs on the same machine naturally should be put into the same group (possibly with addition of other nodes), as the communication within one node should be very fast.

Applications

The cluster used for the tests contained 128 nodes, each equipped with a Pentium III 1 GHz CPU, 512-MB RAM and running RedHat 7.1 Linux with the 2.4.9-31 kernel. The nodes were connected by Fast Ethernet, as shown in Figure 2. GAMESS was compiled with the 7.0 version of the Intel compiler.[14]

All timings are wall-clock and do not include system startup time taken to run rsh processes and copy the input files to corresponding nodes (this can take several minutes on 128 nodes). As indicated below, the error bounds for the timings are estimated to be less than 1%, due to slight differences in runs using dynamic load balancing and system conditions.

Fragment Molecular Orbital Method

To discuss parallelization issues, the main steps of the FMO method are briefly introduced here. A molecule (or a molecular cluster) is divided into fragments (also called monomers). A single-point RHF calculation is then performed on each fragment separately in the mean Coulomb field (electrostatic potentials, ESPs) of the other monomers. Each single point is performed by one DDI group, nearly independently of the others. One cycle of single-point monomer SCF runs generates a new density, thus changing the Coulomb field, so these cycles are repeated until convergence is reached. This generates the monomer densities that are then used during the next step, namely, the dimer single-point SCF runs, also done independently by DDI groups. During this second step dimers are constructed from each pair of monomers, and the initial density is taken to be the sum of the two monomer densities. Then, an SCF calculation is performed once for each dimer. If two monomers are well separated, an approximation is used that essentially ignores exchange and self-consistency, and only the Coulomb interaction contribution is computed. All these steps are illustrated in Figure 3.

The computation of the Coulomb interaction requires one- and two-electron integrals involving both the n-mer ($n = 1, 2$) and the external monomers, one at a time (that is, the total entity is a ($n + 1$)-mer). The calculation of ESPs is a very time-consuming step, and it takes on the order of half of the total time. Several approximations have been developed for well-separated monomers; thus, in practice, most ESPs due to such separated monomers can be computed practically without any loss of accuracy by using point charges, reducing the costs from two-electron to effectively one-electron. Finally, note that breaking bonds in a molecule when defining fragments necessitates the computation of one-electron projection operator contributions. They are parallelized in a similar manner to other one-electron integrals, and require little time to compute; thus, they are not mentioned in the discussion below.

Returning to Figure 3, after initialization, the preparation of starting orbitals begins by a load-balancing scheme (ILB1) that assigns one fragment to one DDI group. As noted above, load balancing can be static or dynamic. Next, a group divides the work for one fragment via the ILB2 step. This includes computation of one-electron integrals, performed by each node (denoted by "W" in Fig. 3). After all work at the intragroup level is finished, data exchange (IX2: global sums of one-electron integrals) occurs. Finally, after all monomers have been treated, data exchange (IX1: exchange of monomer densities) is done. There is other minor communication (indexing arrays, interfragment distances, etc.), part of which involves only exchange between masters.

The other two steps, monomer SCF and single point dimer SCF runs, have exactly the same structure, with some differences in the type of work done (e.g., the addition of two-electron integrals, ESPs are computed once before each SCF). MX2 and DX2 thus involve global sums of ESPs, one-electron integrals, and Fock matrices; MX1 involves density and energy exchange (global

