Efficient execution of electronic structure calculations on SMP clusters

by

Nurzhan Ustemirov

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:
Masha Sosonkina, Co-major Professor
Yan-Bin Jia, Co-major Professor
Mark S. Gordon
Hylke Vandewetering

Iowa State University

Ames, Iowa

2006

Graduate College
Iowa State University

This is to certify that the master's thesis of

Nurzhan Ustemirov

has met the thesis requirements of Iowa State University

Co-major Professor

Co-major Professor

For the Major Program

DEDICATION

I would like to dedicate this thesis to my family especially my mom, Bibihadisha E. Erdenova, and my sister, Raushan L. Ustemirova for their support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES vi

ACKNOWLEDGEMENTS viii

ABSTRACT ix

CHAPTER 1. Introduction 1

1.1 *Ab initio* Quantum Chemistry 1

1.2 Related Work 2

1.3 Thesis Layout 4

CHAPTER 2. Background 5

2.1 General Atomic and Molecular Electronic Structure System 5

2.2 NICAN Middleware Tool 6

2.3 Symmetric Multiprocessor Architecture 8

CHAPTER 3. Adapting GAMESS 10

3.1 Adapting Sequential GAMESS in Dedicated Environment 11

3.2 Adapting Parallel GAMESS in SMP Clusters 13

3.3 Dynamic Algorithm Selection in GAMESS Calculations 15

CHAPTER 4. Test Results 18

4.1 Sequential GAMESS 18

4.2 Parallel GAMESS 20

4.3 Dynamic Algorithm Selection in GAMESS 23

4.4 Hyper-Threading 27

CHAPTER 5. Conclusion and Future work 30

5.1 Future Work 31

BIBLIOGRAPHY 32

APPENDIX . GAMESS Capabilities 35

LIST OF FIGURES

2.1 NICAN Layout 7

2.2 Sample XML file monitoring load on CPU 7

2.3 SMP architecture. 4 CPUs on a single node. Node resources, such as memory, are equally shared among CPUs 8

3.1 GAMESS-NICAN Integration model for dedicated environment 11

3.2 NICAN Manager decision tree 12

3.3 Decision making by NICAN to invoke GAMESS adaptations 14

3.4 Illustration of the SCF algorithm as *conventional* (C) and *direct* (D) implementations 16

3.5 GAMESS-NICAN Integration 17

3.6 GAMESS adaptation scheme 17

4.1 Execution time of 12 identical jobs 18

4.2 Time to complete a pool of 12 jobs 19

4.3 AM1 structure of Penicillin molecule obtained by GAMESS RHF calculation 21

4.4 Four-processor calculations of twelve identical molecules 22

4.5 Eight-processor calculations of twelve identical molecules 22

4.6 Execution of all GAMESS eight-processor jobs in a queue 23

4.7 Luciferin molecule structure obtained by GAMESS RHF calculation . 24

4.8 Two-processor execution of parallel GAMESS job for different amounts of I/O congestion, C_{io} 25

4.9 Total time to complete two simultaneous parallel GAMESS jobs for different adaptivity settings 26
4.10 Concurrent jobs: single CPU 28
4.11 Concurrent jobs: SMP node with dual CPU 29

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank Dr. Sosonkina for her guidance throughout the research and valuable input in writing this thesis. Without her leading role and advising skills none of this work would have been possible.

Special thanks to Dr. Mark Gordon and Dr. Mike Schmidt, for their patience in explaining quantum chemistry and GAMESS calculations.

I would also like to thank my committee members for their efforts: Dr. Mark Gordon, Dr. Yan-Bin Jia and Dr. Hylke Vandewetering.

This work was performed at Ames Laboratory under Contract No. W-7405-Eng-82 with the U.S. Department of Energy. The United States government has assigned the DOE Report number IS-T 2595 to this thesis.

ABSTRACT

Applications augmented with adaptive capabilities are becoming common in parallel computing environments. For large-scale scientific applications, dynamic adjustments to a computationally-intensive part may lead to a large pay-off in facilitating efficient execution of the entire application while aiming at avoiding resource contention. Application-specific knowledge, often best revealed during the run-time, is required to initiate and time these adjustments. In particular, General Atomic and Molecular Electronic Structure System (GAMESS) is a program for *ab initio* quantum chemistry that places significant demands on the high-performance computing platforms. Certain electronic structure calculations are characterized by high consumption of a particular resource, such as CPU, main memory, or disk I/O. This may lead to resource contention among concurrent GAMESS jobs and other programs in the dynamically changing environment. Thus, it is desirable to improve GAMESS calculations by means of dynamic adaptations. In this thesis, we show how an application- or algorithm-specific knowledge may play a significant role in achieving this goal. The choice of implementation is facilitated by a module-driven middleware easily integrated with GAMESS that assesses resource consumption and invokes GAMESS adaptations to the system environment. We show that the throughput of GAMESS jobs may be improved greatly as a result of such adaptations.

CHAPTER 1. Introduction

Cluster computing has emerged as a mainstream platform for high-performance applications. More recently, low-cost CPUs enabled assembling clusters from symmetric multiprocessor nodes (SMP). This affordable means of parallel computing enabled wide-spread usage of parallel applications for scientific computing. However, achieving a sustainable performance across multiple SMP nodes became a challenge in part because of inadequate load management leading to resource contention. Specifically, in a typical multiuser cluster environment, where users could concurrently run a variety of high performance applications, simultaneous processes may compete for shared resources.

1.1 *Ab initio* Quantum Chemistry

Chemistry is the science dealing with construction, transformation, and properties of molecules (15). Traditionally molecules are considered as a collection of positively charged nuclei and negatively charged electrons. The number of these particles and the physical force between them shape the energies of the molecule and the distribution of electrons and nuclei in molecule space (i.e., geometry). The subfield of chemistry that studies the geometry, energies, and properties of molecules is called *quantum theory* and is based on the simplified time-independent Schrödinger equation:

$$H\psi = E\psi, \tag{1.1}$$

where H is a Hamiltonian operator, ψ is set of wave-functions and E is the total energy of the system. Solving the time-independent Schrödinger equation directly for molecules is very

complicated. Though, there are ways (*semiempirical* and *ab initio*) to approximate the solution to the given equation. *Semiemprical* approximation uses experimental data to substitute some of the mathematical terms in the equation. On the other hand, *ab initio* (Latin: "from the beginning") is a collection of methods that uses only mathematical means to iteratively generate the approximate solution (15). Avoiding the use of experimental results comes at the expense of significantly larger computational resources needed for *ab initio* methods. The problem size for a typical computation depends on the accuracy desired for the final result. An upper bound for a problem size is N^4, where N is a number of basis functions, describing an atom, which could go up to hundreds.

The report in (9) lists several applications that perform *ab initio* calculations. Just to mention some of them, *MOLPRO* is a complete package of *ab initio* programs for molecular electronic structure calculations. Using recently developed techniques, it is capable of highly accurate and efficient calculations mostly small molecules (32). *NWChem* is a computational chemistry package that is designed to run on high-performance parallel supercomputers as well as conventional workstation clusters. It aims to be scalable both in its ability to treat large problems efficiently, and in its usage of available parallel computing resources (12). MPQC is another computational chemistry package that was built from the start to run in parallel. Designed in object oriented fashion, MPQC runs on a wide range of computing architectures (14). *GAMESS* is a popular generic *ab initio* program freely distributed and developed at Iowa State University (24). Our research is applied to GAMESS, thus more detailed discussion of the program is in the subsequent chapters.

1.2 Related Work

A crucial part of high-performance computing (HPC) systems is proper resource management, such as load balancing. To share the common purpose of load balancing, preserve and protect jobs until the end of execution, *batch schedulers* are commonly used tools for scheduling jobs. There exists much research on batch schedulers: Many were abandoned and a few are under constant improvement. Some of the popular batch schedulers currently available are.

Portable Batch Scheduler (PBSPro/OpenPBS) is one of the most popular "all-purpose" batch job and computer system resource management packages. PBS accepts batch jobs and a shell script with the control attributes, preserves and protects the job before it starts, runs the job, and delivers output to the submitter (2). PBS tries to map a job based on three factors: required resource attributes set by the user, available system resources, and the implemented scheduling policy. Depending on system parameters such as scale of the managed system, OS and level of fault tolerance there are two versions of PBS. *OpenPBS* is a freely available Open Source version of PBS, limited to Linux and UNIX Operating Systems. It supports fair scheduling up to hundreds of CPUs and suits well the needs of relatively small systems. *PBSPro* is a highly scalable commercial version that has better scheduling algorithms and supports multiple platforms, operating systems and thousands of CPUs.

LoadLeveler (LL) manages both serial and parallel jobs over a network of machines. Initially developed for the IBM SP machines, LL supports now a variety of platforms. Along with the standard features for batch schedulers such as queueing submitted jobs, and running them based on preferences set by the user and system administrator, LL supports an individual load configuration for each computing unit. In addition, LL provides three different ways to interact with the user, command line interface, graphical user interface and an API for a direct interaction with the application (18).

