Parallelizing legacy applications using message passing programming model and the example of MOPAC

by

Tseng-Hui Lin

Abstract of Dissertation

April, 2000

The main purpose of parallel processing technology is to reduce the long execution time problem of big jobs. Many "legacy" programs used today were developed for running on traditional single processor machines only. In addition to parallel programming skill, parallelizing a legacy application requires knowledge in the filed of the application, which is usually difficult to get for computer scientists. The complexity and size make the totally rewriting of these legacy programs a very painful job. Moreover, large amount of code change in a legacy application may in-validate the legacy application. The parallelization should focus on improving the performance while keeping the amount of code change minimized.

We will propose a process to parallelize a legacy application from computer scientist's perspective. This process includes a series of analyses on the legacy application to estimate the performance improvement for different types and sizes of inputs and optimize the parallelized code for maximum performance. The process improve the performance of a legacy application with minimum domain expertise and keep the legacy application certified.

MOPAC is a general purpose semi-empirical molecular orbital package for the study of chemical structures and reactions developed more than thirty years ago. It runs days for molecules consist merely several tens of atoms. We will use MOPAC as the example to express how the process we propose improves the performance of a real legacy application while keeping it validated.

Parallelizing legacy applications using message passing programming model and the example of MOPAC

by

Tseng-Hui Lin
Bachelor of Engineering, National Chiao-Tung University, Taiwan, May 1986
Master of Engineering, National Central University, Taiwan, May 1988
Master of Science, Syracuse University, Syracuse, NY, May 1999

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

April, 2000

Approved ______________________

Date ______________________

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor Professor Geoffery C. Fox for the research directions and the excellent computing environments he gave me. His uninterrupted support helped me stay in research for seven years. I may have given up without his help. He played a parental role in my research.

I would like to thank Dr. Tomasz Haupt. He took care of every detail of my research. He even shared the advisor's role in some projects I was working on in NPAC. Dr. David E. Bernholdt, the most friendly researcher I have ever met, taught me the chemistry knowledge to work on computational chemistry problems. He also helped me debug my huge code.

Professor Danny C. Sorensen kindly taught me some concepts on eigensystem problems. George Fann, the author of PeIGS, showed me much of the internal detail of the optimized parallel eigensolver. They are the best eigensystem experts I could reach on internet.

I also thank my friends Dr. Jhy-Chung Wang, Shennon Shen, Chao-Wei Ou, Steve Cooper, and August Calhoun. They gave me so many valuable suggestions on my research and the writing of this dissertation.

Finally, I thank my family, who were always there whenever I got into trouble, became upset, or lost my confidence. Without their love and encouragement, I would have given up a long time ago.

Chapter 1

Introduction

The requirements for computing power are never satisfied. Although the new semiconductor technology makes computer hardware several times faster every year, software developers and computer users can always find more complex problems to consume the computing power. With higher computing power, scientists can run larger calculations, add more parameters to their models, get more precise results, and improve the quality of their experiments.

Parallel computing has become an important way of achieving higher computing power since the Illiac IV parallel computer became operational in 1975. Parallel processing speeds up the program execution by distributing the computation work load to a set of computation nodes. Each computation node shares a part of the work load. In principle, the more computation nodes are used, the smaller piece of work load is distributed to each computation node; and the shorter the execution time is needed. The most attractive feature of parallel computing technology is that it can be applied to any processor, no matter how fast it is, and boosts the computing power even higher. A hundred-fold or thousand-fold performance boost on large parallel computers is possible for some applications.

The most desirous users of this computing power are computational scientists. It is not unusual for computational scientists to require hundreds of megabytes of memory and weeks of execution time, thus computational scientists have attempted to use parallel computers to solve their problems for years. Moreover, new applications are designed directly for parallel computers to take full advantage of parallel computing technology. Many legacy programs have also been ported to parallel computers. While the performance of some legacy applications does improve dramatically, others may gain very limited or even no performance improvement due to inappropriate approaches, poor tunings, heavy communication or sequential natural of the algorithms.

Legacy applications are programs that were designed decades before parallel computing concepts were introduced. Often over time, new functions may have been added while many changes may not have been precisely well documented. Developers preserve everything they do not fully understand when making changes to legacy applications to prevent breaking the entire program. Programming errors, complicated program codes, obsolete documents, and the growing program size make legacy programs very difficult to deal with.

Some important legacy applications attract many users during their lifetimes. Developers are forced to maintain and improve these applications to satisfy a large population of users. The most common request for improvements are for better user interfaces, new function, fewer limitations, and better performance. All these requirements, of course, require more computing power. Greater performance is what parallel computing technologies were invented for and, thus, parallel computing may be the best solution of improving these legacy applications.

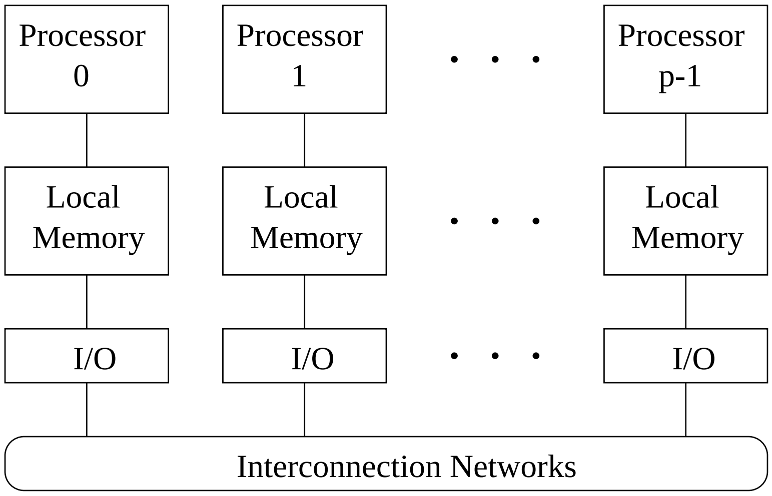
Legacy applications were designed for traditional single processor system environments. Some algorithms optimized for single processor systems may be difficult to parallelize. In contrast, some plain algorithms that perform poorly on single processor systems may scale very well on parallel systems. This means that parallelizing legacy programs involve new concepts which are different from the traditional ones. In short, sometimes we need to parallelize an application with a non-optimized sequential algorithm instead of an optimized one.

One such legacy code is MOPAC, a semi-empirical molecular orbital chemistry package developed thirty years ago [14, 88]; and it surely has evolved over time. Chemists, it is true, have added many new features to make it distinctly more powerful. It has also been ported to many platforms. Many studies were performed utilizing MOPAC. Unfortunately, by the nature of matrix diagonalizations used in semi-empirical quantum chemistry, the execution time required by MOPAC grows roughly cubic of the input molecular sizes. The memory and CPU requirements of MOPAC are beyond the capabilities of most modern workstations for a molecule consisting of only one hundred atoms. In sum, MOPAC is therefore altogether a challenge for parallel processing and constitutes the subject of this dissertation.

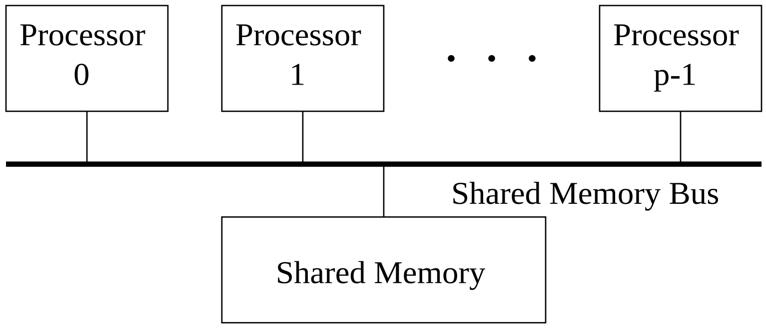
1.1 Parallel Computers

The basic idea of parallel computing is to make a set of processors working on a single problem cooperatively to improve the performance of running a single program or the throughput of multiple programs. In order to work on a single problem cooperatively, the processors must exchange information during program execution. As a result, the processors of a parallel computer must be connected by some kind of communication network.

Based on different types of connections between processors, modern parallel computers can essentially be classified into two basic categories: distributed memory and shared memory parallel computers. Processors in distributed memory parallel computers are connected via an I/O channel while those in shared memory parallel computers are connected via a memory bus [69]. Distributed memory parallel computers use explicit I/O instructions to pass information entirely between processors while shared memory parallel computers directly access all data through a shared memory bus.



(a) distributed memory



(b) shared memory

Figure 1.1: Parallel computer architectures

1.1.1 Distributed Memory Parallel Computers

Distributed memory parallel computers, as shown in Figure 1.1(a), are a set of processors connected to each other via interconnection networks. Not surprisingly, each processor has its own memory. Data sets are distributed and stored in each processor's local memory. There is no direct way to share information stored in the local memory. All information that resides in another processor's local memory must be sent explicitly through the interconnection networks.

Indispensable to a fuller view, the concept of a distributed memory parallel computer is a simple and natural one: adding a communication channel such that a set of independently working computers can cooperate working on a single task to reduce the total execution time. It requires a very small change to the existing computer architecture. In such a case, any kind of interconnection network can be used as long as it can pass information. A specially designed high-speed network is evidently more efficient, but a general purpose local area network (LAN) is effective also. By the type of interconnection network, we can classify distributed memory parallel machines into workstation clusters and massive parallel processors (MPP).

Workstation Cluster A workstation cluster is a set of stand-alone workstations connected by a local area network. To this end, one can simply connect several workstations by a local area network interface which is a built-in device for most modern workstations in order to construct a low-cost distributed memory parallel computer. For example, the Intel 82586 ethernet controller was used as the communication chip of iPSC-1, an early distributed memory parallel computer developed by Intel [69].

By the same token, the local area networks have made significant speed improvements. Fast ethernet is ten times faster than the original 10Mb/sec ethernet at almost the same cost. Gigabit ethernet can do even better [41]. With the dramatic improvement of high-performance network hardware, a low-cost workstation cluster is not necessarily a low-performance cluster [11]. The Beowulf project [81] —run by NASA Center of Excellence in Space Data and Information Sciences (CESDIS)— developed a low-cost parallel workstation cluster for scientific computations. The Beowulf workstation cluster successfully demonstrated high-end parallel computer power from a low-cost workstation cluster in the conference of Super Computing 97. In a word, Beowulf extends the horizon of low-cost workstation clusters. Given these preoccupations, CESDIS has set up a web page [22] to provide software and overall instructions for people to build Beowulf clusters.

Massive Parallel Processors Since all processors may exchange information through the interconnection network, this particular network may become a bottleneck of computation. By and large, specially designed high-speed networks are used for high-end distributed memory parallel computers. These high-end networks not only fundamentally improve the bandwidths but also the latency. Special hardware designs such as worm-hole routing [67] are used to reduce the network routing problem. The Thinking Machine CM5 [90], nCUBE nCube-2 [66], Intel iPSC/860 [55], Intel Touchstone Delta [53], IBM SP2 [58] were all equipped with intensely high-speed interconnection networks.

The interconnection network of Thinking Machine CM5 [90] can invariably perform arithmetic operations while data passes through the network. On the whole, this further improves the global communication operation understandably. The custom-designed routing chips give the mesh topology interconnection network of the Intel Touchstone Delta the ability to deliver several messages on the same link at the same time.

Unlike shared memory parallel computers, distributed memory parallel computers can scale up easily. The interconnection networks can usually connect more computation nodes than shared memory buses do. At a given cost, modulized connection schemes allow the combining of smaller machines into a larger configuration. Thus, large systems can be built by connecting several smaller systems [10, 26, 70]. Distributed memory parallel computers with more than one thousand processing nodes have been seen in the Connection Machine CM-5 [90] and the Intel Paragon [53]. IBM has also built a large SP2 which contains more than 1400 nodes.

1.1.2 Shared Memory Parallel Computers

Shared memory parallel computers, as shown in Figure 1.1(b), have a set of processors connected to a shared memory via a memory bus. Every processor has direct access to the shared memory. Information stored in shared memory is available to all processors from the moment they are loaded into the shared memory. Linked in terms, processors read and write data directly from/to shared memory. No explicit information between processors exchange is needed. As a matter of course, shared memory parallel computers save data distribution and result collection time. As a rule, the direct access of the shared memory creates heavy bus traffic for the shared memory. The shared memory serves all the processors, and it must be extremely fast to keep up with the processors. When the shared memory is not fast enough to bear the heavy traffic, memory access contention occurs and therefore makes memory access the bottleneck of the shared memory parallel computers.

A super-high-speed shared memory which can keep up with the speed of all p processors is the dream of shared memory parallel computer designers. Undeniably, modern processors are so fast that building a shared memory that runs several times faster than processors is almost impossible. Indeed, this limits the maximum number of processors a shared memory parallel computer can have. Although methods of reducing traffic of shared memory accesses have been proposed [60], shared memory traffic is still the most widely acknowledged headache of shared memory parallel computers.

Shared memory with local memory An accurate observation is that the heavy traffic problem of shared memory parallel computers can be reduced by introducing local memory as shown in Figure 1.2. In addition to the shared memory, each processor has a local memory. Only shared data is specifically kept in the shared memory. Non-shared data can supposedly be put in the local memory to reduce the traffic to the shared memory.

Equally important, cache memory is the most commonly used technology to solve the memory-processor speed gap problem on single processor machines [43, 79]. Frequently used information is copied to cache memory. Vitally necessary, cache memory is local to its processor. All data accesses to information that is kept in cache memory requires accesses to cache memory only. No main memory access occurs until a cache miss occurs. This reduces the traffic to main memory dramatically. Cache-hit ratio, which is defined as the ratio of the frequency of cache accesses to total memory

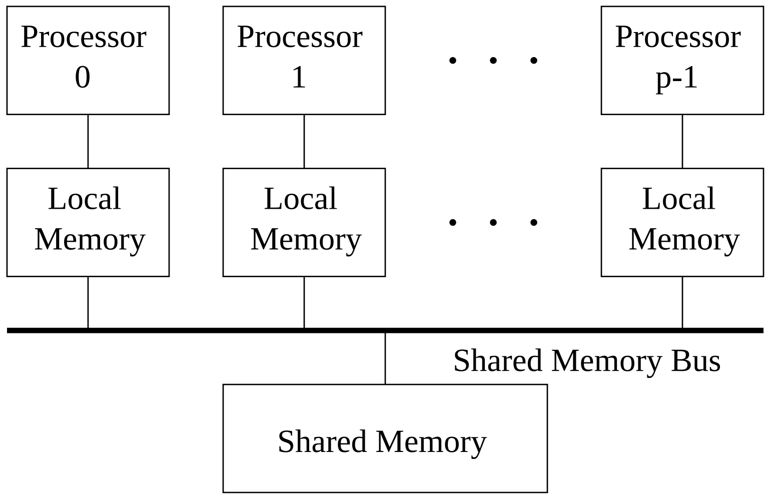


Figure 1.2: Shared memory with local memory

accesses, depends primarily on many factors such as the sizes of cache memories, the cache replacement algorithms, and the data access patterns. A cache hit ratio of 95% reduces the memory access to merely 5%. Multiple levels of cache are also possible for high-speed processors. With its 64KB L1, 2MB L2 cache and enhanced memory bus, the shared memory of SGI Power Challenge can serve up to 36 MIPS R10000 processors without a problem. Although cache memory reduces the memory access traffic problem, it introduces a new problem. In shared memory parallel computers, it is possible that a piece of data is cached in the cache memory of two or more different processors. Cache coherence is required to ensure data consistence between different local copies of data in cache memory of different processors [27].

Some machines put a local memory between cache and shared memory. Read-only information such as code or constants can be duplicated to local memory. Local information, such as working variables, can be kept in local memory, too. The shared memory keeps only information which must be shared and may be altered. This reduces shared memory access even further. Understandably, the local memory serves as an extra level of cache memory for shared memory.

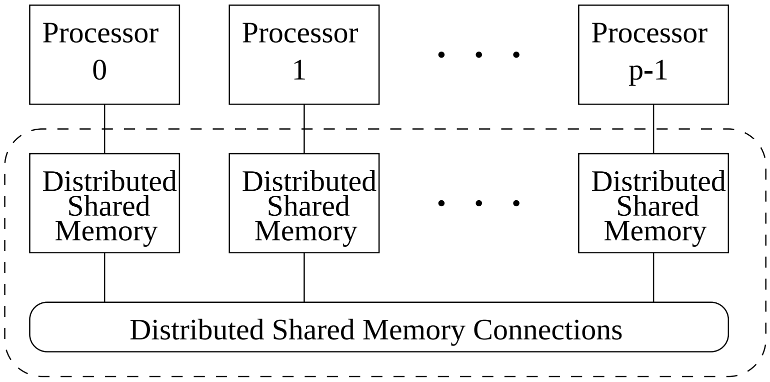


Figure 1.3: Distributed shared memory

Distributed shared memory Although the local memory does reduce the traffic of shared memory dramatically, shared memory still serves only one processor at a time. Vitally necessary, this makes the shared memory the greatest sequential part of shared memory parallel computers. The extensive shared memory access still makes the maximum number of processors of shared memory parallel computers much smaller than that of distributed memory parallel computers.

Since a processor usually accesses a small area of shared memory at a time, it is unreasonable to lock out other processors from accessing other areas of shared memory. Breaking the shared memory into several pieces allows simultaneous accesses to different pieces of shared memories and can improve the performance of the shared memory system. As shown in Figure 1.3, the distributed shared memory is, essentially, the local memory of its processor. In many ways, the distributed shared memories are connected by a special memory interconnection that consists of many memory buses instead of a single bus. The distributed shared memory allows simultaneous accesses of shared memory as long as they are not on the same piece of shared memory. The multiple memory buses design allows all processors not to be attached to the same bus. What emerges is that this distributed shared memory technology not only improves the performance of shared memory but also allows for larger configurations.

SGI Origin 2000 S2MP (Scalable Shared Memory Multi Processor) [78] is a distributed shared memory parallel computer. Instead of a single big shared memory connected on a single memory bus, every prevailing processing node hosts a piece of distributed shared memory connected by a multi-dimensional hypercube shared memory connection. Thanks to the SGI ccNUMA (cache coherent Non-Uniform Memory Access) architecture, the SGI Origin 2000 can connect to as many as 128 processors on a single machine. The Scalable Computing Architecture (SCA) used on the HP 9000 V2600 Enterprise Server [50] is a crossbar topology ccNUMA architecture. The HP 9000 V2600 Enterprise Server at once supports up to 128 PA-8600 processors.

1.1.3 Combined Parallel Architecture

As discussed in previous sub-sections, the interconnection networks used in distributed memory parallel computers give the distributed memory parallel computer better scalability but higher latency, the network initialization time, or set-up time. The shared memory has lower latency but worse scalability. The new design of distributed memory parallel computer moves toward merging the two architectures to minimize the communication latency of distributed memory parallel computers and to improve the scalability of shared memory parallel computers.

Aside from the performance issue, programming models are different between distributed and shared memory parallel computers. By comparing Figure 1.1(a) and Figure 1.3, the architectures of distributed shared memory parallel computers and the distributed memory parallel computers are very similar. The difference between these two architectures is the distributed shared memory connection that enables direct access of off-processor data for distributed shared memory parallel computers while explicit data exchange is required for distributed memory parallel computers.

Approaching both theoretically and practically, SGI Origin 2000 S2MP physically distributes its memory into multiple nodes to improve the scalability while still keeping its appearance of shared memory by using a ccNUMA architecture. In the final analysis, the ccNUMA architecture of SGI Origin 2000 S2MP uses distributed memory technology to improve the scalability while still retaining the advantage of low latency directly shared memory accesses.

On the other hand, the active message [93] implemented on nCUBE nCube-2, Thinking Machine CM5 [91], and IBM SP2 [58] created a low latency method of communication on distributed memory parallel computers. In conjunction with faster communication, active message also provides a way to access off-processor memory directly. With active message, distributed memory parallel computers are able to access off-processor memory more efficiently and run applications written in shared memory model.

In general, since shared memory works more effectively on smaller systems and distributed memory performs better on large configurations, using smaller shared memory parallel computers as the computation nodes to construct a large distributed memory parallel computer is a natural way to take advantage of both technologies. A new IBM SP2 that connected 500 4-way SMP nodes by its high-performance SP switch [84] shrinks the footprint of SP2 while enlarging its computing power.

1.2 Parallel Computing

Parallel computing is used to improve the performance of running a single program or the throughput of multiple programs. The IBM's work introduced in Section 3.3.1 focuses on the throughput of running multiple copies of MOPAC while the SDSC's work introduced in Section 3.3.2 focuses on the performance of running a single MOPAC program. Since this dissertation focuses on the premise of performance improvement of running a single legacy program, our discussion of parallel computing will be limited to the performance improvement of running a single program unless otherwise specified.

A computationally intensive sequential program typically contains one or more iterative loops that consume a large amount of CPU time. To a great extent, the main task in parallel computing is to find and break these loops into smaller pieces and share the work load among all processors in order to shorten the computing time.

As has been frequently pointed out, a program consists of many statement blocks.

The statement blocks of a sequential program are executed in a fixed execution flow. The execution flow decides the order in which the statement blocks are executed. Sequential programmers can arrange the order of statements or use execution flow control statements to control the execution order of statements. The fact of the matter is that by arranging the execution order, it is relatively easy to make the data required by a statement block available before the statement block is executed. In fact, data dependency problem can easily be resolved because the execution flow guarantees the sequence of statement execution.

Parallel computing, virtually, needs to break the sequence in order to make statement blocks execute in parallel. Data dependence essentially may prevent a statement block from being executed before another statement block is finished. This forces statement blocks to be executed in some order and prohibits parallelization. We will discuss more about data dependence in Section 2.1.

It may not be possible to totally eliminate all data dependence in a program. Sequence mechanisms must be used to enforce the execution sequences and follow the data dependencies. Message passing is the most natural way to handle the data dependence problem. The data generating statement block that is executed by one processor sends the data to the receiving processors after the data is generated. This means that the processor that needs the data is blocked until the data is received.

Unquestionably, sending and receiving data is the obvious way of communicating on distributed memory parallel computers. Although the physical size limitation may still limit the overall maximum possible speed-up, locking can be used on shared memory machines to simulate this function. A memorable feature, for the advantage of use in large distributed memory systems and smaller shared memory systems, the message passing programming model is used in this dissertation.

A sequential code to be parallelized can be broken down into the program structure shown in Figure 1.4. A parallel code first determines the data and computational space for each processor. Then the computational space is divided into smaller stages which contain no data dependence in the same stage. The input data is consequently distributed based on the data space of each processor. The processors start computing

---

S1 : Determine data and computational spaces for each processor
S2 : Initial data distribution
S3 : For all data dependent stages do
S4 : For each processor: Loop over its computation space in current stage
S5 : Synchronize execution and exchange intermediate results
S6 : End for
S7 : Collect final results

Figure 1.4: Parallel code structure

concurrently and exchanging intermediate information between nodes. Finally and relatedly, after computing is completed, the final results are sent to their destinations.

For a more strategic coherence, compared with its sequential counterpart, the parallel code shortens the execution time distributing computation to p processors in step S4. Important as it is, the parallel code needs to pay an overhead for step S1, S2, S5, and S7. Parallelization is wasted if the overhead is bigger than the benefit. The key is, namely, to maximize the benefit and minimize the extra cost of parallel algorithms.

The shortest possible execution time by using p identical processors is $1/p$ of single processor execution time. In general, since the final results cannot be ready until all processors finish execution, the total execution time is the execution time of the processor which takes the longest time. Ideally, the work load should be evenly distributed such that all of the processors can finish execution at the same time. In a heterogeneous system, the computing power of each processor must also be taken into account in a heterogeneous systems to achieve load balance.

Step S1 requires a fast work load distributing algorithm. A good load distributing algorithm can distribute workload based on the computing power of each processor and make the execution time about the same for all processors. The complexity of the load distributing algorithm is different from program to program. What emerges is that the difference is not a problem for large problems but may be significant in smaller problems. Although the load distributing algorithm is just a few statements in some simple cases, yet it is typically rather complex. A fast, but not perfect, load distributing algorithm may be better than a perfect, but slow, one. The load balance problem will be appropriately discussed further in Section 2.2.3.

Steps S2, S5, and S7 require communication. In particular, step S2 distributes the input data and arguments to all processors. In many cases, broadcasting and point-to-point communications are the most popular ways to distribute data. Step S7 collects the final results from processors. Collective communications like concatenating or global sum are often used to do this job.

The communication needed for exchanging intermediate data is the most meaningful one since it is performed many times while initial data distribution and final results collection are used for only once. Communication in step S5 is usually point-to-point.

An unscheduled communication phase may result in resource contention and cause unnecessary waiting. A prime example, node 0 and node 1 may want to send data to node 2 and node 3. Node 1 may be blocked because node 2 is receiving data sent from node 0. Node 1 cannot send data to node 2 until node 0 finishes sending and releases node 2. In the meantime, a scheduled communication method will find that node 3 is available and, therefore, has node 1 send data to node 3 while node 2 is not available.

Admittedly, a well scheduled send/receive scheme can fully utilize the communication hardware and cut the communication time by half [61]. A study of scheduling of all-to-all communications to avoid node contentions [74] showed the idea of communication scheduling reduced the communication time on distributed memory parallel machines. A series of studies on communication scheduling for all-to-all and all-to-many communications [72, 71, 73] provide more comprehensive information of communication scheduling. In the final retrospect, a distributed scheduling algorithm [95] eliminated the requirement of exchanging communication pattern information before communication scheduling and reduced the cost of communication scheduling.

