Adaptations in Electronic Structure Calculations in Heterogeneous Environments

by

Sai Kiran Talamudupula

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

Lu Ruan, Co-major Professor

Mark Gordon

Iowa State University

Ames, Iowa

2011

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my committee, first Dr. M. Soosnkina for her valuable guidance throught this research. I would also like to thank other committee members, Dr. L. Ruan, and Dr. M. Gordon for their support and contributions to this work. I would additionally like to thank Dr. M. W. Schmidt for his inputs on the experimental setup, Dr. A. Gaenko and Spencer for their help in the initial stages of this work.

This work was performed at the Ames Laboratory under contract number DE-AC02-07CH11358 with the U.S. Department of Energy. The document number assigned to this thesis/dissertation is IS-T 3034. This work was supported in part by Iowa State University under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and by the National Science Foundation grants NSF/OCI – 0749156, 0941434, 001047772.

ABSTRACT

Modern quantum chemistry deals with electronic structure calculations of unprecedented complexity and accuracy. They demand full power of high-performance computing and must be in tune with the given architecture for superior efficiency. To make such applications resource-aware, it is desirable to enable their static and dynamic adaptations using some external software (middleware), which may monitor both system availability and application needs, rather than mix science with system-related calls inside the application.

The present work investigates scientific application interlinking with middleware based on the example of the computational chemistry package GAMESS and middleware NICAN. The existing synchronous model is limited by the possible delays due to the middleware processing time under the sustainable runtime system conditions. Proposed asynchronous and hybrid models aim at overcoming this limitation. When linked with NICAN, the fragment molecular orbital (FMO) method is capable of adapting statically and dynamically its fragment scheduling policy based on the computing platform conditions. Significant execution time and throughput gains have been obtained due to such static adaptations when the compute nodes have very different core counts. Dynamic adaptations are based on the main memory availability at run time. NICAN prompts FMO to postpone scheduling certain fragments, if there is not enough memory for their immediate execution. Hence, FMO may be able to complete the calculations whereas without such adaptations it aborts.

CHAPTER 1. INTRODUCTION

Electronic structure calculations provide a representative example of High-Performance Computing (HPC) applications. They require a large amount of disk space—typically, several gigabytes—to store the integrals and a large I/O bandwidth to use them during the iterative solution process. The need for the CPU and memory resources scales up with the number of atoms as a power of 4–7 and 2–4, respectively, depending on the type of calculation executed. Thus, pooling all the memory and disk resources available in distributed environments emerges as the solution to such an explosion in computational needs and, at the same time, challenges to obtain an optimal parallel performance. The latter is nearly infeasible without an application awareness of parallel architecture, that is, without being tuned to the architecture at hand.

Dynamic adaptations respond in a timely manner to run-time changes in the environments, as described, e.g., in (1). To implement such adaptations, an application may be supplied with sophisticated system monitoring strategies and/or with the analysis of its own (historical) performance. The most non-intrusive and portable way to accomplish this is to empower an application with a middleware that will provide a liaison between the application and system by monitoring the system resources dynamically, making adaptation decisions based on the application performance and the work environment, and invoking application adaptations if needed. An integration of HPC applications with middleware tools may empower the applications with such advanced features as parallel and distributed programming models, system resource management, remote application monitoring, and user-interactive material. Middlewares are more effective if they provide these attractive features dynamically.

The NICAN middleware, proposed in (22), has been extensively used as adaptive mechanism invocation tool in quantum chemistry applications, such as GAMESS (for more details, see (7)). The interaction between NICAN and GAMESS has not been analyzed yet, however.

Modeling GAMESS execution with and without NICAN middleware prompted to consider an asynchronous mode of their interaction that appears beneficial for lightly oscillating system conditions when only few, if any, application adaptations are required. The standard optimizing technique of load balancing along with its variations for different scenarios in heterogeneous environments and their effects on the execution time are also studied in the present work. The adaptations occurring dynamically with the help of the middleware can be thought of as the pivot points for these variations in the standard scheme.

CHAPTER 2. MIDDLEWARE AND APPLICATION USED

As seen in INTRODUCTION, the current work is based on the study of proposed techniques on a test application GAMESS and the middleware used to get different types of adaptations in this application is NICAN. This chapter introduces the application and the middleware.

2.1 GAMESS

GAMESS is a computational chemistry application that performs ab-initio molecular quantum chemistry calculations. It includes a wide variety of Hartree-Fock (HF) wave functions such as Restricted Hartree-Fock (RHF), Unrestricted Hartree-Fock (UHF), Multiconfigurational SCF wavefunction (MCSCF) and Generalized valence bond wavefunction (GVB). Also, electron correlation energy correction methods such as Density Functional Theory (DFT), Configuration Interaction (CI), Coupled Cluster (CC) and Many Body Perturbation Theory (MP2) for the above mentioned SCF calculations are provided. More details on the capabilities of GAMESS can be found out at the (11). GAMESS implementation is mainly divided into two parts: 1) Legacy code, written in F77 that performs actual computation and 2) DDI, written in C to take the advantage of shared memory in SMP nodes. The RHF SCF calculations and the Fragment Molecular Orbital (FMO) method that are a part of the present implementation of GAMESS are the center of discussion in further chapters. More on these implementations and their relevance to this work is provided in the respective chapters.

2.2 NICAN

Network Information Conveyer and Application Notification (NICAN) is a middleware tool that provides the application with the various facilities to adapt itself according to the changing

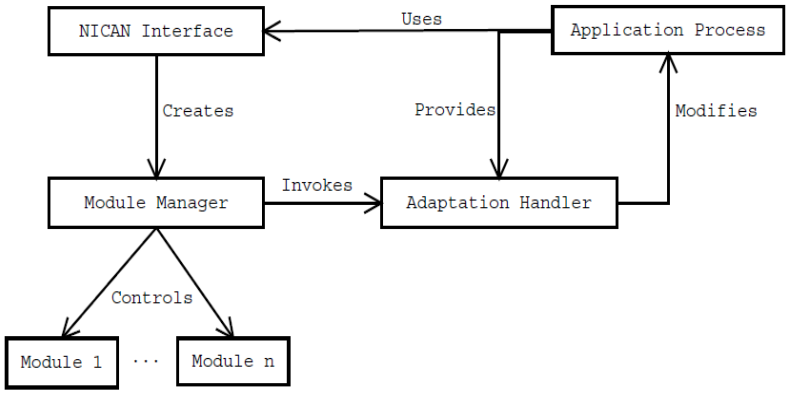

Figure 2.1 NICAN Layout

underlying system conditions. The main idea is to decouple the application from this dynamic decision making overhead. NICAN is started in a separate thread, Manager which inturn loads and starts different modules used for different purposes and Manager is responsible for invoking the adaptations. Thus, there is one manager per job. NICAN is very versatile due to the dynamically loadable modules and provides wide variety of interactions with any application. Each of these "general purpose" modules is designed to perform a specific function such as capturing CPU load, disk I/O, network latency and so on. A layout of NICAN is shown in Figure 2.1

There is only a little overhead of calling NICAN in a separate thread and one function call from application to what is called NICAN daemon. NcnD, the NICAN daemon is started on each node and is used for communication between the NICAN manager and the compute processes. Each process sends it ID and a job description to daemon which delivers it to the manager. Job descriptions might change from application to application. For example, chapter

asyn-adapt and fmo have differnet job descriptions sent to NICAN from application.

CHAPTER 3. LITERATURE REVIEW

For the computational chemistry, adaptive software architecture is an emerging area of research. Existing architectures vary from compiler and model-based engineering approaches to adaptive algorithms. In (2), a component-based approach is described. It interfaces many independently developed numerical adaptation algorithms and implementations. A recent development has been the inclusion of support for CQoS, computational quality of service, which has a measurement, analysis, and control infrastructure for dynamic domain-specific decision making. (3) is an infrastructure in which the application acts as a client when sending information to the server, which, in turn, makes application tuning and adaptation decisions based on this information from the client. (20) focuses on describing the design and early functinality of Active Harmony resource management system. Work migration at the level of procedures, processes and lightweight threads is described as the early online reconfiguration that is a prime feature of Active Harmony. The parameters that represent various kinds of adaptations also prove to be a prominent factor in achieving the most required type of adaptations. (19) presents a parameter prioritizing tool that focuses on parameters that are critical in the performance of an application. The middleware system dQUOB (4) provides continuous evaluation of queries over time sequenced data. The queries are inserted into the data streams at run time and managed remotely during the execution. The goal of IANOS (5) is to provide a general middleware infrastructure that allows optimal positioning and scheduling of HPC Grid applications. (13) is another middleware that provides dynamic adaptations to applications. It provides service and semantics necessary for the follow-me type design of integrating middleware with pervasive applications. The memory requirements are known in advance for many applications. But (14) focuses on work environments changing dynamically and applications whose memory requirements cannot be predicted. For this, knowledge about available memory of a cluster or

multinode environment at run time is absolutely necessary. Simplification of the estimation of the available memory is achieved by a good instrumentation of the kernel. (14) also provides a memory-based load balancing scheme. However, the load balancing scheme discussed in the current work is more specific to iterative jobs such as FMO and is naive in its approach considering more practical scenarios occurring in FMO runs. (21) describes a strategy to decouple the application from the overhead of inserting calls pertaining to decision making. The proposed framework includes a tunability interface component and a virtually executable environment that emulates the execution of application under heterogeneous system conditions.

In (6), it was proposed how to integrate the middleware NICAN with GAMESS for acquiring the dynamic adaptations in the SCF algorithm. The integration of the middleware NICAN with GAMESS in this work closely resembles Nurzhan's model in (6). In (15), MFDn, an application that performs ab-initio nuclear physics calculations was integrated with NICAN to get the dynamic adaptations. The adaptations were based on changing the number of threads that perform Lanczos diagonalization procedure. In (1), a detailed study of exploring tuning strategies for the adaptations for SCF algorithm with NICAN is presented.

CHAPTER 4. ASYNCHRONOUS ADAPTATION INVOCATION

4.1 Introduction

GAMESS is a computational chemistry application which is widely used to perform *ab-initio* molecular quantum chemistry calculations. A wide range of quantum chemistry computations are possible using GAMESS such as calculating Restricted Hartree-Fock (RHF) Self-Consistent Field (SCF) molecular wavefunctions. Using the SCF method, GAMESS iteratively approximates solution to the Schrödinger equation that describes the basic structure of atoms and molecules. SCF is one of the many computationally intensive parts of GAMESS and has two different implementations (i.e., execution modes), *direct* and *conventional*, which differ in the way two-electron (2-e) integrals are computed. In the direct mode, 2-e integrals are recomputed "on-the-fly" for each iteration consuming mainly physical memory and CPU resources. In the conventional, these integrals are calculated before the first SCF iteration and stored in a file on disk. The subsequent iterations fetch the 2-e integrals from disk, consuming mainly I/O bandwidth. So, there are instances when the conventional SCF implementation proves to be better compared to the direct one and vice-verse. It is feasible to alternate between two available implementations at run-time. Therefore, by integrating the SCF process with a middleware that interacts with the system, dynamic adaptations of SCF may be enabled. There are other (higher level of theory) calculations available in GAMESS, such as the second order Möller-Plesset correction (MP2), that are used to get electron correlations and involve post SCF calculations to improve the accuracy of solution. For more details on GAMESS see (7). The work corresponding to this chapter is published as (25).