Load Sharing Facility (LSF), by Platform Computing Corporation, is a software package for managing and accelerating batch workload processing for compute- and data-intensive applications (23). A distributed load sharing package for heterogeneous UNIX environments, LSF manages a network of machines as a single transparent computing system. This feature allows users to run programs remotely that are not available on a local host. As with PBS, there are two different versions of LSF, *Platform LSF* and *Platform LSF HPC*. The first one is a generic scheduling package that could handle batch, interactive, distributed and parallel jobs. The second one is an optimized version to run parallel applications on HPC systems. A distinctive feature of LSF is the ability to dynamically migrate running jobs between nodes.

In an environment where there is a need to execute multiple high-performance applications

while maximizing their throughput, a scheduling that diminishes resource contention based on the knowledge of a current algorithmic stage is highly desirable. Unfortunately, modern "all-purpose" (system-level) schedulers, such as PBS (2) and Load Leveler (18), have limited capabilities in this respect since they do not have a means to "peek" into an application's execution. On the other hand, incorporating self-scheduling mechanisms into an application based on the resource availability often appears infeasible since the system-level details may obscure the application usage and development. The latter is of particular concern since GAMESS is designed to run on a given computational platform by an application scientist without any knowledge of computer science. Therefore, employing some middleware tools (1; 5; 13), which may access the dynamic system conditions and invoke GAMESS adaptations – via a separate function handler, – may be beneficial, assuming that such tools are lightweight and may be turned on/off by the user in a simple manner. One such middleware tool is NICAN, a framework which invokes adaptation functionality of distributed applications (25). In this thesis we present an integration of NICAN into GAMESS as an application-level adaptation facilitator.

1.3 Thesis Layout

The thesis is organized as follows. Chapter 2 gives background information on GAMESS, NICAN, and SMP architecture. In Chapter 3, we present a three stage evaluation of the GAMESS-NICAN integration model, its functional components, and adaptation mechanism. Chapter 4 presents experimental results for each stage of the GAMESS-NICAN integration case. The comparison is made for the original implementation of GAMESS and GAMESS-NICAN integrated software. The 5th Chapter summarizes our research and outlines future work.

CHAPTER 2. Background

2.1 General Atomic and Molecular Electronic Structure System

The General Atomic and Molecular Electronic Structure System (GAMESS) was based on HONDO5 and several other quantum chemistry applications (24). Since then, GAMESS has emerged as a complex, widely used *ab initio* package, with a single code portable on a wide range of computing platforms. Over time, most quantum chemistry applications in GAMESS have been parallelized. This enables GAMESS to run on HPC environments and utilize distributed resources, such as main memory and disk storage. More recent developments of GAMESS include implementation of the Distributed Data Interface (DDI). The DDI data server model takes advantage of and better utilizes the shared memory on symmetric multiprocessor (SMP) nodes (22).

The structure of atoms and molecules in Schrödinger equation is formulated by wavefunctions. In general, these wavefunctions are approximated using iterative *ab initio* methods. Iterative approximation methods include Hartree-Fock (HF) or Self-Consistent Field (SCF). A wide range of HF wavefunction calculations that GAMESS can perform includes Restricted HF (RHF), Restricted Open-Shell HF (ROHF), and Unrestricted HF (UHF). A more detailed list of current capabilities of GAMESS is given in the appendix. There are two different implementations, *direct* and *conventional*, of the SCF method in GAMESS. The *direct* algorithm recomputes integrals "on-the-fly" for each iteration mainly consuming physical memory and CPU resources. Conversely, the *conventional disk-based* algorithm calculates integrals once, stores them on disk, and reuses them during iterations resulting in a heavy disk I/O usage.

We have observed (29) that a parallel job in which computing processes are *equally distributed (scattered)* over different nodes runs faster the one using multiple processors within

the same SMP node. This could be due to the inability of certain communication layer implementations to efficiently handle shared-memory access (26; 7). On the other hand, by using a single processor per node (*scattering*), each process utilizes the networking hardware without waiting for CPU or memory. In addition, we have observed that a GAMESS *conventional* parallel job on two processors of the same SMP node may be actually *slower* compared to its sequential counterpart. The poor performance could be the result of a disk I/O channel bottleneck due to simultaneous access by more than one *conventional* processes constituting a parallel job. On the other hand, running concurrently scattered parallel GAMESS jobs with only one designated as a *conventional* SCF algorithm on the same set of SMP nodes, has shown rather good performance. Thus, the choice of an SCF algorithm has a significant effect on performance.

2.2 NICAN Middleware Tool

The main idea of integrating Network Information Conveyer and Application Notification (NICAN) with an application is to decouple the process of monitoring *system information* from the execution of the application, while providing the application with critical *system knowledge* in a timely manner through adaptation handlers (27). NICAN was successfully integrated with several application to aid their execution in a distributed environment. In his Master's thesis, Gan Chen has used NICAN to monitor the network bandwidth, analyze the collected data and notify the network benchmarking tool NetPIPE (6). In a similar way, Sosonkina and Kulkarni had integrated NICAN with the Parallel Algebraic Multilevel Solver (pARMS), where NICAN augments pARMS's execution to changes in the network condition (16).

The NICAN architecture could be divided into two main parts, the *front end* and the *back end*. The front end consists of a NICAN interface to the integrated application (Figure 2.1, NICAN Interface) and a set of adaptation triggering conditions in an XML file (Figure 2.2). The back end performs necessary monitoring and system resource analysis to aid the execution of the application in a dynamically changing environment. More specifically, the NICAN engine is encapsulated into a separate thread (Figure 2.1, Module Manager). Each module is

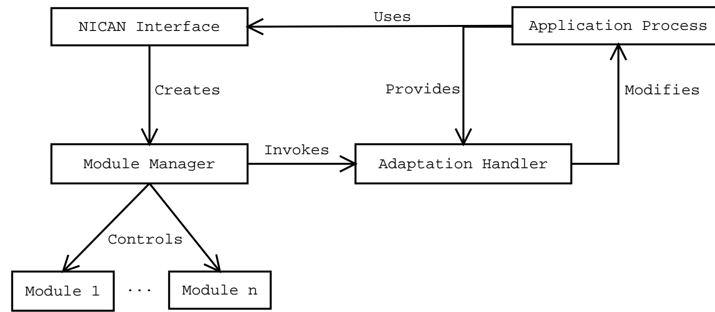

Figure 2.1 NICAN Layout

designated to perform a separable function, such as to watch over a certain system parameter or to validate machine-dependent parameters. Whenever an analyzed system resource monitored by NICAN reaches a critical range the module invokes adaptation handler (Figure 2.1, Adaptation Handler) which in turn changes the execution pattern of the application. Triggering conditions for a critical range are set by the user in the XML file. If, for a particular resource, a critical range is never met, then the adaptation is never triggered and the application will proceed as if NICAN had never been started. Enhanced with "general-use" modules, such as CPU monitoring or Packet Probing, NICAN may not require customized integration with an application – these modules may be utilized "as is" in a variety of applications. However, to explore application-specific trigger conditions, "specific-use" NICAN modules may also be needed.

```
<specification>
    <module_group handler = ‘‘HandlerOne’’>
        <module file = ‘‘cpu_load.so’’
            interval = ‘‘10’’
            monitor = ‘‘ONE’’
            trigger = ‘‘ABOVE’’
            value = ‘‘.6’’ />
    </module_group>
</specification>
```

Figure 2.2 Sample XML file monitoring load on CPU

In a sample XML file (Figure 2.2) NICAN *cpu_load.so* modules monitors CPU load every ten seconds (*interval="10"*) for a one minute (*monitor="ONE"*) period. Whenever the CPU load rises above a triggering point (*value = ".6"*), NICAN calls the adaptation handler *HandlerOne()* which makes the necessary adaptation for the integrated application .

2.3 Symmetric Multiprocessor Architecture

Symmetric Multiprocessor (SMP) is an architecture with multiple processors on the same system that have identical access to the system resources (11). A single Operating System (OS), specifically build for SMP, controls all CPUs and interchangeably uses them to run simultaneous processes. OS could map multiple sequential applications at the same time or run in parallel a single multithreaded application. Such concurrent execution of multiple processes is only efficient if processes ideally overlap with one another. Since system resources are equally shared among processors, simultaneous access could lead to resource contention due to the bottleneck of resource hardware or communication media (30).

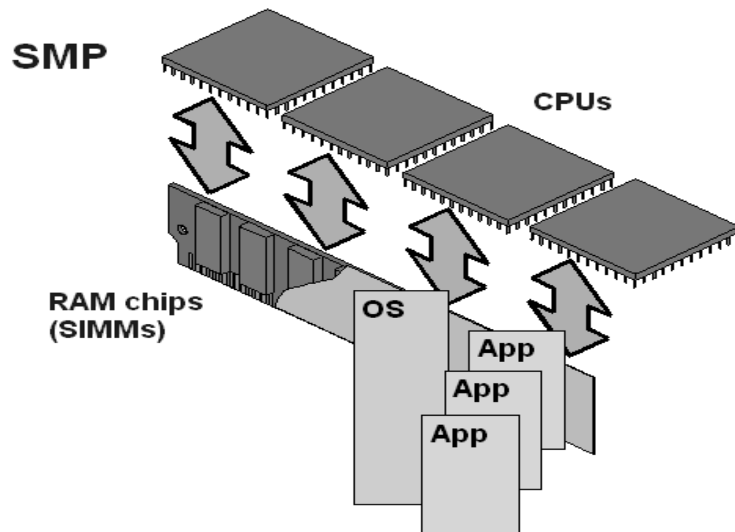

Figure 2.3 SMP architecture. 4 CPUs on a single node. Node resources, such as memory, are equally shared among CPUs

Figure 2.3, from Desktop Computer Encyclopedia, shows a diagram of the SMP architec-

ture. All CPUs have equal access to the physical memory (RAM) and OS could concurrently run multiple applications. The number of CPUs on a typical SMP node could vary from two processors up to 32. One of the first SMP systems was built by Sequent, Pyramid and Encore (10). Currently, most HPC vendors such as IBM, HP and Dell provide solutions based on SMP systems.