As a rule, communication is relatively slow compared to the computation. Processor speed is by far faster than memory and I/O channels. As described in a previous section, cache memory is needed to fill the speed gap between processors and memories. Two levels of cache memory have been used for most modern processors. As the processor clock rates approach the gigahertz range, three levels of cache memory have appeared on some DEC Alpha machines. I/O operations, it appears, that send data to communication hardware are even slower than memory access. The computation and communication performance gap has, one must concede, made communication very expensive.

Since communications can be one or two orders of magnitude slower than computation, communication has a higher priority over computation when choosing algorithms. To be sure, an algorithm that reduces computation but requires more communication may not be as good as an algorithm requires more computation but reduces communication.

Many studies have been done to improve the communication performance on parallel computers. Topology specific studies on a hypercube [17, 61] were made on iPSC/2 and iPSC/860. Algorithms for different applications [33, 51] were designed to take advantage of hypercube network topology.

The communication cost is so significant that it may significantly affect the performance of a parallel program. The communication costs in some parallel algorithms grow as the number of processors increases. This may prevent the algorithms from scaling well. Algorithms with high communication costs may gain very little or even slow down instead of speed-up when too many processors are used. What this suggests is that we will need to determine the best number of processors to use.

We will expand upon communication in Section 2.2.2.

1.3 Legacy Applications

As computers become more affordable, many people join the computer industry to develop software for the vast number of computer users. New software rolls out to replace old products rapidly. Software may be replaced by new ones or become obsolete in a few years. Nonetheless, there are many legacy applications still in use despite their ages. These legacy applications neither get replaced nor abandoned. Legacy applications are exceptions to the rapidly changing software industry.

There must be reasons that potentially make these legacy applications survive. In other words, these legacy applications must have hight value to keep themselves alive. Some possible reasons are:

Functions are validated/certified : Over the long period of lifetime, the bugs and problems of the legacy applications have been found and fixed. The functions of the legacy applications have been proved to be correct and users are confident to use them.

Well known standard : The legacy applications have been widely used and well known. People knows what other people talk about as soon as the names of the applications are mentioned.

Data compatibility : Some legacy applications are used as a part of a sequence of processing. The input and output data formats cannot be changed.

Users get used to it : Users get used to the out look, data format and the way the legacy application handles data. Any change to the legacy application may cause inconvenience to the users. The users prefer keeping the way the legacy application is than making changes to it, even if the changes improve the functionality of the legacy application.

Investment : Users have made big investment in customizing the code, installing peripheral and training their staff to use the legacy application. They want to stick with the legacy application to preserve their investment.

Since these legacy applications are still used, they must be attended to even though their design concepts are old and require an update. The legacy applications to be considered presently have the following characteristics:

Computationally intensive : People would not mind if a program runs for only a few seconds. These small programs do not need to be improved even if they are old. The legacy applications we need to deal with are large legacy applications which require huge amount of CPU time. These programs can take advantage of parallel computing.

Big in size : Small programs can be easily re-written. A small program that contains only a few thousand lines may have been rewritten before they become obsolete.

Sequential design concept : "Legacy applications" in the present context means applications designed before the age of parallel processing. They are designed for uni-processor architecture computers without any parallel computing concepts.

Complicated : The program has been modified/improved by many different people. Program structures, variable names, coding styles are different from one part to the other. Documentation may be missing or not up-to-date. Some optimization may be too tricky to understand. It may take a long time to discover how the code works. This may be the most important reason that people do not want to make changes on these applications.

Improve, not rewrite : People invested a lot of money on the program. It may require even more money to rewrite it. Since there are people still using it and want to improve it, parallelizing it may be a good way to increase the efficiency.

These legacy applications perform functions people need with some limitations due to insufficient computing power. Users may be limited to small size of data, low resolution experiments, or need to wait for a very long response time. Besides waiting for faster hardware, people will apply parallel technology to cure the computing power problem.

Since the legacy applications are designed before parallel processing was born, the algorithms and data structures used in those legacy applications are optimized for uni-processor architecture. One is conscious that some optimizations cause data dependence and make parallelization very difficult.

During the long life of a legacy program, many people enhance the program by adding a new code or changing the existing code. Some of the changes are documented while some of them are not. To avoid breaking something, some unused pieces of codes are kept in the program. These new functions, problem fix patches, undocumented changes and dead code increase the difficulty of tracing and re-writing legacy programs.

The CPU time a program spends is not uniformly distributed in every part of the program. Some parts are executed repeatedly and take most of the CPU time. Other parts are executed only once or even not used in normal cases and take very little CPU time. For instance, most software uses large portions of code to check abnormal situations and error conditions such that the application can produce correct results for the users. This checking usually takes many more program statements since abnormal cases are likely to happen. However, most of the code is made up of "*if-then-else*" structures instead of loop structures. The codes are skipped if no error is found. Even when an error is found and the error handling codes are emphatically invoked to resolve the problem, the codes are run only once and take very little CPU time.

Since we are interested solely in performance improvement, we do not want to touch these seldom used codes. We are more interested in the frequently used data processing parts. These parts may not take as many program statements as the error handling parts do, but they use much CPU time.

It is good practice to divide the whole legacy application into computationally intensive and non-computationally intensive parts. We can leave the non-computationally intensive parts alone and save a lot of work since we are not interested in that segment. The computationally intensive parts, which we are interested in, are now much smaller and easier to deal with. We are first going to extract only the computationally intensive parts from a legacy application, and then parallelize them to boost the performance.

1.3.1 MOPAC

There are essentially two commonly employed theoretical methods for the study of molecules: quantum chemical and classical models of molecular structure. Quantum chemical models can be further divided into two categories: *ab initio* and semi-empirical. Large basis set *ab initio* methods that include correlation offer much better accuracy. It can provide a successful and thoroughly tested framework for molecular calculations. There is, not withstanding, an important drawback of *ab initio* calculations. They require a huge amount of computer resources. In the extreme, a system with just 10 atoms can take hundreds of megabytes of memory and hundreds of hours of CPU time on a workstation. This limits its application, in a sense, to small sized molecular systems. On the other hand, the molecular mechanics method, based on classical mechanical concepts, is extremely fast and requires much less memory than *ab initio* methods. This gives it the ability to handle very large systems. Some molecular mechanics methods can be as accurate as some *ab initio* methods, particularly for hydrocarbons. Still, most classical methods are generally parameterized only for ground state systems and only for common bonding situations. Inevitably they are unable to anticipate unusual bonding situations and the making and breaking of bond, which require a more sophisticated treatment.

Between *ab initio* and molecular mechanics methods are the semi-empirical quantum chemical methods. Like *ab initio* methods, they are quantum-mechanical in nature. However, they greatly simplify the problem in part by substituting empirically obtained parameters for more rigorous and time-consuming computations. Increasing the computing power of a computer by a factor of 100 allows molecular mechanics methods to treat 10 times larger molecular systems, but only 3 times larger for *ab initio* methods. Utilizing today's high-performance scientific workstations, molecular mechanics methods can be applied to thousands of atoms, semi-empirical quantum chemistry to hundreds, and *ab initio* quantum chemistry to tens of atoms [96].

MOPAC is a general-purpose semi-empirical molecular orbital package for the study of chemical structures and reactions [14, 88]. The semi-empirical Hamiltonians MNDO [30], MINDO/3 [15, 28], AM1 [29], and PM3 [83] are used in the electronic part of the calculation to obtain molecular orbitals, the heat of formation, and the derivatives with respect to molecular geometry. Using these results MOPAC can compute the vibrational spectra, thermodynamic quantities, isotopic substitution effects and force constants for molecules, radicals, ions, and polymers. For studying chemical reactions, a transition-state location routine and two transition state optimizing routines are available. A summary indication, for users to get the most out of the program, they must understand how the program works, how to enter data, how to interpret the results, and what to do when things go wrong.

Here is the summary of MOPAC capabilities:

1. MNDO, MINDO/3, AM1, and PM3 Hamiltonians.
2. Restricted Hartree-Fock (RHF) and Unrestricted Hartree-Fock (UHF) methods.
3. Extensive Configuration Interaction
   (a) 100 configurations
   (b) Singlet, Doublets, Triplets, Quartets, Quintets, and Sextets
   (c) Excited states
   (d) Geometry optimizations, etc., on specified states
4. Single SCF calculation
5. Geometry optimization and gradient minimization
6. Transition state location
7. Reaction path calculation including dynamic and intrinsic reaction Coordinate calculations

8. Force constant calculation and normal coordinate analysis
9. Transition dipole calculation
10. Thermodynamic properties calculation
11. Localized orbitals
12. Covalent bond orders
13. Bond analysis into sigma and pi contributions
14. One dimensional polymer calculation

While MOPAC calls upon many concepts in quantum theory and thermodynamics and uses some fairly advanced mathematics, the user primarily needs not be familiar with these specialized topics. MOPAC is written with the non-specialist in mind. The input data is kept as simple as possible so users can give their attention to the chemistry involved and not concern themselves at times with the program itself.

The simplest description of how MOPAC works is that the user creates a data file which describes a molecular system and specifies what kind of calculations and output are desired. A justifiable preference, the user then commands MOPAC to carry out a calculation using that data-file. Finally the user extracts the desired output from the output files created by MOPAC. A sample input file will be shown in Section 3.2.

1.3.2 Data Visualization

Logical arguments can be offered that a computer is a tool which is used to solve real world problems. The results of a computation are numbers that represents real world objects [42]. However various, the real world objects may be locations of planets, velocity of air streams, or the shape of a molecule. Directly printing out these numbers often does not directly give much meaning to a user. Showing a map and the location of hurricanes is much easier for people to understand how hurricanes move rather than just reporting the coordinates of hurricanes. You can more remarkably see how much a stock price rises from a graph instead of a table. It would be very difficult to control air traffic by a set of airplane coordinates. At this juncture, the visual representations allow a quick understanding of the results of computation.

Data visualization has constituted a worthwhile technology for almost all kinds of applications. People used alphabets and punctuation marks to plot tables and draw some low resolution graphs on text terminals before graphic terminals and workstations were invented. Today, with high resolution workstations, people can easily display graphs, charts, and even animations. Pointing devices such as mice, track balls, and touch pads have become important input devices in graphic environments. By moving and clicking on point devices, typing computer commands from keyboards is no longer necessary.

MOPAC focuses on the chemical calculations. All results are represented by numbers and tables. Even the optimized atom geometries are represented by Cartesian coordinates or internal coordinates. A graphic representation of atom geometries thereby helps the study of the geometry optimization process. At this stage, a graphic control panel helps the user input the parameters of calculations by simply clicking on the buttons on the control panel. Data visualization eases the input of parameters and the understanding of the results.

We would like to provide a GUI (graphic user interface) in addition to the traditional text file based MOPAC user interface. To a great extent, the graphic user interface not only provides visualization of molecule structures but also ease the use of parallel MOPAC under parallel environments.

X-windows is considerably the most popular graphic system on workstation environments. Programming by using X-windows directly is a complex job. Higher level GUIs like MOTIF are developed to make graphic programming easier. This means that even with MOTIF, it still needs to add a lot of codes on graphic programming. We would like to use AVS (Application Visualization System) to save efforts required by the graphic programming and focus on parallelizing MOPAC.

For a more strategic coherence, AVS is a visualization system which was designed especially for scientific and engineering communities to analyze and view their data in a real-time interactive fashion. AVS has Image, Geometry, and Graph Viewer subsystems to render impressive visual data on screen. AVS CLI control language can control the viewpoints, angles, and lights projection of objects. By the same token, as many as 16 cameras can be used to view an object. Reduced color is automatic adjusted if the hardware does not have true color. Besides static graphs, animation is also supported by the coroutine module [2]. Using AVS makes the data visualization of parallel MOPAC much easier.

1.4 Outline of this Dissertation

Correctly understood, in Chapter 2, we will describe important parallel processing concepts. What should be sequential and what can be parallelized? Moreover, the advantages and limitations of parallel processing are covered. The factors which limit the performance of parallel processing, mostly related to communications, will be discussed. The data dependence relationships for scalars and arrays in loops are also indicated in this chapter.

More quantum chemistry and MOPAC background will be described in Chapter 3. Prior to making valid conclusions, related MOPAC research will be mentioned in this chapter also.

The procedure to parallelize a legacy application will be proposed in Chapter 4. Closely allied the general procedures of analyzing and parallelizing a legacy application to clearly improve the performance while keeping the legacy application validated are described. The methods that urgently estimate the degree of performance improvement for a certain type of data are described. The methods to tune a parallel program to fully utilize a parallel computer and approach the theoretic upper speed limit are also proposed in this chapter.

In Chapter 5, with unfaltering conviction, we will focus on the implementation and integration of MOPAC. It is used as an example to describe how the procedures described in Chapter 4 are applied to paralleling an actual legacy application.

Of paramount importance, the out look of parallel MOPAC and some benchmark results are shown in Chapter 6. Some discussions are undertaken to explain the result curves and the behavior of parallel MOPAC.

Keeping in mind this spectrum of orientations, the discussion of further improvements and their distinguishing characteristics along with future work as well as some major limitations of today's parallel machines architectures will be demonstrated, finally, in Chapter 7.

Chapter 2

Parallel Environments

In this chapter, the intent is to introduce some background knowledge about message passing parallel computing.

The main goal of parallel processing is to shorten the total execution time. The performance boost of parallelization comes from the fact that many processors share the work load and thus reducing the total execution time. The more processors are used, the smaller piece of work load each processor receives, and the shorter the execution time that is needed. Ideally, we can always use more processors to achieve higher performance. Unfortunately, this is somehow not always possible. In practice, the data dependence forces the execution sequence of related code and, thereby prohibits parallelization. To be properly understood, three kinds of data dependence and how to resolve the dependent relationship between statement blocks will be discussed in Section 2.1.

Parallelization introduces some extra costs, mainly communication overheads. It is efficient to use more processors when the computation time is often much longer than communication time because reducing computation time significantly reduces total execution time. When the number of processors sharing a fixed total work load is large, the computation time, as a rule, may be comparable to the communication time even if the required communication time does not grow as more processors are used. Reducing computation time reduces only part of the total execution time. The performance gain of using more processors becomes negligible after a certain point. In general, the communication costs grow as more processors are used. This limits the maximum possible speed-up of a parallel application. Scalability, the measure of performance gain when more processors are used, is the most important measure of the performance of parallel algorithms.

By largely understanding the factors that reduce the scalability of parallel applications, we can perhaps make some adjustments to avoid the performance degradation and keep better scalability. Even so, the study of scalability can give us the information of maximum possible speed-ups and the most cost efficient number of processors to use.

Communication cost is the most significant factor of the scalability of parallel programs. A coarse grain approach generally requires less communication and fewer synchronizations and leads to better scalability. Conversely, some tightly coupled applications require frequent data exchange and execution synchronization and can use only fine grain parallelization. Certainly, due to the different characteristics and requirements of coarse grain and fine grain parallelism, different considerations for the use of the communication sub-systems of coarse grain and fine grain parallel programs need to be carefully examined. Section 2.4 will give more discussion about coarse and fine grain parallelism.

Parallel computer vendors usually design special communication libraries to fully utilize the communication hardware. After all, using these native communication libraries can achieve the maximum communication performance. Unfortunately, the native communication libraries have different user interfaces. A program written for one parallel computer can not run on another one. This makes migration and porting of parallel applications difficult. Some "standard" communication library user interfaces have been proposed to solve this problem. Using these interfaces may be slightly slower than using the native communication libraries, however, it gives parallel applications portability.

```
INTEGER I
INTEGER A(1000)

DO I = 1,1000
      A(I) = 0
END DO
```

Figure 2.1: Example of parallelizable code

2.1 Data Dependency

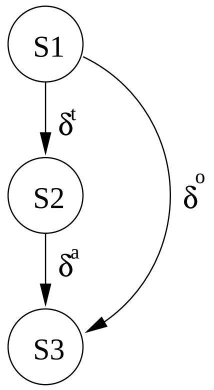
The semantics of sequential languages specifies a linear order on statement execution. Program statements are executed one after another in a predefined fixed order. Sometimes, changing the execution order of some statements may be semantically irrelevant. We can distribute the computation to several computation nodes and combine the results after computing without changing the semantics of the program. For example, we have the FORTRAN code shown in Figure 2.1. Although the execution sequence assigns 0 to $A(1)$ first then $A(2)$, $A(3)$, $\ldots$, $A(1000)$, the semantics will not be changed if we do the assignment in reverse or any different order. In other words, all 1000 iterations can be executed concurrently. In this case, we can easily distribute the 1000 iterations to the computation nodes without any problem.

To be sure, not all loops can be parallelized as easy as the above example. If a loop contains some statements have data dependence relation the iterations of the loop may not be able to execute concurrently. The data dependence relation can be classified as:

- True dependence
- Anti dependence
- Output dependence

S1 : $A = PI$
S2 : $B = 2 * A * C$
S3 : $A = 5 * C + D$

(a) example code



(b) dependence graph

Figure 2.2: A data dependence example and its dependence graph

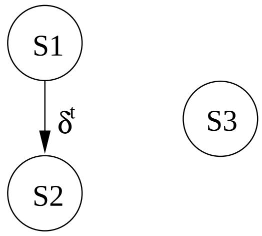
Consider statements S1 and S2 in the example shown in Figure 2.2. The value A used in S2 is given value in S1. Evaluation of the right hand side of S2 is truly dependent on the result of S1. S2 cannot be executed until S1 is executed. Execution of S2 before that of S1 will change the semantics of the program. We call this a *true dependence* from S1 to S2 or S2 is true dependent on S1. Consider statements S2 and S3. The variable A used in S2 is reassigned in S3. Executing S3 before that of S2 will cause the evaluation of the right hand side of S2 to be incorrect. We call this an *anti dependence* from S2 to S3. The relationship between S1 and S3 is called an *output dependence* from S1 to S3 since the two statements both assign value to variable A. The final value of variable A should be the one assigned by S3. An interchange of S1 and S3 will result in the wrong final value for variable A.

The data dependence relationship is directional. A directed graph called dependence graph can be used to represent data dependence. The nodes represent the statements of a set of code and the direct arcs represent the dependence relationship. The symbols δ^t, δ^a, and δ^o denote the true, anti, and output dependence of the arcs. Figure 2.2(b) shows the dependence graph of Figure 2.2(a).

A value must be assigned to a variable before the variable can be used. The variable cannot be reassigned a new value before the previous assigned value has been used. The use and assignment of the same variable in one sense forces a designate

S1	:	$A' = PI$
S2	:	$B = 2 * A' * C$
S3	:	$A = 5 * C + D$

(a) example code



(b) dependence graph

Figure 2.3: Data dependence resolved example and its dependence graph

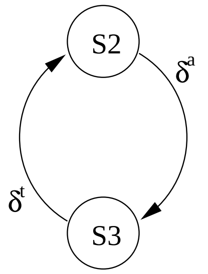
execution sequence between the statement that assigns a value and the statement that uses the value. There are three dependence relationships: true, anti and output dependencies. True dependence expresses a sequential relationship that must be ultimately followed. Anti and output dependencies are caused by the use and re-use of variables. These dependencies can be increasingly eliminated by introducing new variables.

Consider the anti dependence S2 $\xrightarrow{\delta^a}$ S3. The problem that prevents statement S3 from being executed before statement S2 is that S2 will receive an incorrect value of variable A. Since the value of variable A will be decidedly overwritten in statement S3, the problem can be overcome by using a temporary variable A' to replace all instances of variable A before statement S3. The anti dependence between statement S2 and S3, hence, can be resolved.

Consider the output dependence S1 $\xrightarrow{\delta^a}$ S3. The problem that prevents statement S3 from been executed before statement S1 is that the final value of variable A will be reassigned by S1. Since the value produced by statement S1 will only be used by statement S2, again, we can eliminate the output dependence by using a temporary variable A' to replace all instances of variable A between statement S1, the statement that produces the value of variable A, and statement S2, the statement that uses the value of variable A. The new code and its dependence graph is shown in Figures 2.3(a) and (b).

S1 :	DO $I = 2, 100$	
S2 :		$A(I) = B(I) + C(I)$
S3 :		$D(I) = A(I-1) * E(I)$
S4 :	END DO	

(a) example code



(b) dependence graph

Figure 2.4: A loop data dependence example and its dependence graph

The above example illustrates the method of determining and removing the data dependence in a sequence of statements. Since all statements in a loop are repeatedly executed many times, we are more obviously interested in determining if the iterations of a loop can be executed concurrently. Since loops are usually used in conjunction with arrays, we need to analyze thereby the subscripted variables if we want to determine the data dependence between iterations of a loop.

Consider the example in Figure 2.4(a). The loop is executed in the order controlled by the loop index I. The iteration where $I = 2$ is executed first, then the iteration where $I = 3$, $\ldots$, and finally the iteration where $I = 100$. Let $X_{(I=i)}$ denotes the object X at the instance of the iteration that $I = i$. Statement S3$_{(I=3)}$ requires the value $A[I-1]_{(I=3)}$ which is produced in statement S2$_{(I=2)}$ since $A[I]_{(I=2)} = A[I-1]_{(I=3)}$ $= A[2]$. A true dependence S3$_{(I=i)} \xrightarrow{\delta^t}$ S2$_{(I=i-1)}$ for $i = 3, 100$ is therefore found. The iterations of the loop, I, hence cannot be executed concurrently. The dependence graph of Figure 2.4(a) is shown in Figure 2.4(b).

2.2 Scalability

Before talking about the scalability of a parallel application, we need to define the term "speed-up" of a parallel algorithm. "Speed-up" is defined in [3] as :

$$S_p = \frac{T_s}{T_p} \tag{2.2.1}$$

where S_p is the speed-up of an algorithm using p processing nodes, T_s is the time required for the fastest sequential algorithm of the problem, and T_p is the time required for the parallel algorithm using p processing nodes.

In practice, we define the term *speed-up* of a parallel application running on a certain parallel machine using p processors as above equation by replacing the meaning of T_p by the measured run time of the application and T_s by T_1.

As noted in Section 1.2, the basic idea of parallel processing is distributing work load to processors to speed up the execution of programs. Each processor takes only $1/p$ work load when the work load is evenly distributed to p processors. Ideally, without any extra cost, we expect to get $S_p = p$ on a p-processor computer. We call this "linear speed-up" since the speed-up is direct ratio of the number of processors. Linear speed-up is the best scalability. Speed-up grows as the number of processors increases without limitation. A linear speed-up program not only fully utilizes the processing power of all processors but also readily allows unlimited speed-up if you have unlimited number of processors. Linear speed-up is the theoretical maximum that a parallel program can achieve.

As more processors are used, by this means each processor gets a smaller piece of workload. A higher speed-up can be expected if no overhead is introduced. As far as possible, to overcome the data dependence problem and exchange information between processors, some overhead is simply introduced in the process of parallelization. The overhead comes from the following parts:

- Computation for distributing work load.

- Communication for distributing input data sets.

- Communication for information exchange during computations.
- Synchronization of computations.
- Communication for collecting results.

If the overhead is insignificant compared to the total computation, we can get an almost p-fold of speed-up on a p-node parallel machine. Frequently, however, the extra cost grows with p and causes the speed-up to get worse. Even if the overhead does not grow, it becomes more significant and the scalability gets worse when the system size gets large due to the fact that each processor gets smaller piece of workload in larger systems.

Some people do observe "super linear scalability", $S_p > p$, in certain cases. It has originally been traced to cache memory. Cache memory is a fast memory sitting between the CPU and main memory to reduce the speed gap between CPU and memory. Main memory is accessed only when the required data is not already in cache. A "cache-miss" occurs. In any given instance, the memory controller will fetch one cache line from main memory when a cache-miss occurs. If the cache is full, a cache line will be selected and the contents will be flushed to main memory before the cache-missed data can be fetched. CPU needs to wait for a few clocks until the memory controller finishes the cache-miss handling.

In a parallel environment, the data is distributed to p processors when p processors are used. Every processor holds only $\frac{1}{p}$ data. A higher percentage of the distributed data segments can be fit into cache memory and improve the cache-hit rate. The higher cache-hit rate results in a faster average memory access time and shorter execution time. By then, the execution time T_p is shorter than $\frac{T_1}{p}$ and super linear scalability $S_p > p$ can be observed.

Modern computers have several levels of memory hierarchies. Within these confines, the super linear phenomenon happens to memory accesses of all memory hierarchy. The higher level memory is usually faster but smaller. Large program cannot fit into the faster higher level memory and require lower level slower memory accesses.

The lower level slower memory accesses become unnecessary and save a little execution time when the data size becomes small enough to fit into faster higher level memory. Virtual memory is another possible reason to that may promptly cause super linear phenomenon in parallel computing.

Unfortunately, most parallel applications suffer perceptibly from the extra costs of parallelization and have $S_p < p$. We will describe how these extra costs affect the performance of parallel applications in the following subsections.

2.2.1 Non-Parallelizable Part of an Algorithm

Due to the data dependency described in previous section, there are usually some parts of an application that must be executed in some particular order. Data input modules run before processing modules, and most variables must be initialized before they can be used. All data accesses with true dependent relationships must preferably be executed sequentially.

