Hybrid programming in high performance scientific computing

by

Jonathan Lee Bentz

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ricky Kendall, Co-major Professor
Robyn Lutz, Co-major Professor
Mark Gordon

Iowa State University

Ames, Iowa

2006

Copyright © Jonathan Lee Bentz, 2006. All rights reserved.

UMI Number: 1439837

UMI Microform 1439837
Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. Introduction

1.1 Scientific computing

The advent of the computer has revolutionized the practice of science. The use of high-performance supercomputers allows scientific calculations to be performed on a previously unthinkable scale. Supercomputers tackle extremely large computational problems such as sequencing genomes, modeling sophisticated weather patterns, simulating nuclear explosions, calculating molecular properties, etc.

One of the most prevalent supercomputer architectures consists of using multiple nodes of symmetric multi-processor (SMP) machines, where all the processors resident on a node have direct access to the local memory on that node. Each node is then connected to a high-performance network. To access memory in an inter-node fashion, the data is transferred via the network. The popularity of the SMP cluster is due in no small part to its modularity. Increasing the size of the cluster (as well as increasing the computational power) requires only adding more nodes and incorporating the additional nodes into the existing network. SMP clusters come in a variety of sizes, i.e., there is a good deal of variability regarding the number of nodes in the cluster, as well as the number of processors per node. SMP clusters can be constructed from commodity hardware components or specially designed for large scientific computer centers.

1.1.1 Memory management and data movement

1.1.1.1 Message Passing Interface

To achieve practical performance on parallel computers one has to have an efficient method for transferring data between and among nodes. One of the most popular mechanisms for data movement in parallel computing is the Message Passing Interface (MPI) [1]. MPI is a standard which defines mechanisms for parallel data movement; currently most computer vendors provide an MPI implementation to facilitate this data movement. MPI is a process based message passing library. Each process operates independently of all other processes and has access to its own local memory. When data needs to be transferred from process A to process B, process A calls a "send" function and process B calls a "receive" function. MPI can be used to send messages between processes on the same physical node, or between processes on different nodes. An advantage of MPI is that the user has direct control of memory which offers the opportunity to highly optimize the communication of data. A disadvantage also stems from the high level of control in that the user *must* control all the data movement explicitly. This can lead to a more difficult programming endeavor.

1.1.1.2 OpenMP

OpenMP [2] is a mechanism for shared memory parallelism, i.e., utilizing multiple processors on an SMP. OpenMP consists of compiler directives which are inserted into existing code to divide up the computation steps and distribute the work via threads. OpenMP is primarily a loop-based parallelism scheme. One of its main forms of parallelism is splitting a loop into equal sized pieces of work and distributing the pieces to all the threads available. While MPI can be used on any size system (multiple SMP nodes), OpenMP can *only* be used to distribute work among one node as it relies on direct access to memory. Because explicit memory management is not required when using OpenMP, the programming effort is normally much less than when writing MPI code.

1.1.2 Multi-level parallelism

The popularity of supercomputers in the form of SMP clusters provides an opportunity for implementing unique methods of data movement. All the processes on a node have direct access to the memory of that node and as such a model like OpenMP, which already allows direct access to memory, is an attractive choice for parallel programming. However, OpenMP cannot be used to share data among multiple nodes, so an inter-node data movement mechanism is required, such as MPI.

One would like to write parallel programs which can be executed on large numbers of nodes, and this requires an efficient message passing library (such as MPI). One would also like to access the shared memory of an SMP node directly (as in OpenMP) instead of calling functions to handle data access explicitly. The motivation for writing hybrid code is to optimize both the inter and intra node communication of data. A number of studies have been done combining MPI and OpenMP to implement a so-called hybrid or multi-level parallelism [3, 4, 5, 6, 7, 8]. The most common hybrid model is the so-called "master/worker" model. One MPI process is executed per node (irrespective of the number of physical processors per node) and that process spawns a master thread and a number of worker threads as well (commonly the number of threads is set equal to the number of physical processors on the node). The MPI function calls are only executed by the master thread, and when MPI calls are being executed, the rest of the threads are sleeping. Once any remote data required for a particular phase of the algorithm is obtained, the master thread on each node then partitions the work evenly between the threads. The clear performance penalty paid for this model is that of sleeping threads not doing anything during inter node communication. Also, if the network is fast enough, during inter node communication the master thread might not be able to saturate the bandwidth of the network using only one thread. If multiple threads were communicating across the network, it could be used more efficiently, but in the master/worker model this is not normally considered an option.

To alleviate the problem of sleeping threads wasting CPU cycles, one can also augment the master/worker model by overlapping communication and computation among the threads

in a node. This would allow one (or possibly more) threads to be in a communication phase and the rest of the threads could be executing computations that do not depend on the data being communicated across the network. This is suggested as being one of the better solutions to utilize system resources in the most efficient manner and obtain the best performance from the machine. However, this type of algorithm is notoriously difficult to implement from a programming standpoint.

The studies cited above showed that the choice of hybrid programming vs. pure MPI programming is highly dependent on both the machine architecture and the structure of the algorithm. If the algorithm has multi-level structure inherent in it, it may be a good candidate for a hybrid parallel algorithm. The network performance impacts the choice of algorithms as well. This is somewhat problematic as one would like to design one algorithm which will perform well on a variety of machine architectures. The algorithm we present in this work necessarily uses shared memory not because of any preconception about network architecture, but because of the large memory requirements that are associated with it [9]. The only assumption made in designing the algorithm is that, for large input sizes, it will be run on a cluster of SMPs.

1.2 Computational chemistry in GAMESS

One of the scientific disciplines in which computational technology has made a tremendous impact is the realm of chemistry. Computational chemistry has evolved into a discipline of chemistry in its own right, as an extension of theoretical chemistry. Computational chemistry is used both as a predictive and a confirmative method. Predictions about molecules can be made and these can, in some cases, be tested via experiment. Conversely, experimentalists may achieve some unusual and/or unexpected results in the laboratory and through computational chemistry calculations, obtain a more complete understanding of the chemical phenomena taking place. There exist computational chemistry software packages appropriate for a desktop workstation and packages used on the largest of supercomputers. Computational chemistry has traditionally been one of the scientific fields in which supercomputers play a large role

since many chemistry algorithms are resource intensive, with algorithms that stress the CPU, memory, disk I/O and network. The ever-increasing computational power and architectural complexity of supercomputers brings with it the opportunity and challenge of designing and implementing efficient algorithms to properly utilize the most advanced systems.

This work uses one of the most widely used, free computational chemistry software packages available, The General Atomic and Molecular Electronic Structure System (GAMESS) [10]. GAMESS is developed and maintained by Mark Gordon's research group at Iowa State University [11] and runs on almost all current computer architectures. The chemistry code is written in FORTRAN 77, and the communication and memory management routines are written in C. At the time of this writing, there are approximately 22,800 registered users/groups of GAMESS [12]. The number of actual users is most certainly higher as one license for GAMESS is generally used for an entire research group or organization.

1.2.1 Distributed Data Interface

The parallel implementation of GAMESS is built on a communication library written specifically for GAMESS, called the Distributed Data Interface (hereafter abbreviated DDI[1]) [13, 14]. It is a process based approach, with two levels of memory. Each process has access to a specified amount of local memory (invisible to all other processes), which is termed "replicated memory." All processes also have indirect access (via function calls) to what is termed "distributed memory" or global memory. Figure 1.1 shows an example of the memory hierarchy of a small system. The distributed memory is divided evenly among all physical nodes of the system and the amount of memory required for a particular computation is specified as an input parameter to the GAMESS program. DDI provides mechanisms for one-sided data operations such as put, get, and accumulate (see Appendix). Collective operations are also provided such as broadcast, global sum, and process synchronization. When performing these operations DDI will use system memory calls if the distributed data accessed by a process

[1]Both MPI and DDI operate with the same goals, i.e., communicating data between processes. However the developers of GAMESS have chosen not to use MPI for a variety of reasons, one being that they want GAMESS to be a stand-alone software package, not dependent on external software.

happens to reside on the same node as the process making the request. If the data is not local to the node, then the appropriate interconnect network is utilized to transfer the data between nodes. Note that in this scheme there is no support for direct access to shared-memory arrays (accessible to all processes on a node).

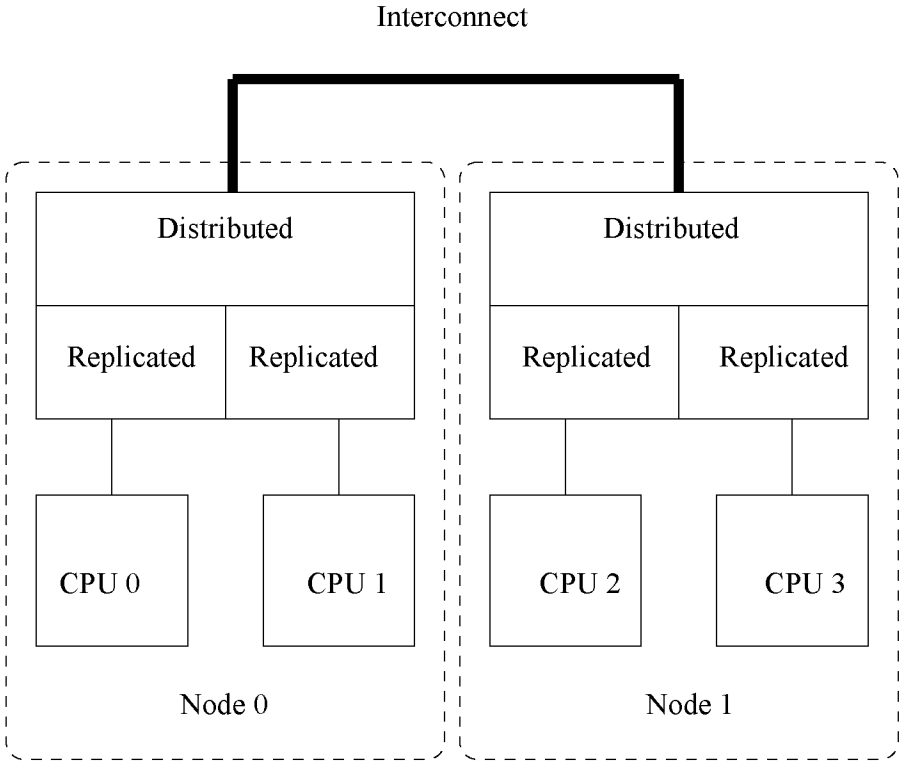

Figure 1.1 Memory hierarchy for the current DDI implementation on a two node, four processor system.

DDI provides support for two dimensional arrays. At array creation, the number of rows and columns are specified, and DDI partitions the array equally among all the processes in the system. The default partitioning scheme is to assign each process an equal number of complete columns of the array. The partitioning by columns (as opposed to rows) is used in part because FORTRAN arrays are stored in column major order. (DDI operates under the assumption that each process will be assigned at least one column of the array. If there are more processes than array columns, DDI kills the job and requests the user to specify fewer DDI processes to run the calculation.) Thus when a put/get operation is performed between

distributed memory and a local buffer, the memory pieces can remain contiguous. Access to DDI distributed arrays is indirect, i.e., to write data to a DDI array, one must have the data in a local buffer and call the DDI_PUT function to put the data into the DDI array. Similarly, to read data from a DDI array, one must call the DDI_GET function which obtains the data from distributed memory and places the data into a local buffer.

A key component of the algorithms we present in this work is the availability of three types of memory; namely the two presented above and SMP shared memory, i.e., memory on a node which any process resident to that node can access directly without the use of a function call such as put or get. Because of this limitation in DDI, we have added to DDI the support for shared memory arrays that can be directly accessed by all processes which are resident on the node. Figure 1.2 shows the memory hierarchy of this new version of DDI. Note the addition of shared memory, which is directly accessed by any processes on the node. There are mechanisms for collective operations such as broadcast and global sum, where the shared memory arrays are operated on by one process per node to avoid multiple processes attempting to write to a shared memory location simultaneously. A synchronization mechanism has also been added which allows all the processes on a node to synchronize with each other while remaining independent of processes on other nodes. It should be stated here that DDI remains a process-based communication library. DDI creates shared memory arrays that are accessible by all the processes on a node. In the parallel algorithm which we present in Chapter 3, when we refer to any shared memory operations, we are referring to the use of shared memory DDI arrays, *not* the use of any thread-based or OpenMP constructs.