CHAPTER 3. Adapting GAMESS

To address the challenges that GAMESS faces running on SMP clusters, we may need an adapting mechanism for improving performance. Modifying GAMESS source code to implement self-adaption is rather difficult. It may decrease its usability by application scientists and its portability, which is highly acclaimed feature of GAMESS.

The GAMESS package consists of two main parts. The first one, written in C, handles the setup of DDI needed for efficient memory management in parallel executions (22). The second part, the native code written in Fortran, does the actual computations. In order to avoid making changes in the computation part of GAMESS we consider a high-level approach to adaptation: Adjust GAMESS input settings based on current system conditions and on the settings of the "peer" GAMESS jobs already running. Therefore, employing middleware tools, such as NICAN, may be beneficial. An attractive feature of NICAN is that it does not require substantial code modifications to the high-performance application with which NICAN is interfaced. In the case of GAMESS, we have only a few changes. First, the application includes a header file that contains declarations for the NICAN interface functions. Second, two single lines are inserted, to initialize and to finalize the NICAN thread, at the beginning and at the end of the execution, respectively. Third, a simple handler function is added to resume GAMESS after NICAN has finished the decision making process.

The efficient execution of GAMESS with the aid of NICAN middleware was evolved in three stages. The first stage focused on improving concurrent execution of sequential GAMESS jobs. In the second, the integrated model was modified to include parallel GAMESS jobs on SMP clusters. In the final stage, with slight modifications to the native code, GAMESS was able to dynamically adapt to the changing environment.

3.1 Adapting Sequential GAMESS in Dedicated Environment

The initial work in improving GAMESS was started with sequential runs. The execution media was assumed to be dedicated to GAMESS jobs only. Thus, the adapting mechanism was aimed to facilitate concurrent electronic structure calculations on a single SMP node.

GAMESS is integrated with NICAN through its interface in *front end* (2.2). Users pass instructions to NICAN through option tags in an XML file. The first option (`do_check = ‘‘YES’’`) activates the system environment checking modules, and the second (`use_sug =‘‘YES’’`) enables or disables the adaptation by NICAN. By default, both of these options are set to **NO**. In any case, NICAN records the execution pattern (*direct* or *conventional*) of the current job in the *active list*. Useful in scheduling concurrent GAMESS jobs that are arriving, the active list is a data structure stored in the shared memory for the interprocess communication.

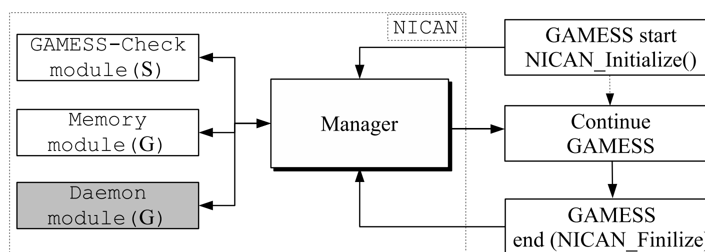

Figure 3.1 GAMESS-NICAN Integration model for dedicated environment

As depicted in Figure 3.1, GAMESS spawns a NICAN thread which, in turn, starts the Manager. General-use modules are labeled with `(G)` and specific-use ones with `(S)`. The Manager parses the XML file for options and checks the *active list* for the presence of other GAMESS-NICAN processes and their execution patterns. In parallel with the Manager, the Check module re-runs GAMESS on the user input file but with the special “check” execution mode. This execution mode is designed in GAMESS to very quickly *estimate* the required main memory with respect to the desired electronic structure calculations without actual computation (24). Then, the input file may be updated with this estimate. Without this estimate,

a user could mispredict the required amount and the computations may fail due to a lack of physical memory. To prevent this, the Manager detects free memory, compares it to the estimate from the Check module, and halts the job until available memory in the system reaches the required amount. The Daemon module is implemented in the second stage and discussed in the next section.

Granted the permission to make changes to the GAMESS input file, the Manager selects an execution pattern according to a predefined set of rules (Figure 3.2). Namely, it updates the input file flag specifying *direct* or *conventional* execution pattern. The Manager also records the execution pattern in the *active list*, and resumes GAMESS. Note that the GAMESS execution is postponed (expressed as the dotted arrow in Figure 3.1) while NICAN modules examine the system and decide on the execution pattern to be suggested for the continuation of GAMESS. After a GAMESS job has completed the task, it calls NICAN_Finalize to remove itself from the *active list*.

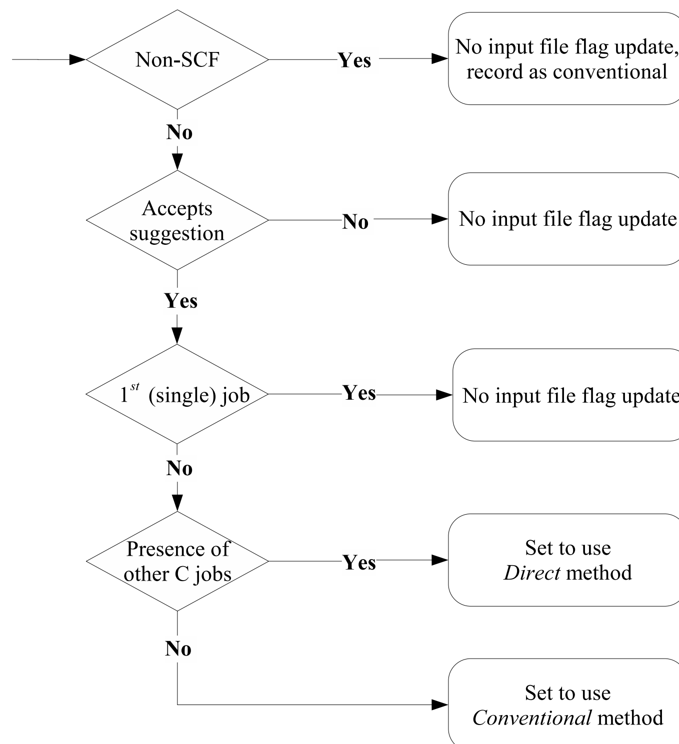

Figure 3.2 NICAN Manager decision tree

Figure 3.2 depicts the decision tree of the co-scheduler. Some types of GAMESS calculations do not have clear separation into the *direct* and *conventional* methods. Thus, regardless of the current job pool, the co-scheduler records them in the *active list* as jobs with *conventional* method because of their possible massive consumption of I/O resources. If none of the active jobs are designated as *conventional*, NICAN will designate $n-1$ (n is a number of processors in the system) jobs to execute using *direct* method. Only a single job is assigned the *conventional* integral calculation method.

The decision tree could be modified to better utilize the Hyper-Threading (HT) technology if the processes are not allowed to migrate to other logical processors. At present, however, such a migration happens on the HT-enabled machines. The scheduling modification to utilize HT could be achieved by setting the CPU affinity mask of a process. The affinity mask is a system data structure in which each bit relates the process to a logical processor. If a given bit is set then the process is allowed to execute on the corresponding logical processor. The affinity system calls (processor assignments) are introduced in the later releases of the kernel (17). Thus a corresponding modification of the decision tree is left for the future work. We study performance of GAMESS in a system equipped with HT technology as a separate section in the Test Results 4.4.

3.2 Adapting Parallel GAMESS in SMP Clusters

Many GAMESS calculations have parallel implementations which enable the utilization of distributed resources, such as main memory and disk storage. Applying the same integration model from sequential GAMESS to parallel was not straightforward. The main constraint was the distributed nature of parallel jobs which complicated analysis of concurrently running peer GAMESS jobs. A part of a distributed GAMESS job could be running on an SMP node alone and another part could be sharing a node with other GAMESS jobs. Since there is only one instance of the NICAN manager per job running on the master host, where GAMESS was launched, remotely maintaining an active list on a local host was impractical. The enhanced integration model presented here handles execution of both sequential and parallel

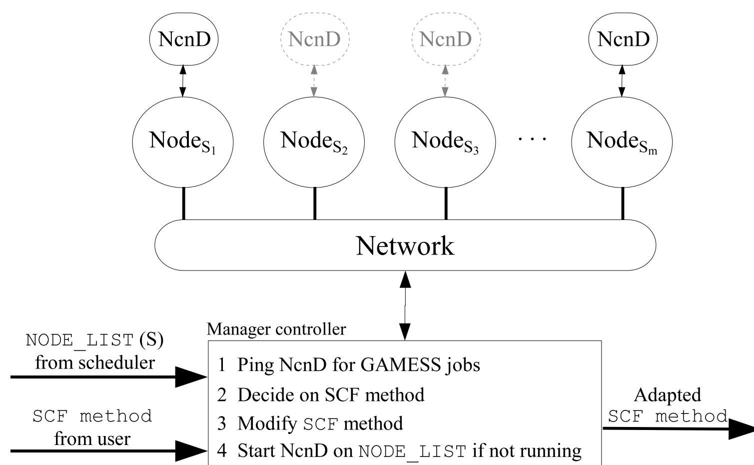

Figure 3.3 Decision making by NICAN to invoke GAMESS adaptations

GAMESS jobs across multiple SMP nodes interconnected in a cluster environment. The same assumption remains that a cluster system is dedicated to run GAMESS jobs only. In other words, no non-GAMESS/NICAN application is expected to run on the cluster. Thus, adapting GAMESS, for efficient execution, could take place at the preprocessing stage of electronic structure calculations.