The theoretic maximum speed-up for parallel programs with P parallelizable and $(1 - P)$ non-parallelizable parts using p processors is:

$$S_p = \frac{1}{(1-P) + \frac{P}{p}} \tag{2.2.2}$$

Equation 2.2.2 shows an important concept of non-parallelizable parts of applications: Suppose we have unlimited number of processors ($p \to \infty$), the maximum possible speed-up will be bounded to $\frac{1}{1-P}$. For example, if $P = 90\%$, then the maximum possible speed-up is only 10. A program with a small P value will not be able to take advantage of parallel technology.

Figure 2.5 shows the maximum possible speed-up a parallel application can reach without considering other costs which may be introduced by the parallelization. The slopes of the curves stand for utilization of processors. All curves in Figure 2.5 start with sharp slopes when the number of processors is small. As the number of processors gets larger, the slopes get flatter and maximum possible speed-up does not increase

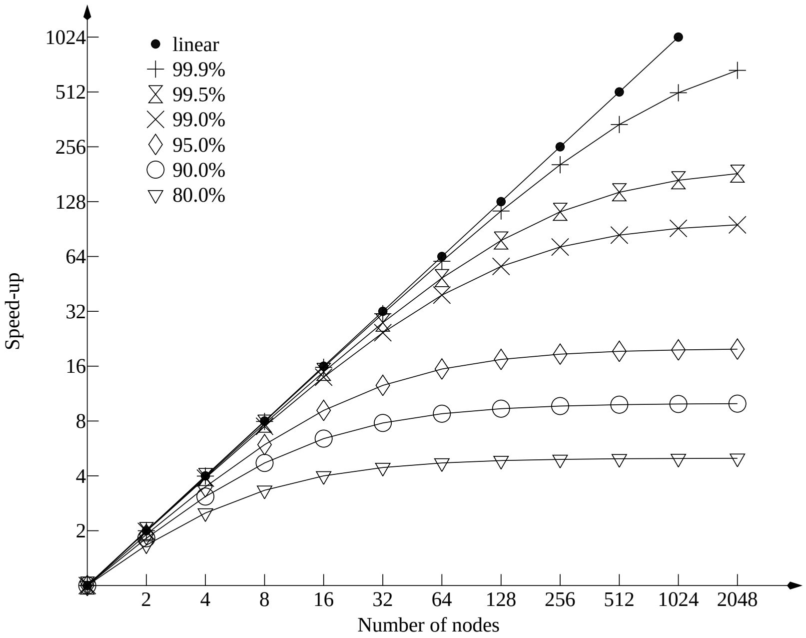


Figure 2.5: Theoretical maximum speed-ups

for more processors. The smaller the parallelizable parts are, the sooner the maximum possible speed-up reaches its saturation point, the point that no significant speed-up can be obtained by adding more processors.

Running a 99.9% parallelizable application on a 2048 processor parallel computer gets 32.8% higher performance than on a 1024 processor one. Putting another 1024 processors on a 1024 processor parallel machine to run a 99% parallelizable application does not get much more of performance gain. Using more than 32 processors to run an 80% parallelizable application is wasting computing resources since the slope of the curve is almost flat after 32 processors. Above all, the more non-parallelizable parts limits the lower maximum possible speed-up and results in a worse scalability.

We can estimate how many times of speed-up a program can get from parallelization by analyzing the parallelizable and non-parallelizable parts of the program. We can also determine the most cost-effective size of the machine to run the application by using the saturation points. The saturation points hence serve as cut points for people who want to find out if it is worth investing more money to add more processors. A time profiling for parallelizable and non-parallelizable parts on a sequential program in conjunction with Figure 2.5 can provide the above useful information before actually parallelizing the program.

2.2.2 Communication Overhead

Communication overhead is an important factor that, in fact, impacts the scalability of a parallel application. Communication overhead depends on the frequency and amount of inter-processor communication. The communication costs vary between programs. The initial data distribution and result collection are the basic communications that are required by all parallel applications. Most parallel programs similarly need to exchange information in the middle of computation. As shown in Figure 1.4, there is a computation phase followed by a communication phase in the loop. The amount of inter-processor communication may be independent of the number of processors. For example, the life problem [6], each processor accesses a clear-cut fixed amount of off-processor data no matter how many processors are used. In other programs, on the other hand, the amount of inter-processor communication increases when more processors are used.

The total communication cost may vary for different systems. In a parallel computer with fully connected interconnection networks, where every processor can send data to any processor without any interference, the total communication cost is identical to that of one processor. Unfortunately, there are few fully connected interconnection networks. Most interconnection networks have some shared links. An off-processor data access from processor A may degenerate the performance or even totally block processor B from accessing off-processor data. Offering further insight, the total communication cost increases by a factor f as the number of processors increases. The total communication cost of an application running on a p-processor system is:

$$c_p = (1 + f\ (p-1))\ c_2 \tag{2.2.3}$$

where c_2 is the fraction of time used for the communication of accessing the off-processor data in a 2-processor system.

The interference factor f can be:

- $f = 0$: for fully connected interconnection networks; absolutely no interference. This kind of interconnection networks can only be seen in small configurations due to the number of links grows rapidly as the number of processors increases. The IBM SP2 high-speed switch can support up to 16 processors fully connected.
- $0 < f < 1$: for interconnection networks with some shared links. Some degree of interference happens when more than one processor injects data into the interconnection network. Extra time is needed to allow the interconnection network to deliver all the messages. Most parallel computers, including both shared and distributed memory parallel machines, fall into this category. Interference occurs when more than one processor is accessing off-processor data and causes an extra f communication cost for each processor.
- $f = 1$: for mutually exclusive used interconnection networks which allow only one processor to send data at a time. The limitation usually comes from the use of one single shared media. Single bus shared memory machines and Token Ring local area network workstation clusters fall into this category.
- $f > 1$: for interconnection networks which have high interference when more than one processor sends data. The interference is so high that p processors sending 1 unit of data each at the same time will take longer than 1 processor sending p units of data. Collision based interconnection networks such as Ethernet require a roll back and re-send when a collision happens. This makes the performance poor for parallel programs that require a lot of communication.

The maximum possible speed-up for a parallel application with a fixed amount of off-processor data on a p-processor system is:

$$S_p = \frac{1}{\frac{1}{p} + (1 + f\ (p-1))\ c_2} \tag{2.2.4}$$

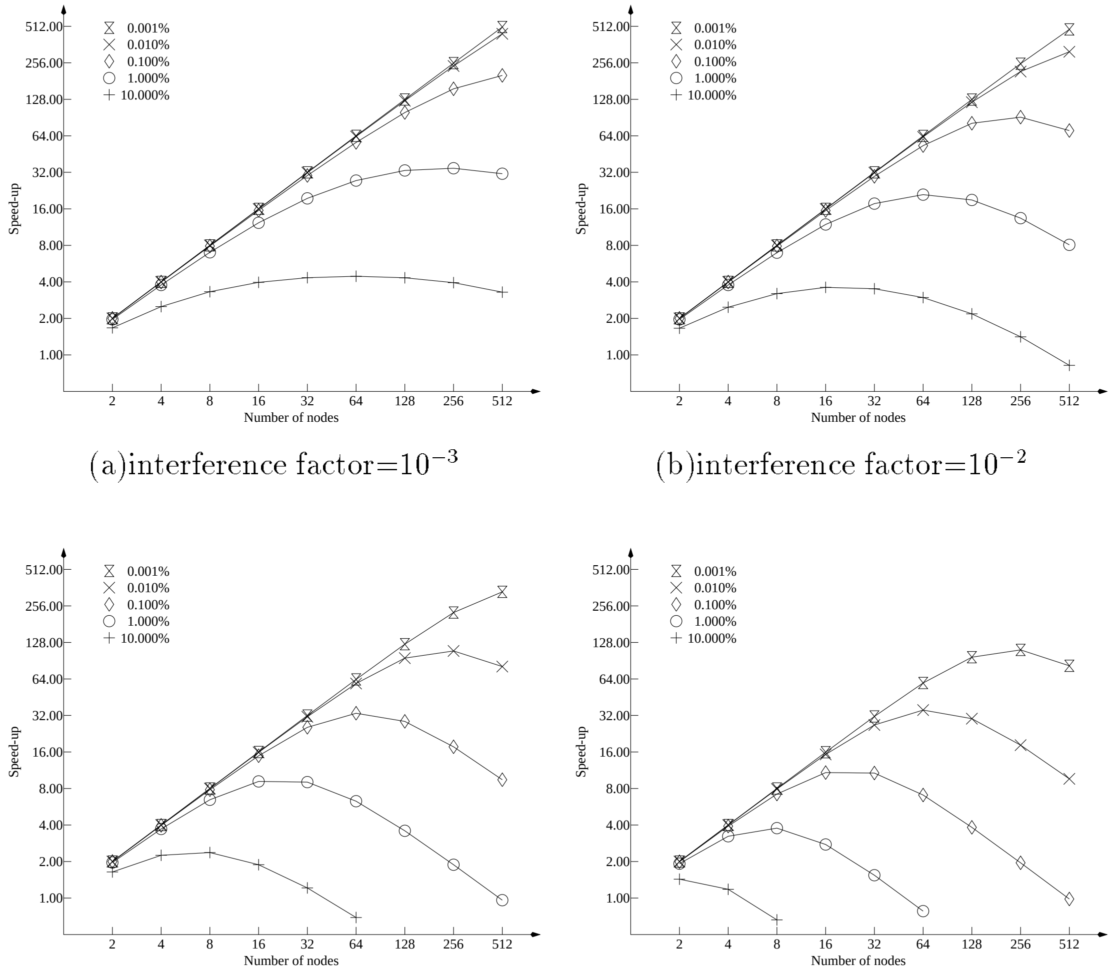
The amount of off-processor data may vary for a different number of processors. In a p-processor system, each processor generally holds only $\frac{1}{p}$ of whole data set. Suppose that each processor in a 2-processor system must access $\frac{1}{2} \times n$ off-processor data, then there will be $\frac{p-1}{p} \times n$ off-processor data accesses for each processor in a p-processor system. The equation 2.2.4 becomes:

$$S_p = \frac{1}{\frac{1}{p} + (1 + f\ (p-1))\ \frac{2\ (p-1)}{p}\ c_2} \tag{2.2.5}$$

Figure 2.6 shows the speed-up curves derived from equation 2.2.5 with communication cost c_2 equal to 0.001%, 0.01%, 0.1%, 1%, and 10% of total computation time and the communication interference factor equal to 0.001, 0.01, 0.1, and 1 of communication cost c_2. The maximum possible speed-up drops sharply as the communication cost rises. The communication interference factor also plays a very important roll in maximum possible speed-up. Even so, unless the communication time is very small, the interference factor will magnify the communication cost and drag down the speed-up when more processors are used.

Note that the interference factor f may vary between different machine sizes. The interference factor f of IBM SP2 is 0 when the machine size is smaller than 16 processors. When the machine size increases to more than 16 processors, a second level of switch is used to connect first level switches. The second level switch is shared between processors and the inter-connection network is no longer fully connected. In this instance, the interference factor f changes to $0 < f < 1$.

Like non-parallelizable parts, communication overhead decreases the scalability of parallel applications. Moreover, the amount and frequency of off-processor access in some parallel applications increase when more processors are used. The interference factor f further magnifies the communication overhead. In the long run, it is advisable not to run parallel applications with heavy communication on parallel machines with large interference factor f.



(a)interference factor=10^{-3}

(b)interference factor=10^{-2}

(c)interference factor=10^{-1}

(d)interference factor=1

Figure 2.6: Estimated maximum speed-up with interfered communications. Curves correspond to fractions of time spent in communications, from 10% to 0.001%.

2.2.3 Load Balance

The equations in previous sections are based on the assumption that perfect distribution schemes are used and the load is evenly shared by all processors such that all processors can finish their computation at the same time. This is the case where there is no CPU idle time. Each processor's computing power is, as a consequence, fully utilized. Even so, sometimes the work load may not be distributed evenly due to the following reasons:

- Non-breakable distribution unit, a segment of code which must be executed by one processor, is too big. The whole work load can be broken into only several parallelizable pieces. Some processors cannot even have one piece of the work load.
- Problem size is too small for the number of processors used.
- The runtime of the parallelized pieces is unpredictable at the time of work load distribution.
- The computing powers of all processors are not identical.
- The sizes of non-breakable distribution units are irregular.

The imbalance load makes some processors finish later than others, leaves other processors idle, reduces the CPU utilization and therefore lowers the scalability.

Besides the above difficulties, sometimes we may not attempt to do perfect load balancing for:

- Perfect distribution scheme entails too much CPU time. A fast but less than perfect distribution scheme is used instead. Fast load balancing algorithms have been studied for many applications [21, 62].
- Sometimes, to avoid communications contentions, the work load may be intentionally distributed in an imbalance manner. The processors which finish

their computation earlier can start exchanging data earlier to avoid the link contention and reduce the communication interference factor. For example, if all processors start the communication phase at the same time on a workstation cluster running over an Ethernet can cause many collisions and re-sends which would degrade performance. A slightly imbalance work load makes processors start their communication at a different time and reduce the collisions and re-sends. This is a useful trade-off for systems with high communication interference factors, especially for those machines with communication interference factor close or greater than 1.0.

2.3 Communication Sub-system Performance

Communication overhead significantly affects the scalability of parallel applications. The performance of the communication sub-system is important to parallel applications with heavy communication. We will discuss the factors that impact the performance of communication sub-systems and some methods that can be used to improve the performance of communication sub-systems.

Communication sub-system performance can be determined by several factors:

- bandwidth: the amount of data a network can send per second after the connection link is established. This is the most commonly seen measuring of interconnection networks. Bandwidth is the most important parameter of interconnection networks. The higher the bandwidth is, the lower the communication time is needed, and the lower the communication overhead will be. Bandwidth is considered "raw power" of a communication channel. It is the most a network can reach. In conjunction with other factors, an application can never observe that high bandwidth. Bandwidth is the major part of the c_2 in the equation 2.2.5 discussed in the previous section.

- latency: the time required from the start of sending a message until the message is actually sent. It is also known as "communication set-up time." Communication latency includes both software and hardware latency. Software latency is the time required from when the system receives the communication calls to the out-going messages which are physically placed into the hardware buffers. It includes the time required for dividing large messages into smaller packets, packaging the data packets, copying data from user buffers to system buffers and initializing the communication device drivers. Hardware latency is the time it takes for the communication hardware to initiate itself and build the connection links between senders and receivers. Latency occurs once per message. For a communication system with long latency, packing small messages into a large message may improve performance a great deal. Long latency will drag down the overall performance if applications require frequent communication. Latency can affect the c_2 in equation 2.2.5 if the data is in small chunks.

- network topology: how processors are connected. Network topology affects the degree of communication interference that occurs when multiple nodes are sending messages at the same time. A fully connected network provides direct links between any two processors. All links are exclusively used by designated processors without sharing with other processors. Any processor can send data to any communication party without waiting for the link becomes available. Most parallel machines do not have fully connected networks and need to share some links sometimes. When a processor is using the shared links, other processors need to wait for the link to become available, which forces all processors send out messages sequentially. Network topology is the main cause of the interference factor f in equation 2.2.5.

- routing scheme: how a message is sent between two non-directly connected nodes. Most interconnection networks are not fully connected. Hardware routing circuits take over the packet store-and-forward operations from host computes. Circuit switching [55] and store-and-forward packet switching were used in older parallel machines. The worm-hole algorithm [67] determines the route as soon as the packet header arrives and does not need to store-and-forward the whole packet. The worm-hole algorithm reduces the routing time to only a few clocks. Above all, worm-hole has become the most widely used routing scheme on distributed memory parallel computers.

Circuit switching allocates all links it requires between the source and destination before a message can be sent. It seriously reduces the utilization of shared links and affects the interference factor f in equation 2.2.5. The routing information added in a packet enlarges the amount of message to be sent. For example, 3 out of 8 words in the message packets of Thinking Machine CM5 are used for routing information. 37.5% of network bandwidth is used for routing. The routing information affects the c_2 in equation 2.2.5.

Communication sub-system performance can be affected by many factors as mentioned above. Unfortunately, only bandwidth can be easily measured with certainty by users. Latency is usually too small to be measured by users. Vendors usually only provide information for hardware latency which is usually much smaller than software latency. The performance impact of network topology and routing scheme depends on the communication patterns and is rather difficult to quantify. Using bandwidth as the only measure can mislead the analysis seriously. CM-5 fat tree interconnection network beats Intel Touch-stone Delta mesh topology in global operations by its network topology. The fact that a 10 Mb ethernet that runs faster than a 4 Mb token ring in a lightly loaded system and slower in a heavily loaded system is another well-known example.

Most communication system parameters like network topology and bandwidth are built into the hardware and cannot be changed. Still, some adjustments to improve communication sub-system performance are still possible:

- avoid node contention: All processors send data to some or all other processors in the communication phase after a computation phase is finished. An unmanaged send may cause more than one processor to send data to the same processor at the same time. In the following example, a program uses a loop for all processors to send messages to all other processors:

```
DO I=0, NPROC − 1
    IF (I ≠ MYID) THEN
        send message to node I
    ENDIF
ENDDO
```

All nodes except for node 0 send data to node 0 at time 0. The data from all other nodes jams the communication channel of node 0. The communication cannot finish until node 0 receive all messages. This problem can be solved by introducing a permutation function $f(i, p, n)$ which generates a permutation of $0, 1, \ldots, p-1, p+1, \ldots, n-1$ for $i = 1, 2, \ldots, n-1$, $p = 0, 1, \ldots, n-1$. The revised algorithm looks like:

```
DO I=0, NPROC − 1
    send message to node f(I, MYID, NPROC)
ENDDO
```

Note that checking if the sender and receiver are not the same is not necessary since we have the permutation not generating p. Suppose we have $f(i, p, n) = (i+p)\ mod\ n$, every node will send to its i-away downstream neighbor at time i. This permutation function assures that every node sends exactly one message and receives exactly one message at all times. The permutation function works if every processor needs to send data to all other processors. Some algorithms that can be used to improve the performance are introduced in [74].

- avoid link contention: Like node contention, it is possible that more than one node may want to send messages through some common communication links. One message may need to wait for another if the common link is not sharable. It is possible to schedule the order of sending such that at any time every non-sharable link is used by at most one node. Above all, this communication scheduling for avoiding link contention is not as straightforward as avoiding node contention since it depends on the network topology. Some studies have been reported in [71, 95].

- user space communication library: The interconnection network used in distributed memory parallel computers is I/O devices. I/O devices are usually controlled by operating systems. User programs need to switch to system mode to perform I/O. Task switching is expensive for every computer since it needs to save all status for the calling user program and restore the status when return. The system needs to check that the arguments the user program passes in are correct and verify that the user program is allowed to perform the I/O requests. A memory copy between user and system buffer may be necessary. These operations cause software latency.

  A user space communication library can send and receive messages in user space. No context switching and access rights verification are necessary since everything is done in the same user process. The data is directly injected to the network hardware without copying to system buffer first. Software latency which is an important limitation for running fine grain parallel programs is, accordingly, reduced.

  Examples of user space communication systems can be found on Thinking Machine CM-5 and IBM SP2.

2.4 Coarse and Fine Grain Parallelization

According to the classification made by Decegama [27], parallelism is loosely classified into three grain sizes: coarse, medium, and fine. Most people refer only to coarse and fine grains since medium grain is just an intermediate between coarse and fine grains.

Coarse grain parallelism involves computational processes at the higher levels of programs. Parallelism at the job or process level is considered coarse grain. Coarse grain parallelism implies large chunks of computations and loosely coupled communications between distributed processing units. In essence, most of today's multiprocessor operating systems are coarse grain. The parallelism in these systems is at job level, where processor handles a set of independent jobs. They communicate to each other only when a system call is made and a system service is requested. The

AVS network [2] (Section B) is also an example of coarse grain parallelism. Remote modules can be distributed to different machines and parallel execution occurs at the process level. With respect to this, coarse grain parallelism introduces the least communications and has a higher CPU utilization.

Fine grain parallelism typically involves computational processes at the inner levels of programs. The unit of work is a statement, a set of statements, or a small loop. Usually, the parallelism is within a subroutine. Clearly, fine grain implies shorter bursts of computation and tightly coupled communications between processors. Fine grain parallelism introduces frequent communications and a high degree of synchronization between processors. Each piece of data in each communication phase may not be very large, but the communication frequency remains high. Since fine grain parallel programs send a lot of small packets, the latency, communication set-up time, becomes an important factor. Communication systems with large set-up time are not useful even if they have high bandwidth. Communication systems with small packets and low set-up time such as Thinking Machines' CM-5 and shared memory parallel computers such as SGI Power Challenge are good platforms for executing fine grain parallel programs.

Medium grain involves computational processes between the above-stated two kinds of parallelism. The unit of work is typically a big loop, a subroutine, or a set of subroutines. Correctly understood, communication requirements are intermediate between the two extremes.

Coarse grain parallelization is not affected by the communication costs very much since the frequency and the amount of data transfer are small. Indeed, because of the low communication cost, coarse grain parallel programs usually have high scalability. Their speed-ups are almost linear in many cases. Unfortunately, not every program can be parallelized in a coarse grain manner. Programs with a high degree of data dependency can only be parallelized in a fine grain manner. Fine grain parallel programs usually have lower scalability due to the high communication cost and may even result in speed-ups $S_p < 1$. When parallelizing a code, it is prudent to choose the method with largest work unit if more than one way is available. It may be advisable to give up on parallelization if the largest work unit we can find is considered too small to gain any performance improvement. Because of these difficulties, most parallel algorithms address medium or coarse grain parallelism only.

In particular, the performance of coarse grain parallelization is not affected by the communication system very much. Due to low communication cost, the overall performance gain from a high-speed interconnection network is negligible for coarse grain parallel programs. Most coarse grain parallel applications do not even need special communication hardware to achieve high performance. To a great extents, running these applications on a workstation cluster can achieve almost the same speed-up as on an expensive parallel computer with a high-speed communication sub-system.

The performance of fine grain parallelization, on the other hand, heavily depends on the performance of the communication system. Fine grain parallelization requires frequent access to interconnection networks which are usually much slower than the processors. Even in a highly optimized massive parallel processors, the interconnection networks are still at least 10 times slower than processors. Besides bandwidth, the latency of interconnection networks is also very important to fine grain parallelization. A communication sub-system with lower latency and higher bandwidth can improve the overall performance dramatically. Judging by this, specially designed hardware and software can be used to improve the performance of fine grain parallel applications.

2.5 Parallel Communication Libraries

As described and illustrated in Figure 1.4 in the previous sections, parallel computers need to exchange information between processors.

- A parallel program needs to distribute data to processing nodes before execution.

- A parallel program needs to exchange results between two computation phases.
- Processing nodes need to synchronize their execution.
- Final results need to be collected after execution.

The above required communication functions can commonly be classified into three main categories:

Point-to-point communications Communications between two processing nodes. It includes the following functions:

Blocking send/receive: communications do not start until both parties are ready. Program execution is blocked until the communication is complete.

Non-blocking send/receive: The sender returns as soon as the data is copied to system buffer. Data may be sent out later. The receiver returns with the data it receives if the data has been in or an error code if the data is not received yet. No program execution is blocked.

Asynchronous send/receive: A status block is returned as soon as an asynchronous send is requested. Data stays in the user buffer without being sent at the time of return. User programs need to check if the data is in user buffer, system buffer, network, or receiver through the status block. The same status block is returned for message status checking on the receiver side.

Status block manipulation: Methods for message status checking and status blocks removing for asynchronous send/receive.

Active message: A small amount for data associated with an action to be performed when the message arrives at the receiver.

Collective communication Communications among more than two nodes. It includes the following functions:

Synchronization: Waiting for other processors to reach the synchronization point.

Broadcast: Sending data to all processors.

Global reduction: For a p-processor parallel machine, commutative operator $\circ$, $i = 0, 1, \ldots, p-1$, processor i sends quantity q_i and receives the results of $q_0 \circ q_1 \circ \ldots \circ q_{p-1}$. The commonly used commutative operators $\circ$ for global reduction operations include add, multiply, minimum, maximum, logical and, inclusive or, and exclusive or.

Scan: also known as a parallel prefix operation. It works similar to global reduction except that processor i receives the result of $q_0 \circ q_1 \circ \ldots \circ q_{i-1}$ where $\circ$ is any commutative operator that can be used in global reduction.

Concatenation: Appending values from each processor in the order of node number and sending to all processors.

Parallel Environment Status and Control Parallel environment information retrieval and parallel system configuration. This catagory includes the following functions:

System information: Returns system information such as the system names, number of processors, my node ID, computation power, and interconnection network topology.

Signaling: Send/receive signals between nodes.

Data conversion: Data format conversion in heterogeneous system configurations.

I/O: File, screen I/O, and I/O redirections.

Timing: Time service for benchmarks.

Debugging: Debugging information.

Massively Parallel Processors Communication Libraries Every parallel machine has at least one set of communication libraries to support part or all of the above functions. Most parallel computers, in fact, have special communication hardware to minimize the communication latency and maximize the communication bandwidth.

For example, the CM5 uses a fat tree network topology to speed up global operations. The Intel Paragon has a custom designed routing chip to handle the traffic in its mesh network topology. The manufacturers also build their specific communication library to fully utilize their hardware [53, 54, 66, 89]. For instance, CM5's fat tree network can do scalar mathematics when data is passing through the network. The result of a global reduction is produced as soon as the data propagates to the root of tree. Its global reduction functions are implemented to take one scalar at a time. The Intel i860 microprocessor used in the iPSC/860 and the Paragon has a CPU built-in vector processor. Sending data to one CPU and using vector units to perform array mathematics are a more efficient means. Its global reduction functions are implemented to handle an array of scalars at a time.