4.2 Integration with middleware

In (8), (9), (6) an integration of GAMESS with NICAN has been discussed for the purpose of GAMESS performance tuning and prediction. One way to use NICAN with GAMESS is to enable GAMESS adaptations in the computationally intensive SCF method. Specifically, NICAN prompts GAMESS to switch from one SCF implementation to another at the iteration following the detection of the decrease in the GAMESS performance as measured by its execution time. An increase in the iteration time typically signifies a change in system conditions. The NICAN architecture, containing a manager dedicated to the application along with the various types of modules to collect the system information and to encapsulate adaptation control, makes possible this switch. The timely nature of adaptations is assured by GAMESS waiting after each iteration for the decision from NICAN. Figure 4.1 depicts this GAMESS-NICAN integration featuring a separate `GMS_NCN` module that implements the adaptation control mechanism and the GAMESS-NICAN synchronization (depicted as lines with double arrows) to communicate the adaptation decision from NICAN to GAMESS. In general, NICAN consists of dynamically loadable modules making it versatile and providing a wide variety of interactions with system or application (24), (23).

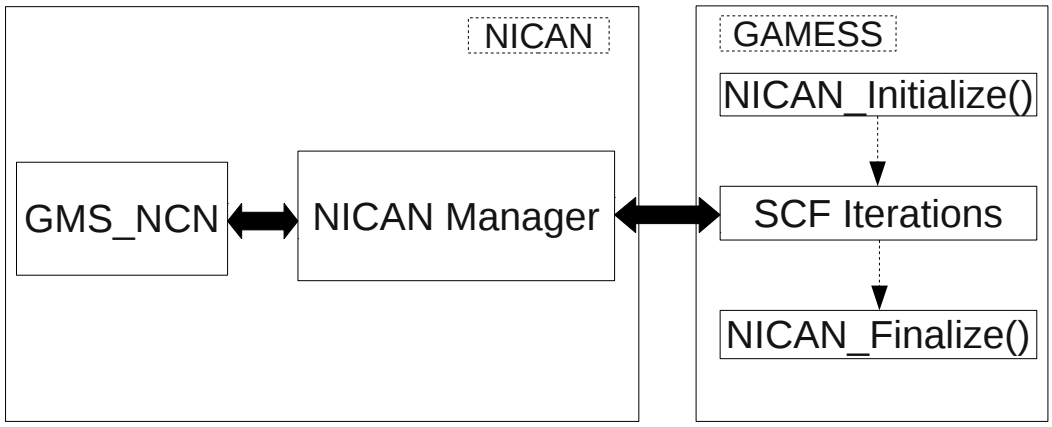

Figure 4.1 GAMESS-NICAN integration in the synchronous model.

4.3 Asynchronous Adaptations

Timely adaptations are an important advantage of the GAMESS-NICAN integration with synchronization, but they come at a price: When no or few adaptations are required the GAMESS-NICAN execution time may suffer. Since a certain system condition typically persists in supercomputers executing HPC applications, it is worth considering an *asynchronous* integration model. This model differs from its synchronous counterpart in that an application proceeds to the next iteration without waiting for the decision from NICAN and adapts, if necessary, later. Certainly, delaying adaptations may be detrimental to the overall performance. Using GAMESS as an example, this section investigates when the delays may be tolerated.

Assume that the execution time of a single iteration, in either direct or conventional mode, does not deviate much from an average iteration time for that mode in a certain system state. Consider two distinct system states, Congested C and Free F, and two SCF execution modes conventional c and direct d. Then, denote a single iteration time as t^S_m, where $m \in \{d, c\}$ and $S \in \{C, F\}$. The execution times T^S and $\overline{T}^S$ for the total N iterations when integrated with NICAN synchronously and asynchronously, respectively, may be expressed as

$$T^C = \sum_{j=1}^{K+1} \delta_j t^C_{m_j} + N\tau \,, \tag{4.1}$$

$$T^F = N t^F_m + N\tau \,, \tag{4.2}$$

$$\overline{T}^C = \sum_{j=1}^{\overline{K}+1} \overline{\delta}_j t^C_{m_j} \,, \tag{4.3}$$

$$\overline{T}^F = N t^F_m \,, \tag{4.4}$$

where τ is the time required to communicate with NICAN including the time to decide on an adaptation. K and $\overline{K}$ are the number of adaptations for synchronous and asynchronous models, respectively, such that $K \geq \overline{K}$ and $j = 0$ at the first SCF iteration while $j = K + 1$ happens at the last (Nth) iteration. Then, δ_j and $\overline{\delta}_j$ are the numbers of iterations between adaptation j and $(j - 1)$ for the synchronous and asynchronous models, respectively.

The expression (4.4) of $\overline{T}^F$ contains no term related to NICAN. Thus, this total time is the same as when GAMESS is executed without NICAN in the non-congested environment, and the integration with the NICAN tool becomes less intrusive.

To analyze the effect of delayed adaptations, consider that δ_1 is always less than $\overline{\delta}_1$. However, for HPC applications, such as GAMESS, it may be assumed that the adaptation decision arrives when only one more GAMESS iteration elapses, i.e., an adaptation is delayed by only one iteration. Hence, $\overline{\delta}_1 - \delta_1 = 1$. An assumption of *persistent congestion* also follows from the fact that an HPC application may heavily consume supercomputer resources, so they are not acquired or released momentarily. Here, persistent congestion is defined as a system resource congestion the detection of which causes the given application to adapt *only once* since the congestion continues to manifest itself until the application termination. This assumption may be reasonable in the adaptive application parts that are short relative to the entire application execution time. For example, SCF iterations are typically much faster than the higher-level calculations, such as MP2. Under these two additional assumptions, the equations (4.1) and (4.3) may be rewritten as

$$T^C = \delta_1 t^C_{m_1} + (N - \delta_1) t^C_{m_2} + N\tau \,, \tag{4.5}$$

$$\overline{T}^C = (\delta_1 + 1) t^C_{m_1} + (N - \delta_1 - 1) t^C_{m_2} \,. \tag{4.6}$$

The asynchronous model becomes beneficial when the total time $\overline{T}^C$ is smaller than T^C, i.e., when

$$t^C_{m_1} - t^C_{m_2} < N\tau \,. \tag{4.7}$$

In other words, the time difference between iterations in different SCF execution modes under the persistent congestion should be smaller than the NICAN execution time for the total N iterations of SCF. This may happen when either N or τ are very large since there is not much sense to adapt if the difference $t^C_{m_1} - t^C_{m_2}$ is close to zero.

To increase the number of cases when the asynchronous integration may be applied, a hybrid model is developed under the same assumptions. It behaves as the synchronous one until the adaptation is invoked and as asynchronous thereafter. For the hybrid model, the total SCF iteration time $\widetilde{T}^S$ is given as

$$\widetilde{T}^C = \delta_1 t^C_{m_1} + (N - \delta_1) t^C_{m_2} + \delta_1 \tau \,, \tag{4.8}$$

$$\widetilde{T}^F = N t^F_m + N\tau \,. \tag{4.9}$$

Note that the first two terms of equation (4.8) come from (4.5) for the synchronous model while the third term in (4.8) is that of (4.5) being reduced by $(N - \delta_1)\tau$. Thus, the difference between $\widetilde{T}^C$ and T^C is always negative, which implies that the hybrid integration model performs better than the synchronous. On the other hand, in the no congestion system state F, both models incur the same overhead.

4.3.1 Implementation

The data regarding the iteration time is collected and used by NICAN to make adaptation decisions. NICAN checks if the iteration time is above a certain limit and decides on the execution mode switch. Following the presentation in (9), the pseudo-code for the jth adaptation decision may be written as:

t_i = Actual time taken for iteration i
t^u = Upper bound for the time per iteration
m = Average iteration time between two adaptations
t^e = Estimated run-time for a single iteration (obtained
by NICAN after running a GAMESS "check" run first.)
$\Delta_i = |t^e - t_i|$
if $t_i > t^u$ OR $t_i > m + \Delta_i$ then
  if (SCF is *conventional*) then
    switch to *direct*
  else if ((no peer *conventional* jobs) then
      switch to *conventional*

The asynchronous model requires an instant return of control to GAMESS after communicating with the middleware. The synchronous model is implemented via Sockets API for the underlying (blocking) communication and can be modified into the asynchronous one by changing the communication to non-blocking. The NICAN Manager-Module interaction is the base for these changes and uses multi-threading. The Manager invokes the module thread and

passes to it the requests from GAMESS. Due to the asynchronous behavior of thread scheduling in NICAN and of integration with GAMESS, race conditions are possible and NICAN "stale" decisions may arise. The race conditions are taken care of by encapsulating the NICAN control algorithm into a critical section executed sequentially by the module threads. To eliminate the production of stale decisions, a global data structure, represented by an array of the size equal to the number of iterations N, has been implemented. Since a stale decision should not be sent to GAMESS, before completing its work a given thread checks whether there is no other thread with a larger number waiting to enter the critical section.

Figure 4.2 shows a timeline diagram for the GAMESS-NICAN interaction while depicting the handling of possible stale results of the decision-making process in the module. `D1` is the decision made by the first thread causing adaptation in the third SCF iteration. A rectangular box next to the timeline of NICAN module represents the critical section encapsulating the control algorithm. The box interior (`C2`) shows that the second thread has entered the critical section and is ready to make a decision while the third thread (`C3` outside the box) is waiting for a lock. To avoid stale results, the second thread is preempted from the critical section execution and has to release the lock for the thread. Thus, the decision `D3` will be made by the third thread and passed to GAMESS (dashed arrow) in this scenario.

The hybrid model of execution starts as the synchronous model and shifts to the asynchronous one when the adaptation occurs once, which is marked by the global variable maintaining the SCF execution mode.