The integration model for parallel GAMESS is the same as that for sequential GAMESS (Figure 3.1). The only difference is the Daemon module which starts the NcnD daemon (shaded color in the Figure 3.1) if there is no other NcnD running on the node. The daemon is self-contained and is independent from the job for which it has been started. The daemon performs standard operations, such as read, write, delete, and self-destroys if there are no more records. Each record consists of a process ID and a description to the job. Thus, the Manager can record any useful information about the processes executed on the node.

Figure 3.3 depicts the decision making process implemented as a controller on the Manager. On input, the controller receives the list of nodes (NODE_LIST) assigned by the scheduler and the GAMESS SCF method. On output, the controller returns the new adapted SCF method for the job. The control algorithm is shown in the rectangular box in Figure 3.3. The Manager communicates (line 1) with NICAN daemons on all the nodes $S_i, i = 1, ..., m$. Each daemon

returns a list of GAMESS jobs running on the node and the corresponding SCF methods. Nodes with no GAMESS jobs (Node_{S_2} and Node_{S_3}) will not have NcnD and will not respond to the request. In that case, the node is regarded as vacant and the job could run on it efficiently with any SCF method. With the information supplied by NcnD daemons, the controller selects an optimal SCF method for the current job (line 3). The decision process is based on the assumption that an SMP node has a single I/O channel and that running concurrent *conventional* jobs on such systems is very costly (30). Thus, the Manager avoids running more than one *conventional* job on the same SMP node. In the case of multiple I/O channels, the Manager controller is easily adaptable. For example, if there are two I/O channels, then the Manager could schedule two *conventional* jobs on the same SMP node. After successfully starting the parallel GAMESS computation, the Manager records its SCF method in NcnD daemon to efficiently run future GAMESS jobs mapped on the same node.

3.3 Dynamic Algorithm Selection in GAMESS Calculations

The first two GAMESS-NICAN integrations models were designed to encapsulate the entire GAMESS computation code (29; 30). A high-level approach was considered for adaptation: Adjust GAMESS *input* settings based on the system conditions and on the settings of the "peer" GAMESS-NICAN jobs co-existing in the system. NICAN Manager made changes to the GAMESS input parameters by selecting the appropriate SCF algorithm at the preprocessing stage. The design assumed that the system is dedicated to run only GAMESS jobs. However, the assumption is too restrictive for modern realistic computing environments when e.g., several applications may reside on the same SMP node or when the I/O channel is shared by many applications. Thus, adaptations made only in the preprocessing stage may not always lead to an improvement under such system conditions, and thus dynamic adaptations are desirable.

The SCF algorithm is one of the computationally-intensive parts of an electronic structure calculation that GAMESS performs. Thus, a proper selection of its implementation has a considerable effect on the overall calculation. Figure 3.4, modified from Introduction to Computational Chemistry (15), sketches the SCF algorithm (24). The main difference between

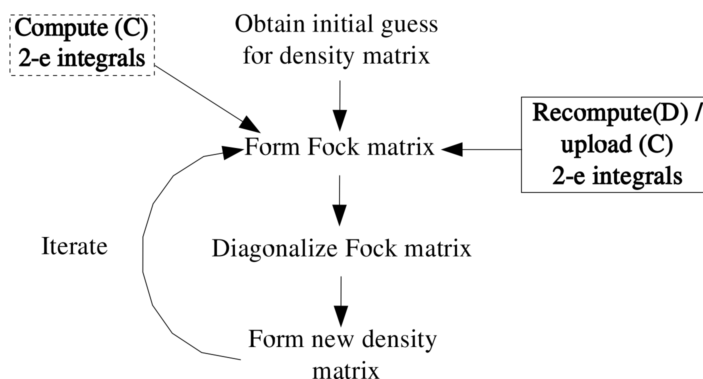

Figure 3.4 Illustration of the SCF algorithm as *conventional* (C) and *direct* (D) implementations

the two implementations is the handling of the two-electron (*2-e*) integrals. In the *direct* SCF method, the *2-e* integrals are recalculated for each iteration. This method has a high demand for CPU time but avoids the possible disk I/O bottleneck. In the *conventional* SCF method, the *2-e* integrals are calculated once at the beginning of the SCF process (box with dashed-line border in Figure 3.4) and stored in a file on disk for subsequent iterations. This implementation performs best on a system where the available physical memory exceeds the file size for *2-e* integrals, so the file is cached in physical memory. Thus, it is faster to re-read buffered *2-e* integrals than to recalculate them as in the *direct* method. Since the SCF algorithm (Figure 3.4) is of an iterative nature, switching between *conventional* and *direct* implementations may be possible in an arbitrary SCF iteration. Although the precise implementation details of the switching are rather complex. The following is a brief outline of the procedure. One may pre-compute the *2-e* integrals before commencing the iteration cycle as required for the *conventional* SCF algorithm. Then at each new iteration, the *2-e* integrals are either fetched from a file or (partly) recomputed for the *conventional* or *direct* implementations to be used, depending on the resource availability. Similarly, any quantum computation algorithm that may be implemented to recompute certain large quantities, such as *2-e* integrals, may be a candidate for the algorithm selection technique presented here.

Following the previous work on improving GAMESS in a dedicated environment, the mid-

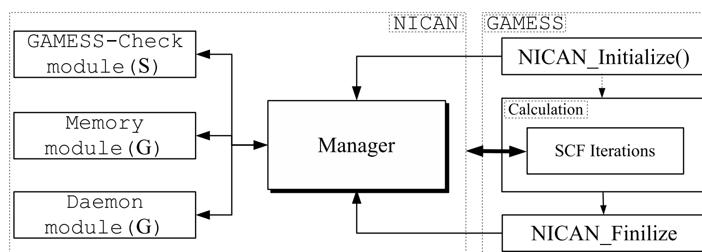

Figure 3.5 GAMESS-NICAN Integration

dleware NICAN (25) was chosen to guide the toggling between the two implementations of the SCF algorithm in GAMESS. Figure 3.5 depicts the GAMESS-NICAN integration for the GAMESS adaptations to be performed at runtime (shown as a bold-faced double arrow). These adaptations are implemented as changes to iteration start parameters and result in switching from one algorithm to the other in between two SCF iterations.

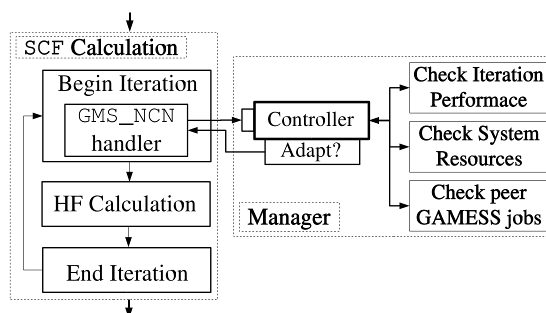

Figure 3.6 GAMESS adaptation scheme

Figure 3.6 details how the interaction between the NICAN Manager and GAMESS SCF calculation is implemented. At the beginning of an SCF iteration, GAMESS calls the adaptation handler *GMS_NCN* which communicates with the NICAN manager and implements adaptation decisions (if any) as returned by the NICAN manager, via the controller port *Adapt?*. The decision is made by Manager's controller, which analyzes the job's past performance, system resource information collected by modules, and the state of peer GAMESS jobs as obtained from NcnD daemons.

CHAPTER 4. Test Results

In this chapter we present test results for each stage of the GAMESS-NICAN integration development as well as performance of GAMESS in hyper-threading (HT) enabled systems.

4.1 Sequential GAMESS

The tests were executed on a 2.8 GHz Intel XEON HT-disabled dual-processor SMP machine. We compared the execution of a pool of concurrent GAMESS jobs with adaptation at preprocessing stage ($GAMESS_{prep}$) and without ($GAMESS_{orig}$). The execution of all the jobs was managed by the PBS scheduler.