1.2.2 Electronic correlation calculations

The coupled cluster method [15] is one of the most accurate *ab initio* methods available to computational chemists for calculating quantum mechanical electronic correlation energies of atoms and molecules. The formal starting point for these calculations is the electronic, time-independent Schrödinger equation

$$\hat{H}\Psi = E\Psi, \tag{1.1}$$

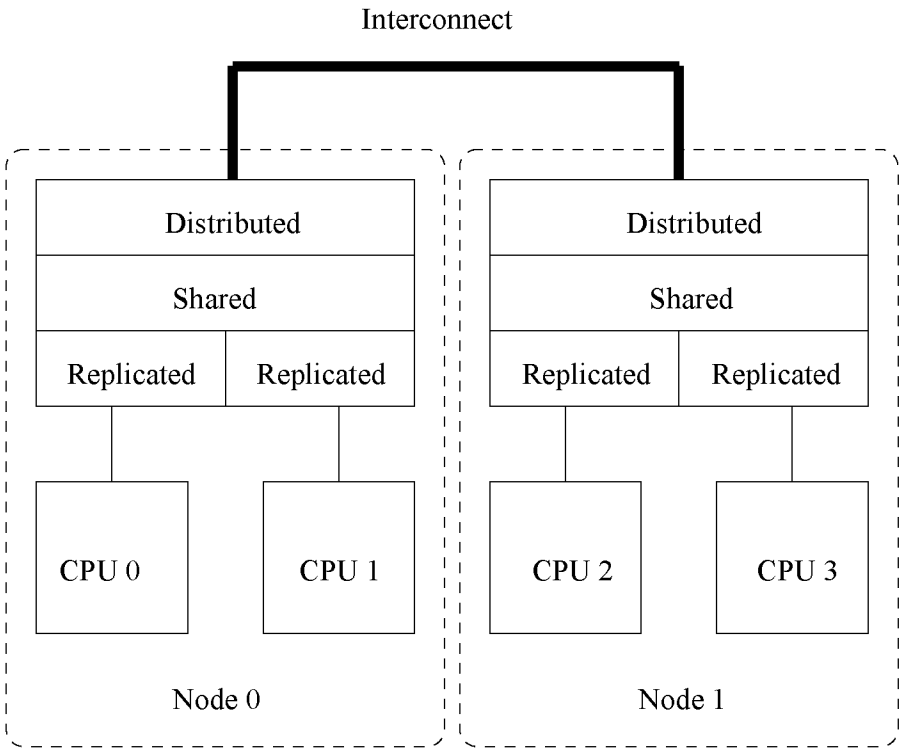

Figure 1.2 Memory hierarchy for the new DDI implementation on a two node, four processor system.

where $\hat{H}$ is the Hamiltonian of the system, E is the energy, and Ψ is the exact wavefunction of the system. The single particle Hamiltonian ($\hat{H}$) is given by

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V, \tag{1.2}$$

where $\hbar$ is Planck's constant ($6.626 \times 10^{-34}\mathrm{m}^2\mathrm{kg/s}$) divided by 2π, m is the mass of the particle in question, ∇ is the three dimensional derivative with respect to position, and V is the potential energy function, which in general depends on both the position and momentum of the particle in question. Thus, the Schrödinger equation is a second order differential equation whose goal is the determination of the wavefunction Ψ, which is also in general a function of position and momentum. Once Ψ is obtained, the physical observables of the particle in question can be calculated.

The Schrödinger equation can be expanded to treat multiple particles and, in fact, is

quite often used to treat entire molecules. One can write down the complete, many-particle Hamiltonian operator as,

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{A=1}^{M} \frac{1}{2M_A}\nabla_A^2 - \sum_{i=1}^{N}\sum_{A=1}^{M} \frac{Z_A}{r_{iA}} - \sum_{i=1}^{N}\sum_{j>i}^{N} \frac{1}{r_{ij}} - \sum_{A=1}^{M}\sum_{B>A}^{M} \frac{Z_A Z_B}{r_{AB}} \tag{1.3}$$

where lower case letters i, j are indices which denote all the electrons in the system and uppercase letters A, B indicate all the nuclei in the system. Also, in this expression atomic units have been used (see Ref. [16]) in place of the SI units which appear in Eq. 1.2, resulting in a much simpler overall expression by factorization of the physical constants. The total number of electrons in the system is N, the total number of nuclei in the system is M, Z_A is the atomic number of nucleus A, M_A is the mass[2] of nucleus A, r_{iA} is the distance between electron i and nucleus A, r_{ij} is the distance between electron i and electron j, and r_{AB} is the distance between nucleus A and nucleus B.

Because the electrons in the system move much more rapidly compared to the nuclei, an approximation (called Born-Oppenheimer) is made which says that the nuclei are essentially "frozen" in space while the Schrödinger equation is solved for the electrons (which move in the field of the "frozen" nuclei). This is the approximation we will assume for the remainder of this thesis as the methods and algorithms outlined below are concerned with solving the Schrödinger equation with respect to the electrons only. This approximation allows us to neglect the second term from Eq. 1.3 and the fifth term can be considered a constant.

At this point it is appropriate to mention that the complete Schrödinger equation is *only* exactly solvable for a system comprised of two particles, e.g., the hydrogen atom (one nuclei and one electron). Once the system is three particles or larger the classic three-body problem is encountered, which says that in the general unrestricted case, the positions and momenta of three interacting particles cannot be determined exactly. Because of this fact, one is then confronted with the choice of making an approximation to the Hamiltonian expression, or an approximation to the wavefunction. In the following work, the electronic Hamiltonian is left exact and an approximation is made to the wavefunction. This approximation is an expansion

[2]To be precise it is the ratio of the mass of nuclei A to the mass of the electron.

of the wave function in terms of basis functions. It involves specifying a set of basis functions, so that the wavefunction is a linear combination of these basis functions. Each basis function is parameterized by an initially unknown coefficient. The task of the computational chemistry application is to calculate these unknown coefficients. Once these coefficients are calculated, the wavefunction is known (to the limit of the basis set accuracy) and physical properties of the system can then be calculated using this wavefunction. As the size of the atomic basis set is increased, the accuracy of the calculation improves. Thus, to achieve the most accurate results, one wishes to use the largest basis sets that are computationally feasible. In computer science algorithm run-time analysis, one specifies how much time the algorithm takes with respect to the input size. In this case, the input size is commonly taken as the size of the basis set, although the size (i.e., number of electrons) of the chemical system in question also practically impacts the time the calculation takes.

The first step in many electronic energy calculations is to calculate the Hartree-Fock Self Consistent Field energy (for background see Refs. [16] and [17]). This method involves calculating each electron in the field of all the other electrons. It is an iterative calculation in which the iterations continue until the electrons' positions remain the same as before and after the iteration. Once the iterations have converged, the calculation is complete. The drawback of this method is that interactions between any two (or more) particular electrons is not explicitly calculated, since each electron only "sees" an overall field of the other electrons. Because of this, methods have been developed which attempt to calculate the so-called correlation energy, i.e., the energy due to the direct correlation of electrons with each other. These are commonly called electron correlation methods and the coupled cluster method is one of these methods used in computational chemistry today. The entire energy calculation is given as,

$$E_{\mathrm{el}} = E_{\mathrm{SCF}} + E_{\mathrm{corr}}, \tag{1.4}$$

where E_{el} is the total electronic energy (neglecting the motion of the heavier nuclei, as noted above in the Born-Oppenheimer approximation). This is the sum of E_{SCF} which is the energy calculated using the Hartree-Fock method, and E_{corr} which is the correlation energy calculated

using one of various correlation methods. The coupled cluster method will be the correlation method described and implemented in this work.

1.2.3 Coupled cluster method

As mentioned above, the starting point for the coupled cluster method is to first solve the Hartree-Fock equations, obtaining the SCF energy and the Hartree-Fock wavefunction. This Hartree-Fock wavefunction is then used as the starting point for the coupled cluster wavefunction, and in turn the coupled cluster energy calculation. The coupled cluster approximation to the wavefunction is the following,

$$\Psi_{CC} \equiv e^{\hat{T}}\Phi_0, \tag{1.5}$$

where Ψ_{CC} is the coupled cluster wavefunction, Φ_0 is the reference (Hartree-Fock [16]) wavefunction , and $\hat{T}$ is the cluster operator given by

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \ldots + \hat{T}_{\mathrm{N}}. \tag{1.6}$$

$\hat{T}_1$ is a single particle operator, $\hat{T}_2$ is a two-particle operator, etc. The number of particles in the system is N. This leads to the following equation

$$\hat{H}e^{\hat{T}}\Phi_0 = Ee^{\hat{T}}\Phi_0, \tag{1.7}$$

which is then used to solve the coupled-cluster equations. The computational demand increases rapidly as the size of the cluster (number of particles in the operator) increases, thus it is common to truncate $\hat{T}$ at at some small value of N. When truncating $\hat{T}$ at the $\hat{T}_2$ term, the calculation is referred to as coupled cluster with singles and doubles (CCSD)[3]. Truncation at the singles and doubles is quite common, and it is an $\mathcal{O}(N^6)$ algorithm. One can do higher order calculations, e.g., CCSDT, CCSDTQ (where T is triples and Q is quadruples) but their

[3]The reader is encouraged to examine Ref. [18] for detailed background information on the coupled cluster method.

asymptotic runtime is much more costly. Consider also that the coupled cluster equations are iterative, that is the runtime analysis applies to *one* iteration, so not only is the asymptotic runtime quite costly, it may have to be run many times before convergence is achieved. To reduce some of this runtime, methods have been developed in which one performs coupled cluster up to a particular order (in this case, second order) and then augments the result with a perturbative, non-iterative correction (in this case, the triples correction). Thus, one performs coupled cluster for single and double excitations (CCSD) and then performs the non-iterative calculation for the triple excitations (T). This results in the CCSD(T) method, i.e., coupled cluster with single and double excitations and perturbative triples. This is a method that affords an excellent amount of accuracy of the calculations, but still completes in a reasonable amount of time and so it is used quite extensively by researchers in the field. This work describes implementing a parallel CCSD(T) hybrid algorithm for GAMESS.

The CCSD algorithm [19, 20] solves for the so-called cluster amplitudes (double precision floating point arrays), t_1 and t_2, where t_1 amplitudes are single excitation amplitudes and t_2 are double excitation amplitudes. These amplitudes are coefficients that are used to make a better approximation to the reference wavefunction. The (T) algorithm adds a perturbative correction to the calculation and is not iterative. The parallelization of the CCSD(T) calculation is unique in that the CCSD component is iterative and as such, to parallelize it requires quite a bit of synchronization between processes as the calculation must proceed in "lock-step" on all processes at the same time. The (T) component on the other hand, is not iterative and requires very little data sharing and thus it can be distributed over multiple processes in a simpler manner. The combination of these two quite different algorithms into one program provides some unique algorithmic challenges.

1.3 Thesis organization

This thesis is comprised of one published paper (Chapter 2) and an in-depth description of the fully distributed algorithm (Chapter 3) that will result in a paper submission as well. The work in Chapter 2 was performed principally by me. The work described in Chapter 3 was

performed jointly by me and Ryan Olson. My contribution included a majority of the CCSD parallel algorithm and a majority of the (T) parallel algorithm. Ryan Olson contributed the direct $[VV|VV]$ integral algorithm (which appears in the CCSD algorithm) and the additions to the DDI communication library which were necessary to operate with three levels of memory.

Chapter 2 describes work performed to parallelize the existing coupled cluster algorithm using *only* shared memory on one node of an SMP. The primary focus was in parallelizing the calls to DGEMM via OpenMP. There it is shown that performance improvements are seen using this scheme. However, to become production code, one must consider more avenues for parallelization and develop a code which runs on multiple SMP nodes. Chapter 3 describes the fully distributed, hybrid parallel algorithms developed for solving the CCSD(T) equations on a cluster of SMP machines. Chapter 4 describes conclusions of the work and possibilities for future research.

CHAPTER 2. Parallelization of general matrix multiply routines using OpenMP

A paper modified slightly from a publication in Lecture Notes in Computer Science[1]

Jonathan L. Bentz[2], Ricky A. Kendall[3]

Abstract

An application programmer interface (API) is developed to facilitate, via OpenMP, the parallelization of the double precision general matrix multiply routine called from within GAMESS [1] during the execution of the coupled-cluster module for calculating physical properties of molecules. Results are reported using the ATLAS library and the Intel MKL on an Intel machine, and using the ESSL and the ATLAS library on an IBM SP.

2.1 Introduction

Matrix multiply has been studied extensively with respect to high performance computing, including analysis of the complexity of parallel matrix multiply (see Ref. [2] and Refs. therein). The Basic Linear Algebra Subprograms [3, 4, 5, 6] are subroutines that perform linear algebraic calculations on vectors and matrices with the aim of being computationally efficient across all platforms and architectures. The Level 3 Basic Linear Algebra Subprograms [7] (BLAS)

[1]Proceedings of the 5th International Workshop on OpenMP Applications and Tools, WOMPAT 2004, Houston, TX, May 17-18, 2004, appearing in Lecture Notes in Computer Science 3349 (2005) 1-11, edited by Barbara Chapman.

[2]Graduate Student, Scalable Computing Laboratory, Ames Laboratory, U.S. DOE, Department of Computer Science, Iowa State University, Ames, IA 50011

[3]Adjunct Professor, Scalable Computing Laboratory, Ames Laboratory, U.S. DOE, Department of Computer Science, Iowa State University, Ames, IA 50011

are subroutines that perform matrix-matrix operations. The double precision general matrix multiply (DGEMM) routine is a member of the level 3 BLAS and has the general form

$$\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}\,, \tag{2.1}$$

where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices and α and β are scalar constants. In the double precision case of real numbers, matrices $\mathbf{A}$ and $\mathbf{B}$ can either be transposed or not transposed upon entry into DGEMM.