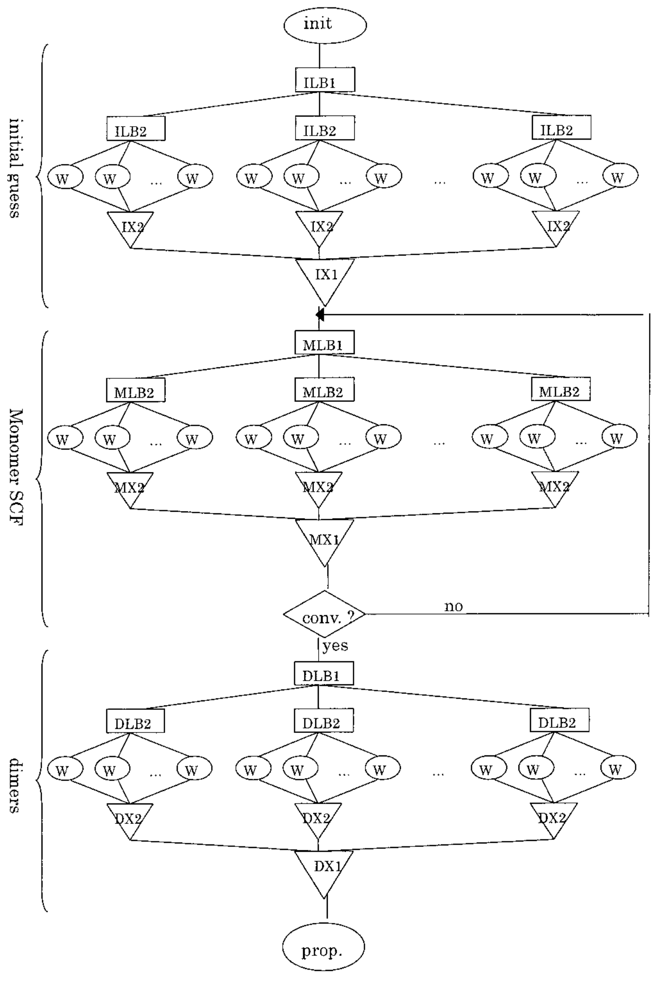

Figure 3. The diagram for the GDDI parallelization of the FMO method. "init" stands for initialization, *Y*LB*n* is the n-th level load balancing of Y = I initial guess, Y = M monomer SCF, Y = D dimer single point runs, n = 1 (intergroup) and 2 (intragroup). For all n = 2 parts the diagram is simplified not showing loops occurring during those steps (such as, e.g., SCF iterations). "W" is some work performed by one node (typically, some fraction of integrals). *Y*Xn is first (n = 1) and second (n = 2) level data exchange. "conv." stands for the test of convergence of the monomer SCF. "prop." stand for properties, currently computed (redundantly) by all group masters. The load-balancing part of the diagram shows the total load that is divided among available nodes, whose number is arbitrary.

convergence is based on energy). DX1 only involves dimer energy exchange. Finally, properties are computed (sequentially).

Now consider the general trends in this scheme, followed by timings for actual computations. Regarding the strategy for dividing nodes into groups, the following points are important: (a) the smaller the group size, the faster the intragroup runs (the more groups the better); (b) the larger the number of groups, the longer the synchronization wait at the exchange points IX1 and MX1. As explained below, DX1 is less affected (fewer groups is better); (c) the fewer the number of groups, the faster the density exchange (especially on networks with large latency, such as Fast and Gigabit Ethernets) (fewer groups is better).

The last point arises from the fact that the density exchange works through a broadcast to *all* nodes by a group master of all densities its group computed. Although the total amount of data is independent of the number of groups (it is equal to the sum of density sizes for all fragments, plus Mulliken charges and populations), due to latency it is faster to broadcast one large block compared to two smaller ones with the same total size.

The reason DX1 is less affected by point (b) above is as follows. The number of dimers is generally many times larger than the number of DDI groups, because the number of dimers is $N(N\text{-}1)/2$, where N is the number of monomers. Then, the work load balancing (see below) changes the order in which dimers are done in such a way that separated dimers (that are treated with an approximation and take little time) are done at the end, so that there is little wait at DX1.

The balance between points (a)–(c) is what determines the node division strategy, and both depend on the network type and topology.

To reduce waiting time at the synchronization points IX1, MX1, and DX1, load balancing (both static and dynamic) is implemented in the descending order of the computational work load. The latter is determined by: (a) n-mer basis set size ($n = 1, 2$), (b) usage of dynamic correlation, (c) applicability of the separated dimer approximation, and (d) in case of the initial guess, the lower triangular matrix of interfragment distances is computed at the same time and it usually requires more time than the Hückel guess; therefore, the lower part of the triangular matrix takes more time (one row is computed for each fragment).

It is easy to take all these points into account and assign a work load weight to each n-mer. By doing the work in descending order, the wait is reduced due to faster final runs before the synchronization.