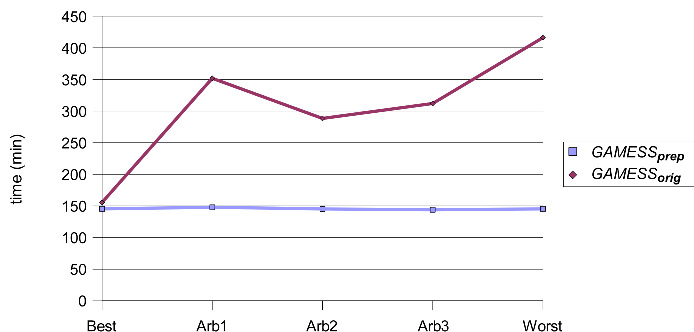

Figure 4.1 Execution time of 12 identical jobs

In the first series of experiments (Figure 4.1), we have queued 12 jobs and recorded their completion time. By choosing 12 jobs, we tried to simulate a rather large pool of similar (identical) jobs. "Identical" means that each job calculates the same analytical gradient for the Luciferin (31) molecule. One half of these jobs was pre-set to use the *direct* (D) SCF method, the other half used the *conventional* (C) method. Concurrent jobs had the best and

the worst performance when they were assigned different and identical patterns of execution, respectively. In the Figure 4.1 we have results for five different orders of the jobs in a queue. $GAMESS_{orig}$ had the best (Figure 4.1, the first point on *y-axis*) performance when the two types of jobs were queued in alternating order (D,C,D,C,...). Conversely, it had the worst (Figure 4.1, the last point) performance when the jobs using the one SCF method were queued following the other method (D,..,D,C,..,C). The other three points in Figure 4.1 correspond to a sample of arbitrary ordered queues. The average time taken to complete these samples is summarized in the first pair of bars in Figure 4.2. This set of experiments shows that the execution of $GAMESS_{prep}$ has a uniform performance, which is no worse than the performance of the best execution of $GAMESS_{orig}$.

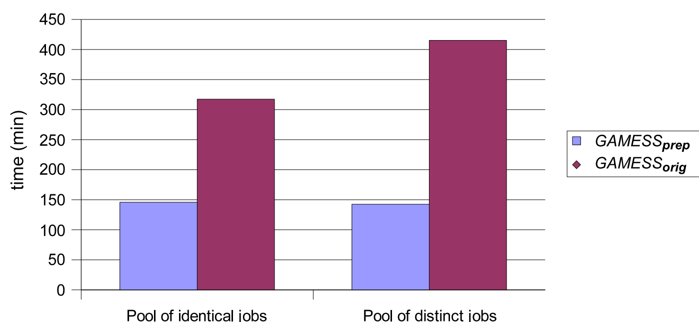

Figure 4.2 Time to complete a pool of 12 jobs

In the second series of experiments (Figure 4.2, the second pair of bars), we simulated a more generic case. A mix of 12 *different* GAMESS jobs, each calculating a structure of a different molecule, was chosen. In contrast with the previous experiment, the jobs in this mix were not pre-set to use a specific SCF method. Instead, to simulate a real-world scenario, GAMESS was started with existing input parameter files. Clearly, both experiments show that concurrent GAMESS jobs, running with $GAMESS_{prep}$, have better utilized the computing resources in an SMP environment, and have resulted in a much better performance as compared with the $GAMESS_{orig}$.

4.2 Parallel GAMESS

The tests were executed on the "4pack" cluster at the Scalable Computing Laboratory (SCL) in Iowa State University (ISU). 4pack is a 32 node Beowulf type cluster consisting of 16 single-processor Macintosh G4 nodes with 512 MB physical memory and 16 dual-processor (Macintosh G4) SMP nodes with 1 GB physical memory (19). The peak processor speed varies from 450 MHz to 999 MHz and the nodes are connected with the Myrinet network. Since our focus is on SMP clusters, we have performed our experiments only on the dual-processor G4 nodes using the PBS scheduler. Each node has its own scratch directory for the temporary storage of files used by parallel applications. This leads to an efficient performance of GAMESS jobs with either of the SCF algorithms. Note that 4pack is configured and routinely used to run GAMESS. In addition to 4pack, we have executed our experiments on the "hpc-class" cluster located also at ISU. The cluster has 44 dual-processor 2.8 GHz Xeon SMP nodes connected with Fast Ethernet. The scratch directory is NFS mounted on a local hard drive of one of the nodes. The latter makes hpc-class not well suitable for the parallel jobs that have a large periodic disk I/O accesses as required by GAMESS with the *conventional* SCF algorithm. Therefore, all GAMESS jobs should use the *direct* SCF algorithm on hpc-class and the test results are not presented here.

First, as in sequential GAMESS, we compare the performance of parallel ($GAMESS_{orig}$) and ($GAMESS_{prep}$) on a pool of identical electronic structure calculations. Particularly, we had submitted 12 identical jobs to calculate the AM1 geometry of antibacterial drug *Penicillin* (Figure 4.3). The molecular structure, as resulted from the calculation, was plotted using the MacMolPlt program (4). One half of these 12 jobs was set to use the *direct* (D) and the other half to the *conventional* (C) SCF algorithm.

Figures 4.4 and 4.5 present results for the five different orderings of the queued GAMESS jobs. $GAMESS_{orig}$ had the best (x-axis label `Best`) performance when the parallel jobs were forced to queue in alternating order (D, C, D, C,...) of the SCF algorithm and had processes scattered across the nodes. Conversely, it had the worst (x-axis label `Worst`) performance when jobs with one SCF method followed the other, for example (D,..., D, C,..., C). Note

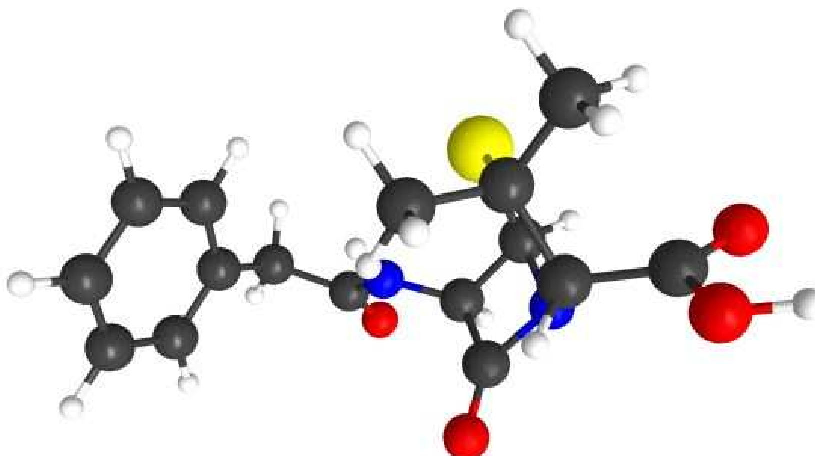

Figure 4.3 AM1 structure of Penicillin molecule obtained by GAMESS RHF calculation

that, at any given time, PBS had a pair of jobs running on the same set of nodes, i.e., parallel GAMESS jobs were equally scattered among assigned nodes. The other three x-axis labels (Figures 4.4 and 4.5) correspond to some arbitrarily ordered queues.

The experiments show that $GAMESS_{prep}$ has a uniformly better performance than the performance of $GAMESS_{orig}$, which is due to a diminished resource contention as a result of GAMESS adaptations.

In Figure 4.6, we a simulation of a more generic case. A mix of eight different GAMESS jobs, each calculating a structure of a different molecule (Absinthin, Adamantane, Cyclic AMP, Darvon, Ergosterol, Luciferin, Penicillin, and Tetrodotoxin), was chosen. The jobs were not pre-defined with a specific SCF method. Instead, to simulate a real-world scenario, GAMESS was started with existing input settings of the SCF method. To see the benefit of adaptive mechanisms, the jobs were chosen such that their execution time is from several minutes to several hours. This set of jobs was executed multiple times, in different orders, on eight processors. The longest time equals to three hours was observed in calculating the Absinthin molecule, while the shortest was one minute for the Adamantane molecule. In Figure 4.6, the legend specifies different orders for executing the test set and the y-axis shows the execution

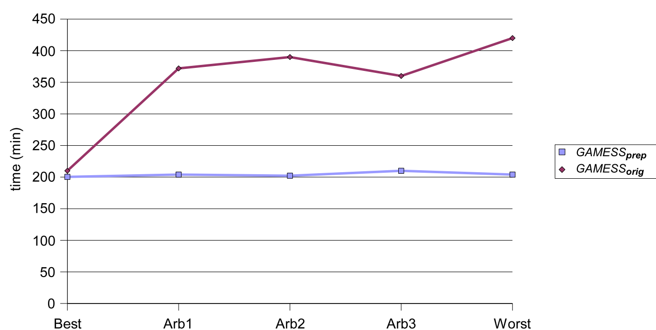

Figure 4.4 Four-processor calculations of twelve identical molecules

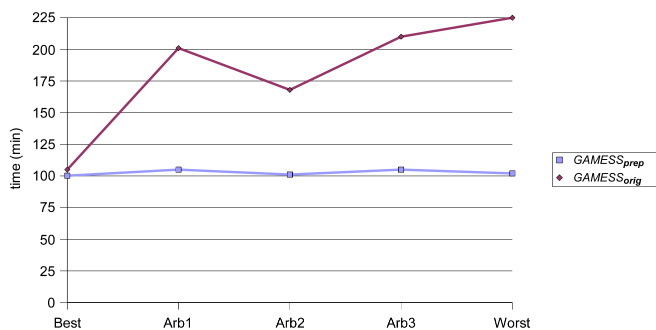

Figure 4.5 Eight-processor calculations of twelve identical molecules

time normalized by the time of $GAMESS_{prep}$ (bar `Best`). The other bars correspond to the execution of $GAMESS_{orig}$ jobs. Regardless of the queue order, $GAMESS_{prep}$ changes the execution pattern to the most efficient one. Note that, with a certain order in a queue, $GAMESS_{orig}$ jobs may perform as well as $GAMESS_{prep}$. However, such a specific order is quite difficult to achieve without application-specific knowledge. The worst performance is observed (bar `Worst`) when all the jobs with one SCF algorithm precede all the jobs with another one and when multiple processes with the *conventional* algorithm run on the same SMP node. This is in agreement with the previous set of experiments. As we see from Figure 4.6, the throughput for $GAMESS_{prep}$ is better than for $GAMESS_{orig}$.