This work uses OpenMP to facilitate the parallelization of the general matrix multiply routines consistently encountered in high-performance computing. Specifically, we are working in conjunction with the ab initio quantum chemistry software suite GAMESS (General Atomic and Molecular Electronic Structure System) [1], developed by Dr. Mark Gordon and his group at Iowa State University. Many of the modules in GAMESS are implemented to run in parallel (via an asynchronous one-sided distributed memory model), but the module used in calculating physical properties via the coupled-cluster (CC) method currently has only a serial implementation [8]. The CC code [9] has numerous calls to DGEMM and to improve the performance of this code when run on shared memory systems, we are developing an intelligent application programmer interface (API) for the DGEMM routine which is called from within GAMESS during its execution. Our wrapper routine (hereafter referred to as ODGEMM) uses OpenMP to parallelize the matrix multiplication. Currently, GAMESS comes with a vanilla source code BLAS (VBLAS) library built in and one can optionally link with any available BLAS library instead. It is not sufficient to simply link all of GAMESS to a multi-threaded BLAS library because then the modules (other than CC) which have previously been parallelized will create numerous threads when the parallelization has already been taken care of at a different level. Because of this difference between the CC module and the rest of GAMESS, shared memory parallelization of DGEMM within the CC module is facilitated by ODGEMM (i.e., **O**penMP **DGEMM**) which is written to work specifically within the CC module. Our API is designed to be directly called by GAMESS, partition the matrices properly and then call a supplied BLAS routine to perform the individual multiplications of

the partitioned patches. We have tested a number of different BLAS libraries in which the library DGEMM is used as a subroutine in our ODGEMM. The ODGEMM routine also calls different BLAS libraries based on the system software infrastructures.

We have tested our routine with the Automatically Tuned Linear Algebra Software library (version 3.6.0) [10] (ATLAS), which is freely available and compiles on many platforms. ATLAS provides a strictly serial library and a parallel library with the parallelization implemented using POSIX threads. The number of threads chosen in the parallel libraries of ATLAS is determined when the library is compiled and is commonly the number of physical processors. The number of threads cannot be increased dynamically but ATLAS may use less if the problem size does not warrant using the full number. The multiplication routines in ATLAS are all written in C, although ATLAS provides a C and FORTRAN interface from which to call their routines. GAMESS is written in FORTRAN and as such our wrapper routine is a FORTRAN callable routine. Because our routine is written in C, we can use pointer arithmetic to manipulate and partition the matrices. This allows us to avoid copying matrix patches before calling the multiplication routine.

Testing has also been performed with the Intel Math Kernel Library (version 6.1) [11] (MKL) and the IBM Engineering and Scientific Subroutine Library (version 3.3.0.4) [12] (ESSL). The MKL is threaded using OpenMP so the threading can be controlled by an environment variable similarly to ODGEMM. The ESSL number of threads can also be changed through the use of an environment variable.

The testing with GAMESS has been run with ATLAS ODGEMM, ATLAS PTDGEMM, MKL ODGEMM, ESSL ODGEMM and ESSLSMP DGEMM (see Table 2.1). MKL does come with a threaded library but it cannot be called with multiple threads from GAMESS currently because of thread stacksize problems[4]. These tests have been performed on a variety of computational resources and associated compiler infrastructure.

[4]This is a vendor specific problem unrelated to the present work.

Library	Platform	Description
VBLAS DGEMM	Intel	Serial Vanilla Blas
ATLAS PTDGEMM	Intel	Pthread built-in implementation using ATLAS
ATLAS ODGEMM	Intel	OpenMP implementation using ATLAS
MKL ODGEMM	Intel	OpenMP implementation using MKL
ESSLSMP DGEMM	IBM	Vendor threaded implementation of ESSL
ESSL ODGEMM	IBM	OpenMP implementation using ESSL
IBMATLAS ODGEMM	IBM	OpenMP implementation using ATLAS

Table 2.1 Acronyms for the libraries used.

2.2 Outline of ODGEMM Algorithm

The ODGEMM algorithm uses course-grained parallelism. Consider, for brevity, that the **A** and **B** matrices are not transposed in the call to the ODGEMM routine. In this case, **A** is an $M \times K$ matrix, **B** is a $K \times N$ matrix, and the resultant **C** is an $M \times N$ matrix. (In subsequent tables of data, M, N and K are also defined in this manner.) Upon entry to ODGEMM, **A** is partitioned into n blocks, where n is the number of threads. The size of each block is M/n by K, such that each block is a patch of complete rows of the original **A** matrix. If n does not divide M evenly, then some blocks may receive one more row of **A** than others. Matrix **B** is partitioned into blocks of size K by N/n. In a similar fashion each block of **B** is a patch of N/n full columns of **B** and again if N/n has a remainder, some blocks will receive one more column of **B** than others.

After this partitioning occurs, the calls to a library DGEMM (e.g., ATLAS, MKL, etc.) are made. Each thread works with one block of **A** and the entire **B**. If A_i is the ith block of **A** and B_j is the jth block of **B**, then the multiplication of A_i by B_j produces the C_{ij} block. Furthermore, since the ith thread works with A_i and the entire **B**, the ith thread is computing the C_i block of **C**, a block of M/n complete rows of **C**. Each thread computes an independent patch of **C** and as a result there is no dependence among executing threads on the storage of **C**.

2.3 Results

2.3.1 Matrix Multiply Testing

We have tested our routine with the libraries mentioned above and provide some results of our testing in Table 2.2.

Library	M	K	N	1	2	4	16	PT
ATLAS ODGEMM	2000	2000	2000	5.05	2.69	1.64	-	1.41
VBLAS DGEMM				99.29	-	-	-	-
MKL ODGEMM				4.62	2.50	1.58	-	-
ATLAS ODGEMM	7000	7000	7000	210.9	108.6	57.98	-	58.5
MKL ODGEMM				196.0	101.8	56.87	-	-
ESSLSMP DGEMM				538.6	-	-	39.9	-
ESSL ODGEMM				538.6	-	-	34.8	-
IBMATLAS ODGEMM				531.3	-	-	49.2	51.4
ATLAS ODGEMM	8000	8000	8000	315.9	162.8	85.5	-	85.74
MKL ODGEMM				293.3	150.4	83.06	-	-
ESSLSMP DGEMM				966.5	-	-	60.69	-
ESSL ODGEMM				966.5	-	-	53.11	-
IBMATLAS ODGEMM				795.7	-	-	78.69	80.42
ATLAS ODGEMM	10000	10000	1000	63.21	30.88	48.56	-	17.27
MKL ODGEMM				57.42	31.21	16.97	-	-
ESSL ODGEMM				163.1	-	-	11.85	-
IBMATLAS ODGEMM				214.3	-	-	15.78	14.26
ATLAS ODGEMM	1000	10000	10000	61.53	31.66	17.18	-	16.67
MKL ODGEMM				58.32	32.22	20.99	-	-
ESSL ODGEMM				153.3	-	-	11.18	-
IBMATLAS ODGEMM				156.9	-	-	15.12	14.10

Table 2.2 Matrix multiplication execution times. Results are reported in seconds of wall-clock time. The numbers in the column headings indicate the number of threads used. The PT column heading indicates the execution time upon calling the ATLAS PTDGEMM threaded routine directly.

These tests were performed by first generating analytical **A**, **B**, and **C** matrices with double precision elements, performing the multiplication using DGEMM or ODGEMM, and comparing the resultant **C** matrix with the analytical **C** matrix. All of our preliminary testing was performed on two machines: an SMP machine (named Redwing) with 4 GB of memory and 4 Intel Xeon 2.00 GHz processors, and one node of the IBM SP (named Seaborg) provided by NERSC (see Acknowledgments) which has 16 GB of memory and 16 POWER3 375 MHz processors.

The data in Table 2.2 exhibits a number of interesting features. The VBLAS DGEMM time is inserted for comparison to the more sophisticated BLAS libraries used in this work. On a relatively small matrix size, the time required for VBLAS DGEMM is almost 2 orders of magnitude larger than either the single-threaded ATLAS ODGEMM or MKL ODGEMM results. Viewing all results from Redwing, the ATLAS ODGEMM results are comparable to the MKL ODGEMM results, and in a few cases, the ATLAS PTDGEMM routine actually runs faster than the MKL ODGEMM routine. Viewing the results from Seaborg, one notices that the ESSLSMP DGEMM and ESSL ODGEMM threaded routines are consistently faster than the IBMATLAS ODGEMM. On the other hand, when only one thread is used, IBMATLAS ODGEMM runs faster for the largest matrix size tested. A rather striking result is that of the last two sections of the table, where the dimensions of the matrices are not equal. Considering the Redwing results, when M is 10000 and 4 threads are used, ATLAS ODGEMM is quite high, but when M is 1000, then ATLAS ODGEMM is quite reasonable. The only difference between these two test cases is that the dimensions of M and N are swapped. Recall that the algorithm partitions the rows of **A**, so as the number of rows of **A** changes, that should affect the outcome somewhat. However, MKL ODGEMM does not show a similar difference between the non-square matrix sizes. These are simply general results so that one can see how these matrix multiply routines compare with one another.

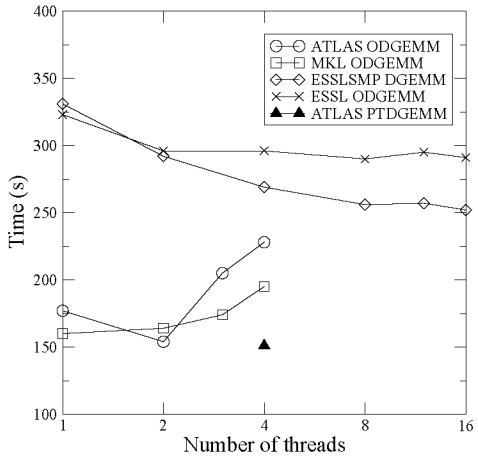

Figure 2.1 Execution time vs. number of threads for the molecule HNO using the cc-pVTZ basis set. ATLAS and MKL calculations were performed on Redwing. ESSL calculations were performed on one node of Seaborg. The basis set cc-pVTZ uses 85 basis functions for this calculation. The x-axis scale is logarithmic.

2.3.2 GAMESS

The results reported in this section are from execution of the GAMESS CC module performing energy calculations.[5] Figures 2.1, 2.2 and 2.3 show timing data vs. number of threads from GAMESS execution runs. In all three figures, the ATLAS PTDGEMM result is simply a single point since the number of threads when using the ATLAS PTDGEMM threaded routine cannot be changed by the user. The compile time thread number (set to the number of physical processors) is really a maximum thread number since ATLAS may use fewer threads if the matrices are sufficiently small.

[5]The basis sets were obtained from the Environmental Molecular Sciences Laboratory at Pacific Northwest National Laboratory, http://www.emsl.pnl.gov/forms/basisform.html. For an explanation of the basis sets see ref. [13].

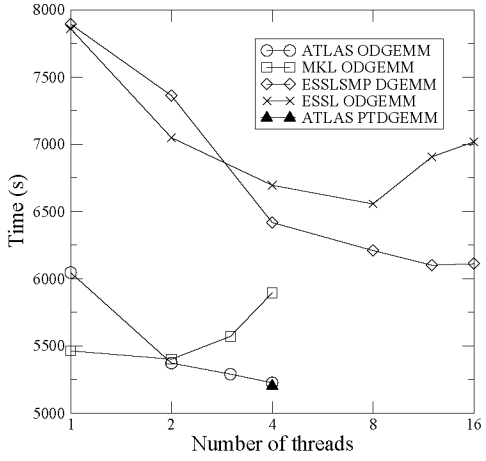

Figure 2.2 Execution time vs. number of threads for the molecule HNO using the cc-pVQZ basis set. ATLAS and MKL calculations were performed on Redwing. ESSL calculations were performed on one node of Seaborg. The basis set cc-pVQZ uses 175 basis functions for this calculation. The x-axis scale is logarithmic.

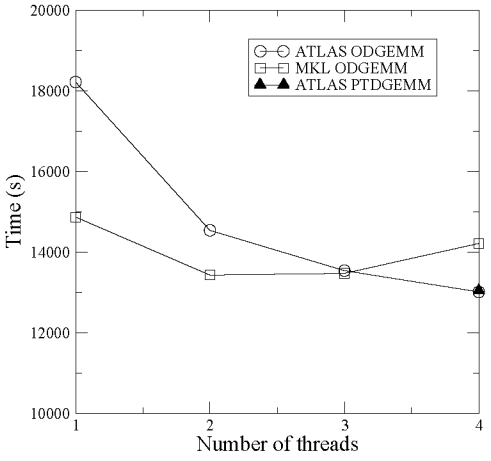

Figure 2.3 Execution time vs. number of threads for the molecule glycine using the cc-pVDZ basis set for the hydrogens and the cc-pVTZ basis set for all other atoms for a total of 200 basis functions. Calculations were performed on Redwing.