Parallel performance can be fine-tuned (with input file options) as follows: (a) division of nodes into groups (implemented independently for each of the three major steps; group division is changed on the fly); (b) the choice of static/dynamic load balancing at each level (inter- and intragroup); (c) the choice of the SCF type, which is direct or conventional (for the sake of sensible analysis we used conventional SCF during all tests; however, for large group sizes direct SCF scales better and should be considered for practical applications); and (d) other fine tuning, such as whether to divide the MLB2, DLB2 work over fragments or shells during ESP calculations.

Although we find that dynamic load balancing definitely wins in general at the intergroup level, static load balancing can be faster at the intragroup level, especially on networks with large latency, due to significant communication to obtain the job index. Small basis sets seem to prefer static load balancing at the intragroup level. The choice of fragment or shell ESP parallelization is determined by the ratio of the number of fragments that are computed without approximations to the number of group members. If this ratio is large (roughly = 5–10), then the fragment option is faster, due to reduced communication.

The following systems were used for parallel tests: (A) $(H_2O)_{256}$ FMO-RHF/6-31G*, divided into 64 fragments; (B) $(H_2O)_{1024}$ FMO-RHF/STO-3G, divided into 128 fragments; and (C) lysozyme tri[N-acetyl-D-glucosamine] complex, 2036 atoms, FMO-RHF/STO-3G, divided into 128 fragments.

Table 1. The Division of N_{nod} Nodes among N_{gr} Groups, the Number of Groups M_k Is Given for the Group Consisting of k Nodes

N_{nod}	N_{gr}	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
64	17		4	13					
128	17			1			6	8	2
128	28		4	4	20				
128	34		8	26					
128	45	7	38						

All systems were run with conventional RHF and tightened accuracy (SCF convergence 10^{-7}, monomer SCF convergence 10^{-7}, and two-electron integral accuracy 10^{-12}). Hückel orbitals were used for the initial guess and Cartesian d functions (6d option) were used (ISPHER = −1). The fragment size was uniform for tests A and B, and varied (one residue per fragment) for test C. Due to cluster topology, it was not possible to use group division with uniform size for 64 nodes and 128 nodes (it is not necessarily desired in any case). Therefore, the manual group division option was used with varying group size given in detail in Table 1. All nodes within a group were always connected to the same hub.

For each system, several sensible options (such as altering the node division among groups while fixing the total number of nodes) were considered. The fastest option is presented below. The general trends are explained for each system, and options used on 128 nodes are clarified. The superlinear scaling that is observed in some cases (verified on different clusters) must come from either CPU or system-level caching that was not explicitly controlled.

Initialization and initial guess are included in the monomer timings and scalabilities. The initial guess takes 7, 17, and 16 s on one node for the systems A, B, and C, respectively, so it is unnecessary to discuss its scalability separately. On 128 nodes the timings are, respectively, 15, 20, and 20 s. This is despite the gain from dividing work among nodes. The data exchange at IX1 makes the overall timing worse on 128 nodes compared to 1 on Fast Ethernet. For system A, there is some loss of efficiency due to the single point initial guess scalability, because of larger group size.

Summarizing general trends prior to considering the details, in all cases the scalability of monomer SCF is somewhat low. This is explained by large communication to exchange density between nodes and the long wait at the synchronization point MX1. In contrast, the dimer efficiency is high. This is because there is little communication cost and smaller group size. Using fewer nodes per group during monomer runs reduces the efficiency due to the increased wait that overcomes the efficiency boost because of better intragroup scaling. *The necessity to strive to form very small groups, thereby increasing the wait at MX1, is, however, the sole consequence of using a slow network.* It should also be noted that dimer runs can also be saturated, meaning the group size has to be decreased if the ratio of the number of dimers to the number of groups is not large enough (as observed in case A on 128 nodes). In reality, it is not just a simple ratio but a more complicated

Table 2. Analysis of the Parallellization Costs: Dependence of the Average Wait and Data Exchange Time in Seconds at the MX1 Step, for One Monomer SCF Iteration (the Total Time for All Iterations Is Shown in Parentheses).