4.4 Results

To obtain performance results, two molecules Adenine-Thymine DNA base pair (AT) and Guanine-Cytosine DNA base pair (GC) are chosen (Figure 4.3). AT and GC are represented using 321 and 316 basis functions, respectively. The tests were conducted for two different input combinations: AT is treated by SCF only while GC continues to the second order Möller-Plesset (MP2) after performing the SCF iterations, thus giving a more accurate energy value. The SCF RHF iterations were performed for each combination. Both molecules start in the conventional mode, i.e., store the 2-e integrals in a file on disk and fetch the integrals from the file when

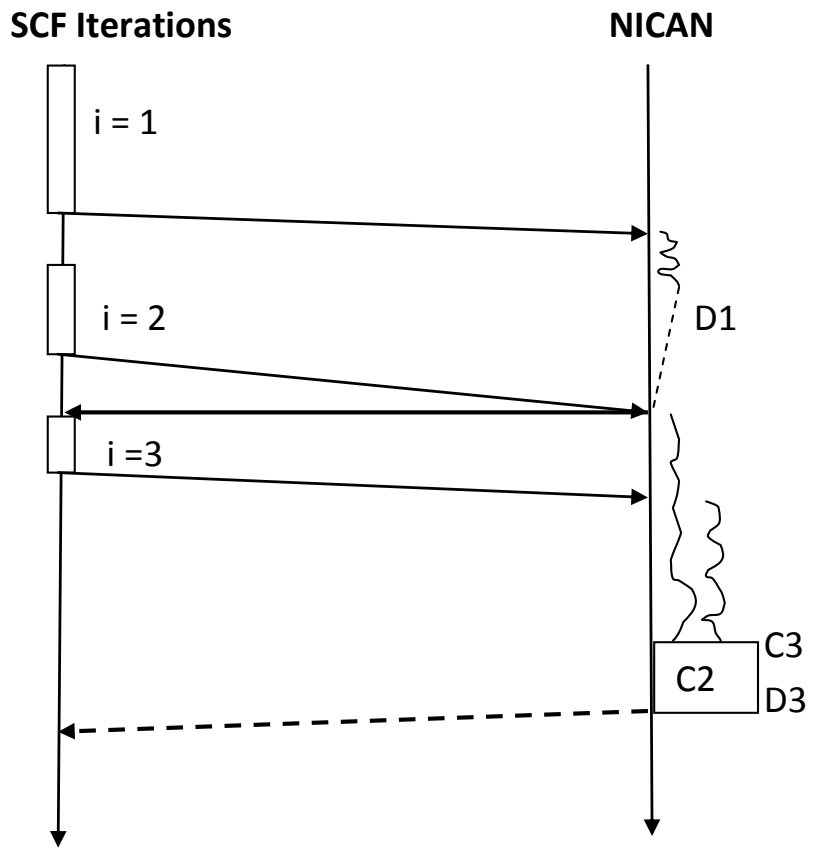

Figure 4.2 Timeline diagram indicating the temporal dependence between SCF iterations and NICAN.

Figure 4.3 Molecule structure: AT (left) and GC (right)

required.

4.4.1 Architecture and software used

The Dynamo cluster, located in the Scalable Computing laboratory, is dedicated to GAMESS and other HPC application research. There are 35 Intel Xeon E5420 nodes with two quad-core processors running at 3 GHz. Each node has a RAM of 16 GB, a 1.5 TB scratch space and runs a 64-bit Redhat Linux. For the experiments presented here, the GAMESS jobs integrated with NICAN are run on up to five entire compute nodes of the cluster, i.e., on 8, 16, 24, 32, 40 cores.

All the three models, synchronous, asynchronous, and hybrid, are considered for the integration of parallel GAMESS calculations and the middleware tool NICAN. Resource congestion is introduced in all the nodes using a file-system benchmarking utility IOzone (10) during the GAMESS run, such that the congestion persists until the GAMESS completion. NICAN is assumed to take 10 seconds to decide on one adaptation. For the experiments reported here, the smallest difference between executing any two SCF iterations in two different modes was observed to be 160 seconds. This value is greater than the time $N\tau$ consumed by NICAN for $N = 15$ iterations typically allocated to SCF and $\tau = 10$ seconds. Furthermore, the largest difference between any two adjacent SCF iterations was observed to be about 199 seconds. So, the NICAN execution time was deliberately chosen to demonstrate how the asynchronous model behaves under the unfavorable conditions and why the hybrid model may be needed.

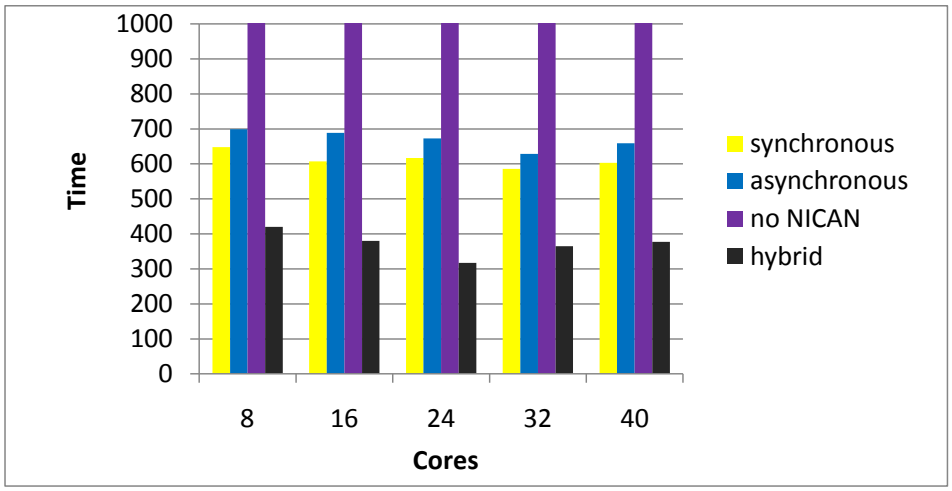

Figure 4.4 AT molecule input with synchronous, asynchronous, hybrid integration, and without NICAN in the presence of persistent congestion.

4.4.2 Discussion of results

For the AT and GC molecules in the presence of congestion, the GAMESS performance is shown in Figure 4.4 and Figure 4.5, respectively, with the synchronous, asynchronous, and hybrid NICAN integration models, as well as when no middleware is used. The number of cores is represented on the X-axis whereas the wall-clock time in seconds is shown on the Y-axis. The congestion forces an adaptation as decided by the control algorithm (Section 3.1).

SCF iterations, starting in the conventional mode and running in the presence of congestion (disk I/O), benefit when they use synchronous model rather than asynchronous model of integration. This gain can be attributed to the timely availability of the adaptation decision given by NICAN and the adaptation happening promptly in the iteration following the congestion discovery. The asynchronous model of integration is outperformed slightly by the synchronous

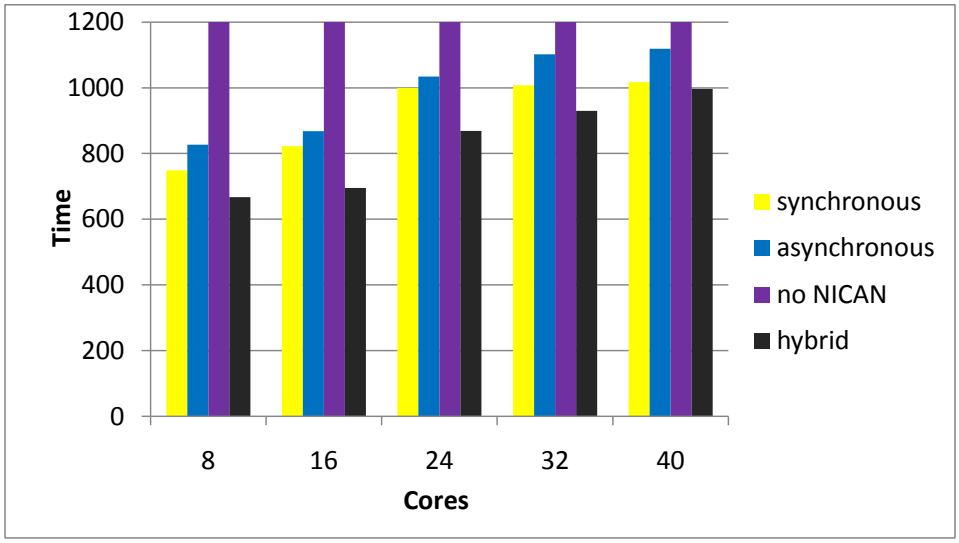

Figure 4.5 GC molecule input with synchronous, asynchronous, hybrid integration, and without NICAN in the presence of persistent congestion.

one due to the additional iteration executing in the conventional mode. For both molecules (Figure 4.4 and Figure 4.5), the number of adaptations is one, i.e., $K = \overline{K} = 1$. The numbers of iterations δ_1 and $\overline{\delta}_1$ before the adaptation are equal to one and two for the synchronous and asynchronous models, respectively, while the iteration numbers equal 14 and 13 after the adaptation.

Even with the adaptation delayed, the asynchronous GAMESS-NICAN integration outperforms GAMESS running without adaptations as depicted by the tallest bars in Figure 4.4 and Figure 4.5. The "no-NICAN" case runs much slower than 1,000 and 1,200 seconds serving as a cut-off in Figure 4.4 and Figure 4.5, respectively, to preserve a good exposition scale. A key observation is that, the workloads as well as dynamic resource conditions contribute to adaptations at the run-time, and thus, they both account for the number and frequency of execution mode shifts. As mentioned earlier, the persistent congestion limits the number of adaptations to one.

Consider the hybrid model of integration, which is a mixture of synchronous and asynchronous ones. This model eliminates adaptation delays for the SCF iterations—thus, alleviating the drawbacks of the asynchronous model—by remaining in the synchronous state while expecting an adaptation and then switching to the asynchronous one with an assumption that the triggering event (i.e., congestion) will persist. This switch improves the GAMES-NICAN performance by eliminating subsequent synchronizations. As predicted theoretically in Section 3, the hybrid model performs better than the synchronous one under this assumption.

The wall clock times of the GAMESS runs without disk I/O congestion for synchronous and asynchronous models and without middleware integration for the AT and GC molecules are shown in Figure 4.6 and Figure 4.7, respectively. When there is only a single conventional job per node, i.e., there is no heavy disk I/O congestion, the synchronous model of integration makes computationally intensive SCF calculations wait for the return of NICAN. In the absence of congestion, the hybrid model reduces to the synchronous model, since there is no call for the adaptation. This may be viewed as the drawback of the hybrid model. The asynchronous model of integration alleviates this situation since it returns immediately to continue with the subsequent SCF iterations. Thus, for the AT and GC (Figure 4.6 and Figure 4.7), the

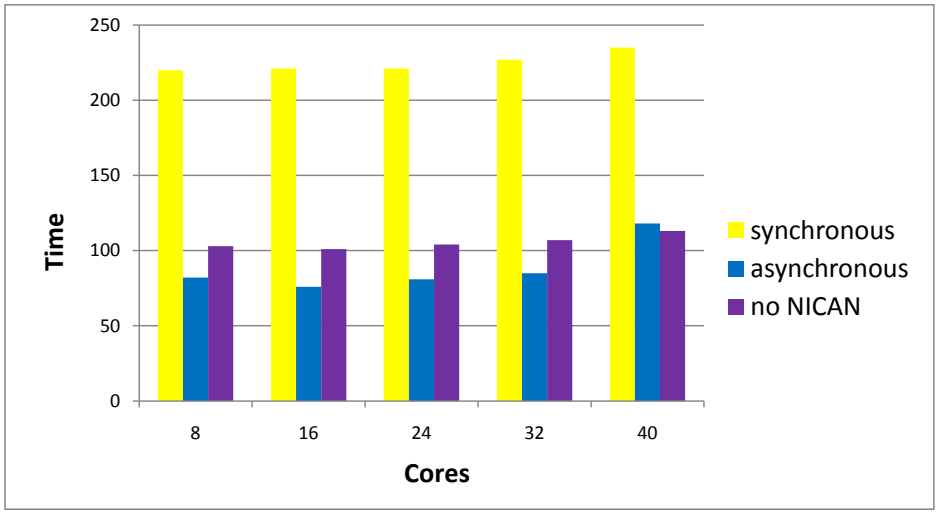

Figure 4.6 AT molecule input for synchronous, asynchronous, no-middleware integration in absence of congestion.