Figure 2.1 shows execution time vs. number of threads for the HNO molecule using the cc-pVTZ basis set. The ESSL ODGEMM and ESSLSMP DGEMM results are almost identical for 1 and 2 threads, but they diverge quickly when 4 or more threads are used. This is probably due to the fact that in these testing runs, ESSL ODGEMM always uses the specified number of threads, and with small matrix sizes this can cause an unnecessary amount of overhead. With ESSL ODGEMM there is no performance improvement after 2 threads but with ESSLSMP DGEMM the performance continues to increase. The ATLAS ODGEMM and MKL ODGEMM results are more sporadic. With 1 thread, MKL ODGEMM is faster, with 2 threads, ATLAS ODGEMM is faster, and then with 3 and 4 threads, MKL ODGEMM is faster. Both curves, with the exception of ATLAS ODGEMM using 2 threads, show an increase in execution time with increase in the number of threads. Again this is most likely due to the thread overhead of partitioning matrices which may not be large enough to require parallelization. The fastest overall execution time is obtained by calling ATLAS PTDGEMM. As a reference point, on Redwing, the wall time for this calculation using the VBLAS DGEMM that comes packaged with GAMESS is 705 seconds.

To investigate this behavior further, some test cases were calculated using matrix sizes that occur frequently in this GAMESS calculation. The results of these test cases are shown in Table 2.3. The three test cases shown account for about 75% of the matrix multiplications found in this GAMESS calculation. The first thing to note when viewing these results is that the dimensions of the matrices are quite unequal. The results show good agreement with what was shown in Fig. 2.1. As the ESSLSMP DGEMM thread number increases, the execution time decreases, while the ESSL ODGEMM execution time is at its lowest using either 2 or 4 threads, then stays flat and even increases in some cases. A similar analysis of the ATLAS ODGEMM and MKL ODGEMM results shows a good agreement with the timing data of the actual GAMESS executions and shows that the execution of GAMESS is dependent on the repeated multiplication of only a few different size matrices.

Library	M	K	N	1	2	4	8	12	16	PT	Frequency
ATLAS ODGEMM	77	77	5929	26.1	20.1	18.4	-	-	-	14.2	420
MKL ODGEMM				44.4	35.7	56.3	-	-	-	-	
ESSLSMP DGEMM				70.8	39.9	20.4	12.8	11.4	11.9	-	
ESSL ODGEMM				87.6	52.4	39.6	33.8	35.4	37.2	-	
ATLAS ODGEMM	5929	6	77	7.67	9.63	8.44	-	-	-	4.84	420
MKL ODGEMM				8.13	9.03	7.89	-	-	-	-	
ESSLSMP DGEMM				48.4	11.9	8.55	6.23	5.51	5.42	-	
ESSL ODGEMM				34.9	18.8	17.7	17.3	20.5	27.9	-	
ATLAS ODGEMM	36	5929	77	28.0	20.5	64.0	-	-	-	31.9	1461
MKL ODGEMM				26.2	19.6	31.5	-	-	-	-	
ESSLSMP DGEMM				43.8	19.3	13.1	13.1	12.1	11.2	-	
ESSL ODGEMM				39.1	21.1	28.7	32.3	37.2	35.9	-	

Table 2.3 Matrix multiplication execution times using test cases where the matrix sizes are equal to the matrix sizes used on the HNO molecule with the cc-pVTZ basis set. Results are reported in 10^{-3} seconds of wall-clock time. The PT column results were obtained by calling ATLAS PTDGEMM directly. The numbers in the column headings indicate the number of threads. Frequency is the number of times DGEMM is called with matrices of that size in the GAMESS execution. Total number of calls to DGEMM in the GAMESS execution is ≈ 3060.

Figure 2.2 shows execution time vs. number of threads for the HNO molecule using the cc-pVQZ basis set with 175 basis functions. The results using ESSL show that the ESSL ODGEMM is slightly faster than ESSLSMP DGEMM for 1 and 2 threads, but when more than 2 threads are used the ESSLSMP DGEMM continues to decrease in execution time while ESSL ODGEMM decreases more slowly and even increases for 12 and 16 threads. The increase when using ESSL ODGEMM is again probably attributed to the overhead of partitioning and using more threads than necessary on some matrix sizes. The MKL ODGEMM results in particular are striking. The execution time decreases slightly from 1 to 2 threads, but using 3 and 4 threads increases the execution time. The ATLAS ODGEMM results are as one would expect, namely that as the thread number increases, the time decreases, until the fastest time is obtained when 4 threads are used. Also note that the direct call of the threaded ATLAS PTDGEMM is essentially the same as that of ATLAS ODGEMM when 4 threads are used. As a reference, the wall time for this calculation on Redwing when using the default VBLAS DGEMM in GAMESS is 35907 seconds.

As in the earlier case, Table 2.4 was prepared with three test cases using matrix sizes that occur frequently in this GAMESS calculation. The results are similar to the earlier case in that the execution time of GAMESS with respect to the matrix multiplication is dominated by the multiplication of relatively few different sizes of matrices. The three test cases shown in Table 2.4 account for about 80% of the matrices multiplied. The matrix dimensions are much different and it is clear that the execution time with respect to the matrix multiplication is dominated by the repeated multiplication of only a few different size matrices.

Library	M	K	N	1	2	4	8	12	16	PT	Frequency
ATLAS ODGEMM	167	167	27889	54.7	32.6	23.0	-	-	-	21.7	420
MKL ODGEMM				44.6	44.1	57.1	-	-	-	-	
ESSLSMP DGEMM				132.	66.0	33.7	33.2	14.2	12.6	-	
ESSL ODGEMM				139.	77.9	47.9	33.1	28.9	27.6	-	
ATLAS ODGEMM	27889	6	167	8.01	8.46	7.84	-	-	-	5.45	420
MKL ODGEMM				7.89	9.05	8.21	-	-	-	-	
ESSLSMP DGEMM				21.8	11.3	5.88	3.09	2.71	2.63	-	
ESSL ODGEMM				29.0	18.7	13.5	11.6	11.2	11.1	-	
ATLAS ODGEMM	36	27889	167	28.4	20.5	17.6	-	-	-	17.8	3172
MKL ODGEMM				22.7	19.1	32.1	-	-	-	-	
ESSLSMP DGEMM				28.9	15.5	8.95	6.22	5.29	5.91	-	
ESSL ODGEMM				28.9	18.9	13.9	15.2	27.4	32.3	-	

Table 2.4 Matrix multiplication times using test cases where the matrix sizes are equal to the matrix sizes used on the HNO molecule with the cc-pVQZ basis set. Results are in 10^{-2} seconds of wall-clock time. The PT column results were obtained calling ATLAS PTDGEMM directly. The numbers in the column headings indicate the number of threads. Frequency is the number of times matrices of that size are called in the GAMESS execution. Total number of calls to DGEMM in the GAMESS execution is ≈ 4860.

Figure 2.3 shows execution time as a function of the number of threads for GAMESS using glycine as the input molecule. An interesting feature of this graph is that the ATLAS ODGEMM timings decrease monotonically and there is a difference of over 500 seconds between one thread and four threads of execution. The MKL ODGEMM does much better than ATLAS ODGEMM when using one thread, and is slightly faster using two threads. But then the total time increases for three and four threads using MKL ODGEMM. ATLAS DGEMM shows consistent decrease in time of execution upon the addition of processors, but MKL ODGEMM actually increases its execution time for three and four threads. Note that the ATLAS PTDGEMM result is almost exactly the same as the ATLAS ODGEMM result using 4 threads.

2.4 Conclusions and Future Work

One conclusion of this work is that if one wants to improve the performance of the CC module of GAMESS, an external BLAS library should be used. With unsophisticated testing on Redwing it was shown that almost two orders of magnitude improvement can be gained by linking with ATLAS or MKL.

With respect to results calculated on Redwing, when using 1 thread, the MKL ODGEMM is faster than ATLAS ODGEMM. When 2 threads are used, the results are mixed with MKL ODGEMM running faster in some cases and ATLAS ODGEMM running faster in other cases. When 3 and 4 threads are used, especially in the GAMESS executions, ATLAS ODGEMM is consistently faster than MKL ODGEMM. When one looks at GAMESS execution time irrespective of the number of threads used, ATLAS PTDGEMM is almost always the fastest, and in Fig. 2.2 and Fig. 2.3 the ATLAS ODGEMM times are comparable. An unexpected result is that of the MKL ODGEMM when multiple threads are used. When more than 2 threads are used in the GAMESS execution using MKL ODGEMM, the times actually increase.

The results obtained using Seaborg are worthy of note as well. When considering the results of the generic matrix multiplication of Section 2.3.1, the ESSL ODGEMM actually runs faster than the built in ESSLSMP DGEMM library. When compared to ATLAS, the

threaded versions of ESSL are faster than the threaded ATLAS versions. The results from the GAMESS tests show that for thread numbers less than 4, ESSL ODGEMM and the ESSLSMP DGEMM give similar results. When adding more threads to the calculation, the ESSLSMP DGEMM consistently yields faster results than the ESSL ODGEMM, especially at high numbers of threads. This seems to suggest that the ESSLSMP DGEMM library has some built in mechanisms to determine better partitioning of the matrices.

After testing on two different architectures, it is clear that calling ODGEMM with the number of threads equal to the number of processors does not always yield the fastest execution times. Depending on whether the system is an Intel system or IBM, our results show that the fastest execution time varies significantly based on the number of threads chosen and the BLAS library used. This information will be incorporated into the ODGEMM routine. The ODGEMM library will then be able to choose the appropriate call mechanism to the library DGEMM with properly partitioned matrices and an appropriate number of threads.

For future work we wish to incorporate some matrix metrics into ODGEMM to make decisions about parallelization. For example, considering GAMESS executions, it is quite probable that some calls to DGEMM will run faster in parallel while some calls may be fastest running on one thread only, or with the number of threads less than the number of processors. These decisions will depend on factors such as matrix size, dimensional layout, and processor speed. Especially when using the vanilla BLAS which has no parallel implementation, the opportunity for performance enhancement is available via ODGEMM properly parallelizing the matrix multiplication. We are also going to perform testing and hence provide portability to more architectures, as GAMESS is currently available for a wide range of architectures and systems. The use of parallel DGEMM in GAMESS is a beginning for exploiting the advantages of SMP machines in GAMESS calculations. We will investigate the utilization of OpenMP at other levels within the chemistry software as well.

Acknowledgements

This work was performed under auspices of the U. S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing Research. This material is based on work supported by the National Science Foundation under Grant No. CHE-0309517. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Basis sets were obtained from the Extensible Computational Chemistry Environment Basis Set Database, Version 12/03/03, as developed and distributed by the Molecular Science Computing Facility, Environmental and Molecular Sciences Laboratory which is part of the Pacific Northwest Laboratory, P.O. Box 999, Richland, Washington 99352, USA, and funded by the U.S. Department of Energy. The Pacific Northwest Laboratory is a multi-program laboratory operated by Battelle Memorial Institute for the U.S. Department of Energy under contract DW-AC06-76RLO 1830. Contact David Feller or Karen Schuchardt for further information. The authors wish to thank Ryan Olson for his helpful comments with respect to implementation details of GAMESS.

References

[1] M. W. Schmidt et al.: General Atomic and Molecular Electronic Structure System. J. Comput. Chem. 14 (1993) 1347-1363.

[2] E. E. Santos: Parallel Complexity of Matrix Multiplication. J. Supercomp. 25 (2003) 155-175.

[3] C .L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh: Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Soft. 5 (1979) 308-323.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh: ALGORITHM 539, Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Soft. 5 (1979) 324-245.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson: An Extended Set of FORTRAN Basic Linear Algebra Subprograms. ACM Trans. Math. Soft. 14 (1988) 1-17.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson: ALGORITHM 656, An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. ACM Trans. Math. Soft. 14 (1988) 18-32.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff: A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Soft. 16 (1990) 1-17.

[8] GAMESS User's Guide, accessed March 1, 2004, (http://www.msg.ameslab.gov/GAMESS/GAMESS.html).

[9] P. Piecuch, S. A. Kucharski, K. Kowalski and M. Musial: Efficient computer implementation of the renormalized coupled-cluster methods: The R-CCSD[T], R-CCSD(T), CR-CCSD[T], and CR-CCSD(T) approaches. Comput. Phys. Commun. 149 (2002) 71-96.

[10] R. C. Whaley, A. Petitet and J. J. Dongarra: Automated Empirical Optimization of Software and the ATLAS project. Parallel Computing. 27 (2001) 3-35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (http://www.netlib.org/lapack/lawns/lawn147.ps).

[11] Intel Corporation. Intel Math Kernel Library Version 6.1, Reference Manual, accessed March 1, 2004. (http://www.intel.com/software/products/mkl/docs/mklman61.htm).

[12] IBM Corporation. Engineering and Scientific Subroutine Library for AIX Version 3 Release 3: Guide and Reference, accessed March 1, 2004. (http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/sp34/essl/essl.html).