	System A ($N_D = 192896$, $N_B = 76$)			System B ($N_D = 214528$, $N_B = 56$)			System C ($N_D = 217795$, $N_B = 28 \ldots 94$)		
N_{nod}	$N_{\text{gr,m}}$	Wait	Exchange	$N_{\text{gr,m}}$	Wait	Exchange	$N_{\text{gr,m}}$	Wait	Exchange
1	1	0.0 (0.)	0.0 (0.)	1	0.0 (0.)	0.2 (1.)	1	0.0 (0.)	0.0 (0.)
2	2	4.2 (33.)	0.4 (3.)	2	7.7 (46.)	0.0 (0.)	2	0.2 (4.)	0.1 (2.)
4	4	5.5 (44.)	0.5 (4.)	4	18.3 (110.)	0.3 (2.)	4	4.0 (68.)	0.5 (9.)
8	8	13.8 (110.)	0.7 (5.)	8	15.4 (92.)	0.6 (4.)	8	2.1 (35.)	0.6 (11.)
16	8	5.7 (45.)	1.1 (8.)	16	8.5 (51.)	0.9 (6.)	16	4.2 (71.)	0.8 (13.)
32	16	14.0 (112.)	3.3 (26.)	16	8.1 (49.)	3.9 (23.)	16	2.0 (33.)	4.1 (69.)
64	17	3.9 (31.)	5.3 (42)	17	6.4 (38.)	6.2 (37.)	17	1.6 (27.)	5.9 (100.)
128	17	2.7 (22.)	9.6 (77.)	28	4.5 (27.)	12.4 (74.)	34	3.3 (56.)	14.1 (239.)

Systems A, B, and C are defined in the text. N_{nod} is the total number of nodes and $N_{\text{gr,m}}$ is the number of groups during the monomer SCF. N_D is the amount of data transferred at each iteration, in words (1 word = 8 bytes). N_B is the fragment basis set size.

relation involving the number of SCF and separated dimer calculations.

Table 2 provides a detailed analysis of the parallelization costs for the monomer SCF. Comparing the results for two nodes, note that system C has shorter waiting times than A and B. This is because system C fragments have varying sizes that are processed in decreasing order. So, the wait time is small because small fragments are done last. It appears to be typical to have such "coarse-grained" performance on a few nodes, when $N_{\text{nod}} - 1$ nodes wait for one node to finish. Wait times for system C are consistently shorter than those for the other two systems (with one small exception). The fragment basis set size is larger for system A; therefore, one often observes longer wait times for the same number of groups, compared to systems B and C.

It is interesting to notice that in almost all cases the wait time exceeds the exchange time, except for 64 and 128 nodes. No doubt the exchange time could improve with an improved network topology, for example, with more direct access between nodes that are now localized at different hubs. It can be computed from data presented below for individual systems that the total direct parallel communication on 128 nodes at the MX1 point consumes 38, 28, and 33% of the monomer SCF wall time for systems A, B, and C, respectively. Without this overhead, the monomer SCF scalability would have been 82, 95, and 94%, respectively. The scaling is less for system A due to the larger group size. Other parallel communication occurs at the MX2 point, within single-point SCF calculations. In contrast, parallel communication for dimers is nearly negligible, if the group size of 1 is a sensible option and is usually quite small otherwise.

Note that the number of fragments into which a given molecule is divided has a dramatic effect on scalability. This number is usually determined from physical rather than computational considerations, as increasing the number of fragments decreases the accuracy of the FMO method. It is both customary and physically reasonable to cut proteins so that amino acid residues are not cut internally (e.g., by assigning one residue per fragment). For molecular clusters it is natural to assign a fixed number of molecules per fragment.

The scalability results for system A are given in Table 3. The scalabilities S_A in Table 3 and subsequent tables refer to the wall time ratio $S_A(n) = t_A(1)/(n^*t_A(n))$, where n is the number of nodes and A is either "mon" or "tot," for monomer SCF and total, respectively, so that ideal linear scaling is equal to one. Parallel efficiency remains high up to eight nodes. Thereafter, the longer wait at MX1 forces the group size to be set to 2, 2, 4, and 8 for the monomer SCF calculation, on 16, 32, 64, and 128 nodes, respec-

Table 3. Scalability Tests for $(H_2O)_{256}$ FMO-RHF/6-31G*, Evenly Divided into 64 Fragments, on the 7-Hub Star-Topology Fast Ethernet PC Cluster.