Since these native communication libraries are specially designed to fully utilize the hardware, they are usually known to be the fastest communication libraries people can use on a given machine [51, 52]. Increasingly, differences between these native communication libraries inhibit the porting of parallel programs. It stands to reason that the idea of defining a standard communication interface was proposed to solve this problem. Communication libraries such as Chameleon [46], Express [68], P4 [19, 20], and PVM [40, 59, 86], usually implemented on the top of the native communication libraries, were popular packages intended to address portability concerns.

Workstation Cluster Communication Libraries Workstation clusters are usually connected by LAN. Although they do not have specially designed communication hardware as a sustained source, they still can run parallel programs as long as they can communicate with each other. Most workstations run at least one of the TCP/IP, XNS, SNA, or NetBIOS protocols [25, 80, 82]. Besides supporting NFS, E-mail, printer service, and other network applications, these network protocols can also serve as communication libraries for workstation cluster parallel computers. Like MPPs, workstation clusters can utilize standard communication libraries built on the top of the LAN protocols. Chameleon, Express, P4, and PVM all support workstation clusters running over LAN protocols.

As network performance improves over time, workstation cluster LANs run much faster than their predecessors. 100Mb/sec token ring, fast Ethernet, and ATM have blurred the border between workstation clusters and massively parallel processors. Gigabit ethernet runs even faster than some specially designed interconnection networks on massive parallel processors [41]. In contrast, some companies implement LAN protocols over the interconnection networks of their massive parallel processors, so that heavily used network activities such as NFS and database accesses can run more efficiently. Software latency often remains an issue, nevertheless, for example, IBM implements the IP protocol over the SP2 high-speed switch, therefore, an SP2 looks like a workstation cluster in many ways. The performance gap between massive parallel processors and workstation clusters may totally disappear in the near future.

Standard Communication Library MPI The standard communication libraries are designed to make porting of applications easier. Application program writers do not need to worry about porting programs as long as the target machine has the same standard communication library installed. By all accounts, the rarity of standard communication libraries raises the porting problem again.

The Message Passing Interface (MPI) Forum, which involved most major vendors of parallel machines and researchers from universities, government laboratories, and industry, designed the standardized communication library [65]. The preliminary draft proposal was put forward by Dongarra et al. in December 1992 whereas a revised version was completed in February 1993.

The MPI Forum defined the MPI standard but left the implementation for the vendors. IBM and SGI both have their own MPI implementations [57, 78]. In many instances, research laboratories and universities also implemented different MPIs. Edinburgh Parallel Computing Center's CHIMP [4], Ohio Supercomputer Center's LAM [18], Argonne National Laboratory and Mississippi State University's mpich [45], and Mississippi State University's UNIFY [92] are examples of the more popular ones. Unlike the manufacturer implementations, these MPI implementations support more than one platform. Moreover, because these MPI implementations work across platforms, they not only make parallel programs portable to different machines but also make heterogeneous parallel machines possible.

These MPI implementations pay only a small performance penalty for their portability. The Argonne National Laboratory shows their mpich performs almost as fast as the native communication library on an IBM SP-1 [44].

Irregular Communication Libraries Besides the basic regular pattern communication functions, some applications have irregular data accesses and require irregular pattern communication libraries to handle irregular data accesses for them [77]. For example, in a 2-processor machine, processor 0 needs to access data item 1, 5, 0 while processor 1 needs to access data item 4, 8, 3, 2. The access pattern is totally random and can only be specified by an index array rather than a formula. Hence the data distribution needs to pack off-processor data in some ways such that both data access and communication is efficient [75]. Irregular communication libraries are designed to specifically meet these requirements. University of Maryland's PARTI was the first irregular communication library [76, 87]. With substantial adeptness, Syracuse University's NICE improves the communication performance by integrating communication scheduling schemes [94].

Chapter 3

Quantum Chemistry and MOPAC Background

Using computer programs to derive and analyze chemical properties respectively has become an important tool of study in theoretical chemistry. Solving the Schrödinger equation which describes the relationship between molecular structure and energy is the basis of theoretical chemistry. Hartree-Fock (HF) theory, as an immediate task, provides a successful and throughly tested framework for molecular calculation. The rigorous *ab initio* method calculates the molecular system using all basis functions and provides accuracy results. Still, the memory capacity and computing power are limited resources. With the high complexity of calculation, the maximum molecule size modern computers can handle is limited in about 100 basis functions [8]. Molecular Mechanics methods replace the complicated calculation with empirical results to simplify the calculation. These methods sacrifice the generality and accuracy for speed. These methods can easily handle molecular structures consisting of thousands of molecules on modern computers. Semi-empirical methods are intermediate to *ab initio* and Molecular Mechanics methods. Semi-empirical methods are quantum mechanical like *ab initio* methods with more approximations based on experimental data.

MOPAC is a general-purpose semi-empirical molecular orbital package developed several decades ago. It is a commonly used utility for studying chemical structures and reactions. It was submitted to Quantum Chemistry Program Exchange (QCPE) for distribution in 1985. Because MOPAC is powerful, easy to get, and work on many platforms, there is, by and large, a huge population of chemists using MOPAC for their studies and research.

Due to the fact that semi-empirical methods use an eigensolver in their calculations, the computational complexity of MOPAC is $O(N^3)$ or higher. It requires large amounts of memory and days of CPU time for a molecule with less than 100 atoms. Even though MOPAC can handle larger molecules, the long execution time and large memory requirements prevent users from taking advantage of this.

Add to the foregoing that parallel computing is a good solution for computationally intensive applications like MOPAC. Unfortunately, decades of changes and improvements not only make MOPAC a large program but also add to the difficulty in tracing the program. It is a challenging task to understand every detail inside the MOPAC code on any level. A full parallelization may not be reasonable due to the high level of programming effort that would be required. At the same time, the amount of effort which has gone into the development of MOPAC itself over the years distinctly makes it unreasonable for most researchers to consider a complete rewrite to obtain parallelization either.

Fortunately, the heart of the computational problem in MOPAC involves matrix diagonalization. We can, therefore, parallelize the legacy application MOPAC by extracting the older sequential diagonalizer and replacing it with a more powerful parallel engine for a more successful ascertainment.

In this chapter, a background in quantum chemistry and the MOPAC package is presented. Some related research at IBM, San Diego Supercomputing Center (SDSC) and Fujitsu company is also described in this chapter for a fuller analysis.

3.1 Molecular Orbital Methods

Theoretical chemistry has been used to study chemical properties for some time. Theoretical chemistry is very useful for transient species that occur in combustion, in interstellar space or as proposed reaction intermediates. Theoretical calculations can be the only way to obtain the data for these transient species. Theoretical calculations can also be used to study the chemical properties and screen out some unnecessary experiments to reduce the cost and time required for new products such as drugs.

Statistical mechanics deals with large numbers of molecules. This allows the prediction of properties like pressure, free energy, enthalpy, etc., and of course, provides the theoretical framework for macroscopic experiments at finite temperatures. For a more detailed scrutiny, to actually carry out such predictions from first principles requires a knowledge of the individual electronic, vibrational and rotational energy levels of the component molecules, which can be obtained from quantum chemistry.

3.1.1 Hartree-Fock Self-Consistent Field Theory

Quantum chemistry and quantum chemical concepts have had an enormous impact on chemistry. Many experimental studies also report quantum calculations. Quantum chemistry has clearly passed beyond the hands of the theorist and has become yet another tool for the experimentalist to interpret and understand his data.

The electronic wavefunction $\psi(1, 2, \ldots, n)$ describes the n electrons in a molecule. $\mid \psi(1, 2, \ldots, n) \mid^2$ gives the electronic probability distribution. The wavefunction is a solution of the the Schrödinger equation, $\hat{H}\psi = E\psi$ where $\hat{H}$ is the Hamiltonian operator and E is the energy. Directly solving the eigenvalue problem is rather difficult. One initial approximation to $\psi(1, 2, \ldots, n)$ is a product of molecular orbitals (MOs), $\{\phi_i\}$, each representing a single electron, so that

$$\psi(1, 2, \ldots, n) \sim \phi_1(1)\phi_2(2)\ldots\phi_n(n) \tag{3.1.1}$$

An *ab initio* (from first principles) theory with optimum molecular orbitals constitutes the Hartree-Fock Self Consistent Field (HF-SCF) solution.

Based on chemical principles, these MOs are often expanded in a basis of functions associated with the individual atoms of the molecules known as atomic orbitals (AOs), $\{\chi_\mu\}$. When an MO is formed from a combination of AO's, we need to adjust the proportions of the AOs. The MO $\{\phi_i\}$ in equation 3.1.1 can thus be described as $\phi_i = \sum_\mu c_{\mu i}\chi_\mu$. The coefficients $c_{\mu i}$ are the weight of χ_μ and can be determined from the Schrödinger equation. This is called the Linear Combination AO (LCAO) approximation.

Calculating the energy of an n-electron system requires calculating the energy of all combinations of elections which leads to $n!$ computational cost. Instead of calculating the energy of each electron pairs, we can simplify the calculation by calculating the energy of each electron and the average energy of the rest of electrons. When inserting equation 3.1.1 into the Schrödinger equation, the electronic Hamiltonian for all n-electrons is replaced by the sum of n one-electron effective Hamiltonians, $\hat{f}$, whose eigenfunctions are the MO's.

$$\hat{f}\phi_i = \varepsilon_i \phi_i \tag{3.1.2}$$

Equation 3.1.2 can be represented in the matrix equation form

$$FC = SC\varepsilon \tag{3.1.3}$$

where S is overlap of AO basis functions, C is the eigenvectors of the matrix equation and ε is the eigenvalues.

Because $\hat{f}$ depends on the electronic charge density and the density must be consistent with $\hat{f}$, the final wavefunction must be "self-consistent." We can build $\hat{f}$ by specifying the AO's $\{\chi_\mu\}$ and solving equation 3.1.2. The matrix representation Fock matrix $F_{\mu\nu}$ can be derived by:

$$\begin{aligned} F_{\mu\nu} &= H^{core}_{\mu\nu} + \sum_a^{\frac{n}{2}} \sum_{\lambda\sigma} C_{\lambda a} C^*_{\sigma a}[2(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma)] \\ &= H^{core}_{\mu\nu} + \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\lambda|\nu\sigma)] \end{aligned} \tag{3.1.4}$$

where H^{core} is the core-Hamiltonian matrix, C is the expansion coefficient and P is the density matrix.

S1 : Calculate molecular integrals required in equation 3.1.3 and 3.1.4.

S2 : Guess initial eigenvector C.

S3 : Use C to compute Fock-matrix F.

S4 : Transform Fock-matrix F to orthogonal basis and diagonalize it to get a new C.

S5 : Compare the new and original C to see if C has converged.

S6 : If not converged, guess next C based on the new and original C and goto S3.

Figure 3.1: SCF procedure

For a more strategic coherence, the whole SCF procedure is shown in Figure 3.1. Steps S1, S3 and S4 require extensive calculations. The calculation of the integrals in S1 requires $O(n^4)$ computational time, as does their use in S3. The diagonalization in S4 requires $O(n^3)$ computational complexity.

The fact that charged electrons avoid each other instantaneously lowers the energy of the many-body system more than an "average" SCF treatment of electron repulsion can. The correlation we ignored in equation 3.1.2 can be significant. Constructing a Configuration Interaction (CI) wavefunction by replacing some occupied orbitals by some of the unoccupied orbital functions can improve the product approximation.

3.1.2 Semi-empirical SCF

Ab initio methods calculate the SCF approximation rigorously. But full inclusion of a large set of basis functions and rigorous calculations introduce huge numbers of calculations which require $O(n^4)$ computation time [9].

Instead of introducing a large set of AOs and doing everything rigorously, semi-empirical methods like CNDO, INDO, MNDO, MINDO, etc. simplify the SCF calculations by using some carefully selected pieces of experimental information. The most computationally intensive part of the SCF calculations are steps S1 and S3 in

Figure 3.1. Most semi-empirical methods use the following methods to reduce the computation requirement:

- Find fast ways to compute the integrals. The most commonly used method is to substitute the integrals with empirical results.
- Ignore some insignificant integrals.
- Assume AO basis has already orthogonalized. i.e. let $S = 1$ in equation 3.1.3.

The computational complexity of steps S1 and S3 in SCF calculations can be reduced to $O(n^2)$ in most semi-empirical methods. This makes the diagonalization in step S4 the most computationally intensive part of the SCF calculations. The computational complexity of the diagonalization is $O(n^3)$. The overall computational complexity of SCF calculation in semi-empirical methods is, as a consequence, reduced from $O(n^4)$ to $O(n^3)$.

Since the experimental data may include some electron correlation effects, semi-empirical methods account for some electron correlation effects in addition to the faster calculation. However, unlike *ab initio* methods, most semi-empirical methods cannot be systematically improved upon because the use of experimental data incorporates some electron correlation effects; but henceforth the degree of incorporation is uncertain.

3.2 MOPAC

MOPAC is a general-purpose semi-empirical molecular orbital computer software package for the study of chemical structures and reactions [14, 88]. The semi-empirical Hamiltonians MNDO [30], MINDO/3 [15, 28], AM1 [29], and PM3 [83] are used in the electronic part of the calculation to obtain molecular orbitals, the heat of formation, and the derivative with respect to the molecular geometry. Using these results MOPAC calculates the vibrational spectra, thermodynamic quantities, isotopic substitution effects and force constants for molecules, radicals, ions, and polymers. For studying chemical reactions, a transition state location routine and two transition state optimizing routines are available. For users to get the most out of the program, they must understand how the program works, how to enter data, how to interpret the results, and what to do when things go wrong regarding unforeseen aspects.

While MOPAC calls upon many concepts in quantum theory and thermodynamics and uses some fairly advanced mathematics, the users are not required to be familiar with these specialized topics. MOPAC is written with non-specialist is mind. The input data is kept as simple as possible so users can give their attention to the chemistry involved.

The main developing of MOPAC is under James J. P. Stewart at Frank J. Seiler Research Laboratory, U.S. Air Force Academy. Several important MOPAC releases are described in Section 3.2.1. The most recent version is MOPAC 7. Ever since the main developer, James, J. P. Stewart became associated with the Fujitsu company, Fujitsu company have worked on commercial versions of MOPAC. The commercial versions MOPAC93, MOPAC97 and MOPAC2000 as well as the Microsoft Windows port, WinMOPAC, will be described in Section 3.3.3.

The version of MOPAC in our study is the sixth edition. MOPAC 6 contains about 30,000 lines of FORTRAN code in 160 subroutines and 20 functions. It runs well on CDC, Data General, DEC-3100, Gould, and Digital computers. It also works on CDC 205 and Cray X-MP mainframe computers. The Cray version has been partly optimized to take advantage of the Cray architecture. Some versions have also been ported to personal computers. The MOPAC used in our study is the UNIX port for the SunOS.

3.2.1 MOPAC releases

In 1985 MOPAC 3.0 was submitted to the QCPE for distribution. Since 1985, MOPAC development has been focused on providing a highly robust program. Reliability that gives precise answers, as well as portability which benefits users on different platforms, are the main concerns in the subsequent versions. MOPAC 3.1 was ported to more platforms including VAX 11, IBM PC and Cray X-MP machines. The vector operation of Cray X-MP machines show large performance improvement over regular sequential machines. According to the MOPAC manual, the Cray X-MP/48 runs about 60 times faster than the VAX 11/780 for small jobs (less than about 40 atoms) and about 300 times faster for large jobs (more than about 130 atoms).

IBM PC introduced a low-cost computation platform to scientists. Although the computation power of early IBM PCs was not big enough to handle large molecules, many people had ported MOPAC 4 to IBM PCs to take advantage of the low-cost platform.

- Santiago Olivella at University de Barcelona, Spain used IBM Professional FORTRAN-77 (PROFORT), version 1.00 to port MOPAC 4 on IBM PC running MS-DOS versions 2.1. In order to work on IBM PCs with only 512 KB of RAM, this porting of MOPAC 4 was broken up into four modules. The largest molecule which can be dealt with is 7 heavy atoms and 7 light atoms.

- Norman E. Heimer, Jon T. Swanson and James J. P. Stewart at Frank J. Seiler Research Laboratory, U.S. Air Force Academy ported MOPAC 4 to IBM PC without splitting MOPAC into smaller modules. The largest molecule which can be dealt with by this version is 8 heavy and 8 light atoms.

- Jon T. Swanson, Terri A. Miller and James J. P. Stewart at Frank J. Seiler Research Laboratory, U.S. Air Force Academy modified the above port to use either the Ryan-McFarland or IBM Professional FORTRAN compilers.

- Jon T. Swanson, Herbert E. Klei and James J. P. Stewart at Frank J. Seiler Research Laboratory, U.S. Air Force Academy have another 32-bit port of MOPAC. This port requires a special 32-bit 032/PC board. It is capable of handling up to 38 heavy atoms and 38 light atoms.

- G. Kapsomenos at Aristotle University of Thessaloniki, Greece ported MOPAC 4 to IBM PS/2. With 4 MB of memory, this version can handle 40 heavy and 40 light atoms and also performs C.I. calculations with 8 orbitals.

MOPAC 5 incorporates all the previously available Hamiltonians, namely, AM1, MINDO/3, and MNDO. In addition there is also incorporated the MNDO-PM3 Hamiltonian. The input file is unchanged. The output is also modified to include several changes recommended by users. This causes the output not to be compatible with other software which reads MOPAC output.

In addition to tradition platforms, MOPAC 5 is also ported to IBM 3090 with vector facility. The IBM's ESSL is extensively used in this port.

Under these circumstances, after Version 5.0 of MOPAC was published, several users made available software which they had developed or modified to run on MOPAC. Most of these modifications have been incorporated into MOPAC Version 6.0 and Version 6.1. The most important of these are:

- A geometry optimizer, G-DIIS, written by Csaszar and Pulay.
- The electrostatic potential fitting routine of Besler, Merz and Kollman, as implemented in MOPAC 5.0 Electrostatic Potential, has been added.
- The eigenvector following routine of Baker, Jensen, Rzepa and Stebbings has been added.

The entire MOPAC 6 is written in FORTRAN 77 except for CPU time and date routines. This makes MOPAC 6 easy porting to other machines.

The new MS-DOS based MOPAC 6 port is divided into 3 modules. With the improved Windows operating system, this version can handle 60 heavy and 60 light atoms with 16 MB of memory.

MOPAC 7, the most recent MOPAC development release, has been ported to more UNIX platforms as well as Microsoft Windows systems.

3.2.2 MOPAC Input File and Keywords

The simplest description of how MOPAC works is that the user creates a data file which describes a molecular system and specifies what kind of calculations and output are desired. To this end, the user then commands MOPAC to carry out the calculation using that data file. Finally the user extracts the desired output on the system from the output files created by MOPAC. There are at most four possible types of data files attainable for MOPAC, but the simplest data file is the most commonly used.

Fig 3.2 shows a MOPAC sample input data file. As can be seen, the first three lines are textual. The first line consists of keywords. There are seven keywords shown in this sample. These keywords control the behavior of MOPAC. The keyword directs MOPAC to "do an unrestricted Hartree-Fock calculation", "use Pulay's converger to obtain an SCF", "use the MINDO/3 Hamiltonian", "print final eigenvectors", "print final density matrix", "print localized orbitals", and "write restart file every 300 seconds." MOPAC checks the input keywords before execution. If two keywords which are incompatible, e.g. UHF and CI, are supplied then error trapping will occur and an error message will be printed. MOPAC recognizes more than 130 keywords.

The next two lines are comments or titles. They will be printed in the corresponding output file. Users can use these lines to put the name of the molecule or the purpose of the data file.

Lines 41 to 46 define the geometry of molecular structure. Note that the geometry is defined in internal coordinates format or Gaussian Z-matrix instead of Cartesian coordinates. Each line represents one atom. For each atom a_i, the internal coordinates consists of an inter-atomic distance in Angstroms from a previous atom a_j, an inter-atomic angle in degrees between atom a_i and a_j and a previous atom a_k, and finally a torsional angle in degrees between atoms a_i, a_j, a_k and a previous atom a_l. Line 44 specifies a hydrogen is located in the space that is 1.098326Å from the atom defined in Line 42, with an inter-atomic angle 123.572063° between the atom defined in Line 41, and a dihedral of 180° with the atoms defined in Lines 42, 41, and 43.

Line 5 should be a blank line or 0 to terminate the geometry definition.

```
Line 1  :    UHF PULAY MINDO3 VECTORS DENSITY LOCAL T=300
Line 2  :    EXAMPLE OF DATA FOR MOPAC
Line 3  :      MINDO/3 UHF CLOSED-SHELL D2D ETHYLENE
Line 41 :  C
Line 42 :  C    1.400118  1
Line 43 :  H    1.098326  1  123.572063  1
Line 44 :  H    1.098326  1  123.572063  1  180.000000  0   2  1  3
Line 45 :  H    1.098326  1  123.572063  1   90.000000  0   1  2  3
Line 46 :  H    1.098326  1  123.572063  1  270.000000  0   1  2  3
Line 5  :
```

Figure 3.2: A MOPAC sample input data file

3.3 Related Work

In this section, we will is essence briefly introduce MOPAC works done in other organizations. The MOPAC parallelization developed by IBM focuses on improving the throughput of MOPAC. Kim Baldridge at SDSC parallelized MOPAC to improve its performance. The MOPAC2000 released by Fujitsu company is a commercial product integrating the new patented MOZYME algorithms to calculate large molecules of up to 10,000 atoms and MOS-F V4.0 to calculate UV-visible spectra. The Windows port of MOPAC, WinMOPAC V2.0, adds graphic user interface to MOPAC.

3.3.1 IBM

In reference to this criterion, Brian T. Luke at International Business Machines Corporation was also working on parallelizing MOPAC. Both IBM's and our work is based on MOPAC version 6. We call our work "parallel MOPAC" while IBM people call their work "MOPAC-7."

Our work on parallel MOPAC focuses on reducing the runtime of a single large job (improving performance), while IBM's approach assumes that users will perform many closely related calculations (such as changing bond angles), and simply provides a mechanism to spawn multiple single-processor jobs on the nodes of a parallel machine (improving throughput). Parallel MOPAC focuses on improving the performance of a single run of MOPAC. That is, we are trying to reduce the time of a single MOPAC run from, say, hours to minutes. We analyze the MOPAC code to find the most computationally intensive portions and use parallelizing technology to speed up the computationally intensive portions of MOPAC in order to reduce the total execution time.

IBM MOPAC-7 takes a different approach. They assume that chemists mainly use MOPAC on conformational search, i.e. running MOPAC many times with slightly modified input files. For example, someone may want to run MOPAC to see the heat difference for the skew of three atoms from 30° to 40°. They may need to repeatedly run MOPAC for 11 times with slightly different data files, which differ from its neighbor only in the skew of that three atoms for 1°. IBM exploits the coarse grained parallelism in this kind of application: the 11 runs can execute in parallel.

IBM MOPAC-7 introduces new control keywords to MOPAC's input file that specifies what and how input data changes. In our example, the skew is specified as a range from 30° to 40° with 1° difference instead of a constant. MOPAC-7 analyzes the new data file and finds out how many runs are needed. It then generates MOPAC 6 format data files. In above example, MOPAC-7 knows there will be 11 runs and generates 11 data files with the skew 30° for the first data file, 31° for the second data file, ..., and 40° for the last data file. It then starts up 11 MOPAC processes on 11 IBM SP2 nodes and feeds the 11 MOPAC processes with the 11 data files. The 11 MOPAC processes do not need to communicate with each other until they finish running. MOPAC-7 then collects the result files from the SP2 nodes for users.

Figure 3.3 shows the IBM MOPAC-7 parallelism. Users specify the data range by the new IBM MOPAC-7 keywords in the MOPAC input file. IBM MOPAC-7 then generates the input file sets and distributes the data files as well as MOPAC program

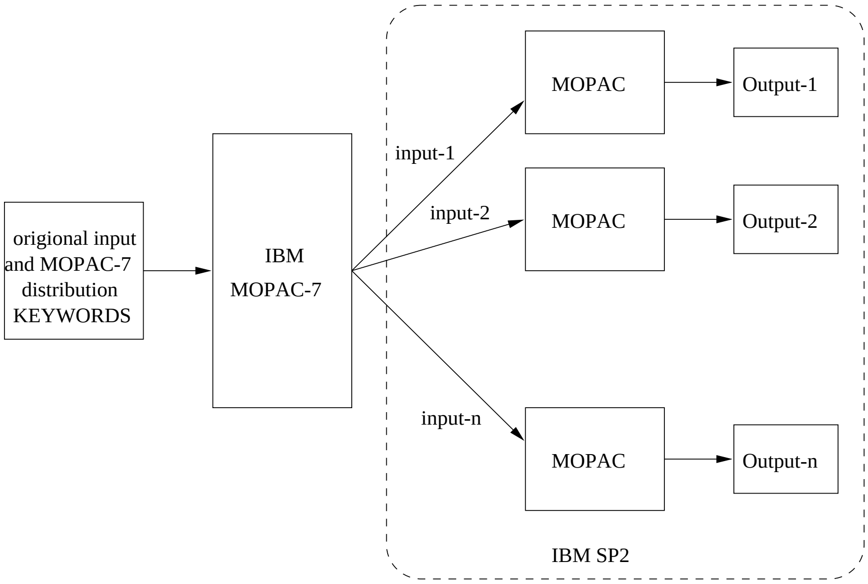


Figure 3.3: IBM MOPAC-7 parallelism

to SP2 nodes. IBM MOPAC-7 distributes work load at the job level, which is the coarsest grain MOPAC has. The communication is not an issue since there is no communication during execution time except for the data file distribution and result file collection. Moreover, since the calculations are very similar, the execution time of each job is about the same. Insofar as activity, load balancing is, thus, achieved. This approach yields nearly linear speed-up.