[13] T. H. Dunning Jr.: Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. J. Chem. Phys. 90 (1989) 1007-1023.

CHAPTER 3. Hybrid memory parallel algorithm

3.1 Serial algorithm in GAMESS

The current implementation of coupled cluster in GAMESS was contributed by Piecuch and coworkers [21]. They did extensive work to determine an optimal ordering of computation steps and factoring the requisite arrays to call DGEMM as much as possible. This was done with the goal of using highly optimized mathematical libraries, supplied by most computer vendors (including DGEMM), which are often hand-tuned for maximum performance. Their contribution is strictly a serial algorithm, in contrast to most of the other major functionalities of GAMESS, which have parallel implementations. This work focuses on the parallel implementation of the CCSD(T) algorithm, coupled cluster with single and double excitations, and the inclusion of perturbative triple excitations.

The input to a GAMESS calculation consists of the user choosing various computation and chemistry parameters, and also specifying a molecule. One input parameter which determines how accurate the calculation will be is the choice of a basis set. There are a number of shorthand variables that will be used, and Table 3.1 outlines three of them. The number of basis functions is completely specified by the choice of basis set and the size of the molecule[1]. The value n_o is the number of occupied orbitals[2], and n_v is the number of virtual orbitals (also called unoccupied orbitals). In the case of the GAMESS CCSD(T) algorithm, the value of n_o denotes only the valence electrons. That is, when considering all the electrons of an atom/molecule, there are a number of electrons that are not used in the CCSD(T) computation

[1]The basis set specifies a fixed number of basis functions for each type of atom. To determine the total number of basis functions in the calculation, one looks at each atom present in the molecule of interest and the basis set will indicate what type and how many basis functions will be used.

[2]An orbital is a mathematical function which specifies a region of space where the electron is likely to be found. Each electron is placed in one orbital.

Abbreviation	Numerical value
n_o	Number of occupied orbitals
n_v	Number of virtual (unoccupied)
n_{bf}	Number of basis functions

Table 3.1 Abbreviations used for array sizes.

Class	size
$[OO\|OO]$	n_o^4
$[OO\|OV]$	$n_o^3 n_v$
$[VO\|VO]$	$n_o^2 n_v^2$
$[VV\|OO]$	$n_o^2 n_v^2$
$[VV\|VO]$	$n_o n_v^3$
$[VV\|VV]$	n_v^4

Table 3.2 The classes of integrals used in the CCSD calculation, arranged in size order from smallest to largest.

because their contribution is negligible due to their location. Thus, n_o is always less than or equal to the number of electrons. The number of virtual orbitals, n_v is bounded by the size of the basis set, n_{bf}. Because $n_o \leq n_{bf}$ and $n_v \leq n_{bf}$, the runtime analysis of computational chemistry programs is commonly given in terms of n_{bf}, i.e. when CCSD is cited as an $\mathcal{O}(N^6)$ algorithm, $N = n_{bf}$.

The first step in the overall coupled cluster algorithm is to calculate the Hartree-Fock wavefunction and energy. This is an iterative calculation and is used as a starting point for many different algorithms in GAMESS. The next step is to perform an integral transformation, which prepares most of the numeric data necessary for the CCSD(T) calculation. The resultant integrals are stored on disk as double precision floating point numbers. Each class of integrals is stored as a four index array. These integral values constitute the majority of the memory requirements for the CCSD(T) algorithm, as they are typically calculated before the CCSD(T) calculation begins and are stored for future retrieval. Table 3.2 shows the different integral classes[3] and their space requirements with respect to occupied and virtual orbitals.

[3]Throughout this work we will use a bracket notation to indicate the different integral classes, e.g., the $[VV|VO]$ integral class is a four index array with the first three indices having size n_v and the last index size n_o. Thus, when discussing the $[VV|VO]$ integrals, we are referring to a four index, double precision array with the four dimensions given as (n_v, n_v, n_v, n_o) and subsequently the total size of the array is equal to $n_v^3 n_o$.

Class	size
t_1	$n_o n_v$
o_1	$n_o n_v$
t_2	$n_o^2 n_v^2$
o_2	$n_o^2 n_v^2$

Table 3.3 The classes of amplitudes used in the CCSD calculation, arranged in size order from smallest to largest.

The next step is to solve for the CCSD amplitudes. All of the integrals except for the $[VV|VV]$ integrals are read into memory and kept there for the entire time. Solving for the amplitudes (the final t_1 and t_2 arrays) is an iterative procedure. The working copies of the amplitudes are labeled t_1 and t_2, and the previous iterations' copies are labeled o_1 and o_2. When the difference between the amplitudes of successive iterations is sufficiently small, convergence has been achieved and the iterations terminate. The sizes of the amplitudes are given in Table 3.3.

A CCSD iteration consists of using the o_1 and o_2 amplitudes, along with all the integral classes shown in Table 3.2 to calculate new values for the t_1 and t_2 amplitudes. Throughout an entire iteration, neither the integral classes nor the o_1 or o_2 amplitudes change values; only the t_1 and t_2 amplitudes are updated. The details of the CCSD iterations are quite complicated. The general form of an iteration involves taking two arrays (integral classes or one of the o_1 or o_2 amplitudes), performing various permutations[4] on them, and then finally multiplying them together (using matrix multiplication) and accumulating the resultant matrix into either the t_1 or t_2 array. This happens no less than 30 times for one CCSD iteration. Run time analysis of an iteration involves calculating the cost of the matrix multiplications since they are the most expensive (in terms of floating point operations) step. The largest matrix multiplication which occurs in the CCSD iterations is the multiplication of o_2 (as a $n_o^2 \times n_v^2$ matrix) by $[VV|VV]$ (as a $n_v^2 \times n_v^2$ matrix). This matrix multiplication has an asymptotic runtime of $\mathcal{O}(n_o^2 n_v^4)$. Since

[4]There are many permutations performed on the arrays in CCSD(T). Some are simple matrix transposes, some involve permuting two of the four indices, e.g., exchanging index 1 and 3. In some cases, a permutation and a multiplication is performed on the array. The exact permutations performed are not instructive but suffice it to say that these permutations are in general very cache inefficient as memory is read and written in many different parts of the array.

$n_o < n_{bf}$ and $n_v < n_{bf}$, this is more generally analyzed as an $\mathcal{O}(n_{bf}^6)$ algorithm. The following pieces of code are presented to illustrate a few sample pieces of the serial CCSD code including its more expensive (in terms of runtime) operations.

```
DO 123 I=1,NU
         IOFF=NO2U*(I-1)+1
         CALL RDVPP(I,NO,NU,TI)
         CALL DGEMM('N','N',NO2,NU,NU2,ONE,O2,NO2,TI,NU2,ONE,
     &   T2(IOFF),NO2)
  123 CONTINUE
-------------------------------------------------------------------------------
      CALL TRMD(O2,TI,NU,NO,20)
      CALL TRMD(VR,TI,NU,NO,21)
      CALL VECMUL(O2,NO2U2,HALF)
      CALL ADT12(1,NO,NU,O1,O2,4)
      CALL DGEMM('N','N',NOU,NOU,NOU,ONEM,VR,NOU,O2,NOU,ONE,VL,NOU)
      CALL ADT12(2,NO,NU,O1,O2,4)
      CALL VECMUL(O2,NO2U2,TWO)
-------------------------------------------------------------------------------
      CALL TRMD(O2,TI,NU,NO,27)
      CALL TRMD(T2,TI,NU,NO,28)
      CALL DGEMM('N','N',NOU,NOU,NOU,ONEM,O2,NOU,VL,NOU,ONE,T2,NOU)
      CALL TRANMD(O2,NO,NU,NU,NO,23)
      CALL TRANMD(T2,NO,NU,NU,NO,23)
      CALL DGEMM('N','N',NOU,NOU,NOU,ONEM,O2,NOU,VL,NOU,ONE,T2,NOU)
```

The first section of code illustrates the most expensive term, namely the DGEMM call using the $[VV|VV]$ term. It is so large that it is actually performed in n_v loops with a runtime of $\mathcal{O}(n_o^2 n_v^3)$ in each loop iteration. The DGEMM is broken up in this fashion because storing the entire $[VV|VV]$ array in memory is only feasible for small system sizes. Instead, a n_v^3

piece of the array is read from disk and used. The second and third sections of code show slightly smaller DGEMMs with a runtime of $\mathcal{O}(n_o^3 n_v^3)$ each. Notice how the function calls to DGEMM are surrounded by other function calls. In these illustrative pieces of code, all the other subroutines surrounding the DGEMMs are calls to some sort of permutation routine (except for the RDVPP routine which reads in a portion of the $[VV|VV]$ array).

Once the CCSD iterations have converged, the (T) calculation is performed. Overall the (T) has three nested DO loops of size n_o. In each of these loops there are multiple DGEMM calls, the largest of which is n_v^4. Therefore, the asymptotic runtime of the (T) step is $\mathcal{O}(n_o^3 n_v^4)$, or more generally $\mathcal{O}(n_{bf}^7)$. Asymptotically, the (T) calculation dominates the runtime of the CCSD(T) algorithm, although practically this domination shows up more prevalently when large sized inputs are used.

The example code serves to illustrate the computationally intensive coupled cluster code currently in GAMESS. Coupled cluster is a CPU intensive method due to the many matrix multiplication calls. It is memory intensive due to the large integral and amplitude arrays which have to be in memory. And it is I/O intensive because it stores all the integrals on disk with the $[VV|VV]$, the largest integral class, being read from disk during each iteration.

3.2 Performance goals

When designing the parallel algorithm there were some goals in mind which we wanted our algorithm to strive for, while remaining within the bounds of the current status of high performance computing architecture. We wanted to be able to run a computation with n_o approaching 100, and with n_v approaching 800. The accuracy of the calculation increases with increasing n_v and so that value should be chosen as high as computationally possible. We also wanted to minimize the disk I/O and hopefully achieve this by eliminating the storage of the integrals on disk. The storage space of disk is quite inexpensive (financially speaking) and so from this perspective storing integrals on disk is attractive. However, depending on the I/O and network capabilities of the machine, the performance penalty can be very high due to the high amount of I/O associated with reading and writing large amounts of data to disk. So our

	100	200	300	400	500	600	700	800
10	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01
20	0.01	0.01	0.02	0.02	0.03	0.04	0.04	0.05
40	0.05	0.10	0.14	0.19	0.24	0.29	0.33	0.38
60	0.16	0.32	0.48	0.64	0.80	0.97	1.13	1.29
80	0.38	0.76	1.14	1.53	1.91	2.29	2.67	3.05
100	0.75	1.49	2.24	2.98	3.73	4.47	5.22	5.96

Table 3.4 The $[OO|OV]$ integral class memory requirements in gigabytes. The horizontal axis is n_v and the vertical axis is n_o.

	100	200	300	400	500	600	700	800
10	0.01	0.03	0.07	0.12	0.19	0.27	0.37	0.48
20	0.03	0.12	0.27	0.48	0.75	1.07	1.46	1.91
40	0.12	0.48	1.07	1.91	2.98	4.29	5.84	7.63
60	0.27	1.07	2.41	4.29	6.71	9.66	13.14	17.17
80	0.48	1.91	4.29	7.63	11.92	17.17	23.37	30.52
100	0.75	2.98	6.71	11.92	18.63	26.82	36.51	47.68

Table 3.5 The $[VO|VO]$ and $[VV|OO]$ integral class memory requirements, as well as the t_2 amplitude memory requirements, in gigabytes. The horizontal axis is n_v and the vertical axis is n_o.

goal is to use main memory as the primary vehicle for the data storage.

To examine the memory requirements of the various array sizes, consider Tables 3.4–3.7. Here the memory requirements (in gigabytes) are shown for the different integral classes. Recall that DDI was originally formulated as a process based communication model. To access data, it either needs to be stored as replicated data (meaning each process on a node would have its own copy, resulting in lots of duplicated data per node) or a library function has to be called to get the data from distributed memory and put it into a local buffer. One sees from the memory requirements tables that as the n_o and n_v are increased, the possibility of replicating these integrals per process becomes less feasible and even a smaller integral class $[OO|OV]$ (Table 3.4) requires almost 6 GB of memory per process for the largest case.

This memory concern led to the augmentation of the current DDI communication library with support for shared memory arrays which can be accessed by all the processes resident on the node. This is especially relevant for data such as the integrals as they are calculated once

	100	200	300	400	500	600	700	800
10	0.07	0.60	2.01	4.77	9.31	16.09	25.56	38.15
20	0.15	1.19	4.02	9.54	18.63	32.19	51.11	76.29
40	0.30	2.38	8.05	19.07	37.25	64.37	102.22	152.59
60	0.45	3.58	12.07	28.61	55.88	96.56	153.33	228.88
80	0.60	4.77	16.09	38.15	74.51	128.75	204.44	305.18
100	0.75	5.96	20.12	47.68	93.13	160.93	255.55	381.47

Table 3.6 The $[VV|VO]$ integral class memory requirements, in gigabytes. The horizontal axis is n_v and the vertical axis is n_o.

n_v	100	200	300	400	500	600	700	800
	1	12	60	191	466	966	1789	3052

Table 3.7 The $[VV|VV]$ integral class memory requirements, in gigabytes.

at the beginning of the code and after that they are only read. The only arrays of significance which are written are the amplitudes.