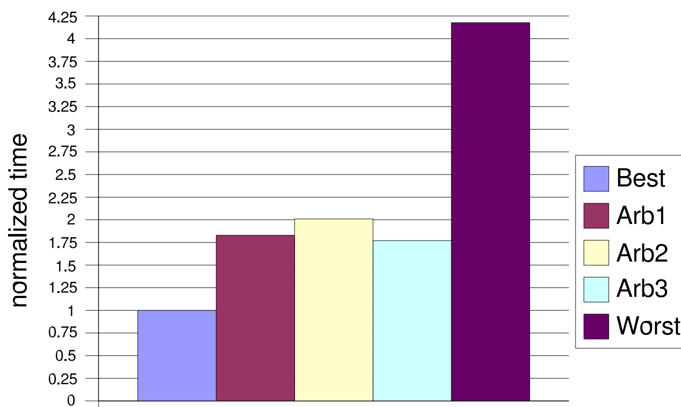

Figure 4.6 Execution of all GAMESS eight-processor jobs in a queue

4.3 Dynamic Algorithm Selection in GAMESS

We consider the behavior of parallel and sequential GAMESS calculations in a dynamically changing computing environment. In particular, the computing platform is shared by the GAMESS calculations integrated with NICAN and other applications that burden the resources critical for GAMESS execution. To study the adaptive features of the SCF algorithm in GAMESS which, in the *conventional* mode, requires much disk I/O, another application competing for this resource has been introduced during the GAMESS runtime. The *iozone* benchmarking tool (21) has been taken as such an application. Although this tool is used mainly for measuring a variety of operations on files, we are interested in its ability to congest the disk I/O channel by writing to disk files of a particular size, and thus occupying a certain percentage of the I/O channel.

The tests were executed on the Tools cluster owned by the SCL group. This cluster has eight SMP nodes, one half of which are 2.2 GHz Intel XEON dual-processor nodes and the other half is 1.7 GHz AMD Athlon MP 2100+ dual-processor nodes. All the nodes have 768 MB of physical memory. Each XEON node has its own scratch directory mounted on a 250 GB 7200 RPM hard drive. On the other hand, the hard drives for the AMD nodes have varying characteristics. To correlate the performance of GAMESS with the I/O bandwidth consumed

by *iozone* and to exclude the effects of processor differences, all the experiments were performed on the XEON nodes only.

As a test problem, we consider the computation of the Luciferin molecular structure using the RHF energy calculation. In Figure 4.7, the molecule structure is plotted using MacMolPlt program (4; 31). For the luciferin energy calculation, GAMESS converges in 15 iterations with the *conventional* SCF algorithm requiring files of at most 3.57 GB to be stored. By default, GAMESS is given the conventional mode on input.

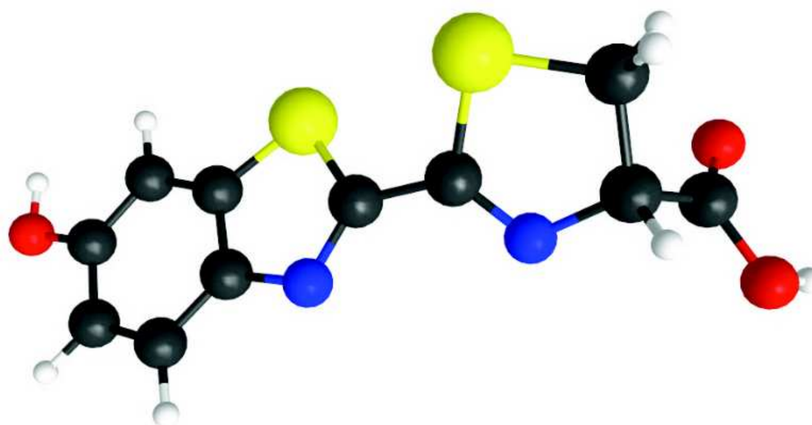

Figure 4.7 Luciferin molecule structure obtained by GAMESS RHF calculation

In the first set of experiments, we compare the performances of the original GAMESS with the *conventional* mode ($GAMESS_{orig}$), i.e., GAMESS without any kind of adaptation, and of GAMESS in which NICAN triggers the dynamic selection of the SCF algorithm. The latter is denoted as $GAMESS_{dyn}$. Each job was executed on two processors, one per SMP node, such that any GAMESS process may not share the (same SMP) I/O channel with another process of the same parallel GAMESS calculation. At the GAMESS runtime, *iozone* starts to introduce a certain amount of I/O Congestion (denoted here as C_{io}) by writing a large file to disk on one of the SMP nodes to which the GAMESS calculation is mapped. Since parallel GAMESS algorithms synchronize at certain times, the C_{io} perceived on an I/O channel may affect the execution of all the GAMESS processes.

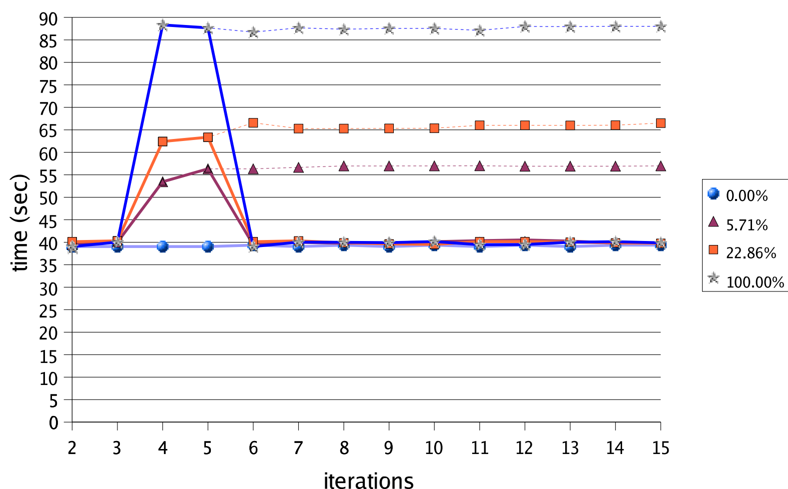

Figure 4.8 Two-processor execution of parallel GAMESS job for different amounts of I/O congestion, C_{io}

Figure 4.8 depicts the time per SCF iteration. Each curve represents the GAMESS performance when different percentages, C_{io}, of I/O bandwidth are consumed by *iozone*, which starts at the point of the fourth GAMESS iteration and continuous to the GAMESS convergence. (Note that iteration four was chosen arbitrarily to start I/O congestion at runtime. If the I/O congestion is present at the very start of a GAMESS calculation then the *preprocessing* adaptation of GAMESS, described in (29) and denoted in this section as $GAMESS_{prep}$, will detect this congestion and adjust the SCF algorithm accordingly.) Until the fourth iteration, $GAMESS_{orig}$ and $GAMESS_{dyn}$ have the same performances. Starting at the fourth iteration, the execution time for the next SCF iterations has increased, depending on C_{io}. In $GAMESS_{dyn}$ (solid lines in Figure 4.8), NICAN monitors the time spent to perform an SCF iteration and compares it with the time required to perform the previous two iterations. If the time has first increased by 10% and then either keeps constant or increases further, the adaptation handler is invoked and the SCF algorithm is toggled from conventional to direct. In other words, this adaptation control strategy is based on the window of three iterations and on the increase of 10%. In the present experiments, these parameters have been dictated mainly by the type of the molecule and type of the GAMESS calculation considered. In addition,

the architecture parameters, such as the I/O bandwidth and the main memory size, may need to be considered. This is left as a future research. The dashed lines in Figure 4.8 show the performance of $GAMESS_{orig}$, while the solid lines reflect the change in the course of GAMESS execution at the sixth iteration when $GAMESS_{dyn}$ is used. For any C_{io}, the performance of $GAMESS_{dyn}$ is almost indistinguishable from the "no-congestion" case (denoted with circles in Figure 4.8) and it may be more than *two times* better than when $GAMESS_{orig}$ is used with an I/O bandwidth fully consumed.