IBM MOPAC-7 works excellently if the users use MOPAC in the above manner. When this becomes clear, it will not benefit those who run MOPAC with a set of different input data files or a single large input data.

3.3.2 SDSC

As this work was proceeding, a similar effort was undertaken by Kim Baldridge at the San Diego Supercomputer Center also tried to implement a parallel version of MOPAC [7, 8]. Her work focused on improving the performance of the following four parts:

- evaluating the electronic repulsion integrals
- calculating first derivatives
- calculating second derivatives
- solving the resulting eigensystem

Her approach is very similar to ours. One distinction is that she implemented her work from a chemist's point of view while we did it from a computer scientist's point of view. She knows the above 4 parts are computationally intensive based on the functions of the 4 parts and her chemistry expertise. We find out these computationally intensive parts through time profiling analysis in Section 4.1.2. She used time profiling to demonstrate that these parts are really the time-consuming parts of MOPAC. She also made computational complexity analysis for these parts. Of vital importance, she did not know that the work load distribution can be changed dramatically for large molecule systems. Section 4.1.3 shows how we use computational complexity analysis with time profiling analysis to determine the time-consuming parts.

The parallel algorithms were implemented on a 64 processor Intel iPSC/860 hypercube, and subsequently on an Intel Paragon. Calculations were performed on node combinations up to 128 processors. Optimization level 3, which incorporates global optimization and software pipelining, was invoked during code compilation. There is no data dependency analysis and communication optimization discussion in her paper.

The results are categorized in two categories: geometry optimization and vibrational analysis. The speed-up of geometry optimization was disappointing. The best result is 5.2 times faster by using 64 nodes. The speed-up of vibrational analysis was better. A 40.8 times speed-up is observed using 64 nodes.

The problem of the poor performance of geometry optimization is because of the memory limitation of iPSC/860. iPSC/860 nodes have 8 MB real memory and no virtual memory. The operating system and communication library take about 3 MB of memory and at best leave about 5 MB for application code and data. The replicated data parallel decomposition used in this implementation requires large amount of memory. Only molecules with less than 20 heavy atoms can run on an iPSC/860. Above all, the results on Paragon are similar except for Paragon has faster CPU and larger memory (32 MB/node).

Drawn into a closer collaboration, the other problem is the communication. There are a lot of small data packets exchanged in the eigensolver in geometry optimization. They were implemented by using fast-library global routines. This was observed to be extremely expensive.

Global communications are expensive unless hardware supports them. iPSC/860 and Paragon both use mesh topology. Global communications are implemented by point-to-point send and receive operations. All nodes sending data to communication networks at the same time may congest the network links and reduce the communication performance. In accordance with this, using global communications to replace regular send and receive communication simplifies the programming but increases the communication cost and should be avoided if not supported by hardware.

3.3.3 Fujitsu

As Dr. James J. P. Stewart, the main developer of MOPAC, began as a consultant with Fujitsu company in 1991, Fujitsu company has been working on commercializing MOPAC. Unlike the research branch, the commercial branch of MOPAC is named after the year of release. The first commercial release of MOPAC is MOPAC93, followed by MOPAC97 and MOPAC2000. MOPAC93 was forked from research release MOPAC 7. A Microsoft Windows port of MOPAC, WinMOPAC V2.0, is also released by Fujitsu.

MOPAC2000

MOPAC2000 [38] was released in March 1999. The most important new feature of MOPAC2000 is its new MOZYME algorithm. Memory size and CPU time requirements limited conventional quantum mechanics packages to a few hundred atoms. The patented MOZYME algorithms enables biochemists and materials chemists to readily take advantage of the powerful property prediction capabilities of quantum mechanics on systems of over ten thousand atoms. MOPAC2000 can perform optimizations orders of magnitude faster, using dramatically less memory than conventional semiempirical and *ab initio* packages. By this trend, the electronic properties of systems with thousands of atoms, including proteins, polymers, semiconductors and crystals, can now be calculated in just minutes or hours. With MOZYME algorithm, MOPAC2000 allows chemists to perform very fast quantum mechanics computations on macromolecular systems such as proteins, polymers and crystals.

MOZYME algorithm reduces the scaling by reformulating the problem in such a way that the Fock matrix construction becomes linear and the diagonalization step is avoided. This dramatically reduces the time required for SCF calculations. However, it is not applicable to every molecular system and all the properties provided by the traditional SCF calculations. The traditional SCF calculations are still needed for those systems.

MOPAC2000 also incorporates over thirty other new capabilities and features, including d-orbitals and metals, crystal properties, geometry in electric fields, NLO, 2D and 3D periodic boundaries, band structures, photon spectra, Young's modulus, tensile strength, MOS-F for UV spectra, inter-system crossing, excited states in solution, etc.

MOPAC was written in FORTRAN. The required memory space was statistically allocated. The maximum array and matrix sizes defined in file SIZES must be changed and the whole MOPAC source needs to be re-compile to change the maximum array and matrix sizes. The dynamic memory allocation introduced in MOPAC97 eliminates this problem and make more efficient memory usage.

MOPAC includes the semiempirical Hamiltonians, MNDO, MINDO/3, AM1, PM3 and MNDO-d calculation methods. These methods have been calibrated using experimental data for thermodynamic properties such as heats of formation.

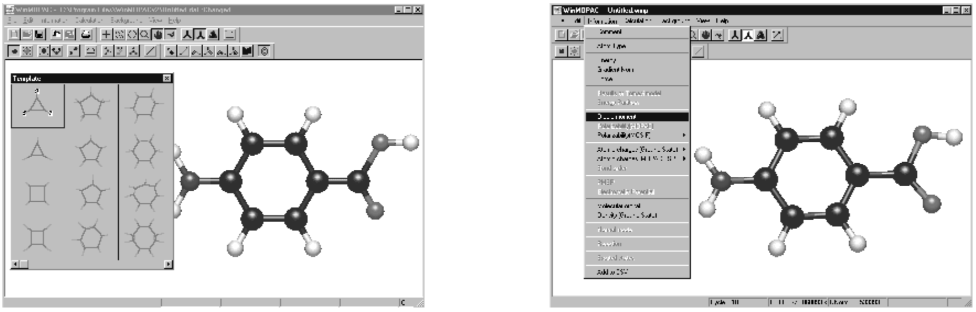
Besides the traditional UNIX workstations, MOPAC2000 is also ported to parallel computers using distributed memory message passing programming model and MPI communication library. Fujitsu AP3000, VX/VPP300/VPP700, SGI Power Challenge and Sun Ultra Enterprise 10000 are the first parallel machines MOPAC2000 was ported to at the time of release. No detailed description was published at the time MOPAC2000 was released.

WinMOPAC

To support the popularly used Microsoft Windows platforms, Fujitsu developed WinMOPAC based on MOPAC97. WinMOPAC V2.0 [39] is a 32-bit Microsoft Windows application running on Windows 95, Windows 98 and Windows NT 4.0 released in August 1998.

The most important new feature of WinMOPAC V2.0 is its graphic user interface. WinMOPAC V2.0 supports a nice 3D graphic user interface. Besides traditional text input files, it also allow input molecule structures using graphic user interface. Its 3D structure building function allows the user to create or change the bond lengths, angles and dihedral angles of molecule structures by dragging and clicking on a mouse. It also provides about 100 templates to help users to build molecules easily. Figure 3.4, downloaded from [39], shows the molecule sketching and property calculating using the WinMOPAC V2.0 graphic user interface.

WinMOPAC V2.0 also includes MOS-F, a semi-empirical molecular orbital calculation program, which was developed and has been used for material design by Fujitsu Laboratories Ltd. The main function of the MOS-F program is to assist in the screening and molecular design of organic dyes and related light-sensitive materials, especially organic nonlinear optical materials, by using the INDO/S method. The MOS-F program can calculate over 1,000 atoms of large molecules because of



(a) Sketch the molecule (b) Calculate the property

Figure 3.4: WinMOPAC V2.0 graphic user interface

efficient memory usage.

WinMOPAC V2.0 supports AM1, PM3, MNDO, MNDO-d, MINDO/3, CNDO/2, CNDO/S, CNDO/S2, CNDO/3 and INDO/S computional methods. It requires Pentium 100MHz or higher with 32 MB or more RAM. In such matters, the maximum size of calculation is limited to 100 hydrogen atoms and 100 heavy atoms due to the memory and computing power limitations at the time of release.

Chapter 4

Parallelizing a Legacy Application

The purpose of parallelization is to decrease the total execution time constrained in a single program or increase throughput of multiple processes. Since we focus on improving the performance of legacy applications, how to reduce the execution time is the main topic of the analysis.

A legacy application, as stated in Section 1.3, is usually large and complex. A complete re-write of the program takes enormous effort. Moreover, the new code may introduce new bugs and invalidate the legacy application. Fortunately, the computationally intensive part is usually concentrated in a small part of the program. In general, in scientific codes, we expect that most of the CPU time will be spent in various loops within the code. Parallelizing these parts results in approximately the same performance boost as totally parallelizing the entire program but takes a much lesser effort. Moreover, most code is not changed and kept validated.

Since we are interested in revealing small portions of a legacy application instead of totally rewriting it, we need to find out which parts take the most CPU time. We apply the parallelization technology to the time-consuming parts to reduce the total execution time but leave the other parts unchanged as illustrated in Figure 4.1. Only small computationally intensive parts are parallelized and run on a parallel computer. Most parts of a legacy application are not changed and run on a workstation or one

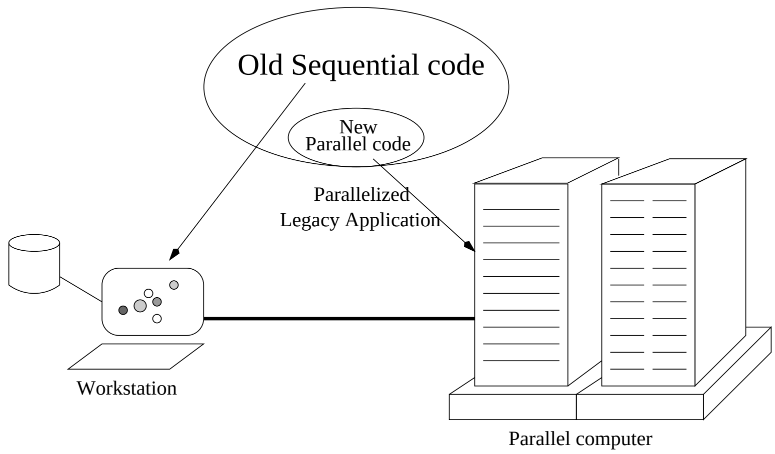


Figure 4.1: Parallelizing a legacy application

of the nodes of the parallel machine. The effort is minimized while the performance is improved and, the most important, the legacy application is still validated.

In this chapter, we will give the procedures used for parallelizing a legacy application in the model described above. A sequence of sequential analysis, including program flow analysis, time profiling analysis and complexity analysis, for parallelizing a legacy application are used to identify the time-consuming parts from the legacy application efficiently. Before we start parallelizing the time-consuming code, we need to perform a data dependency analysis to ensure the code can be parallelized without altering the semantics of the code. Data dependency can prevent programs from being parallelized. We need to make sure that there is no data dependency before beginning to parallelize a segment of code to ensure the program integrity. Some data dependency relationships can be resolved by code rearrangement or transformation. Amdahl's law described in Chapter 2 states that the non-parallelable parts can seriously limit the maximum possible speed-up of a parallel program. We need to resolve all resolvable data dependency to maximize the parallelable portion of the legacy code to improve the overall performance of the legacy program. Data dependency analysis and some commonly used loop parallelization optimization methods are described in Section 4.2 and Section 4.3. At last, integration issues are discussed in Section 4.4.

4.1 Sequential Analysis

A segment of time-consuming code can be as small as a few statements or may span several subroutines. The first step in analyzing a legacy application for parallelization is to browse through the code to get an idea about the program execution flow and which subroutines are used together.

Time profiling is the second step of analyzing a legacy application. We know the program structure from the program flow analysis. We may be able to identify which subroutines serve as data input, which handle error checking and recovering, which do results output, and which perform the real computation. Program flow analysis is important to time profiling because some code may be complicated and consume CPU time, but they may be used only occasionally or even not used at all in normal cases. For example, some code may be used to handle special cases. These special cases may be complex and require a significant amount of computation. However they are not used very often. Program flow analysis can identify these code and keep them out of time profiling. These subroutines need not be parallelized unless other more frequently used subroutines have been parallelized. We cannot find out which part is frequently used unless we feed some typical test data to the legacy application and measure the CPU time these subroutines use.

Time profiling analysis widely shows the time distribution of a legacy application for input data. However, the work load distribution can be changed for different sizes of input data. A complexity analysis is useful to find out how the CPU time distribution changes when larger data files are used. We can estimate the speed-ups for different sizes of input data files by combining the results of time profiling and complexity analysis.

4.1.1 Program Flow Analysis

A program usually consists of many subroutines and functions. Each subroutine usually performs one or more tasks. Subroutines performing large tasks may divide the tasks into smaller sub-tasks and call other subroutines to perform these sub-tasks. A subroutine is a good starting point for analyzing program flows. A flow diagram can show the inevitable relationships between subroutines. We can easily find out which subroutines are together and what functions they perform.

Legacy applications, however, are usually large and complex. Precisely finding the function of every subroutine is very time-consuming and requires expertise in the field the legacy application is designed for. It is difficult for a computer scientist to do these kinds of analysis without years of training.

Fortunately, the information we need form program flow analysis is to identify time-consuming parts. Not knowing the functions of most parts is not a problem since we do not change them at all. We only need to determine if a subroutine is used for normal data processing, which is used regularly, or abnormal data and error handling, which is seldom used in normal cases. This can be done with basic knowledge.

In general, in scientific codes, most of the CPU time will be spent in various loops. It is easy for computer scientists to identify loops. A loop can be in several forms:

deterministic loop : The number of iterations can be determined when the loop starts. These are typically, but not necessarily expressed using language constructs such as *DO* loops in FORTRAN and *for* loops in C. A loop index variable serves as the iteration counter of the loop. The number of iterations is fixed and can be calculated from the loop parameters before the loop is executed.

indeterministic loop : The number of iterations cannot be determined when the loop starts. The loop keeps iterating until some conditions are satisfied. Since we have no way of knowing in advance when the conditions will be satisfied, we do not know how many iterations an indeterminate loop will run. *REPEAT* and *WHILE* loops in C belong to this category.

Backward goto : Programs are executed statement by statement. A backward *GOTO* can change the execution flow may be used to construct a loop, especially in FORTRAN codes.

Recursive : The code itself does not contain the loop structures specified above. However, it still can form a loop by calling itself. A recursive subroutine does not necessarily directly call itself. It can call other subroutines and then call back to itself from these subroutines to form a recursive execution sequence indirectly. Recursive subroutines are more difficult to identify and parallelize than other types of loop. Fortunately, earlier FORTRAN languages, which most scientific legacy applications were written in, do not allow recursion.

Program initialization, data input and result output are used only once in the beginning or end of program execution. Abnormal case and error handling routines are seldom used in normal cases. They are not eligible for parallelization even if they appear to have high computational complexity and be time-consuming. We need to exclude them from the rest of analysis to avoid accidentally picking them up in time profiling analysis.

4.1.2 Time Profiling Analysis

A program flow analysis shows the structure of a legacy program. We can intentionally learn the various functions of the subroutines. The flow diagrams also give us some rough ideas about the execution time of each segment. At another point, the time required for a program segment does not solely depend on the complexity of the code. The frequency of calls also plays a very important role in CPU time consumption. Unfortunately, the behavior of program execution depends heavily on the input data. Different sets of input data may use different parts of a program. At this stage, we do not know the actual execution time and the frequency that these subroutines are called from flow diagram. Time profiling is the only way to know how much CPU time a segment of code needs to run a set of input data.

Care must be taken when doing time profiling analysis. The input data must be carefully chosen and the time measurement cannot interfere the regular execution. The most common way to do time profiling analysis is to write time stamps at the start and end point of the code we are interested in. The execution time of each code segment can therefore be calculated from these time stamps. As a consequence, the writing of the time stamps should be as fast as possible to reduce the interference of time measurement.

Time stamps are kept in a large memory buffer. Memory access is fast and the access time is uniform. We can ignore it or measure and subtract it from the results. However, in many cases, time stamps take large space. They need to be written to disk files. The time used in writing data to a disk file is clearly quite un-predictable. It may be very fast if the data goes to the file buffer only. It may take longer if the file buffer is full and needs to physically write the file buffer to a hard disk. The time required to write a file buffer to hard disk depends on the position of the hard disk head and the location of the data on the disk platter. The time may vary from nano-seconds to a few hundred milli-seconds in normal operation. In some very special cases, e.g. disk head re-calibration, it may take up to several seconds for very large disks. We can neither ignore it, since it may be very long, nor measure it, since it varies in large range.

Since UNIX systems count resource usage by process, each process has a set of counters to count its resource usages. The best way to do time profiling is to use separate processes. As shown in Figure 4.2(a), the results of time profiling includes the timing and tracing and data logging and analysis modules. The timing and tracing module reads system time and increases some tracing counters. These operations are fast and do not vary very much. The data logging and analysis module, which involves disk accesses, can be slow and non-uniform. Dual process time profiling, as shown in Figure 4.2(b), moves the data logging and analysis module to a separate process. Instead of disk files, the time stamps and trace counters are sent to the the data logging and analysis module through faster UNIX Inter Process Communication (IPC), e.g. shared memory. Data is written to a memory buffer, whose access time is fast and uniform. When the memory buffer is full, the context switches to the data logging and analysis module. The timers pause when the context switches out of time profiling process stops and resume when the context switches back. Notably, the operating system takes care of the accounting. The time used by the data logging

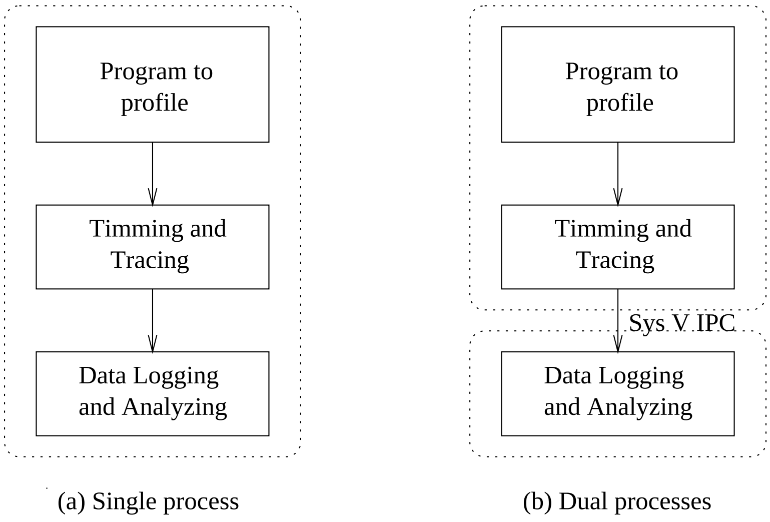


Figure 4.2: Single and dual process time profiling

and analysis module is not counted since it is in different process. The interference introduced by the time stamp writing is minimized.

The behavior of a program is heavily dependent upon the input data. Evidently, giving unusual input data will mislead the analysis. For example, we may observe certain routines are used very frequently and take a lot of CPU time if the input data we use contains many unusual situations. A set of "typical" input data should be carefully chosen for the time profiling.

The more test data we run, the better accuracy with which the time distribution can be measured. It is good to run as many test data as possible to reduce the bias caused by the choice of input data. Moreover, the results of running the test data files need not carry the same weights. By contrast, some test data may be more "typical" and carry more weight than the others. Weights should be added to the results of the time profiling to reflect the types of data that these test data files represent. A larger job should carry more weight too.

Time profiling analysis shows the time distribution of a legacy application. We can easily find out which parts consume most of CPU time. Moreover, we can also estimate how much speed-up we can gain from parallelizing these segments of codes for the test data. For example, we find a legacy application spends 99% of CPU time in a data processing subroutine. Even with unlimited number of processors and perfect parallelization, the speed-up of the legacy application will not exceed 100. Actually, by looking at Figure 2.5, we find the maximum possible speed-up does not indeed increase when the number of processors is larger than 256. A more cost-effective cut point would be 128 processors.

4.1.3 Complexity Analysis

Time profiling analysis finds the relatively time-consuming parts for certain test data sets. By carefully choosing different types of data files, we can throughly test all major execution paths of a legacy application and find out the work load patterns. However, beside of the types of data files, the size of data sets changes the work load patterns also.

As a set-piece, we usually use smaller data at the analysis stage to simplify the analysis. In production, on the other hand, the input data files are usually much larger. The work load patterns found in time profiling analysis may not reflect the real work load patterns in production. An adjustment on data size needs to be applied to the results of time profiling analysis to correctly show the work load patterns of large data used in production.

Computational complexity affects the CPU time required for a subroutine when the input data size changes. Codes with higher computational complexity gain higher work load than those with lower computational complexity does when the data size gets larger. Computational complexity is originally the function of the change of CPU time required and the change of input data size. We can find the change of CPU time of each parts of a legacy application from the change of data size.

For example, a legacy application spends 50% of CPU time in subroutine A with computational complexity $O(N^3)$ and the rest 50% of CPU time on subroutine B with computational complexity $O(N)$. Likewise, when a 10 times larger data file is given, subroutine A needs 1000 times of CPU time while subroutine B needs only 10 times.

The work load pattern is changed to 99% and 1% for subroutine A and B respectively. Subroutine A will be more time-consuming than subroutine B in production.

The work load change caused by data size change not only changes the time-consuming degrees of various subroutines but also changes the maximum possible speed-ups. For example, if we have a program that contains 2 parts. Part A has computational complexity $O(n)$ while part B has $O(n^2)$. The time profiling on a data of size $n = 100$ gets the result that part A takes 1 second and part B takes 9 seconds. The time-consuming part B takes 90% of CPU time and has a maximum possible speed-up of 10. Now we run a larger data of size $n = 1000$. Part A takes 10 seconds and part B takes 900 seconds. This time part B takes 98.9% and raises the performance ceiling to a factor of 91.

By combining time profiling and computational complexity analysis, we can better estimate the work load patterns of a legacy application for any data size of every type of input data. We can find which parts of a legacy application are the most time-consuming in production. These parts are utterly the targets of parallelization.

Beside of identifying the targets of parallelization, by using Figure 2.5, we can also find the maximum possible speed-up of a legacy application and the most urgent cost-effective cut points for running the legacy application.

4.2 Data Dependence Analysis

As described in Section 2.1, data dependence can prevent parallelization. It is therefore prudent to check data dependence before performing parallelization in order to avoid changing program semantics. Since parallelization typically takes advantage of concurrently executing iterations of a loop in order to shorten the total execution time, the data dependences between iterations of a loop are the most important subject of parallelization. However, the data dependence between the iterations of a loop is, nonetheless, not as simple as within a sequence of statements. Some data dependence relationships can be very difficult to check due to the complexity of a program. We need to define a formal procedure to ensure that the valid data dependence analysis can correctly determine if there are data dependencies between iterations of a loop. Some tools will be introduced to ease the analysis.

Data dependence happens between two statements that need to access the same variable. By the very definition and use of a variable in different orders, we can have true, anti-, and output dependencies. We will need to check the execution order of instances of loops and the use of common variables to determine if there are data dependences between any statements.

4.2.1 Determination of Loop Data Dependence

The first step of data dependence analysis is to define the execution order. It may be remarked that the sequential code implies an execution order for all statements. A program counter points to the next statement to be executed. The program counter moves to the next statement in the program unless the current statement is a branch statement. Variables are defined and used in this execution order.

In order to simplify the analysis, the loop we discuss in this section is a normalized loop. A normalized loop is a loop with an index variable that has a start value 1, a positive final value n, and an increment value of 1. Any non-normalized loop can be normalized by the transformation described later. Suppose we have a normalized m-level loop L with loop indices I_1, I_2, ..., I_m and final values n_1, n_2, ..., n_m where $1 \leq I_1 \leq n_1$, $1 \leq I_2 \leq n_2$, ..., $1 \leq I_m \leq n_m$. An iteration vector is defined as an m-tuple $\bar{I} = (i_1, i_2, \ldots, i_m)$ to represent the instance of the m-level loop where $I_1 = i_1$, $I_2 = i_2$, ..., $I_m = i_m$.