3.3 Parallel algorithm in GAMESS

3.3.1 Hierarchical memory partitioning

One of the first steps in designing the parallel algorithm is to determine an efficient partitioning of the system memory. Our target architecture for this algorithm, considering the large memory requirements, is a cluster of SMP nodes. The types of systems we have in mind are e.g., SMP nodes with 8 or 16 processors per node, with a minimum of 16 (and up to 64) GB of memory per node.

The choice of how to properly store the integrals depends a great deal on the system one is using. However, when one examines the memory requirements for the $[VV|VV]$ integrals given in Table 3.7, it is apparent that an alternate strategy is required for them, since even a relatively modest calculation ($\sim$ 400 basis functions) requires almost 200 GB of memory for this integral class alone. As stated above in the current serial implementation, all of the integrals are calculated before the CCSD iterations begin, stored on disk, and read into memory as needed. To alleviate the tremendous storage requirements of the $[VV|VV]$ integrals, we have

implemented a direct integral transformation routine for computing these integrals. Instead of computing the entire set of $[VV|VV]$ integrals at the beginning and storing them, we instead recalculate them "on the fly" as needed during each CCSD iteration. This greatly reduces the memory requirement. There is obviously a tradeoff between memory and CPU time, since the same computation is performed each iteration instead of one time at the beginning of the algorithm. However, it is often a trade that computational scientists are willing to make since storing the $[VV|VV]$ integrals would likely have to be done on disk, and accessing the disk might take as long or longer than recomputing the integrals, and might exhaust disk space as well. Also, even with the extra computations performed due to recomputing the integrals, the overall size of the problem which can be computed on a particular system is increased due to the smaller memory requirements of the integral computation. Memory is often the bottleneck in scientific computations and conserving memory at the expense of CPU time is frequently an advantageous tradeoff. This concept of direct computation of integrals in coupled cluster calculations has also been implemented in another computational chemistry package NWChem [22], where they not only recompute the $[VV|VV]$ integrals, but the $[VV|VO]$ integrals as well [23].

The $[VV|VO]$ integrals are stored in distributed memory (i.e., one copy distributed across all the nodes) and due to symmetry of the data, they are stored in about half[5] the space of the actual size of the array. When a section of the array is needed in a computation, it is expanded to its full form at that time. This requires the array to be expanded multiple times in the course of the CCSD iterations, but that expansion time is small compared to the memory savings gained.

The $[VV|OO]$ and $[VO|VO]$ integrals are also stored in distributed memory. Because these two classes of integrals are used so frequently and are permuted in complicated ways, they are actually stored multiple times in distributed memory, in different permuted orders which are used in the CCSD iterations. The rest of the arrays, including temporary arrays are stored in shared or replicated memory. Table 3.8 summarizes the relevant arrays, where they are

[5]The exact size is $n_v(n_v+1)/2 \times n_v n_o$

Array	Distributed memory	Shared memory	Replicated memory
$[VV\|VO]$	$n_o n_v^3/2$	0	0
$[VO\|VO]$	$5n_o^2 n_v^2$	0	0
$[VV\|OO]$	$2n_o^2 n_v^2$	0	0
$[OO\|OV]$	0	$n_o^3 n_v$	0
$[OO\|OO]$	0	n_o^4	0
t_2	0	$n_o^2 n_v^2$	0
o_2	0	$n_o^2 n_v^2$	0
temp	0	$n_o^2 n_v^2$	0
t_1	0	0	$n_o n_v$
o_1	0	0	$n_o n_v$

Table 3.8 Array sizes and memory requirements. The coefficients of the values in the distributed memory column reflect the use of symmetry or the multiple copies of arrays stored in different orders.

located, and how much memory they require.

3.3.2 Process based algorithm

Depending on the memory of the system and the size of the calculation, the CCSD(T) parallel algorithm is designed to operate on either a process based or node based footing. The process based algorithm has a larger memory requirement due to the necessity of more replicated temporary memory. There are multiple sections of the algorithm where more than one process is calculating contributions to the same section of a shared memory array. When multiple processes attempt to write to the same memory location, that memory location will likely be corrupted. In the process based algorithm, each process writes the results of these types of calculations into a temporary, replicated memory buffer, and at the conclusion of that particular part of the calculation, each process performs a global sum of the temporary data, which is then accumulated into the shared data.

The overall algorithm begins by using existing portions of GAMESS to get the data set up properly. There already exists a parallel version of Hartree-Fock and of the integral transformation, so these modules were used with no modification. Once the integral transformation is completed the integrals are placed into distributed or shared memory (according to the scheme showed in Table 3.8) by the master process. Recall from §1.2.1 that the DDI arrays are two

index arrays. The distributed integral arrays are four index arrays, so an intelligent partitioning of the integrals needs to be performed. From an algorithmic standpoint, the simplest distribution of the integrals would be on the fourth index, e.g., for the $[VO|VO]$ integrals, set the number of rows equal to $n_v n_o n_v$ and the number of columns equal to n_o. This is the easiest because there are multiple phases of the algorithm where a permutation is performed on either the first and third or second and third indices of the array. If the array is distributed on the fourth index, then one can do a DDI_GET on some number of columns of the array, and the data that is obtained can be easily permuted on the third index of the array. However, from a memory standpoint distributing the integrals on the fourth index results in non-uniform memory distribution. A close examination of Table 3.8 shows that all three integral classes which are stored in distributed memory have n_o as their last index. Because n_o depends on the molecule and not the basis set, one is limited (by DDI, see §1.2.1) to setting the number of processes to be less than or equal to n_o. Considering that the value of n_o is less than 100, and in some cases much less, this restriction severely limits how large a machine one can use to run the calculation. One would prefer to be able to distribute the array on an index of size n_v, because n_v can be increased (in principle) arbitrarily and is not directly dependent on the molecule of interest. To alleviate the memory distribution problem, the DDI arrays are distributed on the third index. With this partitioning scheme the number of columns of the distributed arrays is either $n_o n_v$ or n_o^2, both of which allow the number of processes to be increased to a significantly large value[6]. This completes the setup phase of the algorithm.

3.3.2.1 CCSD

At this point the CCSD iterations begin. The first part of the CCSD iteration is the calculation of the direct $[VV|VV]$ contribution. As mentioned above, this requires very little memory because the integrals are calculated during each iteration as needed and are not stored. Then the iterations proceed in essentially the same order as the serial algorithm,

[6]This necessary partitioning of the DDI arrays on the third index leads to the duplication of the $[VV|OO]$ and $[VO|VO]$ arrays. Since one cannot perform permutations on the third index if only the first two indices are present, the duplicated arrays are the result of permutations involving the third index. This duplication simplifies the algorithm as the data is already in the proper order.

with the work of the DGEMM calls being broken up and given to each process. If the data required by a process happens to reside in distributed memory, then that process executes a DDI_GET function call to retrieve the requisite data. In a majority of the DDI_GET calls, the process *only* gets the distributed data that is associated with itself. Recall from Chapter 1 that DDI assigns equal amounts of data in a DDI array to each process. If a particular process only requests the data that it was already assigned (when the DDI array was created), then obtaining that data does not involve any network communication; a system memory call is all that is required. This mechanism allows DDI_GET calls to execute quickly because there is no network communication required to obtain the data.

Since this is an iterative calculation, all the processes must move in a lock-step fashion. There simply is no way for processes to work completely independently because they all contribute iteration dependent data necessary for the amplitudes. Because of this, there is a fair amount of process synchronization. Every time a portion of the algorithm requires access to a shared memory array, a SMP_SYNC[7] is required.

As an example of how one type of matrix multiplication is partitioned in our process based algorithm, consider Figure 3.1 and the following description. Assume the shared memory array B (with matrix dimensions of $m \times k$) is going to be multiplied times a block of a distributed array A (where the total array A has matrix dimensions of $k \times n$) to produce the matrix C.

1. Call DDI_DISTRIB[8]

2. Call DDI_GET, obtain the block of the array (consisting of complete columns) which is local to this process and place it in buffer A. Treat A as a $k \times n'$ matrix.

3. Each process performs B times A, as a $m \times k$ times $k \times n'$ matrix multiply, resulting in a $m \times n'$ matrix.

[7]There are two types of synchronization calls in DDI. One is DDI_SYNC, which is a global barrier. All processes must get to the same point in the program before continuing execution. A SMP_SYNC is a barrier such that all processes on a node must be at the same point in the program before continuing and processes on other nodes are ignored. Since the shared memory is *only* accessed by the processes resident on the node, a SMP_SYNC is sufficient to guarantee that data in the shared memory array will be accessed correctly

[8]DDI_DISTRIB is a function that takes the process rank and the array identifier and returns the number of columns and rows of the distributed array which are associated with that process, and therefore stored on the local memory.

Note that since each process is generating a unique portion of matrix C (a block of complete columns), there is no contention for C because each process is writing to a different block of C. If C is in replicated memory, at the end of the computation a global sum operation can be performed on the entire C matrix to obtain the full aggregated result of the matrix multiplication.

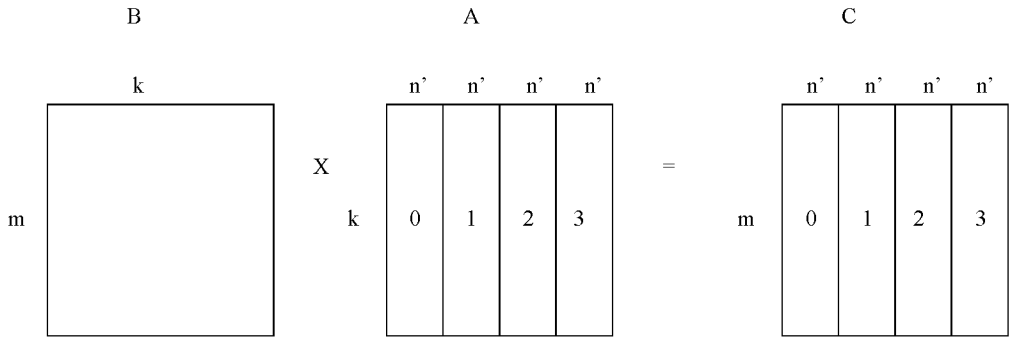

Figure 3.1 Matrix multiplication using four processes where B is in shared memory, A is in distributed memory, and C is either in shared or replicated memory. The number inside the block labels the process that operates on that portion of the matrix.

Another type of matrix multiplication occurs when A and B are multiplied in the opposite order. In this case the matrix multiplication proceeds as follows (see Figure 3.2):

1. Call DDI_DISTRIB
2. Call DDI_GET, obtain the block of the array (consisting of complete columns) which is local to this process and place it in buffer A. Treat A as a $m \times k'$ matrix.
3. Partition B into blocks of complete rows, based on the number of columns of A that each process receives. Treat the portion of B as a $k' \times n$ matrix.
4. Each process performs A times B, as a $m \times k'$ times $k' \times n$ matrix multiply, resulting in a $m \times n$ matrix.

Note that each process produces a $m \times n$ matrix, which then has to be summed together (element by element) to form the final C matrix. This is accomplished by doing a global sum operation on the entire C matrix.

The detailed replicated memory requirement for the CCSD iterations has a leading term of MAX($n_v^3 + 2n_o n_v^2$, $2n_o^2 n_v^2 + 2n_o n_v^2$). Asymptotically this reduces to MAX($\mathcal{O}(n_v^3)$, $\mathcal{O}(n_o^2 n_v^2)$). The n_v^3 memory requirement is due to portions of the algorithm in which a permutation of indices one and three is performed on the $[VV|VO]$ integrals; thus, an entire n_v^3 portion of the array must be in memory at the same time. The $n_o^2 n_v^2$ memory requirement comes from the DGEMMs as shown in Figure 3.2, where the entire resultant matrix must be replicated per process to avoid overwriting shared memory data.

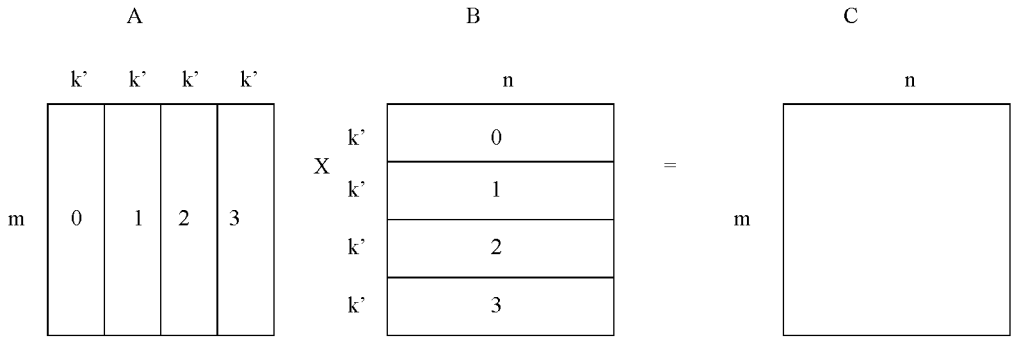

Figure 3.2 Matrix multiplication using four processes where B is in shared memory, A is in distributed memory, and C is in replicated memory. The number inside the block labels the process that operates on that portion of the matrix.