N_{nod}	$N_{\text{gr,m}}$	P_{ESP}	t_{mon}, s	S_{mon}	t_{tot}, s	S_{tot}
1	1	—	14602.2	1	87818.9	1
2	2	—	7225.0	1.0105	43493.1	1.0096
4	4	—	3673.0	0.9939	21760.6	1.0089
8	8	—	1912.3	0.9545	10959.4	1.0016
16	8	S	1023.4	0.8918	5575.0	0.9845
32	16	S	596.5	0.7649	2885.8	0.9509
64	17	S	354.6	0.6434	1544.3	0.8885
128	17[b]	S	258.6	0.4411	876.8	0.7825

N_{nod} is the number of nodes, equal to $N_{\text{gr,d}}$, the number of groups during the dimer runs, $N_{\text{gr,m}}$ is the number of groups during the monomer SCF. The load balancing was dynamic at the ILB1, MLB1, DLB1, and static at the ILB2, MLB2, and DLB2 levels. P_{ESP} stands for parallelization of ESPs in MLB2 and DLB2:—(none: due to one node in a group), S (over shells), and F (over fragments). t_{mon} and t_{tot} are the wall clock monomer SCF[a] and total times, respectively. S_{mon} and S_{tot} are the monomer SCF and total scalabilities, respectively. See caption to Figure 3 for the FMO step notation.

[a]Initial guess is included into monomer timings throughout all results.

[b]The number of dimer groups was set to 45.

Table 4. Comparison of DDI vs. GDDI Scalability Tests for $(H_2O)_{256}$ FMO-RHF/6-31G*, Evenly Divided into 64 Fragments, on the 7-Hub Star-Topology Fast Ethernet PC Cluster (DDI is the First Row in the Table, $N_{gr} = 1$)

N_{nod}	N_{gr}	LB	t_{mon}, *s*	S_{mon}	t_{tot}, *s*	S_{tot}
16	1	D	1516.8	0.6017	9616.6	0.5707
16	2	D	1177.6	0.7750	7315.2	0.7503
16	4	S	1055.4	0.8647	6218.9	0.8826
16	8	D	1009.9	0.9037	5777.9	0.9499
16	16	—	1066.9	0.8554	5614.7	0.9776

N_{gr} is the number of groups, the same during both monomer and dimer steps. The number of nodes N_{nod} is fixed and equal to 16, shown to avoid confusion. LB denotes load balancing within each group (*S* for static, *D* for dynamic and none for a group with one node). See caption to Table 3 for other notation.

tively. On the other hand, the group size was fixed at 1 for the dimers, except for 128 nodes where the group size was 3. Shell parallelization is preferred over fragment parallelization, due to the small number of fragments that are computed without approximations. The low efficiency of monomer SCF calculations at 128 nodes comes from the low efficiency of single point SCF runs (including ESPs) due to the slow network. As can be seen by subtracting the monomer SCF part, the scalability of the rest of the code, even on 128 nodes, is 0.9252. This is actually a bit low, because groups with average size of 3 were used during the dimer runs, as explained above.

The efficiency of GDDI vs. DDI is presented in Table 4. DDI corresponds to having just one GDDI group, and for all cases both static and dynamic load balancing was tried and the fastest timing is shown (for large groups dynamic load balancing within a group somewhat suffers from high latency of FastEthernet). GDDI outperforms DDI in all cases. In the best set of options (given in Table 3, for 16 nodes), the total scalability with GDDI is 0.9845, whereas the DDI scalability is only 0.5707. It is instructive to observe the drop in monomer parallel efficiency when going from 8 to 16 groups. This is due to the increased wait at MX1, as discussed above. This difference becomes dramatic when then number of nodes increases further, as shown in Table 5. *For 128 nodes the scalability of the standard DDI becomes disastrous:* it is faster to run on 1 node than on 128, and the scalability of DDI is merely

Table 5. Comparison of DDI vs. GDDI Scalability Tests for $(H_2O)_{256}$ FMO-RHF/6-31G*, Evenly Divided into 64 Fragments, on the 7-Hub Star-Topology Fast Ethernet PC Cluster (using Static Load Balancing).