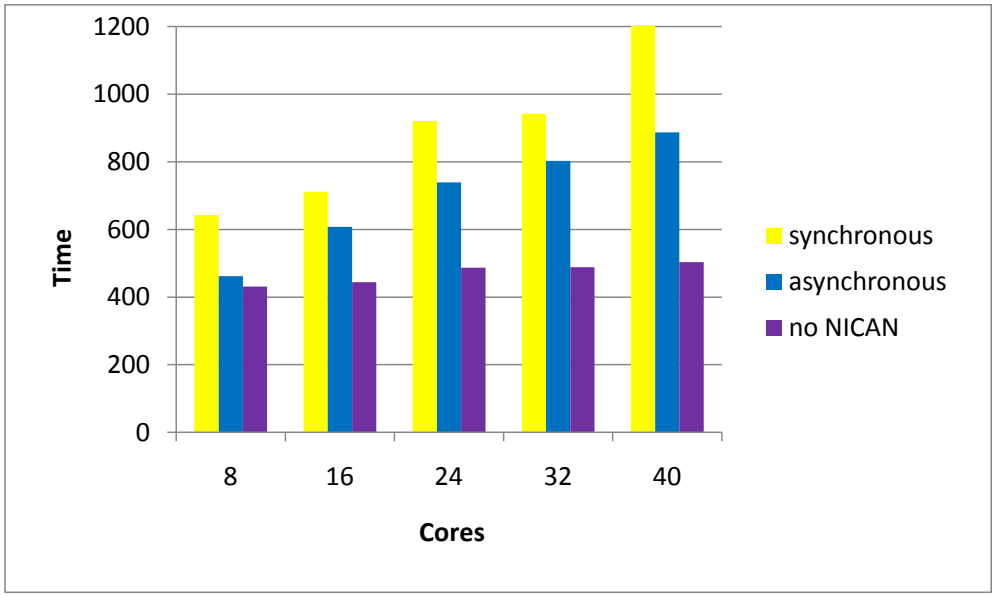

Figure 4.7 GC molecule input for synchronous, asynchronous, no-middleware integration in absence of congestion.

Table 4.1 MP2 electron correlation and SCF as percentages of the total execution time for GC in the absence of congestion and no NICAN.

Cores	8	16	24	32	40
MP2, %	74.4	80.4	86.4	86.6	85.8
SCF, %	15.8	17	12.5	12.3	13

asynchronous model delivers almost the same performance as GAMESS without NICAN in the environment requiring no adaptations.

In the case of GC, an MP2 calculation is used after the SCF iterations complete and takes a significant percent of the execution time as shown in Table 4.1. Therefore, the gains over the synchronous execution are less pronounced. Also, note that, for the GC calculation (Figure 4.7), the wall clock time does not decrease with the increase in the core numbers, which may be attributed to the problem size not being sufficiently large to outweigh the communication and other parallel overhead.

CHAPTER 5. STATIC AND DYNAMIC ADAPTATIONS IN FRAGMENT MOLECULAR ORBITAL METHOD

5.1 Introduction

The optimizations in geometry of large molecules are very hard to perform. Therefore, some more methods are proposed to deal with problems involving large molecules. The method proposed by (16), FMO divides the molecule into fragments and electrons of the molecule are assigned to these fragments. The MO calculations, assuming the MO is local to the fragment, are performed for the electrons in that fragment. After all the single fragment MO (monomer) calculations are done, the MOs for the fragment pairs (dimer) are calculated for the total energy of the molecule. This is expected to reduce the computational time for large molecules drastically, as it avoids performing MO calculations on the whole molecule at once.

After the initial decision of how to divide the molecule into fragments, comes the implementation part. A new hierarchical parallelization scheme GDDI proposed by (17) forms groups out of the nodes at hand and assigns tasks to these groups. This is the parallelization at the outer level. Then each group in turn does parallelization at inner level by diving its task into smaller work loads. A group may contain one or more nodes and each node may contain one or more CPUs.

The FMO method mentioned above can be used with this scheme as it provides separate computational tasks (monomers MO calculations and dimers MO calculations) that can be assigned to groups. A molecule is divided into fragments and RHF calculation is performed on each fragment separately by one group. After density convergence is reached for all monomers, these densities are used to perform SCF runs on dimers by each group independently. There is a need for synchronization point after the monomer MO calculations involving density and energy

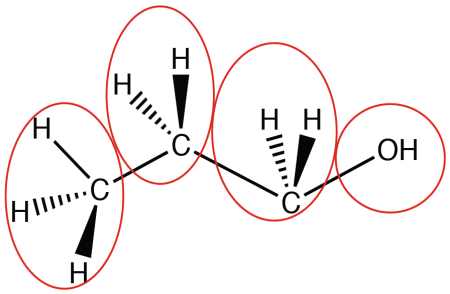

Figure 5.1 A molecule is divided into fragments circled in red

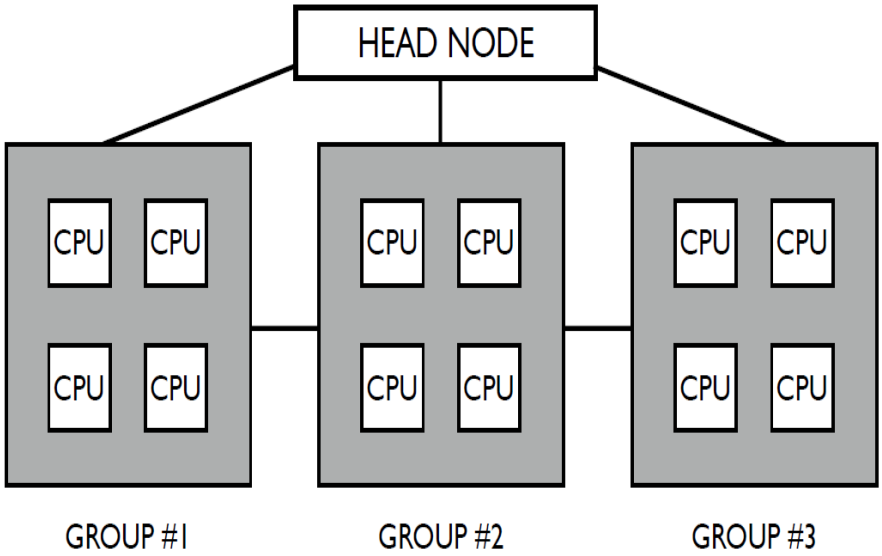

Figure 5.2 Nodes divided as groups in GDDI

exchange, and after the dimer MO calculations involving dimer energy exchange. (12) gives more documentation of the standard FMO procedure. (18) provides advanced three-body terms in FMO method. To reduce the amount of time involved in waiting at these synchronization points, load balancing is used. Both monomers and dimers are executed in the decreasing order of their computational work. The present work focuses on fine-tuning this srtategy based on different load balancing techniques at inter group or outer level. Load balancing can be static featuring fixed-division of work or dynamic involving flexible scheme that changes on-the-fly where groups can ask for the next available work item upon the completion of the current work. Dynamic load balancing proves to be very helpful when the characteristics of the system are changing at run-time and the cluster or the work environment is heterogeneous in nature. Group heterogeneity can be described in terms of different number of nodes, nodes in turn comprising different number of CPUs (cores), changing CPU loads, and varying available memory.

The current implementation of FMO in GAMESS assumes homogeneous system and load balancing is done based on group size. Largest task is assigned to largest group (group with most nodes). Present work focuses on tuning this strategy to accommodate static adaptations based on group heterogeneity. Also, dynamic adaptations based on the changing available memory on groups (essentially nodes) are introduced to give more resilient FMO runs. With the help of these dynamic adaptations, an FMO run does not abort as soon as a dimer cannot be fit on a group due to lack of memory but delays its execution and allows it to execute at a later time if memory becomes available.

5.2 Integration with middleware

Static adaptations are based on differing number of cores on nodes in the present work. This requires keeping track of the list of the computational tasks executed on each group at any time and also be able to decide on the next task going on each group. Dynamic adaptations require a continuous monitoring of the system and, based on available memory, and be able to decide on the next task going to each group. The strategies for each type of adaptation are described in the next section. As seen, this decision-making can be decoupled from the application using

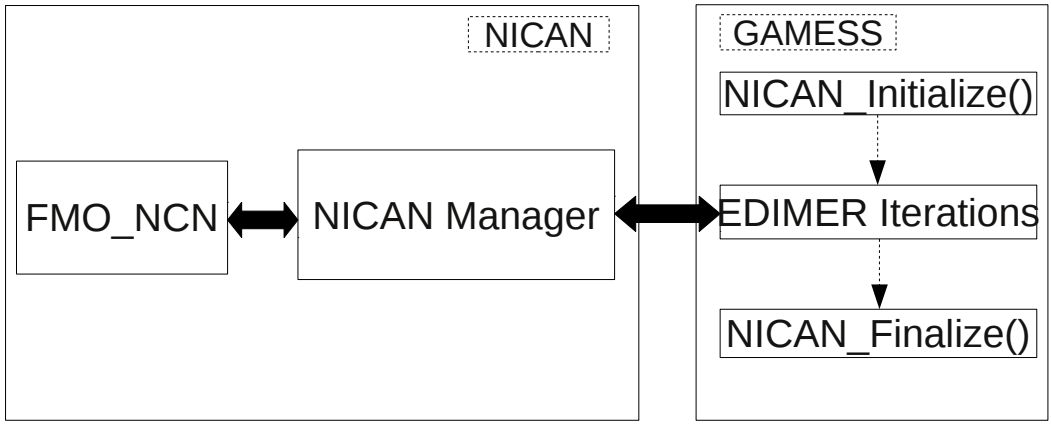

Figure 5.3 Integration model of FMO with NICAN

NICAN, which enables these adaptations in the FMO method. NICAN suggests the groups at the starting of a new task-to-group assignment as to what computational task is to be performed now.

To be able to take this decision in the NICAN manager, certain information is needed from the FMO side. The task description is expected to include the group number, node name, iteration number, previous task number and current task number (required for dynamic adaptations only) and a process ID.

5.3 Adaptations

"Largest task first" is a decent load balancing strategy for parallel runs. A simple scheme of task assignment to groups in FMO is shown in Figure 5.4 where the tasks F1 through Fn are ordered in the decreasing order of their sizes. Note, a task with respect to an FMO run might mean a monomer or a dimer or a trimer.