3.3.2.2 (T)

The (T) portion of the algorithm is more straightforward to parallelize than the CCSD component. It consists of three nested loops, each of size n_o. Within each loop, a number of DGEMM calls is made, the largest of which is $\mathcal{O}(n_v^4)$. The nice feature of the (T) calculation is that each loop iteration can be performed independently of all other loop iterations. The only data obtained from the (T) calculation are two double precision values. These are additive values in the sense that each loop iteration contributes a portion of both values, and all the portions are summed together to form the overall result. This allows a partitioning based on processes; i.e., each process executes a chunk of the overall loop iterations. The leading term of the replicated memory requirement for the (T) code is $2n_v^3 + n_o^2 n_v^2$. This requirement comes

from the necessity of storing two temporary work arrays of size n_v^3 and storing the $[VO|VO]$ integral class during each loop iteration.

3.3.3 Node based algorithm

The process based algorithm is successful in parallelizing the work and the load balancing is fairly even. There are portions of the algorithm in which the master process on each node is working (while the other processes are sleeping) to manipulate the shared memory arrays, but that amount of execution time is not significant in the overall runtime. The main downfall of the process based approach is the replicated memory requirement. When one has to replicate (per process) arrays of size n_v^3 or $n_o^2 n_v^2$, the memory cost[9] is simply too high to perform calculations with large input sizes.

Due to the memory cost of the process based algorithm, a complementary node based algorithm has also been designed. The majority of the changes between the process based and node based algorithm are implemented with the goal of conserving memory. The node based CCSD(T) computation begins in the same way as the process based code (see §3.3.2). That is, the Hartree-Fock energy and wavefunction are computed and the parallel integral transformation is performed to setup the various integral arrays for the CCSD(T) calculation, including putting the requisite arrays into distributed or shared memory.

3.3.3.1 CCSD

The CCSD iterations begin in the same fashion by executing the direct algorithm to calculate the contribution of the $[VV|VV]$ integrals to the amplitudes. The overall order of operations for each iteration does not change between the process and node based algorithm. But, whenever there is a DGEMM in the process based code which requires either a n_v^3 or $n_o^2 n_v^2$ replicated array, this portion of the code is adapted to execute in a node based fashion. Generally, this involves partitioning the DGEMM evenly by the number of nodes. Each node gets one portion of the DGEMM to work on. Then each node divides the DGEMM into equal sized

[9]Especially when one considers that larger SMP machines can commonly have from 8 to 16 processors per node

work portions for each process to work on. There are two levels of division, and the rationale is that the large temporary arrays which previously resided in replicated memory now reside in shared memory, one time per node, instead of one time per process. With this being the case, there must be more synchronization of processes in a single node to avoid data corruption due to multiple processes writing to a shared memory array location concurrently. The following is a description of one DGEMM found in the process based algorithm. For reference, the current serial algorithm multiplies the $[VV|VO]$ array (as a $n_v^2 \times n_o n_v$ matrix) times the o_1 amplitudes (as a $n_o n_v \times 1$ matrix) and generates a $n_v^2 \times 1$ matrix. For comparison, the process and node based algorithms will be presented. The process based algorithm runs as follows:

1. Divide n_o by the number of processes so as to assign each process an equal amount of work.

2. Each process obtains a complete n_v^3 portion of the $[VV|VO]$ integrals based on the index calculated in the previous step, resulting in a 4-index array with dimensions $(n_v, n_v, n_v, 1)$. This array is stored in replicated memory.

3. Each process performs a permutation of the first and third index.

4. Each process executes a local DGEMM (as a $n_v^2 \times n_v$ matrix times a $n_v \times 1$ matrix resulting in a $n_v^2 \times 1$ matrix). The second and the resultant matrix are stored in replicated memory.

5. If n_o is greater than the number of processes, then some (possibly all) processes will execute steps 2–4 again with a different portion of the $[VV|VO]$ array until the entire, overall matrix multiplication is performed.

6. After all processes complete their respective work, a global sum is called on the resultant matrix.

The replicated memory requirement for this DGEMM is n_v^3, which is much too large to be stored once per process. The following description outlines the adaptations that are made to convert the process based algorithm to a node based one.

1. Divide n_o by the number of nodes so as to assign each node an equal amount of work.

2. Each node obtains a complete n_v^3 portion of the $[VV|VO]$ integrals based on the index calculated in the previous step, resulting in a 4-index array with dimensions $(n_v, n_v, n_v, 1)$. This array is stored in shared memory.

3. Each node performs the permutation of the first and third index, using a routine which allows all the processors on the node to do the permutation in parallel, without overwriting shared memory data.

4. Each node executes a DGEMM (as a $n_v^2 \times n_v$ matrix times a $n_v \times 1$ matrix resulting in a $n_v^2 \times 1$ matrix). This DGEMM is further split among the processes on the node, by dividing n_v^2 (the row dimension of the first matrix) by the number of processors. The actual DGEMM executed by each process consists of a portion of the first matrix, times the entire second matrix to yield the entire resultant matrix. In this way, each process works on a different portion of the array. The second and the resultant matrix are stored in replicated memory.

5. If n_o is greater than the number of nodes, then some (possibly all) nodes will execute steps 2–4 again with a different portion of the $[VV|VO]$ array until the entire, overall matrix multiplication is performed.

6. After all nodes complete their respective work, a global sum is called on the resultant matrix.

This node based algorithm eliminates the memory requirement of n_v^3 per process and instead stores it once per node. In the case of a node with many processors, e.g., greater than 8, this is a substantial savings. The overall replicated memory for the node based CCSD algorithm has a leading term of $n_v^2 + n_o n_v^2$. Because n_o is typically much less than n_v, this replicated requirement is small enough to allow large sized calculations to be performed. The extra shared memory requirement (in addition to the values given in Table 3.8) is MAX$(n_v^3 + 2n_o n_v^2, n_o n_v^2 + n_o^2 n_v^2)$. The shared memory requirement is equal to or less than the process based replicated memory requirement, illustrating the significant memory savings when moving to a node based algorithm.

Algorithm	Routine	Shared memory	Replicated memory
Process based	CCSD	0	$\mathrm{MAX}(n_v^3,\, 2n_o^2n_v^2) + 2n_on_v^2$
	(T)	0	$2n_v^3 + n_o^2n_v^2$
Node based	CCSD	$\mathrm{MAX}(n_v^3 + 2n_on_v^2, n_on_v^2 + n_o^2n_v^2\ )$	$n_v^2 + n_on_v^2$
	(T)	$2n_v^3$	n_v^2

Table 3.9 Memory requirements for the process and node based algorithms, in addition to the memory requirements given in Table 3.8.

3.3.3.2 (T)

The node based (T) algorithm continues along much the same lines as the node based CCSD component. Recall that the (T) algorithm is three nested loops of size n_o each, and each loop iteration can be executed independently of all the other iterations since no data needs to be communicated between loop iterations. However, the algorithm must avoid the high storage requirements of the process based algorithm. To achieve this goal, the large n_v^3 temporary arrays are stored one time per node in shared memory. Then the loop iterations are divided evenly among the nodes. When a computationally intensive routine (such as a permutation or DGEMM) is encountered, the work is partitioned equally among the processors of the node, with strict control maintained to avoid overwriting shared memory array locations by multiple processors. The extra shared memory requirement is $2n_v^3$. But the replicated memory requirement has a leading term of n_v^2, which represents a significant memory savings over the process based algorithm. Table 3.9 outlines the overall memory requirements for both the process and node based algorithms.

3.3.4 Performance results

To assess the performance of our algorithm, some sample calculations have been performed, using different numbers of processors and different numbers of nodes. The primary machine used is an IBM SMP cluster provided by the Scalable Computing Laboratory, Ames Laboratory. This machine has three SMP nodes, each of which has eight power4 processors (1.7 GHz) and 32 GB of main memory. The two molecules of interest in these tests are luciferin and a T-shaped benzophenol-benzene dimer.

Execution time is the main metric in evaluating the performance of a parallel algorithm. However, simply viewing the execution times rarely allows one to obtain the complete picture of algorithm performance. Two common performance evaluation functions used in parallel computing are speedup and efficiency. Speedup is defined as

$$S(p) = \frac{T(n,1)}{T(n,p)}, \tag{3.1}$$

where n is the input size, $T(n,1)$ is the execution time of the fastest known serial algorithm, and $T(n,p)$ is the execution time of the parallel algorithm using p processors. Perfect speedup corresponds to $S(p) = p$, i.e., using p processors makes the algorithm run p times faster. In practice this is rarely achieved. Efficiency is defined as

$$E(p) = \frac{T(n,1)}{pT(n,p)} = \frac{S(p)}{p}. \tag{3.2}$$

When p is fixed, speedup and efficiency are equivalent measures, differing only by the constant factor p.

Tables 3.10 and 3.11 illustrate execution statistics taken from the process and node based algorithms respectively. The molecule in question is luciferin ($C_{11}N_2O_3S_2H_8$). The performance (paying particular attention to the efficiency values) for the process based algorithm (Table 3.10) is quite good, especially the $[VV|VV]$ subroutine. In fact, it actually exhibits superlinear speedup at 2 processors, which is likely due to cache effects. Once multiple nodes are used, the performance drops quite drastically for the CCSD† (The notation CCSD† indicates one entire CCSD iteration minus the $[VV|VV]$ term). This is due to in part to the high degree of synchronization required for the CCSD iterations and also possibly due to network performance. This calculation is relatively small (in terms of basis functions). Thus, when attempting to partition the work in a fashion that is too fine grained, the overhead due to work partitioning and communication begins to overtake the computation cost.

Table 3.11 shows results on luciferin when executing the node based algorithm. Again one notices that on 1, 2 and 4 processors (i.e., one node) the performance is excellent, with

	Routine	1	2	4	8	16	24
Wall time	$[VV\|VV]$	653	319	165	99	52	36
	CCSD†	643	349	208	135	139	156
	CCSD	1297	668	373	234	191	193
	(T)	37619	19890	11313	8460	4455	3070
Speedup	$[VV\|VV]$	1.0	2.05	3.96	6.61	12.55	18.01
	CCSD†	1.0	1.84	3.08	4.76	4.64	4.09
	CCSD	1.0	1.94	3.47	5.54	6.80	6.71
	(T)	1.0	1.89	3.33	4.45	8.44	12.25
Efficiency	$[VV\|VV]$	1.0	1.02	0.99	0.83	0.78	0.75
	CCSD†	1.0	0.92	0.77	0.60	0.29	0.17
	CCSD	1.0	0.97	0.87	0.69	0.43	0.28
	(T)	1.0	0.95	0.83	0.56	0.53	0.51

Table 3.10 Wall clock execution time, speedup, and efficiency for luciferin run on the power4 IBM machines at SCL. The basis set has $n_o = 46$ and $n_v = 114$. Timing for CCSD is one iteration only. The CCSD† entry shows time spent in a CCSD iteration *excluding* the $[VV|VV]$ routine, while the CCSD entry shows time spent *including* the $[VV|VV]$ routine. The numbers in the column headings indicate the number of processors. The process based algorithms are used in these data.

all the subroutines executing at greater than 80% efficiency. The $[VV|VV]$ routine shows excellent speedup, including superlinear speedup at low processor numbers. Going to 8 or more processors sees a significant drop in performance. Again, since this is an input size on the smaller side of the spectrum, the expected performance gain by using more processors is not achieved when using too many processors.

A test of a larger input size running the node based algorithm is shown in Table 3.12. The efficiency of the $[VV|VV]$ routine is quite good, with a value of 91% at 24 processors. This stands to reason as it does not require much synchronization and is dynamically load balanced. The (T) routine efficiency is also quite good, with a value of 72% at 24 processors. The CCSD† efficiency is less than optimal at 24 processors. In fact, its execution time at 16 and 24 processors is almost identical. The big bottlenecks in the node based CCSD† routine are the frequent synchronization steps. This sample calculation is somewhat of a medium sized problem, so one is encouraged by the performance of the $[VV|VV]$ and (T) routines. As n_v

	Routine	1	2	4	8	16	24
Wall time	$[VV\|VV]$	654	319	161	98	53	36
	CCSD†	690	372	215	147	139	148
	CCSD	1344	691	376	245	191	185
	(T)	21204	11414	6536	3944	2925	2385
Speedup	$[VV\|VV]$	1.0	2.05	4.07	6.67	12.46	17.94
	CCSD†	1.0	1.85	3.21	4.69	4.98	4.62
	CCSD	1.0	1.94	3.58	5.48	7.03	7.26
	(T)	1.0	1.86	3.24	5.37	7.25	8.89
Efficiency	$[VV\|VV]$	1.0	1.03	1.02	0.83	0.77	0.74
	CCSD†	1.0	0.92	0.80	0.59	0.31	0.19
	CCSD	1.0	0.97	0.89	0.67	0.44	0.30
	(T)	1.0	0.92	0.81	0.67	0.45	0.37

Table 3.11 Wall clock execution time, speedup, and efficiency for luciferin run on the power4 IBM machines at SCL. The basis set has $n_o = 46$ and $n_v = 114$. Timing for CCSD is one iteration only. The CCSD† entry shows time spent in a CCSD iteration *excluding* the $[VV|VV]$ routine, while the CCSD entry shows time spent *including* the $[VV|VV]$ routine. The numbers in the column headings indicate the number of processors. The node based algorithms are used in these data.

continues to grow, those two routines will dominate the runtime of the entire algorithm. The CCSD† speedup is less than ideal, but as the problem sizes get larger, it becomes less and less important as it does not asymptotically affect the runtime[10].