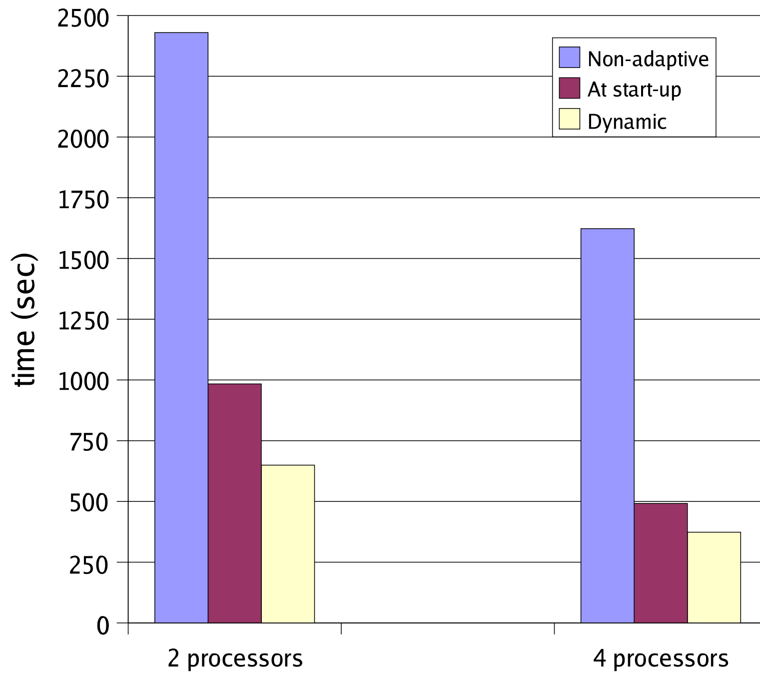

Figure 4.9 Total time to complete two simultaneous parallel GAMESS jobs for different adaptivity settings

The aim of the next set of experiments is to observe the multiple parallel peer GAMESS calculations executing concurrently in the same computing environment that experiences congestion of I/O channels. Two jobs of the same GAMESS calculation using three adaptivity settings — non-adaptive ($GAMESS_{orig}$), at runtime ($GAMESS_{dyn}$), and only in the preprocessing stage ($GAMESS_{prep}$) — are executed on either two or four processors. Figure 4.9 shows the total time taken by two jobs for each adaptivity setting. In each pair, the jobs are

started one after another with minimal delay. $GAMESS_{orig}$ jobs have finished as *conventional* resulting in bad resource utilization caused by mutual competition, as described in (30). With the I/O congestion, the performance of multiple $GAMESS_{orig}$ jobs deteriorates further (gray bars in Figure 4.9). Since GAMESS selects by default the *conventional* mode, this scenario is rather likely to happen in a multiuser cluster environment. When preprocessing is used, in $GAMESS_{prep}$ and $GAMESS_{dyn}$, once the presence of the peer GAMESS *conventional* calculation is detected for the job starting second, its SCF algorithm is switched to the *direct* mode. Thus, the $GAMESS_{prep}$ pair finishes with one *conventional* and one direct SCF algorithm (dark bars in Figure 4.9). The $GAMESS_{dyn}$ pair, however, is adapted further to the I/O channel congestion $C_{io} = 46\%$ by changing the algorithm of the job started first to the the *direct* mode (light-colored bars in Figure 4.9). Similar to the previous set of experiments, this change happens at the sixth iteration since the I/O congestion procedure and the control strategy are taken as in the first set of the experiments in Section 4.3. We observe that dynamic adaptations of GAMESS calculations bring 10% to 15% improvement in the cumulative execution time of two jobs on four and two processors, respectively. We expect this improvement to hold for massively parallel large GAMESS calculations performed on computing platforms with few I/O channels, which are requested simultaneously by many GAMESS processes.

4.4 Hyper-Threading

This subsection notes on the execution of concurrent GAMESS jobs on Intel processor platforms equipped with Hyper-Threading (HT) technology. HT makes a physical processor appear to the operating system (OS) as a dual logical processor. The purpose of HT is to enable an efficient resource utilization by scheduling programs (or their parts) concurrently, such that the idle time for the computational resources involved is minimized. In HT, there is a copy of the architecture state for each logical processor, and the logical processors share a single set of physical execution resources. In fact, OS recognizes the existence of multiple logical processors as an SMP node. Thus, OS and the user programs could schedule processes (threads) on logical processors as they would on conventional physical processors in a multi-

processor system (20). The experiments in this section were tested on a single 2.4 GHz Intel XEON processor machine.

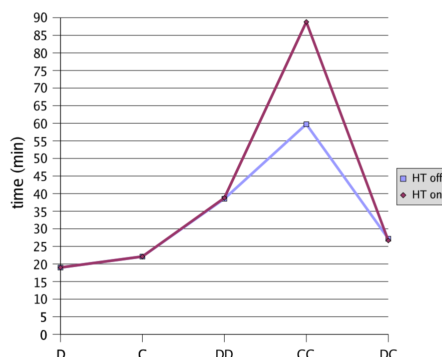

Figure 4.10 Concurrent jobs: single CPU

Initially, results for the separate runs of the *direct* and *conventional* methods were obtained. The execution times appeared rather close (Figure 4.10, the first two points). In Figure 4.10, D and C stand for *direct* and *conventional*, respectively. To observe the effect of the HT technology two concurrent sequential jobs were tested while recording the mean completion time for these two jobs. Three different combinations (DD, CC, and DC in Figure 4.10) have resulted in different execution times. As predicted, two jobs with the *direct* method (DD) took almost twice as much time as a single *direct* execution. Since the *direct* method has high CPU utilization and both jobs are performing the same task, there is a competition for the same resource (28). Two jobs with *conventional* method (CC) resulted in a much worse performance, which is not a surprise either. Due to the slower response time of the I/O channel, both processes had to spend more time idling while competing and waiting for the I/O resources. A combination of two different methods (DC) has resulted in the best performance out of the three latter experiments, requiring only about 35% more time than a single D job. This is due to the fact that D and C burden different functional units of the computer and are scheduled for efficient execution by OS.

Another set of experiments was conducted on a two-processor SMP machine. As mentioned earlier, HT makes a single physical processor appear as two logical processors. In this case,

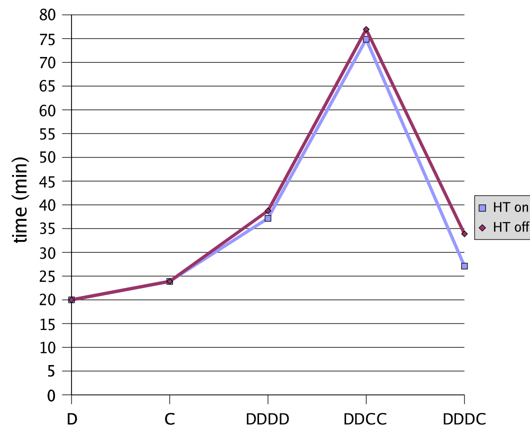

Figure 4.11 Concurrent jobs: SMP node with dual CPU

the system became a four-processor machine when HT-enabled and two-processor SMP node when HT-disabled. In this experiment, HT was analyzed by comparing the performance of four concurrent tasks (DDDD, DDCC, and DDDC in Figure 4.11) and the combination DDDC has the best performance. DCCC and CCCC tests were not included in Figure 4.11 because of their poor results effecting the plot scaling.

The presence of HT (Figures 4.10 and 4.11) did not change the pattern of the graphs. But there is a significant difference in the execution of concurrent *conventional* jobs. Poor performance under HT might be due to a lack of HT-awareness in the Linux kernel scheduler used. In multi-processor systems, the process scheduler could even execute concurrent threads on two logical processors corresponding to the same physical processor, while the other physical processor is vacant (8). New releases of the kernel have patches that make the kernel HT-aware(3). But even in such a case, application-specific execution patterns may remain unknown to the kernel and to the operating system.

CHAPTER 5. Conclusion and Future work

In this thesis, we have shown an improvement of electronic structure calculations executed on SMP clusters. The advantage is observed when a GAMESS calculation adapts its execution pattern by choosing either an in-core or out-of-core implementation. We have enabled the adaptation process by integrating GAMESS with the middleware NICAN. NICAN general- and specific-use modules discover system parameters, analyze peer GAMESS jobs, and choose a particular execution pattern for GAMESS. An attractive feature of the proposed integration is that it requires only a few lines of trivial modifications to the original GAMESS code and provides an example of how to utilize NICAN general-use modules for many other high-performance applications. The throughput for the adaptive GAMESS calculations was observed to be *several times* higher than for non-adaptive ones. The results were obtained for the identical as well as different molecules.

Initially, we have considered specific case of dedicated SMP clusters, where an adaptation at preprocessing stage is sufficient to gain throughput improvement. The adaptation is applied to incoming jobs in order to start execution efficiently based on the current state of the computing environment. This choice depends on system characteristics and on the execution patterns of the peer GAMESS calculations. Later, we have presented a more sophisticated adaptation mechanism that works in realistic environments. The GAMESS-NICAN integration is able to switch dynamically between algorithms to adapt to the changing environment. While a precise calibration of the decision making is of paramount importance, in this research, we concentrated mostly on the toggling abilities of the SCF calculations and their implementation using the middleware. To demonstrate the capabilities of GAMESS to adapt to dynamic changes in vital system resources, we considered the effect of the disk I/O channel congestion.

As the result of adaptations, the SCF algorithm mode may be switched dynamically based on the the current state of the multiuser computing environment. This switching is performed by the NICAN middleware that is tightly integrated with GAMESS and controls the adaptation process.

Although our tests do not cover all the possible GAMESS calculations which use the SCF algorithm, the obtained results already indicate that GAMESS benefits considerably from an application-level middleware that facilitates its execution. The GAMESS-NICAN integration model for dynamic adaptations may be used for any application to achieve similar purpose of improving the resource utilization and application performance. An essential condition for such an integration is that an application has dynamically interchangeable execution modes or algorithms while system monitoring and job performance analysis are delegated to NICAN modules.