However, if the system is heterogeneous, scheduling pattern of the incoming tasks on groups effects the run time considerably. The proposed static adaptations aim at improving the

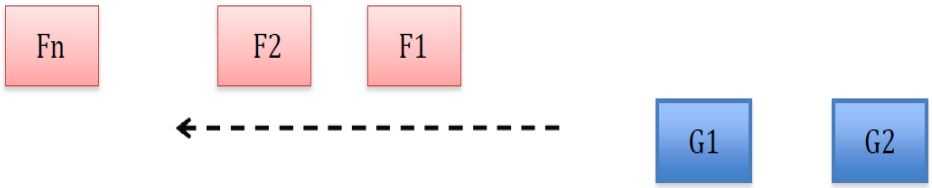

Figure 5.4 The original "largest task first to the first group" task scheduling in FMO

scheduling pattern beyond the ab-initio load balancing. As mentioned earlier, the current implementation of FMO in GAMESS does not account for varying number of cores on groups. A simple case when there is an equal number of nodes in each group but each node having different number of cores fails to satisfy the largest task being assigned to the largest group scheduling. Therefore, this is modified into the state where the decision-making of the task assignment to groups is based on the number of cores per group. This revised assignment is necessary to do statically only upto the first k group requests, where k is the number of groups. After this, a group is assigned the next available task. No group is idle at any point of time is the idea behind this adaptation which hereby increases the throughput of FMO. All the adaptations are based on the information collected from the input file at the start of the run statically. Static adaptations tend to have limited effect under certain circumstances. For example, if the molecule is divided into fragments of almost equal sizes, the tasks formed are of equal sizes too. In this case of little or no variation in size, any change in the scheduling pattern has no effect on the execution timing as every group gets to perform same amount of computation. Similarly, when the groups are of the same size or there is negligible amount of variation in the group size, each dimer is executed on arguably similar group. Hence, no ef-

fect of any change in scheduling pattern. Therefore, static adaptations require heterogeneity in terms of both task and group sizes. Note that it is typical for chemists to divide a molecule into fragments varying in size as, e.g., in protein molecules to be split into amino acids. Figure 5.5 gives an idea of how an FMO run with and without the middleware may look like.

Group heterogeneity as discussed can be in various forms. However, available memory always appears to be a foremost parameter that is essential for any kind of computational task to be be able to complete. But what is different with memory is it can change dynamically. As a single FMO run involves the execution of many computation and memory-intensive calculations, for example, SCF and MP2. The available memory on a group must be able to meet the memory requirements of a task to be executed on that particular group. Often many tasks are performed on a cluster simultaneously, so there is a good chance of certain amount of memory being used up on a particular group and the available memory may not be sufficient for the current run. It would be extremely helpful if the memory requirement of the task is known in advance, because it helps us in deciding whether the next task scheduled by the ab initio load balancing scheme to be executed on a group is a good fit on that particular group. This can be done by checking the task requirement in advance against the available memory on the group dynamically. The available memory in a node can be calculated by acquiring system related information. A very sophisticated method to calculate available memory for dynamic adaptations is given by (14). Dynamic adaptations come into picture if a group fails to meet the memory requirements of any task scheduled to be executed by it. Such tasks are marked *missed* and scheduled to be executed later allowing a next task that would prove to be a good fit on that particular group to be executed. Figure 5.6 gives a scenario where the dynamic scheduling being discussed comes into effect. Tasks F_1, F_i, F_j, F_k are ordered in the decreasing order of their sizes. When an FMO run starts, each task is assigned to groups G_1, G_i, G_j, G_k respectively. Note that the dynamic adaptations are used separately from static ones here although the two may be combined in future. When group G_i fails to satisfy the memory requirements of task F_i, this task is marked as *missed* and stored in the *missed task queue* which is shown adjacent to the groups. Similarly in Figure 5.6 in the case of F_j where G_j lacks enough available memory to execute G_j. After all the tasks form the initial ordering are done with either execution or

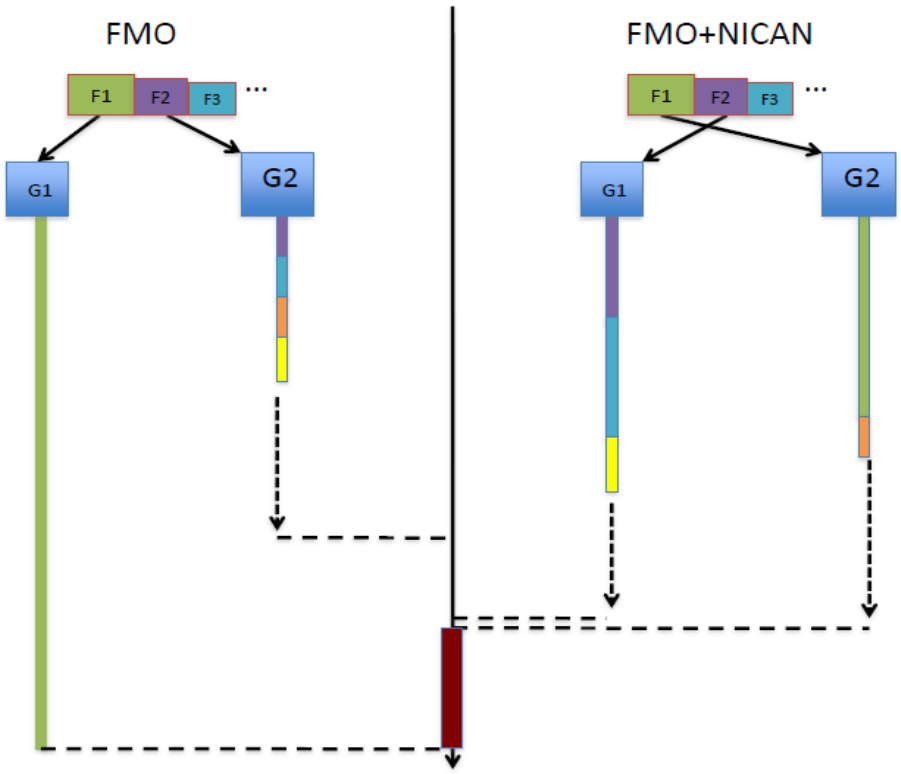

Figure 5.5 Example FMO run w/ and w/o NICAN

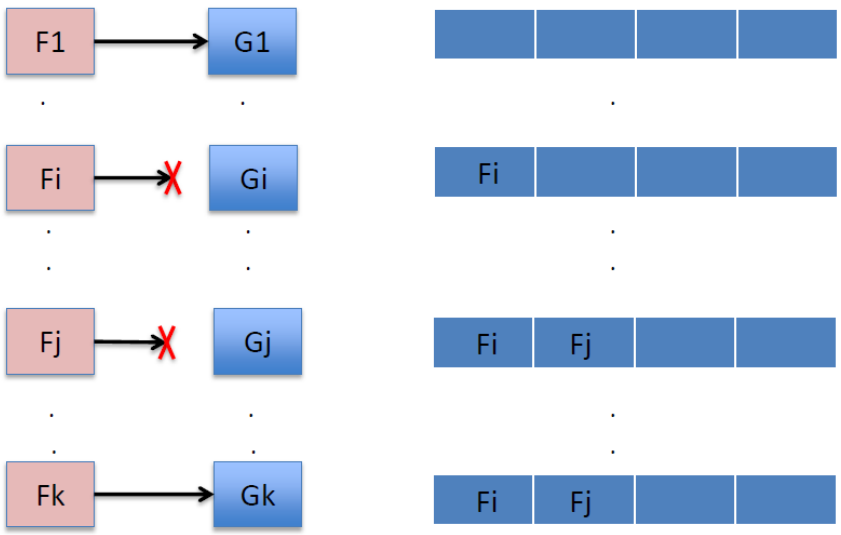

Figure 5.6 Scenario showing *missed* dimers

marked with being *missed*, this *missed task queue* is iterated over to allow the execution of *missed* tasks. This is just basic scheduling which is described as strategy 2 below.

There are two strategies to execute the *missed* tasks based on when they are executed. 1) A periodic check can be made to see if any of *missed* tasks can be a good fit on any of the groups and schedule it for the current iteration of that group. This preserves the essence of "largest task first" principle if not to a full extent but to a certain extent and definitely more than strategy 2. Because *missed* tasks are always bigger or of same size as the tasks occurring after them. 2) After the last task is scheduled, the *missed* tasks are executed in the order they were stored in the queue if the memory check is satisfied. Figure 5.7 shows the algorithm implemented with strategy 2.

More complex strategies can be considered. A *missed* task could be scheduled to run on a different group right away if a group does not satisfy its memory requirements. Lets call this

strategy 3. But this is not fruitful every time as there may be scenarios where this can still lead to task being marked as *missed*. For example, if every other group is busy executing its quota, even though the memory check is satisfied on one or more of them, the current task has to be a executed at a later time. So, it is marked as *missed* and stored in the array. The only scenario in which this scheduling can work is if the task is in set of the first k tasks, where k is the number of groups. This is because there is a chance that other groups are not assigned a task yet and the current task can be assigned to one of the task-not-yet-assigned groups that satisfies the current task's memory requirements. Another strategy, strategy 4 could be scheduling the next *missed* task in the queue in the next iteration of a group that satisfies the memory requirements of this *missed* task. This can be done with every group at the starting of each iteration because this gets rid of the *missed* tasks quickly if memory is available on a group. This resembles the strategy 1, the only difference being the periodic checking is done at the starting of an iteration rather than for every predetermined amount of time. However, strategy 1, 3 and 4 are very complex to implement. Other types of tuning are possible too such as division of nodes into groups, changing group divisions on the fly, choice of SF type (direct/conventional).

5.3.1 Implementation

For the implementation and analysis purposes, the integration of the middle ware with dimer calculations was considered and all the results described in the next section correspond to dimer calculations. The information regarding groups, nodes, cores is read from the input file by the NICAN module before the processes are kicked off on all the groups. The groups are then organized in the decreasing order of their priority. And an incoming MO request is served with the dimer number the current iteration has to execute. The next-in-line processs group is identified by the information provided by the request and then a dimer number is assigned to this process based on the groups priority. This is how static adaptations are obtained.

Implementation of the dynamic adaptations requires the advance knowledge of all the dimer memory requirements. The present work achieves this with a "pre-run". In this pre-run, all the dimer memory requirements are calculated. This is done by monitoring all the malloc calls

inside a dimer calculation. All the dimers with their corresponding memory requirements are stored in a file which is placed on the master (FMO kick-off) node so that the module can access it dynamically. From the subsequent runs, the ab-initio load balancing scheme is continued. If there is a situation where a dimer memory requirement (read from the file) is not met by the available memory on group (read from system information that is again stored on FMO kick-off node), the current dimer is stored in an array and the next available dimer is chosen and is returned to the current FMO request if the memory check is satisfied for the new dimer. This is repeated until a satisfying case is obtained. The dimer that missed the execution first time is called a *missed* dimer. The array in which *missed* dimers are stored is essentially a queue as later they can be executed in the "largest dimer first" order.

Figure 5.7 shows strategy 2 discussed in the previous section.