The instances of the m-level loop are executed in the order

$$
\begin{array}{l}
(1,1,\ldots,1,1),\ (1,1,\ldots,1,2),\ \ldots,\ (1,1,\ldots,1,n_m),\\
(1,1,\ldots,2,1),\ (1,1,\ldots,2,2),\ \ldots,\ (1,1,\ldots,2,n_m),\\
\qquad\qquad\qquad\vdots\\
(n_1,n_2,\ldots,n_{m-1},1),\ (n_1,n_2,\ldots,n_{m-1},2),\ \ldots,\ (n_1,n_2,\ldots,n_{m-1},n_m).
\end{array}
$$

Any two instances $\bar{I}_1$ and $\bar{I}_2$ will have one of 3 execution order relationships $\bar{I}_1 < \bar{I}_2$

which means $\bar{I}_1$ is executed before $\bar{I}_2$, $\bar{I}_1 > \bar{I}_2$ which means $\bar{I}_1$ is executed after $\bar{I}_2$, and $\bar{I}_1 = \bar{I}_2$ which means $\bar{I}_1$ and $\bar{I}_2$ are the same instance. We can tell the execution order relation between any two instance $\bar{I}_1$ and $\bar{I}_2$ by checking their iteration vectors. The following rules hold:

- $\bar{I}_1 > \bar{I}_2$ if $i_{1_j} = i_{2_j}$ for all $j = 1, k$ and $i_{1_{k+1}} > i_{2_{k+1}}$ where $1 \leq k < m$.
- $\bar{I}_1 < \bar{I}_2$ if $i1_j = i2_j$ for all $j = 1, k$ and $i_{1_{k+1}} < i_{2_{k+1}}$ where $1 \leq k < m$.
- $\bar{I}_1 = \bar{I}_2$ if $i_{1_j} = i_{2_j}$ for all $j = 1, m$.

As described earlier in this section, data dependence between two statements occurs when these two statements access the same variables. An input set of statement S is the set of variables whose values are used for the statement S. An output set of statement S is the set of variables whose values are assigned in statement S. Variables used as subscripts of other variables are considered input variables.

Data dependence in the same instance can be determined by the method described in Section 2.1. Now the data dependence between two statements S1 and S2 in different instances $\bar{I}_1$ and $\bar{I}_2$ where $\bar{I}_1 < \bar{I}_2$, with the input sets $\hat{I}(\mathrm{S1})$ and $\hat{I}(\mathrm{S2})$ and output sets $\hat{O}(\mathrm{S1})$ and $\hat{O}(\mathrm{S2})$ can be determined in the following conditions:

- True dependence $\mathrm{S1} \xrightarrow{\delta^t} \mathrm{S2}$ if $\exists v \in \hat{O}(\mathrm{S1}) \cap \hat{I}(\mathrm{S2})$.
- Anti-dependence $\mathrm{S1} \xrightarrow{\delta^a} \mathrm{S2}$ if $\exists v \in \hat{I}(\mathrm{S1}) \cap \hat{O}(\mathrm{S2})$.
- Output dependence $\mathrm{S1} \xrightarrow{\delta^o} \mathrm{S2}$ if $\exists v \in \hat{O}(\mathrm{S1}) \cap \hat{O}(\mathrm{S2})$.

4.2.2 Loop Index and Variable Subscript Transformations

To simplify the determination of data dependence, we assumed that all loops displayed are normalized. However, not all loops are normalized in real programs. Some subscripts of variables have complex forms which make the dependence analysis very difficult. Some program transformation methods [97] transform the program into some standard formats that can be easily checked for data dependency relations without altering the program semantics. These transformation methods can be used to simplify data dependence analysis.

These transformation methods are used to ease the data dependency analysis. Some transformations are the inverses of optimizations. Optimizations should not be given up for parallelization. Optimizations still can be re-applied after the dependence analysis is completed.

DO Loop Normalization A DO loop has an index variable which is assigned an initial value in the beginning, increased by an increment value in the end of each iteration, and stopped when the index variable exceeds the limit value. The increment value can be anything except for 0. Some languages such as C even allow floating point values. The flexibility of the DO loop makes DO loops easier to use. However, it may cause difficulty when checking the dependency in the loop.

The simple way to make the addressed checking easier without losing the functionality of DO loop is to normalize it. A DO loop is normalized if its initial value and increment equal to one.

This normalization is straightforward. The upper bound of the loop is simply changed to *(Exp2 - Exp1 + Exp3) / Exp3* and all loop indices are substituted in the loop to $I_{normalized} \times Exp3 + Exp1$. Thus the DO loop normalization is done. The final value of the loop index variable can be set to *Exp1 + (Exp2 - Exp1 + Exp3) / Exp3* $\times$ *Exp3* if needed. The normalization is shown in Figure 4.3

Scalar Forward Substitution Equally well, programmers often use temporary variables to store the values of common expressions. This reduces the amount of computation necessary. However, the temporary variables may be used in the subscript expressions of statements with data dependency and complicate the data dependency checking. One may falsely think there are data dependences between some statements if there are several statement references to these temporary variables in a big loop.

S1 : DO $I = \mathit{Exp1},\ \mathit{Exp2},\ \mathit{Exp3}$
S2 : DO-Body
S3 : ENDDO

(a) Unnormalized DO loop

S1′ : DO $I_{normalized} = \mathit{1},\ (\mathit{Exp2} - \mathit{Exp1} + \mathit{Exp3})\ /\ \mathit{Exp3}$
S2′ : DO-Body (with all I substituted by $I_{normalized} \times \mathit{Exp3} + \mathit{Exp1}$)
S3′ : ENDDO
S4′ : $I = \mathit{Exp1} + (\mathit{Exp2} - \mathit{Exp1} + \mathit{Exp3})\ /\ \mathit{Exp3} \times \mathit{Exp3}$

(b) Normalized DO loop

Figure 4.3: DO loop normalization

Scalar forward substitution replaces all such temporary variables with their original expressions. This may cause the expressions which in effect use these temporary variables to be more complex and inefficient, but the data dependency of these statements will be easier to check because these expressions now use only the necessary variables.

Induction Variable Substitution Induction variables are variables that used in a loop and its value is generated from the value in the previous iteration of the loop. Induction variables have have the following form:

```
IV=initial value
DO I = 1, N
      IV = F(IV)
          ⋮
ENDDO
```

The value of induction variable IV is generated form its previous value by a function F. A true data dependence of variable IV is found between the current iteration and previous iteration of the DO loop. The induction variable may force the iterations of the DO loop to be executed sequentially. The iterations of the loop cannot be executed in parallel unless we can produce a function of the form $\underbrace{F(\,F(\cdots F}_{N}(\mathit{initial\ value})\cdots))$.

Induction variables are usually used in loops to serve as another index variable of a DO loop. Consider the following example:

A triangular matrix A is stored in a one dimensional array B to save some space. $A[1,\ 1] = B[1]$, $A[2,\ 1] = B[2]$, $A[2,\ 2] = B[3]$, $A[3,\ 1] = B[4]$, $\ldots$, $A[4,\ 1] = B[7]$, $\ldots$, $A[n,\ 1] = B[(n - 1) \times n \ / \ 2 + 1]$, $\ldots$, $A[n,\ n] = B[n \times (n + 1) \ / \ 2]$. Now we want to assign $A[i,\ 1] = i$, for $i = 1,\ N$. One may use the program in Figure 4.4(a).

The variable BI is an induction variable of the loop. Induction variable BI prevents the above program from being parallelized. Fortunately, calculating the value of $\underbrace{F(\,F(\cdots F}_{N}(\mathit{initial\ value})\cdots))$ is possible if $F(x)$ has the form $F(x) = a \times x + b$ where a and b are constant.

Let IV_i denotes the value of induction variable IV at the $i'th$ iteration where $1 \leq i \leq N$. Suppose IV has initial value IV_0.

```
S1  :   BI = 1
S2  :   DO I = 1, N
S3  :       B[BI] = I
S4  :       BI = BI + I
S5  :   ENDDO
```

(a) loop with induction variable

```
S1′ :   BI = 1
S2′ :   DO I = 1, N
S3′ :       B[I × (I + 1) / 2] = I
S4′ :   ENDDO
S5′ :   BI = N × (N + 1) / 2
```

(b) loop without induction variable

Figure 4.4: Induction variable substitution

$$
\begin{aligned}
IV_1 &= F(IV_0) = a \times IV_0 + b \\
IV_2 &= F(IV_1) = a^2 \times IV_0 + (a+1) \times b \\
IV_3 &= F(IV_2) = a^3 \times IV_0 + (a^2 + a + 1) \times b \\
&\vdots \\
IV_N &= F(IV_{N-1}) = a^N \times IV_0 + (a^{N-1} + a^{N-2} + \cdots + a + 1) \times b
\end{aligned}
$$

We can generalize the above equations as follows:

$$
\begin{aligned}
IV_i &= a^i \times IV_0 + (a^{i-1} + a^{i-2} + \cdots + a + 1) \times b \\
&= a^i \times IV_0 + \frac{a^i - 1}{a - 1} \times b \qquad (4.2.1)
\end{aligned}
$$

The induction variable IV can be substituted by equation 4.2.1 to eliminate the data dependency and make the DO loop parallelizable. The DO loop that substitutes its induction variable by applying equation 4.2.1 is shown in Figure 4.4(b).

4.3 Loop Parallelization

Loops are the main causes accounting for time-consuming and the targets of parallelization. Loop parallelization changes the sequential codes to parallel codes and guarantees the final results is the same as the results generated by the sequential codes. As introduced in Section 2.1, only true dependence is unresolveable. Loop interchange exchanges the loops organization to improves the performance while maintain the program correctness. Anti- and output data dependence between loop instances can be resolved by introducing new variables. Scalar expansion and variable copying not only improve the performance but also resolve the perplexing data dependence between loop instances. These optimization methods for loop parallelization are described in this section.

Loop Interchange We would like to have coarse granularity when parallelizing a loop. It reduces the frequency of communication and synchronization. For the above straightforward reason, we prefer distributing the outermost loop if possible.

It is common that the work load cannot be 100% evenly distributed to all processors when the number of distributable work load units is not an integer multiple of the number of processor. Some processors may receive a more generous work load than the others due to the residual. This causes a minor imbalance. The work load distribution will be more balanced if the number of work load units is far bigger than the number of processors. For example, distributing 3 units of work load to 2 processors results 50% imbalance while 101 units results only 2% imbalance. We prefer to distribute bigger loops for load balance reasons.

Loop interchange is a transformation that exchanges two levels of a nested loop.

```
S1  :   DO I = 1, N1
S2  :       DO J = 1, N2
S3  :           DO loop body
S4  :       ENDDO
S5  :   ENDDO
```

(a) Loop I outside of loop J

```
S1' :   DO J = 1, N2
S2' :       DO I = 1, N1
S3' :           DO loop body
S4' :       ENDDO
S5' :   ENDDO
```

(b) Loop J outside of loop I

Figure 4.5: Loop interchange

Consider the example in Figure 4.5(a), loop I is the outer loop. Suppose $N1 \ll N2$. We may not get a good load balance if we distribute the outer loop I. We may get a fine grain parallelization if we distribute the inner loop J. Fairly simple, we can exchange the two loops as long as there is no correlation between the two loop index variables I and J. The loop structure after applying loop interchange is shown in Figure 4.5(b).

By applying loop interchange, we can move the largest loop to the outermost of a multi-level loop structure and distribute the outermost loop. This will entirely satisfy both the concerns of coarse granularity and load balancing.

```
S1  :    DO I = 1, N
S2  :        A = F(B[I]) + G(X)
S3  :        C[I] = H(A)
S4  :    ENDDO
```

(a) Data dependence caused by intermediate scalar variable

```
S1′ :    DO I = 1, N
S2′ :        AA[I] = F(B[I]) + G(X)
S3′ :        C[I] = H(AA[I])
S4′ :    ENDDO
S5′ :    A = AA[N]
```

(b) Data dependency resolved by scalar expension

Figure 4.6: Scalar variable expansion

Scalar Expansion Scalar variables are commonly used to store intermediate results of a complex expression. The complex expression is divided into smaller sub-expressions. The sub-expressions are then executed and the results are stored in some intermediate variables. The final result of the complex expression is calculated from these intermediate scalar variables.

Scalar variable A is used as an intermediate variable to store the result from the right hand side of statement S2 in Figure 4.6(a). The statement S3 cannot execute until S2 is executed. This causes a true dependence from S2 to S3. This true dependence does not cause any problem since S2 and S3 are in the same loop and we are interested in parallelizing the loop only, not statements inside the loop.

However, variable A will be assigned new value at next iteration. Statement S2 cannot be executed if statement S3, in previous iteration, has not be executed yet. This causes an anti-dependence from $S3_{i-1}$ to $S2_i$. Moreover, the final value of variable A should be the value assigned in the last iteration of the loop. The last iteration must, therefore, be executed last. This causes an output dependence from $S2_i$ to $S2_N$ for all $i = 1, N$.

To resolve this problem, one can give each iteration of the loop a variable instead of sharing one. As shown in Figure 4.6(b), an array of size N is introduced to give every iteration its own intermediate variable. The data dependence relations are thus eliminated in Figure 4.6(b).

Variable Copying Consider the example in Figure 4.7(a), array A in statement S2 cannot be re-written until statement S3 in previous iteration is executed because *A[I+1]* in $S3_{i-1}$ is the same as the A_i in statement $S2_i$. This causes an anti-dependence from $S3_{i-1}$ to $S2_i$.

To resolve this problem, one can use a working array *A2* to store the values of *A[I+1]*. As shown in Figure 4.7(b), statement $S1'$ copies the contents of array A to working variable *A2*. By paying the extra cost of array copying and associate storage, we resolve the data dependence problem and parallelize the loop. Not surprisingly, note that the extra cost of copying the array can be parallelized as well.

4.4 Integration

A parallelized legacy application consists at least a sequential and a parallel parts. The parallel parts must run on a parallel machine and the sequential parts can run on a workstation or a node of the parallel machine. In many cases, especially for large shared servers, a user interface module is also needed. Figure 4.8 shows the relationship between these modules. Once again, no matter how the machines are used, these modules need to work cooperatively. Synchronization and data exchange mechanism must be implemented.

```
S1  :    DO I = 1, N
S2  :        A[I] = F(B[I])
S3  :        C[I] = H(A[I], A[I + 1])
S4  :    ENDDO
```

(a) Data dependence caused by anti dependence

```
S1' :    DO I = 1, N
S2' :        A2[I] = A[I + 1]
S3' :    ENDDO
S4' :    DO I = 1, N
S5' :        A[I] = F(B[I])
S6' :        C[I] = H(A[I], A2[I])
S7' :    ENDDO
```

(b) Data dependency resolved by variable copying

Figure 4.7: Variable copying

The data exchanged between these modules are:

- Sequential part and parallel functions: the parameters of the parallel functions.
- User interface and sequential part: input data and output files. Access control if the parallelized legacy application is a shared server.
- Display and user interface: text stream if in text only environment. Graphic data if GUI is used.

These functionally distributed modules can be considered coarse grain parallelism.

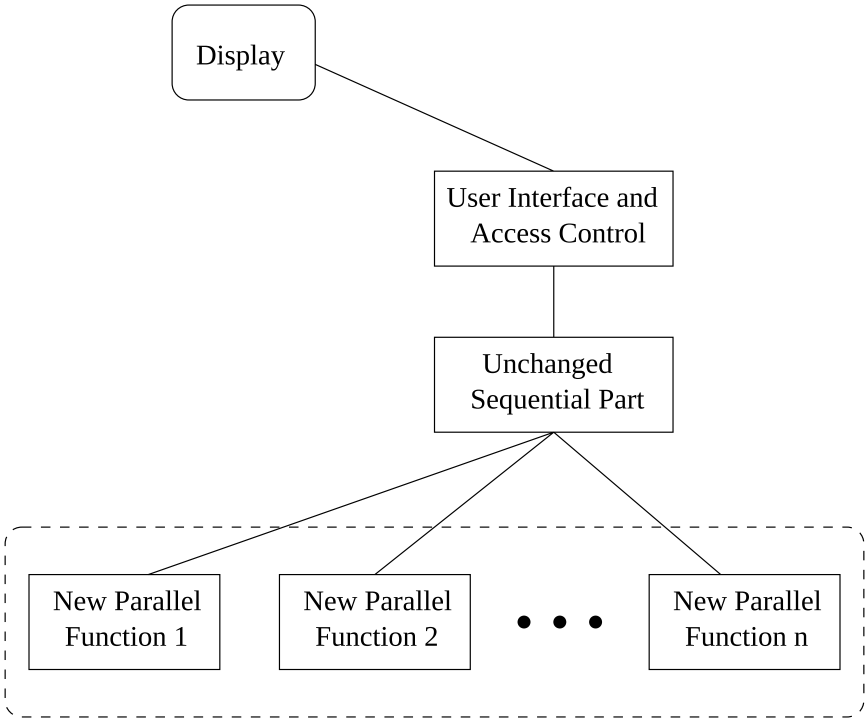


Figure 4.8: Integration

The traffic between these modules are emphatically as low as regular local area network activities. They can use any local area network protocols such as TCP/IP. If both parties run on the same machine, inter-process communication such as UNIX domain socket and shared memory can be used to improve the performance.

By contrast, while the unchanged sequential parts is running, the parallel machine is idle if it is not used by other programs. In this case, we can simply replicate the unchanged sequential parts on all nodes. The results of the sequential parts are available in all nodes and do not need to be distributed. Still more clearly, this saves the data distribution time. However, replication is not recommended in the following cases:

- Heterogeneous node types. Different platform may produce slightly different results for the same program. When the sequential parts are finished, each node starts the parallel parts computation based on slightly different data. The difference may cause some problems for the parallel parts if the program does not tolerate the difference.

- Heterogeneous node performance. When a parallel machine contains slow nodes, faster nodes may need to wait for slow nodes to finish running the sequential parts. Running sequential parts on a fast node and then broadcasting the results may be faster.

- The sequential parts contain non-sharable operations. The most commonly seen non-sharable operations is I/O. Only one node can actually perform output, other nodes need to discard the data. Input data must be retrieved by one node and broadcast to all nodes.

Chapter 5

Parallelizing MOPAC

Parallelizing a legacy application like MOPAC is no doubt a large job to confront. These programs are large, complicated and difficult to trace. The goal is to improve the performance in the most economical way. As the analysis procedures specified in the previous chapter showed, it is necessary to analyze the sequential version of the code in order to find the most time-consuming parts. These time-consuming parts are examined for possible of parallelization. These computationally intensive sections usually reside in a small number of subroutines. Parallelizing only these time-consuming parts and leaving the rest of program unchanged achieve the same performance improvement as parallelizing the whole program while saving a lot of effort. Four subroutines, DCART, DENSIT, DIAG and HQRII are identified as the time-consuming parts of MOPAC in the time profiling analysis.

The work load distribution for each part of program is not a fixed pattern. The pattern depends heavily on the input data. It is very urgent to find out the computational complexity of the whole program when we parallelize the computational intensive parts. The computationally intensive parts that have the highest computational complexity of the whole program are nominally considered the dominant parts of the program. The other parts that have lower computational complexity are non-dominant. The percentage of CPU time required by the non-dominant parts reduces and becomes negligible when N gets large even if they take large portion of CPU time for small data sizes. The dominant parts will benefit from parallelization more than the non-dominant parts.

The computational complexity of the four computationally intensive parts of MOPAC is $O(N^3)$ except for subroutine DCART. Subroutine DCART it is not eligible for parallelization because its computational complexity is $O(N^2)$. The CPU time it needs will not grow as fast as other computationally intensive parts and become very shortly not computationally intensive when the data size grows larger.

The computationally intensive subroutines are parallelized by applying the procedure drawn attention to in Chapter 4. Subroutine HQRII is identified as a fast eigensolver. The sequential code is optimized for sequential execution and difficult to parallelize. Parallel eigensolvers, however, are commonly used and have been well studied. Many algorithms have been implemented, tested and available for public use. Using a well known code not only save us efforts but also reduce the risk of introducing software bugs. Subroutine HQRII will be parallelized by using a pre-exist parallel eigensolvers called PeIGS in the final retrospect.

Finally, the sequential module and parallel modules as well as the newly added data visualization part patently need to be integrated. We use AVS to integrate these coarse grain modules to allow these modules run in heterogeneous platforms.

5.1 Sequential Analysis

In this section, we will show how we perform the three sequential analyses on the sequential MOPAC to identify the target subroutines for ultimate parallelization.

5.1.1 Program Flow Analysis

As described in previous chapter, we do not want to analyses the functions of MOPAC subroutines like chemists' do. We just want to distinguish the frequently used data processing subroutines from MOPAC so that we can focus the time profiling analysis on these subroutines.

The sequential MOPAC code was developed by hundreds of authors over more than 30 years [37]. The version we are using was published in Dec. 1990. It consists of thirty thousand lines of FORTRAN 4 and FORTRAN 77 code. MOPAC version 6 was originally developed on a DEC-3100 and ported to several different platforms. Since UNIX is the most popular OS for scientific workstations, we utilized the SUN4 port as our origin.

The idea behind creating MOPAC was to integrate several independent algorithms into a single package. The program offers different routes at some points. The behavior and execution flow are controlled by the keywords specified in the first three lines of the input data file.

The MOPAC flow diagram is shown in Figure 5.1. The text in the diagram shows the subroutine names. MOPAC allows putting multiple molecule structures in one data file. A loop in the main program repeatedly call subroutine READMO to read in one set of input data at a time. The loop ends when all input data is read and processed. Since we are obstinately interested in determining which are data processing subroutines, Figure 5.1 suppresses the loop. There are two main sequences in MOPAC. They are the geometric sequence and the electronic sequence. The diagram above the subroutine COMPFG shows the geometric sequence. The diagram beneath the subroutine COMPFG shows the electronic sequence. Subroutine READMO reads in one set of data and selects the execution flow in MOPAC geometric and electronic sequences. The molecular structure then feeds one geometric sequence into one of the execution paths, passes through COMPFG, and then again as elsewhere goes through one of the execution paths in the electronic sequence.

Above all, all of the data required for a subroutine is passed in either as arguments or in common blocks. Variables used only inside a subroutine are local to the subroutine and not passed outside of the subroutine. The data that a subroutine is going to work on is passed via its argument list. Reference data is passed into a subroutine through common blocks.

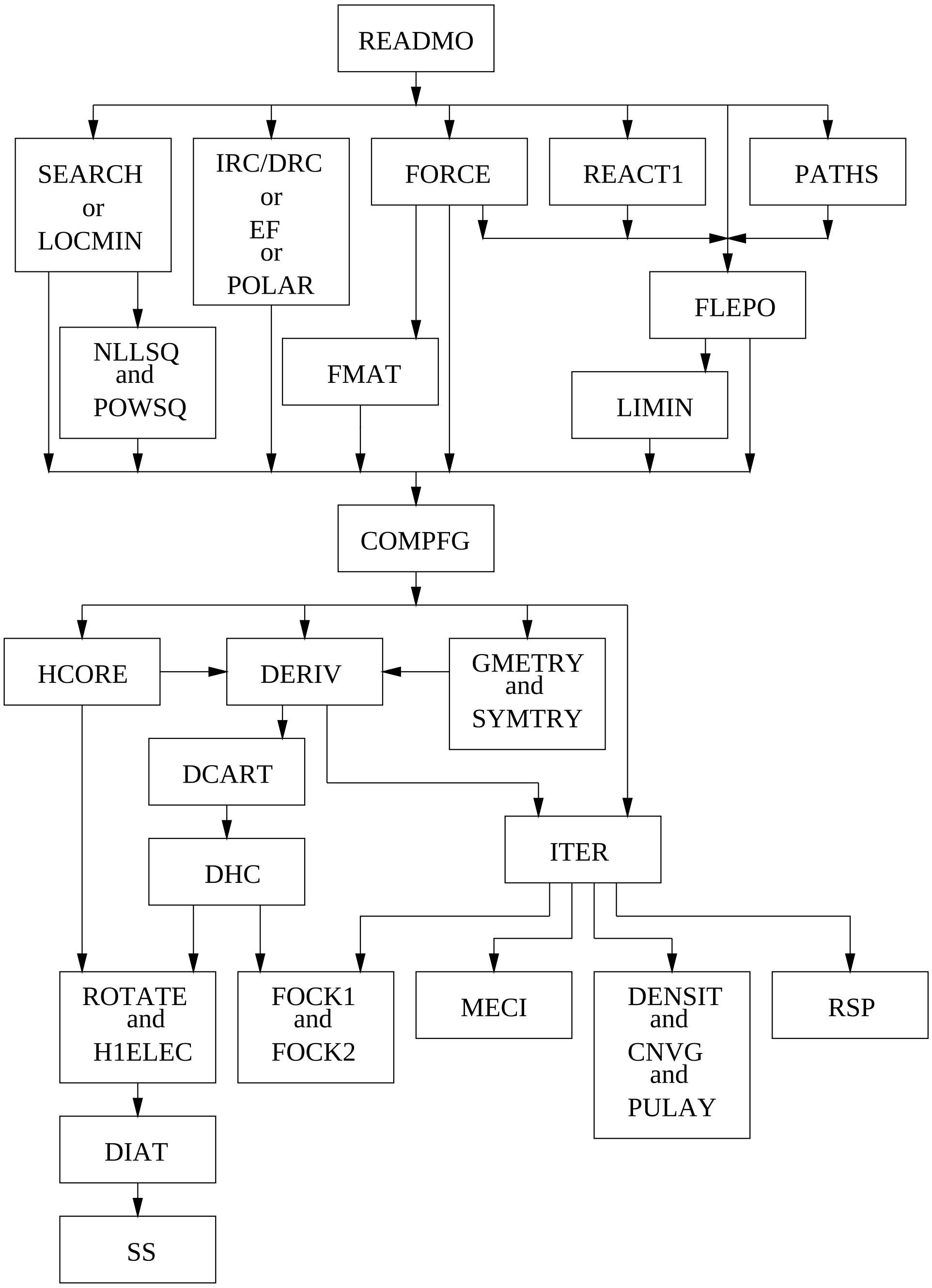


Figure 5.1: MOPAC flow diagrams

The most notable subroutine in Figure 5.1 is subroutine COMPFG. There is no major loop in COMPFG. It acts as a coordinator that calls other main routines to perform computations and organizes the results to feed into another routine if needed. The main data processing and computation start by COMPFG. The major CPU time consumers must be among the subroutines called by COMPFG.