N_{nod}	N_{gr}	t_{iter}, *s*	S_{iter}
1	1	1892.5	1
128	1	2963.8	0.0049
128	17	32.0	0.4620

See caption to for other notation. t_{iter} and S_{iter} are timings and scalability for the first monomer SCF iteration (S_{iter} is a very good estimate of S_{mon}). See caption of Table 4 for other notation.

Table 6. Scalability Tests for $(H_2O)_{1024}$ FMO-RHF/STO-3G, Evenly Divided into 128 Fragments, on the 7-Hub Star-Topology Fast Ethernet PC Cluster.

N_{nod}	$N_{gr,m}$	P_{ESP}	t_{mon}, *s*	S_{mon}	t_{tot}, *s*	S_{tot}
1	1	—	25,945.6	1	201,819.5	1
2	2	—	13,249.9	0.9791	102,192.0	0.9875
4	4	—	6603.5	0.9823	50,738.6	0.9944
8	8	—	3351.0	0.9678	25,489.9	0.9897
16	16	—	1753.0	0.9250	12,821.1	0.9838
32	16	F	955.5	0.8486	6494.7	0.9711
64	17	S	542.1	0.7478	3312.6	0.9520
128	28	S	355.2	0.5707	1742.7	0.9048

See caption to Table 3 for notation.

0.0049. GDDI, on the other hand, shows scalability of 0.4620; not too high, but acceptable. The timing of GDDI is 93 times faster than DDI at the same number of nodes (128) and the same input, other than the group division. Of course, in practice, one would only grow the number of nodes under DDI in accord with the efficiency that hardware and the computational methods provide. Nonetheless, these examples emphasize the great utility of GDDI, even when DDI is not very useful.

Because dynamic load balancing is used throughout, small changes in timings are observed with every run. As can be seen, the t_{mon} timings for 16 nodes, eight groups in Table 3 and Table 4 differ by less than 1%. This is a typical error bound for the timings (note that t_{tot} for the aforementioned entries differs because the dimers are run with 16 and 8 groups, respectively).

The results for system B are summarized in Table 6. The trends are similar to those observed for system A. Due to a larger number of fragments (128 vs. 64), one can increase the group size to 16 nodes. Similarly, at 128 nodes the average group size is 4.5, increasing the scalability to 0.5707. Likewise, fragment parallelization of ESPs is slightly (by 21 s, not shown) preferred to shell parallelization, due to the increased number of fragments. Shell parallelization is preferred for larger group sizes. The total scalability on 128 nodes without including the monomer SCF calculation is 0.9902. The total scalability for system B is better than that for system A. In part, this is due to the decreased fraction of time the monomer SCF calculation takes compared to the total time (13 vs. 17%), because it takes fewer monomer SCF iterations to converge (6 vs. 8).

Table 7 summarizes the results for system C. The node division strategy closely resembles that for system B, because the two systems are similar in size and in the number of fragments. Shell parallelization is faster than fragment parallelization on 32 nodes, by 105 s. This may be due in part to varying fragment size (load balancing according to size is not implemented at present inside of ESPs) and possibly to the smaller number of fragments treated without approximation because of the larger spatial dimensions of the molecule. The overall scalability for C is lower than that for B, due to the increased fraction of the monomer SCF calculation in the total time: 30 vs. 13%. This reflects, at least in part, the 17 iterations it took to converge. The total scalability at 128 nodes excluding the monomer SCF times is 99.5%. It is typical to take 15–25 iterations to converge the monomer SCF calculation vs. 5–10

Table 7. Scalability Tests for Lyso (2036 Atoms) FMO-RHF/STO-3G, Divided into 128 Fragments, on the 7-Hub Star-Topology Fast Ethernet PC Cluster.