5.4 Results

5.4.1 Architecture and software used

The Dynamo cluster as mentioned in the previous chapter is used as a test bed for the experiments in this chapter too. For the experiments presented here, the GAMESS jobs integrated with NICAN are run on up to four entire compute nodes (4 groups). It is known that the FMO runs take the advantage of the fact that one node-per-group design works well for smaller molecules. Hence, in the experiments conducted in this work, single node per group is used. However, the design and the implementation presented earlier work for any configuration of groups. A water cluster may be divided into fragments in the way showed in Figure 5.8

However, the water cluster used for the current set of experiments regarding static adaptations has 16 molecules and the fragment set is [15,12,3,3,3,3,3,3,3,3] where the numbers correspond to number of atoms in each fragment. Dimers are thus 27×1,18×7,15×7,6×21. For dynamic adaptations the water cluster is divided into fragments as [15,9,3,3,3,3,3,3,3,3]. Therefore the dimer set is 24×1,18×8,12×8,6×28; where $A \times B$ has the following meaning: A corresponds to number of atoms in the dimer and B corresponds to the number of dimers with A atoms.

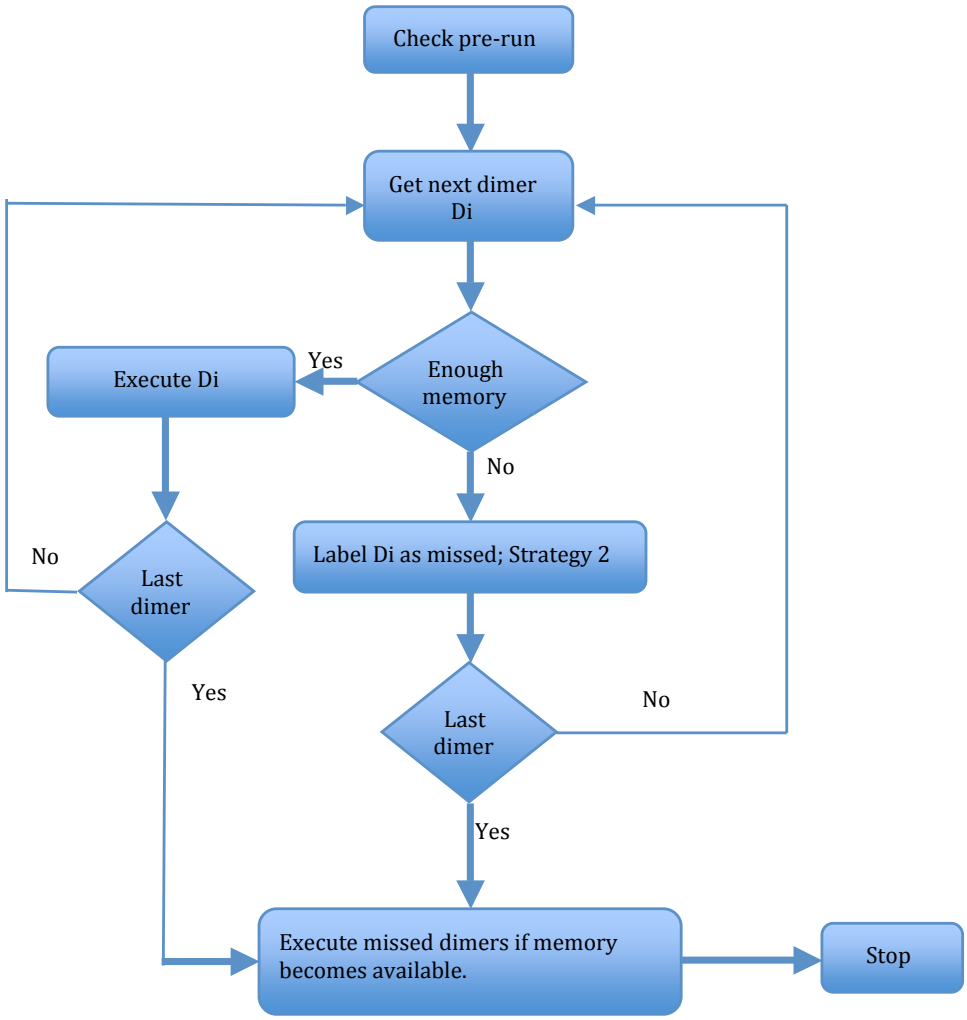

Figure 5.7 Flowchart for the dynamic adaptation implementation

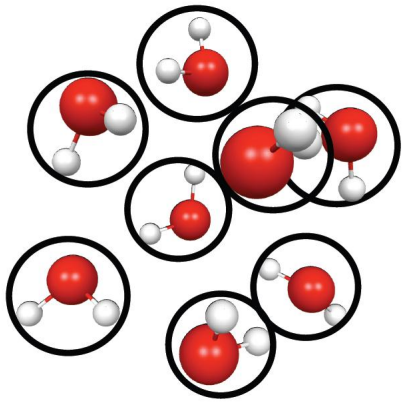

Figure 5.8 A water cluster divided into fragments in black

5.4.2 Discussion of results

Figure 5.9, Figure 5.10, Figure 5.13, and Figure 5.14 represent different scenarios on X-axis based on group division. Each scenario has a different number of cores per group and ith scenario is represented by C_i. Number of cores per group is given right below the scenario name. Y-axis represents the maximum of execution time among all the groups per each scenario. The execution time on a group is the sum of wall-clock timings of the execution of all the dimers scheduled on that group and is measured in seconds. The input for the above mentioned runs corresponds to the fragment configuration mentioned for the static adaptations earlier.

The column w/o NICAN specifies the timings of the FMO run without the integration of the middleware. As seen, the maximum time taken by the application in this case is more than that of the application integrated with the middleware. This is due to the fact that the largest dimer is executed by the largest group in the presence of middleware i.e. on the group with 4, 6, 8 cores in the scenarios shown. In the absence of middleware, the largest dimer is executed by the first group occurring in the input file i.e. the group with 1 core in all the three scenarios. The existing code is written in such a way that the first dimer goes to the group occurring first in the input file. We can get rid of this default assignment with the coupling of middleware

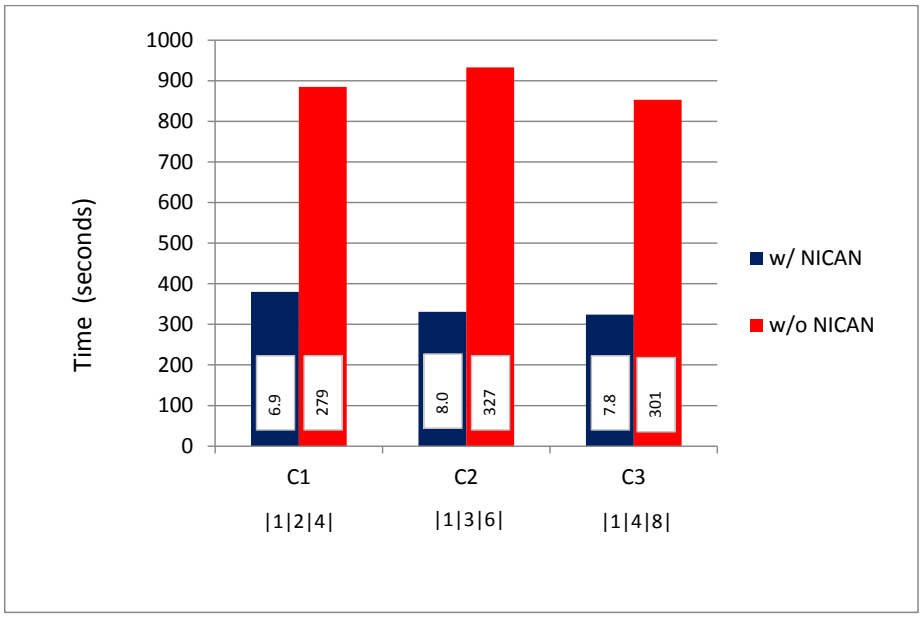

Figure 5.9 Maximum execution time among three groups for three scenarios of an FMO run with and without middleware along with standard deviation for each column

and get more efficient times as shown in Figure 5.9.

Figure 5.10 shows a similar set of experiments as previous one. In this case, code without NICAN performs better than code with NICAN. This is due to the fact that largest dimer which is executed on group with 6 or 8 cores in presence of middleware surprisingly takes almost same amount of time as when executed on group with 2 or 4 cores. The reason for such an abnormality may be the CPU utilization is not 100 percent. As a result the group with least number of cores has to execute more dimers than it has to when no adaptations are forced.

This can be seen in the individual dimer scheduling pattern shown for a scenario C_1 (from Figure 5.10) in the Figure 5.11 for without middleware and Figure 5.12 for with middleware. This will give an insight on the difference between the execution of FMO with and without

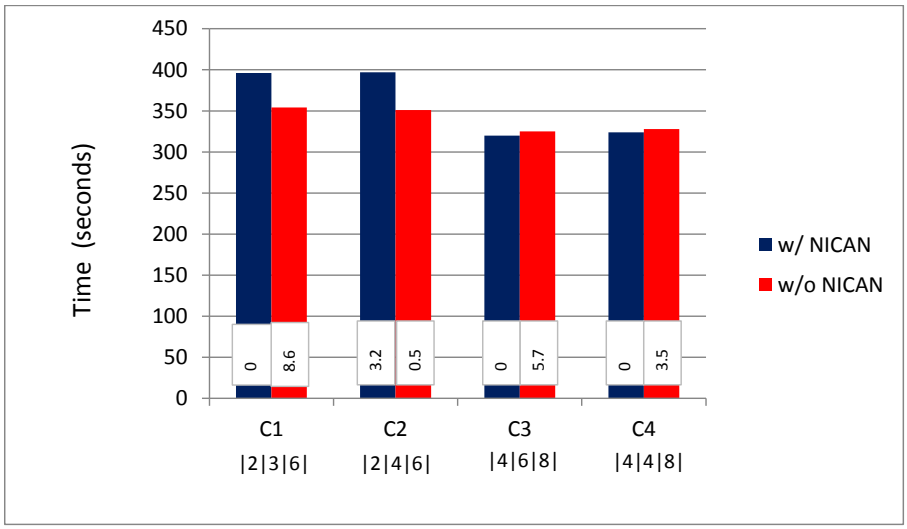

Figure 5.10 Maximum execution time among three groups with different configuration from previous one of an FMO run with and without middleware along with standard deviation for each column

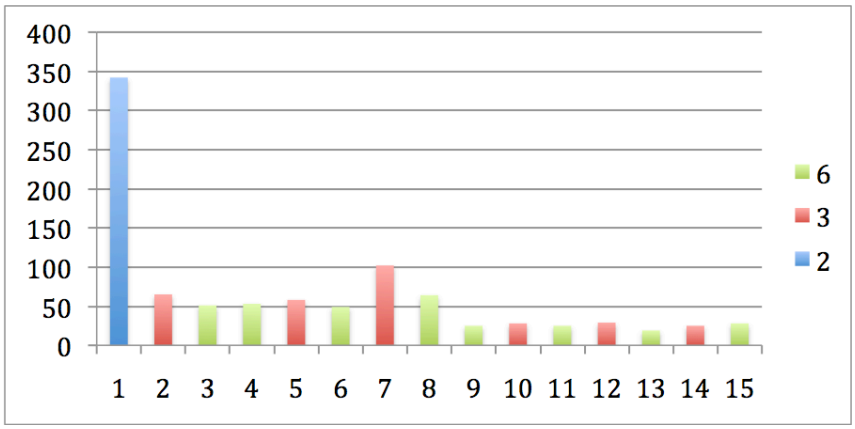

Figure 5.11 Execution timings of first 15 dimers for scenario C_1 from Figure 5.10 without NICAN

NICAN. X-axis shows the dimer numbers and Y-axis shows the execution time of each dimer in seconds. Legend represents the three groups and also shows the number of cores on each group.