When considering the performance of a parallel algorithm, speedup and efficiency are the most common metrics. That is, the quality of a parallel algorithm is based almost entirely on efficient use of CPU. One always hopes to decrease the runtime linearly when adding more processors to the same problem. While this is a valid performance metric, it can be argued that in the case of CCSD(T) (and many other algorithms of scientific interest), *only* viewing CPU efficiency does not give a complete picture. The system resource which is scarce in this algorithm is memory, i.e., the size of the problem that can be calculated is bounded by the amount of system memory available. The utmost effort has been made to conserve memory at all costs, e.g., the $[VV|VV]$ routine recalculates the same integrals every iteration with the

[10]Recall that the asymptotic runtime complexity of CCSD is $\mathcal{O}(N^6)$, due to the $[VV|VV]$ term and the asymptotic runtime of (T) is $\mathcal{O}(N^7)$.

	Routine	4	8	16	24
Wall time	$[VV\|VV]$	5083	2796	1399	936
	CCSD†	1278	808	650	648
	CCSD	6362	3604	2048	1585
	(T)	163403	108202	56280	37770
	CCSD(T)	297154	184151	102131	75861
Speedup	$[VV\|VV]$	1.0	1.82	3.63	5.43
	CCSD†	1.0	1.58	1.97	1.97
	CCSD	1.0	1.77	3.11	4.01
	(T)	1.0	1.51	2.90	4.33
	CCSD(T)	1.0	1.61	2.91	3.92
Efficiency	$[VV\|VV]$	1.0	0.91	0.91	0.91
	CCSD†	1.0	0.79	0.49	0.33
	CCSD	1.0	0.88	0.78	0.67
	(T)	1.0	0.76	0.73	0.72
	CCSD(T)	1.0	0.81	0.73	0.65

Table 3.12 Wall clock execution time, speedup, and efficiency for T-shaped benzophenol-benzene dimer run on the power4 IBM machines at SCL. The basis set has $n_o = 33$ and $n_v = 313$. Timing for CCSD is one iteration only. The CCSD† entry shows time spent in a CCSD iteration *excluding* the $[VV|VV]$ routine, while the CCSD entry shows time spent *including* the $[VV|VV]$ routine. The numbers in the column headings indicate the number of processors. The node based algorithm was used and the performance data were calculated using the 4 process algorithm as the baseline, not the serial algorithm.

sole goal of conserving memory. The use of shared memory (instead of replicated memory) introduces a much smaller algorithmic memory requirement, but also introduces much more process and node synchronization. This necessarily induces a CPU utilization penalty, as processes are sleeping while waiting for synchronization to occur. Less than ideal efficiency of CPU with the benefit of efficient utilization of memory means, in the case of this CCSD(T) algorithm, that larger input sizes can be run.

CHAPTER 4. Conclusions

4.1 General discussion

In this work, a model for the use of hybrid memory programming in scientific computing has been developed and implemented in the CCSD(T) algorithm programmed in GAMESS. Chapter 2 showed the results of using shared memory programming and OpenMP to parallelize the main computation routines of CCSD(T), i.e., the DGEMM routines. This work showed that parallelizing the DGEMMs yielded some speedup, but there are other significant portions of the CCSD(T) algorithm which still executed serially. To achieve a truly parallel code, the work had to be extended from one SMP machine to a cluster of SMP machines.

To facilitate this more extensive parallelization, additions were made to the Distributed Data Interface communication library of GAMESS. Support for shared memory arrays (including direct read and write access) and collective operations such as broadcast and global sum were implemented. This allowed the use of three layers of memory, namely distributed, shared and replicated memory.

Aided with the new memory hierarchy, a fully parallel algorithm was developed for the CCSD(T) equations. The hierarchical algorithm was designed with a primary goal of utilizing the system memory as efficiently as possible. Both process and node based algorithms were developed. The intention is that for the really large calculations, the node based algorithm is the one of choice because the process based algorithm has a replicated memory requirement which is fairly high. This is due in no small part to the fact that when certain partitionings of DGEMM are performed, the entire resultant matrix must be stored on each process. These large replicated memory requirements are eliminated in the node based algorithm by storing the large arrays in shared rather than replicated memory space. This results in more SMP

synchronizations but also resulted in a much smaller replicated memory requirement.

Both the process and node based algorithms were described in detail, and performance results were given to indicate relevant execution time improvement over the serial algorithm. For a relatively small input size (Tables 3.10 and 3.11), the parallel efficiencies were quite good on one node. Adding multiple nodes did not show a proportionate increase in efficiency. For a medium sized input (Table 3.12) the efficiencies were better than for the small input size, especially as the number of processors was increased.

Overall, the node based algorithm was found to have good efficiency for the $[VV|VV]$ and (T) subroutines, while the CCSD† efficiency was not as good. However, this does not pose a major concern in the large input size limit, as the CCSD(T) algorithm complexity is dominated by the $[VV|VV]$ and (T) subroutines. It was also shown that in this particular parallel code, with memory being the greatest bottleneck, to conserve memory at the expense of CPU is sometimes the *only* way, in the context of this algorithm, to run the largest input sizes possible.

4.2 Future research

As has been mentioned a number of times in this thesis, the greatest barrier to really large CCSD(T) calculations is the high memory requirements. With that restriction in mind, one is always looking for methods to reduce the memory requirements. To this end, one could add a direct, "on the fly" calculation which calculates the $[VV|VO]$ each iteration, instead of calculating the integral values once and storing them in distributed memory. This would eliminate the largest distributed array that is currently stored. Once could also implement a fully distributed model, where each of the large arrays which are currently stored in shared memory, would then be stored once in distributed memory. This would eliminate the advantages of shared memory, but would reduce the overall memory requirement for the algorithm. This would also require much more communication and synchronization, as the DDI arrays would be read *and* written, while the current algorithm only reads from the DDI arrays and does not write them.

APPENDIX Parallel communication functions

- `put`

  Function that takes a local memory buffer and places the contents of that memory into distributed memory.

- `get`

  Function that obtains a portion of distributed memory and places it into a local memory buffer

- `accumulate`

  Function that operates similar to `put` but instead of writing the local buffer to distributed memory, it first performs a binary operation (such as add or multiply) between the local memory and the pre-existing distributed memory, element by element. The result of the binary operation is then stored in distributed memory.

- `broadcast`

  Function that takes a local memory buffer and sends it to all other processes.

- `global sum`

  Function that takes a memory buffer from each process, adds them all together and writes the result in the original memory buffer.

- `synchronize`

  Function that places a barrier on process execution. Processes wait at the barrier until all the processes are at the same execution point in the program before proceeding.

REFERENCES

[1] The MPI Forum, MPI-1.1, A Message-Passing Interface Standard, accessed October 16, 2006. (http://www.mpi-forum.org/docs/docs.html).

[2] OpenMP Application Program Interface, Version 2.5, accessed October 16, 2006. (http://www.openmp.org/drupal/node/view/8).

[3] G. Jost, H. Jin, D. an Mey, F. F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster," *Proceedings of the Fifth European Workshop on OpenMP*, EWOMP 03, Aachen, Germany, Sept. 22-26, 2003.

[4] R. Rabenseifner, "Hybrid Parallel Programming on HPC Platforms," *Proceedings of the Fifth European Workshop on OpenMP*, EWOMP 03, Aachen, Germany, Sept. 22-26, 2003.

[5] R. Rabenseifner, G. Wellein, "Comparison of Parallel Programming Models on Clusters of SMP Nodes," *Proceedings of the International Conference on High Performance Scientific Computing*, Hanoi, Vietnam, Mar. 10-14, 2003.

[6] R. Rabenseifner, "Hybrid Parallel Programming: Performance Problems and Chances," *Proceedings of the 45th Cray Users Group Conference*, Columbus, Ohio, USA, May 12-16, 2003.

[7] E. Chow, D. Hysom, "Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters", *Lawrence Livermore National Laboratory Technical Report UCRL-JC-143957*, May 2001.

[8] M. Krishnan, Y. Alexeev, T. L. Windus, J. Nieplocha, "Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays", *Proceedings of the ACM/IEEE Supercomputing Conference,* Seattle, Washington, USA, Nov. 12-18, 2005.

[9] L. Smith, M. Bull, "Development of mixed mode MPI/OpenMP applications," *Scientific Programming,* vol. 9, pp. 83-98, 2001.

[10] M. W. Schmidt et al., "General Atomic and Molecular Elecronic Structure System," *J. Comput. Chem.,* vol. 14, pp. 1347-1363, 1993.

[11] GAMESS User's Guide, accessed August 1, 2006. (http://www.msg.ameslab.gov/GAMESS/GAMESS.html).

[12] B. M. Bode, Associate Scientist, Ames Laboratory, U.S.DOE, Ames, IA 50011. Personal Communication, August 2006.

[13] G. D. Fletcher, M. W. Schmidt, B. M. Bode, M. S. Gordon, "The Distributed Data Interface in GAMESS", *Comp. Phys. Comm.*, vol. 128, pp. 190-200, 2000.

[14] R. M. Olson, M. W. Schmidt, M. S. Gordon, A. P. Rendell, "Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model", *Proceedings of the ACM/IEEE Supercomputing Conference,* Phoenix, Arizona, USA, Nov. 15–21, 2003.

[15] J. Cizek, "On the Correlation Problem in Atomic and Molecular Systems. Calculation of Wavefunction Components in Ursell-Type Expansion Using Quantum-Field Theoretical Methods", *J. Chem. Phys.,* vol. 45, pp. 4256-4266, 1966.

[16] A. Szabo, N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, Dover Publications, Reading, MA, 1986.

[17] I. N. Levine, *Quantum Chemistry*, 5th edition, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2000.

[18] T. D. Crawford, H. F. Schaefer, "An Introduction to Coupled Cluster Theory for Computational Chemists", in *Reviews in Computational Chemistry*, K. B. Lipkowitz and D. B. Boyd, Eds. VCH Publishers, New York, NY, USA, vol. 14, pp. 33-136, 2000.

[19] G. D. Purvis, R. J. Bartlett, "A full coupled-cluster singles and doubles model: The inclusion of disconnected triples", *J. Chem. Phys.*, vol. 76, pp. 1910-1918, 1982.

[20] G. E. Scuseria, C. L. Janssen, H. F. Schaefer III, "An efficient reformulation of the closed-shell coupled cluster single and double (CCSD) equations", *J. Chem. Phys.,* vol. 89, pp. 7382-7387, 1988.

[21] P. Piecuch, S. A. Kucharski, K. Kowalski and M. Musial, "Efficient computer implementation of the renormalized coupled-cluster methods: The R-CCSD[T], R-CCSD(T), CR-CCSD[T], and CR-CCSD(T) approaches," *Comput. Phys. Commun.* vol. 149, pp. 71-96, 2002.

[22] NWChem program package, developed at Pacific Northwest Laboratory, PO Box 999, Mail Stop K1-96, Richland, WA 99352.

[23] R. Kobayashi, A. P. Rendell, "A direct coupled cluster algorithm for massively parallel computers", *Chem. Phys. Lett.*, vol. 265, pp. 1-11, 1997.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the Department of Energy under contract W-7405-ENG-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing Research. This material is based on work supported by the National Science Foundation under Grant No. CHE-0309517. This research was performed in part using computational resources in the Scalable Computing Laboratory which were partially donated from IBM Corporation under the Shared University Research grant program. This research also used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098 with the University of California.

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Ricky Kendall for his guidance, patience, and support throughout this research and the writing of this thesis. I thank Dr. Mark Gordon for his helpful comments and suggestions with respect to GAMESS. Thanks also to Dr. Robyn Lutz for her support during my graduate career.

Thanks are also in order to Ryan Olson, my principal graduate student collaborator, for his helpful suggestions and discussions.

I would like to thank the Department of Computer Science office staff for helping me with all the administrative tasks and meet all the deadlines.

Lastly, I thank my family for their support, and a special thanks to my wife Jennifer and daughter Hannah, who have supported me throughout my graduate education and because of their unselfishness, have allowed me to pursue my academic and personal goals vigorously.