N_{nod}	$N_{gr,m}$	P_{ESP}	t_{mon}, *s*	S_{mon}	t_{tot}, *s*	S_{tot}
1	1	—	69,264.6	1	229,925.1	1
2	2	—	34,792.3	0.9954	115,430.4	0.9959
4	4	—	17,438.7	0.9930	57,684.5	0.9965
8	8	—	8802.6	0.9836	28,958.7	0.9925
16	16	—	4434.6	0.9762	14,484.5	0.9921
32	16	S	2348.2	0.9218	7385.0	0.9729
64	17	S	1297.9	0.8339	3815.0	0.9417
128	34	S	893.4	0.6057	2154.9	0.8336

See caption to Table 3 for notation.

iterations to converge molecular clusters. Thus, in general, the scalability for single molecules is lower compared to molecular clusters.

Data presented in Table 8 demonstrate the importance of the dynamic load balancing and job scheduling (altering the execution order to reduce parallelization costs). It is seen that dynamic load balancing is always the best strategy, and its importance grows with the number of nodes. Approximately equally important is job reordering according to the job size. While for eight nodes the combined effect of the two factors is 5–12% (in terms of scalability), the effect grows with the number of nodes and reaches 27–36%, which in the most pronounced case of system C on 64 nodes translates into the total timing about 1.6 times slower if the two options (dynamic load balancing and job reordering) are not used. It should be emphasized that both options become vital to good scalability on clusters with mixed nodes; but even on identical nodes they significantly affect the performance.

Finally, we note in passing that there is a factor altering the parallel performance that was not very relevant for the systems considered in this work. If large basis sets (and large fragments) are considered, two-electron integrals may not fit into memory, and have to be partially stored on disk. Using larger group size one can manage to fit all integrals in memory, reducing idle CPU time waiting for I/O. Correspondingly, the benefits of in-memory runs will add to the considerations of the most efficient group size and they will compete with other factors reducing performance for large groups.

Table 8. The Total Scalability (S_{tot}) Dependence upon Load Balancing (LB) Type at the Upper (Intergroup) Level: Dynamic (D), Static (S) with the Possible Addition of Job Reordering According to the Subtask Size (Indicated by + Where Used).

N_{nod}	LB	*A*, S_{tot}	*B*, S_{tot}	*C*, S_{tot}
	D+	1.0016	0.9897	0.9925
8	S+	0.9534	0.9753	0.9552
	S	0.9445	0.9688	0.8790
	D+	0.9845	0.9838	0.9921
16	S+	0.9307	0.9548	0.9010
	S	0.8548	0.9198	0.7074
	D+	0.9510	0.9711	0.9729
32	S+	0.8644	0.9373	0.8538
	S	0.8184	0.8300	0.6793
	D+	0.8885	0.9520	0.9417
64	S+	0.5240	0.8853	0.7771
	S	0.6105	0.7336	0.5876

Molecular systems A, B, and C are described in the main text.

Table 9. Scalability Tests for $(H_2O)_{64}$ RHF 6-31G* (Numeric Gradient), on the 7-Hub Star-Topology Fast Ethernet PC Cluster.

$N_{nod} = N_{gr}$	t_{tot}, *s*	S_{tot}
1	51,831.7	1
2	25,697.3	1.0085
4	12,826.2	1.0103
8	6461.4	1.0027
16	3244.2	0.9985
32	1678.1	0.9652
64	900.5	0.8994
128	579.8	0.6984

t_{tot} is the total time and S_{tot} is the total scalability.

Numeric Derivatives

Parallel tests employed the $(H_2O)_{64}$ system, with RHF/6-31G* two-point numeric gradient (naturally divided into 64*3*3*2 + 1 = 1155 single-point runs), using dynamic load balancing. Although a RHF analytic gradient is available, this example is a convenient tool for testing the numeric gradient scalability. The results are presented in Table 9. In all cases the most effective strategy is to create groups of size 1.

The scalability remains high as there is no intragroup communication, and it is hindered only by the wait at the final data exchange points, where $N_{nod} - 1$ nodes wait for the last node to finish its single energy calculation. If the number of nodes increases further, it is quite likely that a group size equal to 2 or larger will be preferred.