Besides the program flow, the core of the calculation of MOPAC consists of the repeated application of a diagonalization routine in subroutine ITER. In order to speed up the calculation, one starts first with a matrix which is close to its diagonal form and performs a fast diagonalization routine. The last step of the iteration uses an exact diagonalization routine. Subroutine DIAG does the fast diagonalization and subroutine HQRII the exact one. These two subroutines consume a lot of CPU time which the time profiling analysis in next section will show.

5.1.2 Time Profiling Analysis

Time profiling must use some time measuring mechanism. The most common way is inserting code to write time stamps in the beginning and end of the codes we want to trace. These time measuring mechanisms may take some disarming CPU time and introduce some errors in the profile. The CPU time used by time measuring must be small enough to be ignored or it should be taken into account at the end of the time profiling.

MOPAC was written in FORTRAN. FORTRAN does not interact with UNIX operating systems very well. The resolution of timing subroutines is one second. This is not enough accuracy to support fast scientific workstations. We have to use C to do a better time profiling. C system calls provide up to micro-second time resolution. The real resolution depends on operating system and hardware implementations. Modern workstations have at least milli-seconds time resolution. Micro-seconds time resolution are provided through real time clock system calls.

The MOPAC time profiling is implemented using dual process. As shown in Figure 5.2, our time profiling program consist of two parts. The first part is linked

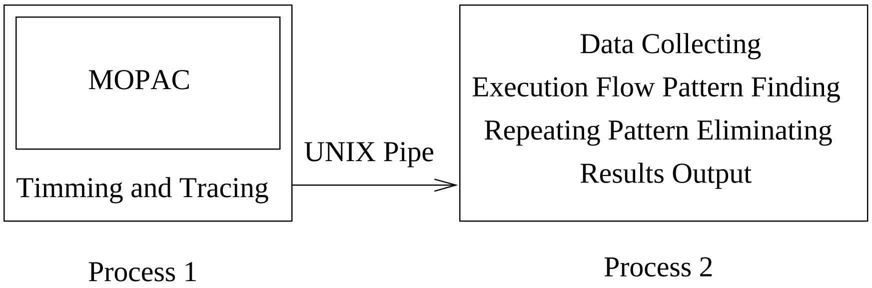


Figure 5.2: MOPAC time profiling tool

with MOPAC so that it can cogently report how much CPU times has been consumed. The second part runs as a separate process to collect the timing information sent out by the first part. One gets time stamps and the other writes time stamps. The two parts are connected by a UNIX pipe.

The CPU time used by the time profiling utility can be divided into two parts, CPU time reading and time stamp recording. The CPU time reading system calls read a system time variable which is updated by the operating system. A UNIX pipe is implemented as a circular queue in main memory. Writing to the pipe is as fast as memory accesses and the CPU time used by writing data to the pipe is about the same for every write operation. Unlike writing to disk, the operating system stops CPU time accumulation and switches to the other process when the pipe is full. The time used by the time profiling utility is small and fully uniform for all data stamps. It can be measured easily or ignored. The time profiling interference to MOPAC is minimized. The data collection process can subtract the time used by the time reading and time stamp writing system calls to get precise time profiling results.

As mentioned previously, the results of time profiling is highly dependent on the test data. Different types of test data at once show different work load distributions. Three test data files are chosen for the time profiling. The tests are run on two Sun Sparc and two IBM RS6000 workstations. We run these test files on the test machines with and without our time profiling routines. The total time required by time profiling utility is less than *1.5%* of the total CPU time.

Different machines have different performance characteristics. The CPU time may

Subroutine	apsbtest	porphin	tetrabenz	
Weight	0.667	0.101	0.232	1.000
	CPU time percentage	CPU time percentage	CPU time percentage	Weighted percentage
DCART	6.51 %	11.56 %	8.22 %	7.42 %
DENSIT	16.18 %	14.14 %	15.83 %	15.89 %
DIAG	63.17 %	63.98 %	68.69 %	64.53 %
HQRII	8.22 %	1.72 %	1.52 %	6.01 %
Sum	94.08 %	91.40 %	94.26 %	93.85 %

Table 5.1: MOPAC execution time profiling

vary from one machine to another. We need to use the CPU time percentages instead of raw CPU time to calculate the average for results coming from different machines.

It is not a correct approach to treat the results equally. The data for larger calculations should be given a larger weight. The weights are therefore given based on the total CPU time. Detailed procedures and processes can be found in [63]. Table 5.1 shows the final results of MOPAC time profiling.

Four subroutines, DCART, DENSIT, DIAG, and HQRII, are found to take the most CPU time. These four subroutines are all called by core subroutine COMPFG directly or indirectly. Subroutine DCART calculates the derivatives of the energy with respect to the Cartesian coordinates of the atoms in the molecule. Subroutine DENSIT is a utility that constructs the Coulson electron density matrix from the eigenvectors. Subroutine DIAG and HQRII are diagonalization routines.

The four subroutines take 93.85% of total CPU time. By the same token, from Amdahl's law, we can expect a maximum of $\frac{1}{1-93.85\%} = 16.3$ times speed-up for the data files of the same size as our test data.

The above timing profiling analysis gives us some idea about the work load distribution of an application running certain input data. We can also estimate the maximum possible speed-up the application can gain from parallelization. The 16.3 times speed-up we get from time profiling MOPAC is not very attractive. Fortunately, from Section 4.1.3, we expressly know that data size affects the maximum possible speed-up. Usually the time-consuming parts have higher computational complexity than other parts. The larger data sizes used in productions raise the maximum possible speed-up of a parallel program. We need to combine the results of time profiling analysis with computational complexities of those subroutines to do profoundly better estimations.

5.1.3 Complexity Analysis

For complexity analysis of the MOPAC time-consuming subroutines, the abbreviations shown in the following are used:

- n_l: number of light atoms, e.g. H.
- n_h: number of heavy atoms, e.g. C.
- N: number of atoms. $N = n_l + n_h$.
- n_o: number of occupied orbitals. Typically, $n_o \sim 0.5n$.
- n_v: number of virtual orbitals.
- n: number of orbitals. $n = n_o + n_v$. Typically, $n \sim 4n_h + n_l$.

Complexity Analysis for DCART

Subroutine DCART calculates the derivatives of the energy with respect to the Cartesian coordinates of the atoms in the molecule. It uses a finite difference method. When DCART is called by subroutine DERIV, it sets up a list of Cartesian derivatives of the energy with respect to coordinates. Subroutine DERIV can then use it to construct the internal coordinate derivatives.

// *nHNCO* : number of H-N-C=O

// *all possible 3-D moves* : single step move for *(x,y,z)* = $(-1{:}1,\ -1{:}1,\ -1{:}1)$

S1 : for $i = 1,\ N$ do
S2 :   for $j = 1,\ i$ do
S3 :   for $k \in$ *all possible 3-D moves* do
S4 :   *make one step move on the second atom*
S5 :   *calculate the energy contribution from the move*
S6 :   endfor
S7 :   endfor
S8 : endfor
S9 : // add in molecular-mechanics correlation to H-N-C=O torsion if any
S10 : if (*nNHCO* > 0) then
S11 :   for $i = 1,$ *nHNCO* do
S12 :   *calculate H-N-C=O torsion*
S13 :   endfor
S14 : endif

Figure 5.3: simplified DCART algorithm for complexity analysis

As shown in Figure 5.3, subroutine DCART can be divided into two parts. The main computation is in the first part while the second part handles a special H-N-C=O structure.

The loop at step S3 is used for a 3 dimensional atom movement calculations. The atom moves one step in either direction of all three dimensions, that is, $x = -1,\ 0,\ +1$, $y = -1,\ 0,\ +1$, $z = -1,\ 0,\ +1$. This yields a constant number of $(3 \times 3 \times 3 = 27)$ iterations. The upper bound variable of the loop at step S2 is the index variable of the outermost loop at step S1. This gives the total of $\frac{N\times(N+1)}{2}$ iterations for the loops S1 and S2. Likewise, the computational complexity of the three loops is therefore

$O(N \times (N+1)/2 \times 27) = O(N^2)$.

The second part of DCART is a single level loop and iterates *nHNCO* times. However, it is used only when the input atom structure contains H-N-C=O structure. We do not count this designated part in the computational complexity analysis since it is not a general case.

Note that the step S5 is not a simple step. Step S5 calls subroutine DHC. The computational complexity should be $O(n^2\times$ *complexity of DHC)*. Appendix C shows at some length that the computational complexity of subroutine DHC is $O(1)$. Therefore, the total computational complexity of subroutine DHC is $O(n^2)$.

Complexity Analysis for DENSIT

Subroutine DENSIT is a utility that constructs the Coulson electron density matrix from the eigenvectors. It takes the eigenvectors, number of singly and number of doubly occupied levels as inputs and produces the density matrix as output.

The program structure of this subroutine is very simple. There are no subroutine calls to any other subroutine. The algorithm of DENSIT is shown in Figure 5.4. Step S1 sets some boundary variables from the number of singly and doubly occupied levels. The values of the boundaries are determined by the *mode* input argument which specifies whether the electron or positron equivalent is used. An "if" structure handles the checking and the range assignments. It is embedded in step S1 and not shown in Figure 5.4 because it has no impact to the complexity analysis.

The second level loop at step S3 depends on the index variable of the outermost loop at step S2. The two outer level loops will iterate $n \times (n+1)/2$ times. The two innermost loops at step S5 and S8 are single level loops. S5 iterates "the number of singly occupied orbitals" times and S8 iterates "the number of doubly occupied orbitals" times. The total number of singly and doubly occupied orbitals is less than n. Therefore the total number of construed iterations that loop S5 and S8 has is less than n times. The three-level loop structure yields a computational complexity of $O(\frac{n\times(n+1)}{2} \times n) = O(n^3)$ for subroutine DENSIT.

```
        // evec : eigenvectors matrix
        // den : density matrix

S1  : nl1 = beginning of one electron sum; nu1 = end of one electron sum;
      nl2 = beginning of two electron sum; nu2 = end of two electron sum;
S2  : for i = 1, n do
S3  :     for j = 1, i do
S4  :         sum1 = sum2 = 0;
S5  :         for k = nl1, nu1 do
S6  :             sum1 = sum1 + evec[i,k] × evec[j,k];
S7  :         endfor
S8  :         for k = nl2, nu2 do
S9  :             sum2 = sum2 + evec[i,k] × evec[j,k];
S10 :         endfor
S11 :         den[i,j] = sum1 × fract + sum2 × 2;
S12 :     endfor
S13 : endfor
```

Figure 5.4: simplified DENSIT algorithm for complexity analysis

Complexity Analysis for DIAG

As described in prescribed program flow analysis, the core of the calculation is the repeated application of a diagonalization routine. Subroutine DIAG is a fast diagonalization routine that is used as a good starting approximation. Typically a few steps of the conventional SCF iteration are then required to bring the suggested secular determinant (Fock matrix) into approximately diagonal form in the orbital basis.

Subroutine DIAG is a pseudo diagonalization routine in that the vectors that are generated by it are more nearly able to block-diagonalize the Fock matrix over molecular orbitals than the starting vectors. It must be considered pseudo for several reinforcing reasons:

- It does not generate a complete set of eigenvectors. The secular determinant is not diagonalized, only the occupied-virtual intersection is. That means only the eigenvectors for the occupied orbitals are generated.
- Many small elements in the secular determinant are ignored as being too small compared with the largest element.
- When elements are eliminated by rotation, the rest of the secular determinant is assumed not to change, i.e. elements created are ignored.
- The rotation required to eliminate those elements considered significant is approximated to using the eigenvalues of the exact diagonalization throughout the rest of the iterative procedure.

The procedure has the following arguments:

- A: contains the lower half triangle of the matrix to be diagonalized in a packed format.
- *evec*: contains the old eigenvectors on input, the new vectors on output.
- n_o: number of occupied molecular orbitals.
- *eval*: eigenvalues from an exact diagonalization.
- n: number of atomic orbitals in the basis set.

The algorithm of subroutine DIAG can be divided into two parts. To this end, the first part of the diagonalization routine constructs the secular determinant over molecular orbitals which connects occupied and virtual sets. The sequential algorithm uses a triangular matrix as input argument in order to save memory. For the parallelization of this sequential algorithm, it is better to keep the whole matrix even though more memory is used. Substituting the triangular matrix by its complete counterpart would enable one to rewrite the sequential algorithm in the way shown in Figure 5.5.

In Figure 5.5, the array w is a temporary variable, matrix *fmo* holds the Fock molecular orbital interaction matrix, and *evec(*,i)* is the $i'th$ column vector of the matrix *evec* of length n. Temporary variable *tiny* holds the value of the largest element of *fmo*. It will be used as a threshold to determine if a 2 by 2 rotation is needed in the second part of subroutine DIAG. The computational complexity of calculating the occupied virtual block of the secular determinant as shown above is $O(n_v \times (n^2 + n_o \times \ n)) \approx O(n^3)$

The second part of DIAG performs a crude 2 by 2 rotation to the eigenvectors to eliminate the significant elements. The simplified algorithm of DIAG part 2 is shown in Figure 5.6. The step S7-9 that perform the 2 by 2 rotation are a set of simple statements and contain no loop. In the worst case where step S4 is satisfied all the time, the computational complexity of the algorithm is $O(n_v \times n_o \times n) \approx O(n^3)$.

Complexity Analysis for HQRII

As a matter of course, subroutine HQRII is a fast standard diagonalization routine designed to solve the standard eigenvalue problems [12]. The name HQRII is short for "Householder-QR-Inverse Iteration method", which are the methods it uses. The HQRII subroutine is divided into three major steps reflecting the names of the routines.

- (Householder) the dense matrix A is converted into the tridiagonal matrix T by the Householder algorithm.
- (QR) all eigenvalues of the tridiagonal matrix T are determined by the QR algorithm.
- (Inverse Iteration) Some of the eigenvectors of the tridiagonal matrix T are found by the inverse iteration algorithm where Gaussian elimination with partial

```
// A : matrix to be diagonalized
// evec : eigenvector matrix
// fmo : Fock molecular orbital interaction matrix
// w : working space

tiny = 0;
for i = 1, n_v do
    for j = 1, n do
        ws[j] = 0;
        for k = 1, n do
            ws[j] = ws[j] + A[j,k] × evec[k,i];
        endfor
    endfor
    for j = 1, n_o do
        fmo[i,j] = 0;
        for k = 1, n do
            fmo[i,j] = fmo[i,j] + ws[k] × evec[k,j];
        endfor
        if (tiny < abs(fmo[i,j]) then
            tiny = abs(fmo[i,j]);
        endif
    endfor
endfor
tiny = 0.05 × tiny;
```

Figure 5.5: simplified DIAG part 1 algorithm for complexity analysis

```
      // fmo : Fock molecular orbital interaction matrix from part 1
      // evec : eigenvector matrix
      // eval : eigenvalues array

S1  : for i = 1, n_v do
S2  :     for j = 1, n_o do
S3  :         calculate threshold from fmo and eval;
S4  :         if (fmo[i,j] ≥ threshold) then
S5  :             calculate alpha and beta from eval[i], eval[j] and fmo[i,j];
S6  :             for m = 1, n do
S7  :                 a = evec[m,j];    b = evec[m,i];
S8  :                 evec[m,j] = alpha × a + beta × b;
S9  :                 evec[m,i] = alpha × b − beta × a;
S10 :             endfor
S11 :         endif
S12 :     endfor
S13 : endfor
```

Figure 5.6: simplified DIAG part 2 algorithm for complexity analysis

pivoting is used. The eigenvectors of matrix A are obtained as the product of the Householder transformation matrix H and the eigenvectors.

In the main, these algorithms are numerically stable even when some eigenvalues might be degenerate, which is common in chemistry applications.

Now this modified matrix can be used to find eigenvalues efficiently with the QR method. Only the eigenvectors for the occupied orbitals in this respect are calculated. These algorithms have been well studied and their computational complexities are known:

- Householder transformation: $O(n^3)$
- QR-Inverse Iteration: $O(n^2)$
- Calculating n_o eigenvectors: $O(n_o \times n)$
- Transforming n_o eigenvectors: $O(n_o \times n^2)$

The above break-down elaborates that the Householder transformation to a triagonal matrix is the most expensive part of subroutine HQRII. It leads to a computational complexity of $O(n^3)$ in subroutine HQRII.

Performance Estimation

The above computational complexity analysis shows that all the independent time-consuming subroutines are $O(n^3)$ except for DCART. Although DCART takes large portion of CPU time in time profiling analysis, we know the percentage of CPU time requirement by DCART will decline when large data files are used in production. Subroutine DCART is not as critical as the other three time-consuming subroutines. We should parallelize DENSIT, DIAG and HQRII first. According to Table 5.1, the total weighted percentage of the three computationally intensive subroutines is $15.89\% + 64.53\% + 6.01\% = 86.43\%$. The maximum possible speed-up for the input data size used in our time profiling analysis is 7.37.

Appendix D shows the data sizes used in time profiling analysis are roughly 50 molecules. As illustrated in Section 4.1.3, the work load distribution and the maximum possible speed-up changes dramatically when the size of input data changes. By the above result from time profiling analysis and the complexity analysis, we can predict the work load of each part and the maximum possible speed-up for large input data. For example, the weighted work load percentage of the three subroutines will be 98.45% and the maximum possible speed-up is 64.52 if the computational complexity of the non-parallelized part is $O(n^2)$ and the data files are 10 times larger.

5.2 Work Load Distribution

Work load distribution is important to parallel programs not only because imbalance work load lengthens total execution time seriously but also because bad work load distribution can result high communication time. As described in previous chapters, communication costs is an important factor of parallelization. Since we use a distributed memory programming model, frequently transmitting small packets degrades the performance of communication significantly. This seriously affects distributed memory machines since distributed memory machines have higher network set-up time. Packing small pieces of data into a larger chunk reduces the frequency of data communication. The bigger the packet size is, the better the performance we can achieve. Thus, it is useful to aggregate data before injecting it into the network. Sometimes we may need to rearrange the program structure, exchange the inner and outer loops so that the data can be packed and the frequency of communication can be reduced.

Distributing a problem of size N to a p-nodes of a parallel machine always causes some load imbalance unless p evenly divides N. Some nodes receive $\lfloor \frac{N}{p} \rfloor$ work load while the others receive $\lfloor \frac{N}{p} \rfloor + 1$. Load balance is unlikely a serious problem if the size of the problem to be distributed is far bigger than the number of nodes. Therefore we need to maximize the number of distributable computational units. If we have a two-level loop L1 and L2. L2 is inside of L1. L1 and L2 share the same loop body and there is no data dependency preventing us from parallelizing both loops. We can merge the two-level loop into a new loop L3. The size of new loop L3 is usually the product of size L1 and L2 if the range of L2 is not a function of the index of L1. The size of L3 is larger than L1 and L2 and L3 can be distributed more evenly than L1 or L2. Multiple levels of loops will be merged for a better balanced work load distribution.

Load distribution is extra computation cost of parallel programs. There are many complex algorithms that can produce optimized work load distribution schemes for very complex problems. However, other research on communication also points out that a perfectly balanced load may result in high communication contention due to all nodes finishing computation and starting communication at the same time. Similarity, a non-optimized work load distribution scheme may not fully minimize the computation time, but takes less total time by avoiding the communication contention.

Figure 5.7 shows the most straightforward block distribution scheme. Subroutine DISTRIB finds the lower and higher bounds of a range [*LB*, *UB*] for processor *MYID* in a *NPROC*-processor system. A commonly made mistake is shown in Figure 5.7(a). Every node receives the same amount of work load except for the last one. The last node receives as much as *NPROC−1* more work load than other nodes if the size of work load *UB−LB+1* does not divide *NPROC*. The last node will need as many as *UB−LB+1* units of extra time to finish its work. Figure 5.7(b) is similar to Figure 5.7(a) but the last node receives less work load than other nodes. In a word, the small difference makes Figure 5.7(b) distributes work load much better than Figure 5.7(a). All nodes except for the last one receive no more than 1 extra unit of work load. Moreover, in programs that need a host node to do some extra work, this last node may be used as the host node because it can finish its computation earlier than other nodes. The work on the extra host node may be for result collecting. Figure 5.7(c) distributes work load evenly. In short, there is no extra heavy or light work load distributed by the program in Figure 5.7(c). The maximum difference is 1 unit. That is, some nodes finish 1 unit of time earlier than other nodes. Figure 5.7(c) is used when no host node is needed.

Communication is generally an expensive process for a parallel program. A processor may take advantage of a multi-level cache memory to operate at a very high-speed while communication can only move data at lower memory bus and network interface speed. The internal processor clock rates are 5 to 10 times faster than memory bus clock rates. At length, the memory buses are usually several times faster and wider than network interfaces. Moreover, communication driver software may further drag down the performance of communication and enlarge the performance gap between computation and communication. It is true, this large performance gap between computation and communication gives us an extra consideration when we determine the

```
        C      Processor ID starts from 0 to NPROC−1.
        C      In (b), (X+Y−1)/Y returns the ceiling of X/Y

S1  :          SUBROUTINE DISTRIB(MYLB,MYUB,LB,UB)
S2  :          INTEGER MYLB,MYUB,LB,UB,NPROC,MYPID,LEN,SHARE
S3  :          SHARE = (UB−LB+1)/NPROC
S4  :          MYLB = SHARE*MYID+LB
S5  :          MYUB = MYLB+SHARE−1
S6  :          IF (MYID .EQ. NPROC−1) THEN MYUB = UB
S7  :          END
```

(a) Bad distribution

```
S'3 :          SHARE = (UB−LB+1+NPROC−1)/NPROC
S'4 :          MYLB = SHARE*MYID+LB
S'5 :          MYUB = MYLB+SHARE−1
S'5 :          IF (MYID .EQ. NPROC−1) THEN MYUB = UB
```

(b) Good distribution when using last node as host node

```
S''3 :         LEN = UB−LB+1
S''4 :         MYLB = LEN*MYID/NPROC+LB
S''5 :         MYUB = LEN*(MYID+1)/NPROC−1+LB
```

(c) Good distribution

Figure 5.7: Simple block distributions

```
// There are p = 2^m processors, where m in a positive integer
// Processor ID=[0..p−1]

S1 : Find Amax, the local maximum of the portion of array A owned locally;
S2 : for i = m−1, 0, −1
S3 :     if (myid ≥ 2^i) then
S4 :         send data Amax to node myid −2^i;
S5 :     else
S6 :         receive data Atemp from node myid + 2^i;
S7 :         Amax = max(Amax, Atemp);
S8 :     endif
S9 : endfor
```

Figure 5.8: Parallel algorithm that finds a global maximum

complexity of an algorithm.

In some parallel algorithms, the total data to be sent may grow when more processors are used. For example, the algorithm in Figure 5.8 finds the global maximum of array A in a p-processor system. The total data to be sent rises as p rises. The total number of communications is:

$$2^{m-1} + 2^{m-2} + \cdots + 1 = \frac{2^m}{2-1} = 2^m - 1 = p - 1$$

The amount of data to be sent rises solely because more processors are used. The communication cost can rise even when the total data that is sent does not change. For example, distributing an array A to p processors requires $p-1$ sends. Each send requires one network setup time. Lastly, this can be significant if the network setup time is long or p is very large.

The communication costs need to be precisely verified to ensure they do not degrade the performance gained from work load distribution. The communication costs include the communication and synchronization during the computing, initial data distribution, final results collection, and global variable synchronization. Global variables are also parts of input or output arguments. They are easily forgotten since they often do not appear in argument lists. FORTRAN supports equivalent variables, which gives the same variables different names. It may be difficult to identify the global variables if they are equivalent to other variables.

In summary, the procedure described in Chapter 4 identifies the code to be parallelized. However, the method of parallelizing a piece of code also relies on some other factors, especially communication. For programs with stern loops but no data dependencies, we need only to add data distribution and results collection code, and modify the upper and lower boundaries of index variables. For programs with data dependencies, we need to resolve the dependencies between variables by applying the procedures described in the previous chapter and re-write the loop. For programs that cannot be parallelized by modifying parts of them, we may need to find new algorithms to replace the part of program totally.

5.3 Parallelizing Subroutine DENSIT

Subroutine DENSIT computes the density matrix, given the eigenvector matrix, and information about the M.O. occupancy. It takes about 16% of the total CPU time in the time profiling analysis. As analyzed in Section 5.1.3, the computational complexity of subroutine DENSIT is $O(N^3)$, which makes it eligible for parallelizing in the first round.