Only first 15 dimers are shown here as the rest of them take more or less same time independent of the number of cores they are executed on as their size is too small to take the advantage of the parallel code. However, we can see that as the number of cores per group are becoming close or there is less variation in the group size, the effect of the middleware is decreased i.e., code with or without middleware performs similarly. This can be seen in the last set of bars in Figure 5.10.

Figure 5.13 and Figure 5.14 display the above discussed scenarios with group number increased by one and two respectively. The code with NICAN outperforms code without NICAN in every scenario. The first and the second sets of bars in Figure 5.13 are self explanatory and there are no anomalies as in the previous case. Also, the third set of bars confirms that as

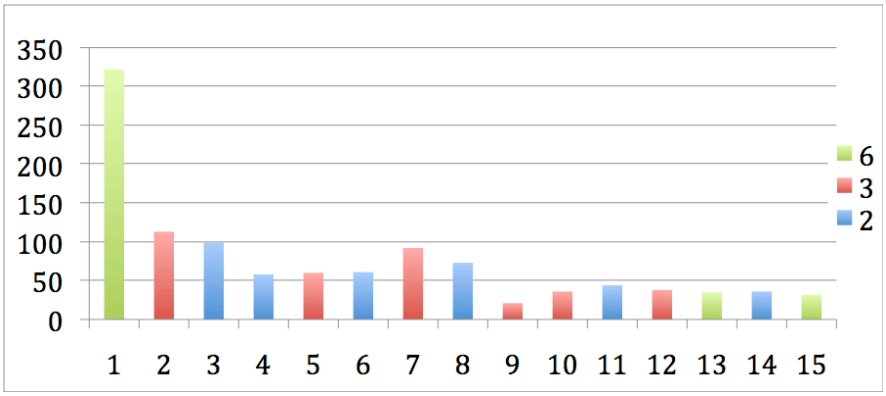

Figure 5.12 Execution timings of first 15 dimers for scenario C_1 from Figure 5.10 with NICAN

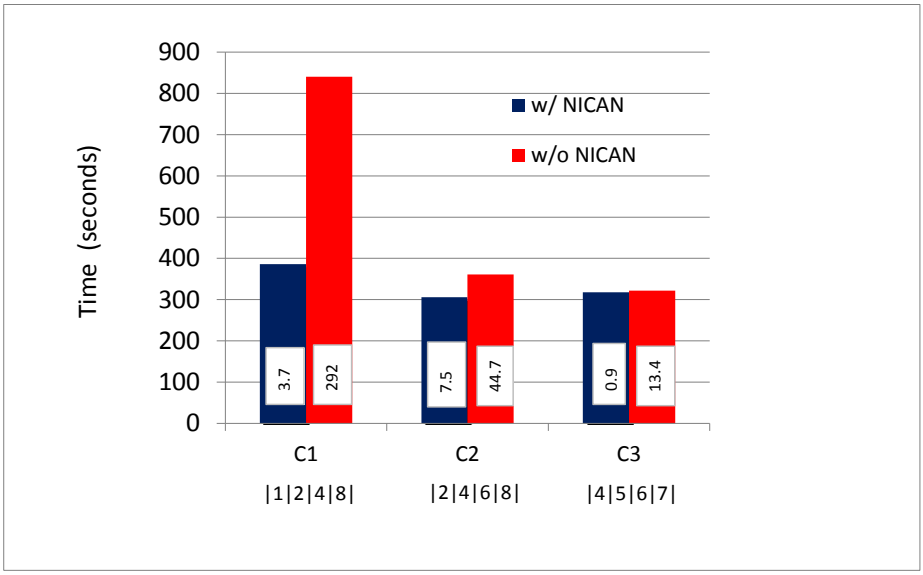

Figure 5.13 Maximum execution time among four groups for three scenarios of an FMO run with and without middleware along with standard deviation for each column

the variation in the group size decreases, the presence of middleware which has an effect on the pattern of the dimer scheduling is not very beneficial. Similarly, in Figure 5.14 the first two sets of bars show that code with NICAN performs better than code without NICAN. But as the variation in group size decreases, there is less performance gain using the middleware. This can be seen in last two sets of bars in Figure 5.14. The standard deviations shown in the figures straight away confirm that using the middleware, the idle time on a group is decreased a lot, infact it is almost negligible unlike FMO without NICAN.

All the discussion until now was regarding the static adaptations. To study the effect of dynamic adaptations, dynamic adaptations were forced to occur by simulating the available memory values according to the values from the "pre-run". Figure 5.15 shows the details of

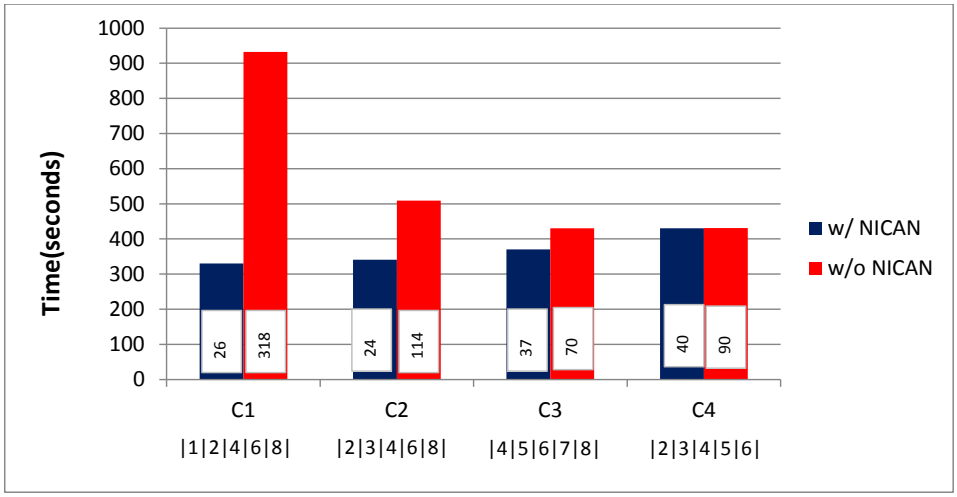

Figure 5.14 Maximum execution time among five groups for four scenarios of an FMO run with and without middleware along with standard deviation for each column

the execution timings of the water cluster with the fragment configuration mentioned earlier corresponding to dynamic adaptations. Two groups with same number of cores were used to nullify the effect of change in pattern of dimer scheduling as it was mentioned earlier that dynamic adaptations were considered independent of the static ones. The available memory values were simulated in the way that dimers 1, 18, 41 were marked as *missed* in the initial ordering of the dimers. Strategy 2 was followed then to execute the *missed* dimers 1, 18 and 41. Also, to show the scenario where the available memory might not be enough to run a dimer even at the end of the initial ordering, it was simulated that none of the groups had sufficient memory to run dimer 1 while the *missed* dimer execution started. This is represented by blue bars in the graph. Otherwise, if a group has enough memory for all the *missed* dimers, then the FMO run is complete and is shown by red bars. Note that more time on group 1 is due to dimer 1 being executed on group 1 and since it is much bigger in size than dimers 18 and 41. Without NICAN, the FMO run aborted due to the lack of enough memory for dimer 1 on group 1. As a comparison between different startegies, startegy 1 was implemented and compared with startegy 2. Again in both the cases, the available memory was simulated in such a way that dimer 1 would be marked as *missed* in the initial ordering. Using startegy 2, dimer 1 was executed by group 1 after the end of initial ordering. Hence, wall clock time recoreded in this case was more than that of group 2. Using startegy 1, a periodic check was made to ensure dimer 1 would fit on either group if available memory was sufficient. For the comparison purposes, in strategy 1, available memory was simulated to be sufficient on both the groups after dimer 3 was executed. Note that, dimer 1 is not executed until now. After dimer 3 has been executed and available memory on either groups is sufficient for dimer 1, group 2 executes dimer 1 before continuing with other dimers' execution in the initial ordering. This preserves the "largest job first" condition better than strategy 2. As seen in Figure 5.16, Stategy 1 performs better compared to Startegy 2.

Note that "pre-run" is required for the dynamic adaptations (with NICAN) and it takes the time of an entire FMO run. There is no overhead due to decision making using NICAN. This can be seen in Figure 5.17 where the total execution time on both the groups with and without NICAN is exactly the same. This simulation considers that there is no lack of memory

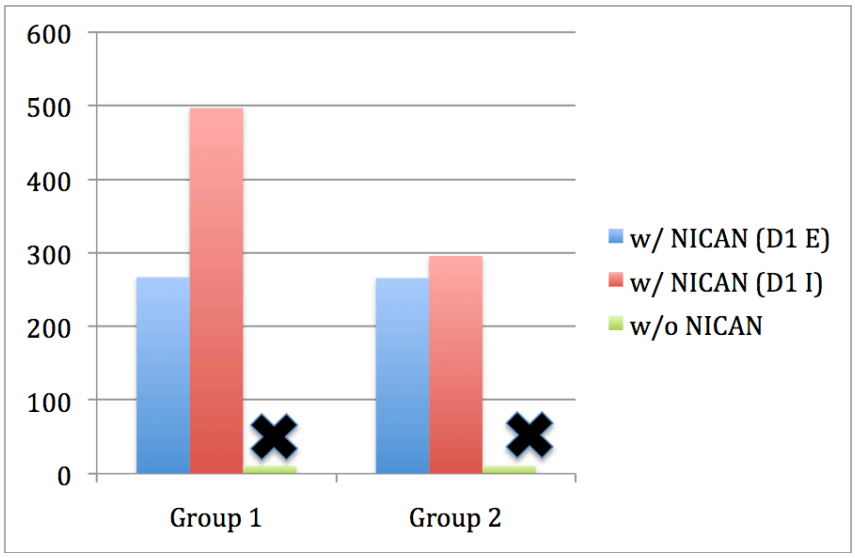

Figure 5.15 Figure showing the effect of dynamic adaptations (w/ NICAN) compared to no adaptations (w/o NICAN)