5.1 Future Work

In the future, we envision to apply similar technique for post Hartree-Fock and non-SCF calculations as well as to develop multiple adaptation control strategies suitable for many GAMESS calculations. In particular, we plan to consider the effect of the available main memory on dynamic adaptations. Our observations have shown that with the sufficient main memory, the data in the out-of-core implementation could be cached in the main memory making it accessible faster than recomputing the data in the in-core implementation. The presented results may also be extended to various types of heterogeneity in computing platforms and resources.

BIBLIOGRAPHY

[1] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of 4th Symposium on Operating Systems Design and Implementation*, pages 213–226, 2000.

[2] Bayucan, et al. Portable batch system: administrator guide. *OpenPBS* 2.3. August 2000.

[3] P. Bemowski. Hyper-threading linux. *Linuxworld Magazine*, 1.1. August 11th, 2003. http://linux.sys-con.com/read/33885.htm Last access April 18, 2006.

[4] B.M. Bode and M.S. Gordon. Macmolplt: a graphical user interface for GAMESS. *Journal of Molecular Graphics and Modeling*, 16, 133-138 (1998).

[5] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing* 4(1): 49-62 (2001).

[6] G. Chen. *Providing dynamic network information to distributed applications.* Master's Thesis, University of Minnesota Duluth, May 2001.

[7] X. Chen, D. Turner. Efficient message-passing within SMP systems. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 10th European PVM/MPI conference, Venice, Italy, pg 286-293 (October 2003).

[8] J. Corbet. Kernel development: The scheduler and hyperthreading. *LWN.net*, http://lwn.net/Articles/8287/ Last access April 18, 2006.

[9] D. Feller. The EMSL ab initio methods benchmark report: a measure of hardware and software performance in the area of electronic structure methods. *Pacific Northwest National Laboratory Technical Report PNNL-10481 (Version 3.0)*, June 1997. http://www.emsl.pnl.gov/docs/tms/abinitio/cover.html Last access April 18, 2006.

[10] A. Freedman, C. Rabinowitz and P. Felperin. *Computer Desktop Encyclopedia.* McGraw-Hill, 2001.

[11] C. Hamacher, Z. Vranesic and S. Zaky. *Computer Organization.* 5th Edition, McGraw-Hill, New York, USA, 2002.

[12] High Performance Computational Chemistry Group. NWChem, a computational chemistry package for parallel computers. *Pacific Northwest National Laboratory*, Richland, WA 99352. http://www.emsl.pnl.gov/docs/nwchem/ Last access April 18, 2006.

[13] J. Hollingsworth and P. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2(3):195-205, 1999.

[14] C. L. Janssen, I. B. Nielsen, M. L. Leininger, E. F. Valeev, E. T. Seidl. The massively parallel quantum chemistry program (MPQC), 2.2.2, *Sandia National Laboratories*, Livermore, CA, USA, 2004.

[15] F. Jensen. *Introduction to Computational Chemistry*. Wiley, Chester UK, 1999.

[16] D. Kulkarni and M. Sosonkina. A framework for integrating network information into distributed iterative solution of sparse linear systems. In José M. L. M. Palma, Jack Dongarra, Vicente Hernández, and A. Augusto de Sousa, editors, *High Performance Computing for Computational Science - VECPAR* 2002, 5th International Conference, Porto, Portugal, June 26-28, 2002, Selected Papers and Invited Talks, volume 2565 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2003.

[17] sched_setaffinity. *Linux Programmer's Manual*, 11, 2002.

[18] LoadLeveler for AIX 5L and linux V3.2: using and administering. *IBM LoadLeveler Publications*, 3rd Edition, May 2004.

[19] Macintosh G4 cluster. http://www.scl.ameslab.gov/Projects/PPC_cluster/ Last access April 18, 2006.

[20] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyperthreading technology architecture and microarchitecture. *Intel Technical Journal*, 1(1), Feb. 2002.

[21] W. D. Norcott and D. Capps. *Iozone Filesystem Benchmark*. http://www.iozone.org Last access April 18, 2006.

[22] R. M. Olson, M. W. Schmidt, M. S. Gordon, and A. P. Rendell. Enabling the efficient use of SMP clusters: The GAMESS/DDI model. *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, p.41, November 15-21, 2003.

[23] *Load Sharing Facility*. http://www.platform.com Last access April 18, 2006.

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

[25] M. Sosonkina. Adapting distributed scientific applications to run-time network conditions. In *PARA'04 Workshop on state-of-the-art in scientific computing*, Denmark, June 20 – 23, 2004.

[26] M. Sosonkina, S. Storie. Parallel performance of an iterative method in cluster environments: an experimental study. In *Proceedings PMAA 2004*, Marseille, October 2004.

[27] S. Storie. *Aspects of Communication Subsystem Analysis for Distributed Scientific Applications.* Masters Thesis, University of Minnesota Duluth, May 2004.

[28] N. Tuck, D. D. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[29] N. Ustemirov, M. Sosonkina. Efficient execution of parallel electronic structure calculations on SMP clusters. *Minnesota Supercomputing Institute Technical Report* umsi-2005-227, University of Minnesota, 2005.

[30] N. Ustemirov, M. Sosonkina, M.S. Gordon, M.W. Schmidt. Concurrent execution of electronic structure calculations in SMP environments. In *Proceedings HPC 2005*, April 2005.

[31] E.H. White, F. Capra, W.D. McElroy. The structure and synthesis of firefly luciferin. *J. Am. Chem. Soc.*, 83(10), 2402-2403(1961).

[32] H.-J. Werner, P. J. Knowles, R. Lindh, M. Schütz and others. MOLPRO a package of ab initio programs. http://www.molpro.net Last access April 18, 2006.

APPENDIX. GAMESS Capabilities

A wide range of quantum chemical computations are possible using GAMESS, which

1. Calculates RHF, UHF, ROHF, GVB, MCSCF or density functional theory (DFT) self-consistent field molecular wavefunctions.

2. Calculates the electron correlation energy correction for these SCF wavefunctions Configuration Interaction (CI), Many Body Perturbation Theory (MP2), coupled-cluster (CC) or equation of Motion CC (EOM-CC) methodologies. (see summary table below).

3. Calculates semi-empirical MNDO, AM1, or PM3 models using RHF, UHF, ROHF, OR GVB wavefunctions.

4. Calculates analytic energy gradients for any of the SCF wavefunctions, closed or open shell MP2, or closed shell reference-based CI.

5. Optimizes molecular geometries using the energy gradient, using internal or Cartesian coordinates.

6. Searches for saddle points (transition states) on the potential energy surface.

7. Computes the energy Hessian, and thus normal modes, vibrational frequencies, and IR intensities. Raman intensities are a follow-up option.

8. Obtains anharmonic vibrational frequencies and intensities (fundamentals or overtones).

9. Traces the intrinsic reaction path from the saddle point towards products, or back to reactants.

10. Traces gradient extremal curves, which may lead from one stationary point such as a minimum to another, which might be a saddle point.

11. Follows the dynamic reaction coordinate, a classical mechanics trajectory on the potential energy surface.

12. Computes excited state energies, wavefunctions, and transition dipole moments at various levels

- SCF (e.g. ROHF or MCSCF) Introduction 1-4
- CIS (RHF plus single excitations)
- more general CI functions
- Equation of Motion Coupled Cluster.

13. Evaluates relativistic effects, including

    - scalar corrections, via 3rd order Douglas-Kroll transformations. Gradients are available.
    - spin-orbit coupling matrix elements and the resulting spin-mixed wavefunctions.

14. Evaluates the molecular linear polarizability and the first and second order hyperpolarizabilities for all wavefunctions, by applying finite electric fields.

15. Evaluates both the static and frequency dependent polarizabilities for various non-linear optical processes, by analytic means, for RHF wavefunctions. Nuclear derivatives of the polarizabilities lead to analytic Raman spectra, also for RHF wavefunctions. The imaginary frequency dependent polarizabilities can also be obtained.

16. Obtains localized orbitals by the Foster-Boys, Edmiston-Ruedenberg, or Pipek-Mezey methods, with optional SCF or MP2 energy analysis of the LMOs.

17. Calculates the following molecular properties:

    - dipole, quadrupole, and octopole moments
    - electrostatic potential
    - electric field and electric field gradients
    - electron density and spin density
    - Mulliken and Lowdin population analysis
    - virial theorem and energy components
    - Stone's distributed multipole analysis

18. Models solvent effects by

    - effective fragment potentials (EFP)
    - polarizable continuum model (PCM
    - surface and simulation of volume polarization for electrostatics (SS(V)PE)
    - conductor-like screening model (COSMO)
    - self-consistent reaction field (SCRF)

19. Performs all-electron calculations based on the Fragment Molecular Orbital (FMO) method.

20. When combined with the add-on TINKER molecular mechanics program, performs Surface IMOMM or IMOMM QM/MM type simulations. Download from http://www.msg.ameslab.gov/GAMESS/GAMESS.html

21. When combined with the add-on VB2000 program, performs valence bond calculations. See http://www.scinetec.com/ vb for more information.

22. When combined with the add-on NBO program, performs Natural Bond Order analyses. This program is available at http://www.chem.wisc.edu/ nbo5, for a modest license fee.