In this case, there is also nearly no difference between dynamic and static load balancing. This arises because there is a single synchronization point at the end, and because every single-point run performed by each group differs very little from one another, physically and computationally, even when integral screening and other geometry dependent factors are taken into account. Such behavior is different from FMO runs of molecular clusters and large molecules. For instance, for the same $(H_2O)_{64}$ cluster using the FMO method, each run deals with a different H_2O molecule that feels a physically and computationally different potential, whereas in case of the numeric derivatives it is a slightly displaced total molecule that is computed.

Conclusions

A general approach to efficiently use network topology has been proposed, based on a two level hierarchical scheme. Practical implementations utilizing this approach are at present limited to the FMO method and numeric derivatives. A detailed analysis of the parallel implementation and its performance was given for both. The new approach was found to drastically outperform the standard DDI scheme, if many nodes are used (93 times on 128 nodes) for SCF calculations. Future work will be concerned with adding electron correlation to the FMO method and studying its scalability, as well as improving the inter-group communication paradigm.

Some problems remain to be solved in the parallel interface itself. One of these is the ability to survive sudden death of a node without interrupting the calculation. There is also a need to accommodate computers of different memory size (at present all are forced to use the same amount). It is likely that a smarter load balancing scheme will be required for electron correlation, as then the memory and computational requirements grow significantly.

Because the current implementation is based on either MPI or socket libraries, both available on most UNIX systems, and can be extended to most other types of computers, it is expected that this new approach will be readily available for efficient use on massively parallel computers.

Acknowledgments

All benchmark tests were performed on the PC cluster in Osaka Prefecture University. This work was supported in part by a Department of Energy Scientific Discovery through Advanced Computing (SciDAC) grant to MSG, administered by the Ames Laboratory. We would like to thank Dr. Michael W. Schmidt and Dr. Tamio Ikeshoji for helpful discussions and assistance.

References

1. Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S.; Windus, T. L.; Dupuis, M.; Montgomery, J. A., Jr. J Comp Chem 1993, 14, 1347.
2. Fletcher, G. D.; Schmidt, M. W.; Gordon, M. S. Adv Chem Phys 1999, 110, 267; Fletcher, G. D.; Schmidt, M. W.; Bode, B. M.; Gordon, M. S. Comp Phys Commun 2000, 128, 190.
3. Umeda, H.; Koseki, S.; Nagashima, U.; Schmidt, M. W. J Comp Chem 2001, 22, 1243.
4. Nakano, H. J Chem Phys 1993, 99, 7983.
5. Alexeev, U.; Kendall, R. A.; Gordon, M. S. Comp Phys Commun 2002, 143, 69.
6. Gan, Z.; Alexeev, U.; Gordon, M. S.; Kendall, R. A. J Chem Phys 2003, 119, 47.
7. Fletcher, G. D.; Schmidt, M. W.; Gordon, M. S., manuscript in preparation.
8. Aikens, C.; Gordon, M. S., J Phys Chem A in press.
9. Snir, M.; Otto, S.; Huss–Lederman, S.; Walker, D.; Dongarra, J. MPI—The Complete Reference, Vol. 1; The MPI Core; MIT Press: Cambridge, MA, 1998. An online MPI reference can be downloaded from: http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245380.pdf.
10. Kitaura, K.; Sugiki, S.; Nakano, T.; Komeiji, Y.; Uebayasi, M. Chem Phys Lett 2001, 336, 163.
11. Nakano, T.; Kaminuma, T.; Sato, T.; Fukuzawa, K.; Akiyama, Y.; Uebayasi, M.; Kitaura, K. Chem Phys Lett 2002, 351, 475.
12. Sato, T.; Akiyama, Y.; Nakano, T.; Uebayasi, M.; Kitaura, K. IPSJ Transaction on High Performance Computing Systems 2000, 41 No. SIG5, 104 (in Japanese); http://moldb.nihs.go.jp/abinitmp.
13. Fukuzawa, K.; Kitaura, K.; Uebayasi, M.; Nakata, K.; Kaminuma, T.; Nakano, T., to be submitted.
14. http://www.intel.com/software/products/compilers/flin/noncom.htm.