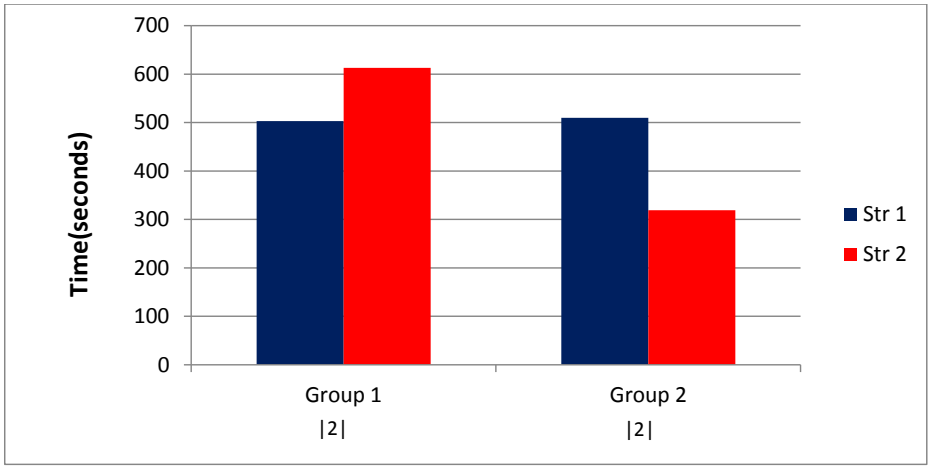

Figure 5.16 Figure showing the effect on execution time while using dynamic adaptations with Stategy 1 vs. Stategy 2

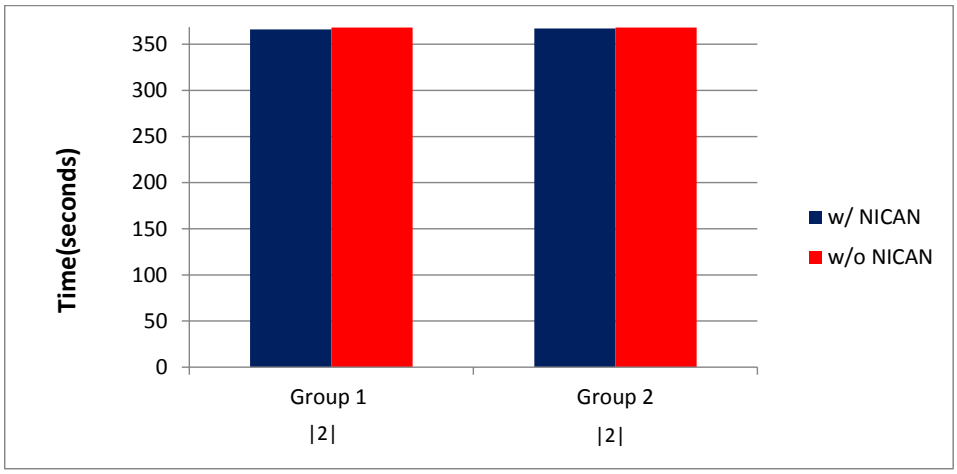

Figure 5.17 Total execution time on two groups of an FMO run with middleware (using dynamic adaptations) and without middleware (no adaptations)

for any of the dimers while using the code with NICAN in order to observe if there is any overhead in decision making particularly. The "pre-run" can be considered as the FMO run without middleware in this case. When there is no adaptation required, codes with and without middleware are equivalent and when there is an adaptation required, the overhead is due to the extra *missed* dimer executing on one of the groups (can be seen in Figure 5.15) but not due to the overhead of decision making.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Former part of the present work investigates three integration models—synchronous, asynchronous, and hybrid—of the NICAN middleware with computationally-intensive applications, such as GAMESS. The total execution times have been compared among these models when GAMESS algorithm (SCF) adaptations are invoked and when no adaptation is needed. It has been demonstrated theoretically and supported by the experiments that, under the assumption of persistent congestion, the hybrid model outperforms the synchronous one, which, in turn, performs better than asynchronous. However, it may be detrimental to decide on the type of integration statically, before the actual execution, since the the presence or absence of congestion is typically a runtime system condition. Therefore, a hybrid model of integration is a better choice to start with when the system conditions may not be ideal and a congestion is probable.

As a future work, the hybrid model is to be extended for the multiple congestion phases and applied in other computationally-intensive parts of electronic structure calculations as well as in other applications that have been integrated with NICAN.

The latter part of the present work deals with the performance enhancement of the FMO runs in heterogeneous systems by applying appropriate load balancing scheme based on the run time system conditions. The execution times of the FMO runs with and without the adaptations provided by the middleware are presented. The advantages of using the middleware (under the assumption that dimers and the groups vary in their sizes) to get the static adaptations are self evident by the results provided. Dynamic adaptations that prove helpful to make the FMO runs more resilient are also shown. However, there is no overhead using the middleware even when adaptations are not required. So, here integrating the application with the middleware is always recommended.

In the future, more static adaptations regarding FMO runs can be acheived. For example, division of groups before the kickoff of the processes can be a potential area to look into to see how the change in the group sizes can effect the execution time. Also, the dynamic adaptations can be extended to include decision making based on CPU load. In the present work, the static adaptations and dynamic adaptations are studied independently. However, these can be combined to give more specific and realistic scenario of group heterogeneity. Decision making can be based both on the number of cores in a group and also available memory.The group with largest number of cores that satisfies the memory check gets to execute the current dimer. A better approach to know the dimer memory requirements is to intelligently guess them based on the number of AOs that constitute the dimer. Several test runs can be taken and a pattern can be found out that would guess the dimer memory requirement based on the number of AOs. Number of AOs in a dimer can always be got from current implementation of FMO in GAMESS.

BIBLIOGRAPHY

[1] L. Seshagiri, M. Sosonkina, Z. Zhang, *Exploring tuning strategies for quantum chemistry applications.* In Proc. 2009 Int'l Workshop on Automatic Performance Tuning (iWAPT-2009), Tokyo, Japan, Oct 1-2, 2009, 2009.

[2] L. Li, J. P. Kenny, M. Wu , K. Huck, A. Gaenko, M. S. Gordon , C. L. Janssen, L. Curfman McInnes, H. Mori, H. M. Netzloff, B. Norris, and T. L. Windus, *Adaptive Application Composition in Quantum Chemistry.* The 5th International Conference on the Quality of Software Architectures (QoSA 2009), East Stroudsburg University, Pennsylvania, USA, June 22-26, 2009.

[3] C. Tapus, I-Hsin Chung, and J. K. Hollingsworth, *Active Harmony: Towards Automated Performance Tuning.* SC'01 November 2002.

[4] B. Plale, K. Schwan, *dQUOB: Efficient Queries for Reducing End-to-End Latency in Large Data Streams.* High Performance Distributed Computing (HPDC-9), August 2000.

[5] H. Rasheed, R. Gruber, V. Keller, W. Ziegler, O. Waeldrich, P. Wieder P. Kuonen, *IANOS: An Intelligent Application Oriented Scheduling Framework For An HPCN Grid.*

[6] N. Ustemirov, M. Sosonkina, M. S. Gordon, M. W. Schmidt, *Dynamic algorithm selection in parallel GAMESS calculations* In Proc. 2006 International Conference on Parallel Processing Workshops, Columbus, OH, pages 489-496. IEEE Computer Society, 2006.

[7] 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, J. A. Montgomery Jr, *General atomic and molecular electronic structure system* Journal of Computational Chemistry, pages 1347-1363, November 1993

[8] L. Seshagiri, M. Wu, M. Sosonkina, Z. Zhang, M. S. Gordon, M. W. Schmidt, *Enhancing adaptive middleware for quantum chemistry applications with a database framework.* In Int'l Parallel and Distributed Processing Symposium (IPDPS 2010), 11th Workshop on Parallel and Distributed Scientific and Engineering Computing, Atlanta, GA, Apr. 2010, 2010.

[9] L. Seshagiri, M. Sosonkina, Z. Zhang, *Electronic structure calculations and adaptation scheme in multi-core computing environments.* In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I, volume 5544 of Lecture Notes in Computer Science, pages 3-12. Springer, 2009.

[10] *IOzone Filesystem Benchmark.* http://www.iozone.org/

[11] *Introduction to GAMESS.* http://www.msg.ameslab.gov/tutorials/tutorials.html

[12] *FMO Presentation.* http://www.msg.ameslab.gov/tutorials/tutorials.html

[13] A. Yamin, I. Augustin, L. Cavalheiro da Silva, R. Araujo Real, A. E. Schaeffer Filho, C. F. Resin Geyer *EXEHDA: adaptive middleware for building a pervasive grid environment.* Proceeding of the 2005 conference on Self-Organization and Autonomic Informatics (I)

[14] L. Xiao, S. Chen, X. Zhang *Dynamic cluster resource allocations for jobs with known and unknown memory demands.* Parallel and Distributed Systems, IEEE Transactions on, Mar 2002, 223-240

[15] A. Srinivasa, M. Sosonkina, P. Maris, J. P. Vary *Dynamic Adaptations in ab-initio Nuclear Physics Calculations on Multicore Computer Architectures.* Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, May 2011, 1332-1339

[16] K. Kitaura, E. Ikeo, T. Asada, T. Nakano, M. Uebayasi *Fragment molecular orbital method: an approximate computational method for large molecules.* Chemical Physics Letters, Volume 313, Issues 3-4, 1999, 701-706

[17] D. G. Fedorov, R. M. Olson, K. Kitaura, M. S. Gordon, S. Koseki *A new hierarchical parallelization scheme: Generalized distributed data interface (GDDI), and an application to the fragment molecular orbital method (FMO).* Journal of Computational Chemistry, Volume 25, pages 872-880, 2004

[18] D. G. Fedorov, K. Kitaura *The importance of three-body terms in the fragment molecular orbital method.* J. Chem. Phys. 120, 6832 (2004); doi:10.1063/1.1687334 (9 pages)

[19] I-Hsin Chung, J. K. Hollingsworth *Using Information from Prior Runs to Improve Automated Tuning Systems.* Proceeding SC '04 Proceedings of the 2004 ACM/IEEE conference on Supercomputing

[20] J. K. Hollingsworth, P. J. Keleher *Prediction and adaptation in Active Harmony.* CLUSTER COMPUTING Volume 2, Number 3, 195-205, DOI: 10.1023/A:1019034926845

[21] F. Chang, V. Karamcheti *Automatic Configuration and Run-time Adaptation of Distributed Applications.* High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on, 2000, pages 11-20

[22] M. Sosonkina *Runtime adaptation of an iterative linear system solution to distributed environments.* In Applied Parallel Computing, PARA 2000. (2001), T. Srevik, F. Manne, R. Moe, and A. H. Gebremedhin., Eds., vol. 1947 of Lecture Notes in Computer Science, Springer-Verlag, pp. 132140.

[23] S. Storie, M. Sosonkina *Packet probing as network load detection for scientific applications at run-time.* Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, April 2004

[24] D. Kulkarni, M. Sosonkina *A framework for integrating network information into distributed iterative solution of sparse linear systems.* In High Performance Computing for Computational Science - VECPAR 2002, Porto, Portugal. (2003), J. M. Palma, J. Dongarra, V. Hernandez, and A. de Sousa., Eds., vol. 2565 of Lecture Notes in Computer Science, Springer-Verlag, pp. 436451.

[25] S. Talamudupula, M. Sosonkina, M. W. Schmidt *Asynchronous invocation of adaptations in electronic structure calculations* Proceedings of the 19th High Performance Computing Symposia, Society for Computer Simulation International San Diego, CA, USA 2011