Subroutine DENSIT does not call any other subroutine. The main loop structure of subroutine DENSIT is shown in Figure 5.4. Four *for*-loops at S2, S3, S5, and S8 form a three level loop. This loop is the computational kernel of subroutine DENSIT. The two innermost loops at S5 and S7 accumulate the values of *sum1* and *sum2*. The results of loop S5 and S7 are used to calculate the density matrix at S11. The true dependency between loop S5, S7 and statement S11 bonds statements S4 through S11 into a non-parallelizable computation unit. Each combination of indices (I, J) is an occurrence of S4-11. Each occurrence of S4-11 produces an element of the density matrix. There is no data dependence between any two occurrences of S4-11.

The skeleton of the loop structure of Figure 5.4 is shown in Figure 5.9(a). The outer two loops S$'$2 and S$'$3 have common loop body. Moreover, the upper bound of loop S$'$3 is the index of loop S$'$2. The two loops S$'$2 and S$'$3 produce $\frac{N\times(N+1)}{2}$ different combinations of the index pair (I, J). We can distribute the work load over either loop S$'$2 or S$'$3. Either way works, but the fact remains that it has two drawbacks:

- The work load is not even for each index i or j. There are N occurrences of S$'$4-6 for $i = 1$ while only 1 occurrence for $i = N$. It is, therefore, difficult to distribute work load evenly.
- The distribution space is small. The size of distribution space is N if we distribute the work load over loop S$'$2. The work load will not be distributed evenly if n is small or the number of processors, p, is large.

We can principally resolve the above two problems by merging the two loops S$'$2 and S$'$3 into one single loop with $\frac{N\times(N+1)}{2}$ iterations. Each occurrence of the new loop contains one occurrence of the loop body S$'$4-6. The size of the distribution space is therefore enlarged. The larger $\frac{N\times(N+1)}{2}$ distribution space size can be distributed more evenly than N.

The parallelized subroutine DENSIT is shown in Figure 5.9(b). A new loop index L is introduced to be the loop index of the merged loop. Since we still need old loop indices I and J to access the eigenvector matrix C, we need a translation from index L to indices I and J to keep I and J synchronized with L.

Analyzing the relation between I, J, and L: I ranges from *1* to N, J ranges from *1* to I, and L is number of all possible combination of I and J.

$(I{=}1,\ J{=}1)\equiv(L{=}1)$

$(I{=}2,\ J{=}1)\equiv(L{=}2),\ (I{=}2,\ J{=}2)\equiv(L{=}3)$

$(I{=}3,\ J{=}1)\equiv(L{=}4),\ (I{=}3,\ J{=}2)\equiv(L{=}5),\ (I{=}3,\ J{=}3)\equiv(L{=}6)$

$\vdots$

$(I{=}N,\ J{=}1)\equiv(L{=}\frac{(N-1)\times N}{2}),\ (I{=}N,\ J{=}2)\equiv(L{=}\frac{(N-1)\times N}{2}{+}1),\ \ldots,\ (I{=}N,\ J{=}N)\equiv(L{=}\frac{N\times(N+1)}{2})$

```
C       FRAC, SIGN, CONST are scalar constant to the loop
C       P is the packed density matrix

S'1 :       L = 0
S'2 :       DO I = 1, NORBS
S'3 :           DO J = 1, I
S'4 :               L = L + 1
S'5 :               Figure 5.4 step S4~S10, O(N)
S'6 :               P(L) = (SUM1*FRAC+SUM2*2)*SIGN
S'7 :           ENDDO J
S'8 :       ENDDO I
```

(a) Sequential

```
P1 :        CALL DISTRIB(MYLB,MYUB,1,NORBS*(NORBS-1)/2)
P2 :        DO L = MYLB,MYUB
P3 :            Calculate I and J values from L using equations 5.3.3 and 5.3.4
P4 :            Figure 5.4 step S4~S10, O(N)
P5 :            P(L) = (SUM1*FRAC+SUM2*2)*SIGN
P6 :        ENDDO L
```

(b) Parallelized

Figure 5.9: Parallelizing DENSIT loop structure

Since the computational space is triangular, the formula of L should be:

$$L = \frac{(I-1)\times I}{2} + J \tag{5.3.1}$$

The range of the inner loop is from 1 to I. The index of the inner loop should be less than or equal to I.

$$0 < J \leq I \tag{5.3.2}$$

From equation 5.3.1, left half of equation 5.3.2, and the fact of $I > 0$, we have:

$$\begin{aligned} (I-1)\times I &< 2\times L \\ (I-\frac{1}{2})^2 &< 2\times L + \frac{1}{4} \\ I &< \sqrt{2\times L + \frac{1}{4}} + \frac{1}{2} \end{aligned}$$

Since I is a positive integer that satisfies the above equation, the final formula for finding I from L is:

$$I = INT\left(\sqrt{2\times L + \frac{1}{4}} + \frac{1}{2}\right) \tag{5.3.3}$$

and the formula for finding J from L and I is:

$$J = L - \frac{(I-1)\times I}{2} \tag{5.3.4}$$

By the above formulas 5.3.3 and 5.3.4. We can distribute the work load over main loop by dividing the range L= 1 to $\frac{NORBS\times(NORBS-1)}{2}$ to all computational nodes and calculate the starting indices of I and J from distributed loop index L.

There is no communication in the middle of computation. There are no global variables either. In actuality, the only communication is the initial data distribution and final data collection. The input eigenvector matrix is used by all processors. It must be broadcast to all processors. The cost of broadcasting an $N \times N$ matrix is $O(N^2)$. The result density matrix P is a packed triangular matrix. The total data item is $\frac{N\times(N+1)}{2}$. Since the result is collected by one node, the communication must be serialized. The total communication cost is, thus, $O(N^2)$.

5.4 Parallelizing Subroutine DIAG

Subroutine DIAG is a pseudo-diagonalization procedure. It generates only the occupied orbitals and ignores small elements so that it runs much faster than a full diagonalization. It is used as a good starting approximation. Diagonalization is the most time-consuming part of the whole MOPAC program. It takes about 65% of total CPU time in our time profiling analysis.

As analyzed in Section 5.1.3, subroutine DIAG can be further divided into two parts. The first part generates the Fock molecular orbital interaction matrix FMO and a threshold $TINY$ from the matrix to be diagonalized A and its eigenvector matrix $EVEC$. The second part then performs a 2 by 2 rotation on the eigenvectors to eliminate the significant elements. The use of the Fock molecular orbital interaction matrix FMO and the threshold $TINY$ generated in part 1 introduces a true dependency between the two parts. This true dependency forces part 2 of subroutine DIAG execute after part 1. Preferably, we need to parallelize these two parts separately. We will refer DIAG1 as the first part of subroutine DIAG and DIAG2 the second part of subroutine DIAG.

The computational complexity of both parts is $O(N^3)$, which makes both parts eligible for parallelization. We will parallelize DIAG1 and DIAG2 in the following sub-sections.

5.4.1 Parallelizing Subroutine DIAG1

As shown in Figure 5.5, the loop structure of DIAG1 contains 5 loops. The loop S2, S3 and S5 forms an $O(N^3)$ loop and loop S2, S9 and S11 forms the other $O(N^3)$ loop. Loop S3 produces a working matrix ws which is used in loop S9. This true dependency forces loop S3 to execute before loop S9.

We cannot simply divide loop S2, S3, S5 and S2, S9, S11 into two three-level loops because the value of the working matrix ws will be re-assigned. Although this output dependency can be resolved by introducing new variables as described in

Section 2.1, we do not want to do that because instead of introducing one variable, it adds another dimension to a matrix. The size of the new matrix will grow in the rate of $\frac{N \times (N+1)}{2} \times N_v \approx O(N^3)$. Consider $N = 1000$, the new matrix will be 1000 times larger than the old one and take a large amount of memory. It is, therefore, not practical. This leaves us no option but to parallelize loop S2.

The skeleton of loop structure of Figure 5.5 is shown in 5.10(a). The parallelization is straightforward. The parallelized DIAG1 is shown in Figure 5.10(b).

There is no communication in the middle of the computation. There are no global variables either. The initial data distribution is straightforward. The matrix to be diagonalized (size $\frac{N \times (N+1)}{2}$), the eigenvalues (size N), and the initial eigenvectors (size N^2) are needed by all nodes. The cost of broadcasting $\frac{N \times (N+1)}{2} + N^2 + N$ data items is $O(N^2)$.

The results produced by the program is the Fock molecular orbital interaction matrix FMO and a threshold $TINY$. Since the Fock molecular orbital interaction matrix, FMO, is generated in parallel, each node holds a slice of the FMO. Since the whole FMO will be needed by all nodes in DIAG2, we need to collect the distributed FMO from all nodes and send it in whole to all nodes. The best way to do the job is letting every node broadcast their own slice of FMO to all other nodes. The cost of broadcasting FMO equals to the size of FMO, which is $O(N^2)$.

The threshold $TINY$ is the maximum of the elements of FMO. Since each node produces a slice of FMO, every node gets only the local maximum of the elements of FMO they hold. The global maximum reduction function in MPI communication library can be used to find the global maximum from the local maximum. The cost of the global maximum reduction on a scalar variable is a constant.

The total communication cost of subroutine DIAG1 is, therefore, $O(N^2)$.

C $NOCC$ (N_o) is the number of occupied orbitals
C $N{-}NOCC$ (N_v) is the number of virtual orbitals

S′1 : $TINY = 0$
S′2 : DO $I = NOCC{+}1,\ N$
S′3 : Figure 5.5 steps S3∼S8 that produce *WS(1:N)*, $O(N^2)$
S′4 : Figure 5.5 steps S9∼S17 that produce *FMO(I,1:I)* and calculate local maximum of variable $TINY$, $O(N \times N_o)$
S′5 : ENDDO I

(a) Sequential

P1 : CALL $DISTRIB(MYLB,MYUB,NOCC{+}1,N{-}NOCC)$
P2 : $TINY = 0$
P3 : DO $L = LB,MYUB$
P4 : Figure 5.5 steps S3∼S8 that produce *WS(1:N)*, $O(N^2)$
P5 : Figure 5.5 steps S9∼S17 that produce *FMO(I,1:I)* and calculate local maximum of variable $TINY$, $O(N \times N_o)$
P6 : ENDDO L
P7 : Find global maximum of variable $TINY$

(b) Parallelized

Figure 5.10: Parallelizing DIAG part 1 loop structure

5.4.2 Parallelizing Subroutine DIAG2

The second part of subroutine DIAG checks on the Fock molecular orbital interaction matrix FMO and performs a 2 by 2 rotation on the eigenvectors to eliminate the significant elements. The algorithm here is simply walking through the entire FMO matrix and performing a 2 by 2 rotation if the value of the FMO element exceeds a certain threshold.

The simplified algorithm is shown in Figure 5.6. The only matrix changed in this part of program is the eigenvector $VECTOR$. As far as possible, it would be easy to parallelize the outer loops if the 2 by 2 rotation did not produce any data dependency on $VECTOR$. Unfortunately, statement S8 in Figure 5.6 uses index j to access the eigenvector matrix in the left hand side and uses both indices i and j to access the same matrix in the right hand side. This causes a true dependency on matrix $VECTOR$ between different occurrences of j and prevents us from breaking loop S2. Statement S9 in Figure 5.6 has the same data access pattern except for using index i in the left hand side. Again, this causes a true dependency on matrix $VECTOR$ between different occurrences of i and prevents us from breaking up loop S1 in any given instance.

The only loop we can try to break up is the innermost loop S6. Fortunately, loop S6 just applies the 2 by 2 rotation to all elements of the vector we want to rotate. It does not matter which element to rotate in a vector. There is no data dependence between different occurrences of the index m.

The skeleton loop structure of Figure 5.6 is shown in Figure 5.11(a). The loop breaking is simply distributing the range of loop S′4. The parallelization of the second part of subroutine DIAG is shown in Figure 5.11(b).

There is no communication in the middle of computation. The communication for data distribution has been covered in the end of DIAG1. The final result is the newly updated eigenvector matrix $VECTOR$. Since every node holds a strip of matrix $VECTOR$, collecting the distributed matrix $VECTOR$ is the main work in results collection. The size of matrix $VECTOR$ is $N \times N$. The total communication cost of

---

```
        C    VECTOR, evec in Figure 5.6, is the matrix that holds eigenvectors
        C    EIG, eval in Figure 5.6, is the array that holds eigenvalues

S'1 :        Figure 5.6 steps S1~S3 loop I, J headers and threshold calculation
S'2 :        IF (FMO(I,J) ≥ threshold) THEN
S'3 :            Calc ALPHA and BETA from EIG(I), EIG(J), and FMO(I,J)
S'4 :            DO M = 1,N
S'5 :                Figure 5.6 steps S7~S9 that does a 2 by 2 rotation on
                         VECTOR(M,I) and VECTOR(M,J)
S'6 :            ENDDO M
S'7 :        ENDIF
S'8 :        Figure 5.6 steps S12~S13 loop I, J tails
```

(a) Sequential

```
P1 :         CALL DISTRIB(MYLB,MYUB,1,N)
P2 :         Figure 5.6 steps S1~S3 loop I, J headers and threshold calculation
P3 :         IF (FMO(I,J) ≥ threshold) THEN
P4 :             Calc ALPHA and BETA from EIG(I), EIG(J), and FMO(I,J)
P5 :             DO L = MYLB,MYUB
P6 :                 Figure 5.6 steps S7~S9 that does a 2 by 2 rotation on
                         VECTOR(L,I) and VECTOR(L,J)
P7 :             ENDDO L
P8 :         ENDIF
P9 :         Figure 5.6 steps S12~S13 loop I, J tails
```

(b) Parallelized

Figure 5.11: Parallelizing DIAG part 2 loop structure

the DIAG2 is $O(N^2)$.

There is another operative way to parallelize the second part of subroutine DIAG. The major problem that forces us to break the innermost loop S6 of Figure 5.6 is the 2 by 2 rotation. Everything else does not cause any exclusive data dependency on breaking the two outer loops S1 and S2. By introducing a matrix *rotate* to keep the result of threshold checking and expand the scalar variable *alpha* and *beta* to arrays, we can divide the algorithm in Figure 5.6 into two disjoint loop sets as shown in Figure 5.12.

Loops $\hat{\text{S}}1$ and $\hat{\text{S}}2$ in Figure 5.12 can easily be parallelized since there is no data dependency on any variable used inside of the loops. Although loop $\hat{\text{S}}12$ and $\hat{\text{S}}13$ still suffer from the data dependency of $VECTOR$ and only loop $\hat{\text{S}}15$ can be parallelized, the parallelization of loops $\hat{\text{S}}1$ and $\hat{\text{S}}2$ gives us advantage for the parallel computation.

However, the alternative needs to redistribute the matrix *alpha*, *beta*, and *rotate*. It introduces an extra $O(N^2)$ communication cost, and parallelizes a portion of the routine that has a computational complexity of $O(N^2)$ instead of $O(N^3)$. This part is not a dominant part of the whole application. The percentage of CPU time required by this part is reduced when N grows and the advantage shrinks as well. This alternative will be beneficial only for small data sizes.

5.5 Parallelizing Subroutine HQRII

A segment of program code recognized as a well known algorithm may not need to be parallelized. Many commonly used algorithms have already been parallelized and optimized. It is a good idea to use those optimized packages if they are applicable.

As analyzed in Section 5.1.3, HQRII is a fast standard eigensolver based on the Householder-QR-Inverse Iteration method. Eigensystems have been actively studied for some time. and many parallel eigensolvers have been developed to solve different categories of eigen-problems. These pre-existing packages have been effectively optimized and well studied. Their characteristics are well known. To find a well

// *fmo* : Fock molecular orbital interaction matrix from part 1
// *evec* : eigenvector matrix
// *eval* : eigenvalues array

$\hat{S}1$: for $i = 1,\ n_v$ do
$\hat{S}2$:   for $j = 1,\ n_o$ do
$\hat{S}3$:   *calculate threshold from fmo and eval;*
$\hat{S}4$:   if (*fmo[i,j]* $\geq$ *threshold*) then
$\hat{S}5$:   *calculate alpha[i,j] and beta[i,j] from eval[i], eval[j] and fmo[i,j];*
$\hat{S}6$:   *rotate* = TRUE;
$\hat{S}7$:   else
$\hat{S}8$:   *rotate* = FALSE;
$\hat{S}9$:   endif
$\hat{S}10$:   endfor
$\hat{S}11$: endfor
$\hat{S}12$: for $i = 1,\ n_v$ do
$\hat{S}13$:   for $j = 1,\ n_o$ do
$\hat{S}14$:   if (*rotate*) then
$\hat{S}15$:   for $m = 1,\ n$ do
$\hat{S}16$:   *a = evec[m,j];   b = evec[m,i];*
$\hat{S}17$:   *evec[m,j]* = *alpha* $\times$ *a* + *beta* $\times$ *b;*
$\hat{S}18$:   *evec[m,i]* = *alpha* $\times$ *b* $-$ *beta* $\times$ *a;*
$\hat{S}19$:   endfor
$\hat{S}20$:   endif
$\hat{S}21$:   endfor
$\hat{S}22$: endfor

Figure 5.12: Modified DIAG part 2 loop structure

developed eigensolver and adopt for use in MOPAC is more efficient in both cost and performance. Allen and Bush have written a very good survey purely about the parallel implementations of eigensolvers [5].

Parallel Eigensolver

The eigensolver algorithms broadly fall into three camps:

Direct methods Directly apply Householder reduction to tridiagonal form followed by QL or QR diagonalization.

Jacobi type iterative methods Repeated application of orthogonal Jacobi rotations to bring the matrix to diagonal form quickly.

Conjugate gradient type iterative methods They are generally used on solving large sparse systems of linear equations. They are also applied to the eigenvalue problems. It includes Lanczos, Arnoldi and Subspace Decomposition methods.

Allen and Bush [5] gave a good survey on parallel implementations of eigensolvers. The parallel eigensolver implementations surveyed are:

- Direct methods
  - Cleve Moler and Richard Chamberlain's EISCUBE released by Intel is the oldest parallel Householder-QR method. No orthogonalization is done in EISCUBE.
  - Scalable Linear Algebra Package (ScaLAPACK) [16] is the most popular linear algebra package. It has been ported to most MPPs like Cray T3D, the IBM SP2, Thinking Machine CM5 and workstation clusters. It does not do orthogonalization.
  - PeIGS [34, 35, 36] released by PNNL is a collection of commonly used linear algebra routines for solving dense real orthogonalization eigensystems.

Orthogonalization in one sense is guaranteed implicitly by using inverse iteration.

  - HJS algorithm [48] is another eigensolver for solving dense symmetric systems developed by Hendrickson, Jessup and Smith of Sandia National Laboratory. It uses a modified Gram-Schemidt procedure thereupon to achieve the orthogonality of eigenvectors.

- Jacobi type iterative methods
  - Ian Bush's one-sided block Jacobi BFG
  - Pittsburgh Supercomputer Center 2-sided Jacobi PJAC (can also do complex symmetric or Hermitian)
  - Parallel NAG library
- Conjugate gradient type iterative methods
  - Kristi Maschhoff's parallel ARPACK
  - PRISM project Symmetric Invariant Subspace Decomposition Algorithm (SYISDA) invariant subspace decomposition approach
  - Chris Potter's parallel divide and conquer algorithm
  - Soren Paedkjar's Spectral Transform Lanchos Method

Allen's experiments compare the execution time of representative routines for a 1024 by 1024 matrix on various parallel machines including Cray T3D and IBM SP2. The results show that the Jacobi methods scale better than QL methods because the QL factorization does not scale very well. For example, PDSPEV scales poorly compared to EISCUBE or ScaLAPACK PDSYEVX due to the Gram-Schmidt orthogonalization, which being essentially a QL step.

The other reason that the Jacobi method scales better is because the serial version of Jacobi method runs slower than the direct method algorithms. The results of course also show that large processor counts favor the best scaling algorithms while small processor counts favor the best serial algorithms. The reason is that large processor counts means smaller data chunks while small processor counts means larger data chunks. Allen concludes overall that for an order N matrix over a p processor system, one-sided Jacobi will be the best for $\frac{N}{p} < 8$ and the direct methods will win for $\frac{N}{p} > 8$.

It is difficult to find a capable parallel machine with a large enough processor count for the matrix sizes of MOPAC. In Allen's example, $N = 1024$, Jacobi methods need $P > 128$ to out-perform direct methods. Consider $N = 5000$, Jacobi methods cannot match direct methods until $P > 640$, which is about the largest substantial configuration available today. Direct methods will better fit into MOPAC rather than the Jacobi methods.

Direct methods

The traditional method for determining the eigensystem of a real, dense symmetric matrix A employs a three-step technique. First, A is reduced to tridiagonal form using a series of Householder transformations (reflections). Next, the eigensystem of the tridiagonal matrix T is computed. The eigenvalues of T are the same as those of A, while the noteworthy eigenvectors of A are found in the third step by back-transforming the eigenvectors of the tridiagonal matrix T via reduction transformations.

Although the sequential algorithms for the tridiagonalization and back-transformation steps are well understood, tridiagonal eigensolvers continue to be a critical area of active research. With respect to parallel algorithms the efficiency criteria and therefore the situation is somewhat reversed. There has been much work on the tridiagonal eigensolver since Huang's 1974 study of multi-section methods on the ILLIAC IV. This continued with the implementation of several other tridiagonal eigensolvers on a variety of shared and distributed memory systems [13, 32, 64].

There is still a problem of assuring orthogonality which is not fully worked out and is of particular importance in this application. Most commonly, the parallel implementations use bisection to find the eigenvalues, followed by inverse iteration to get the vectors. However this does not guarantee orthogonality if the values are clustered. In this case a modified Gram-Schmidt procedure is conventionally needed to re-orthogonalize. However if clustered values belong to different processors, this involves extra communication [35].

Work on parallel tridiagonalization and back-transformation began more recently and has been confined to distributed memory systems. Chang et al. [23] described, but not implemented, a parallel version of the Householder algorithm. Several implementations of the traditional tridiagonalization algorithm have appeared in [24, 48, 64].

Whilst the QL method is often the worthwhile method of choice on sequential machines when the system is not very diagonally dominant, the QL and QR methods are generally considered to be problematic on a parallel computer because of poor scaling. Up to 95% of the time spent in the method is in the Householder reduction to tridiagonal form.

However the suggested QL and QR methods produce eigenvalues which are not ordered. This means that closely similar (or even degenerate to machine precision) values may be on different processors. We cannot guarantee orthogonality of eigenvectors from similar eigenvalues if they are computed on different processors, as usually eigenvectors are orthogonal to the previously found one on the same processor. In general, an explicit parallel Gram-Schmidt orthogonalization step would be required. This could result in poor load balancing in the full algorithm. No orthogonalization is done in EISCUBE or ScaLAPACK, which therefore do not guarantee to give orthogonal eigenvectors.

Allen's experience also shows that among the direct methods on systems with low latency communications, e.g. Cray T3D, block cyclic distribution as in ScaLAPACK using BLACS is an advantage. However on other systems with higher communication latency, e.g. the IBM SP2, EISPACK or PeIGS wins. After all, since distributed memory parallel computers usually have higher communication latency, PeIGS becomes the best choice.

PNNL PeIGS

PeIGS [31] is a collection of linear algebra subroutines for computing the eigensystem of the real standard problem $Ax = \lambda x$ and the general eigensystem $Ax = \lambda Bx$ where A and B are dense real symmetric matrices with B positive definite and λ is an eigenvalue corresponding to eigenvector x.

PeIGS was developed by David Elwood, George Fann, and Richard Littlefield of Pacific Northwest National Laboratory. Its functions are documented in the PeIGS manual [34]. For the moment, there are also papers benchmarking several methods tested at PNNL [35, 36].

PeIGS can solve problems in linear algebra that are associated with the general symmetric and the standard symmetric eigensystem problems. PeIGS can also handle associated computations such as the Choleski factorization of positive definite matrices in packed storage format and linear matrix equations involving lower and upper triangular matrices in distributed packed form.

PeIGS was written to use BLAS (Basic Linear Algebra Subprograms) library calling interfaces for most of its computation. PeIGS comes with a general BLAS library. Users of machines that equip with optimized BLAS libraries can use them to accelerate the computation.

PeIGS uses the SPMD (Single Program Multiple Data) programming model. It was designed for distributed memory parallel computers with high startup communication time like workstation clusters. PeIGS supports Intel iPSC/860 and Touchstone Delta native communication libraries as well as the TCGMSG and MPI communication libraries.

For the most part, the computational kernel of PeIGS is its QL routine PDSPEV. The routine PDSPEV() is a parallel implementation equivalent to the LAPACK routine DSPEV().

The diagonalization method proceeds by Householder reduction to tridiagonal form using a subordinate parallel version of the EISPACK routine TRED2(). Processor 0 broadcasts the diagonal and sub-diagonal of the tridiagonal matrix to all processors, which each used to compute a set of eigenvalues using DSTEBZ() from LAPACK. This uses Sturm sequencing and bisection, some redundant computation is performed if there are closely clustered eigenvalues, just by having the intervals overlapping slightly.

Processor 0 then collects all eigenvalues and block numbers and broadcasts them to all processors. Inverse iteration is used with a modified Gram-Schmidt procedure for re-orthogonalization to find the orthogonal eigenvectors [35]. This gives results comparable to LAPACK routine DSTEIN(). Standard back-transformation with the Householder matrix gives the eigenvectors of the original problem.

PeIGS uses a flexible column distribution with packed storage for real symmetric matrices. The user specifies the processor which stores each column. Basic linear algebra operations, such as triangular matrix multiplication are performed in a panel-block systolic loop. The program is capable of taking data in a column(row)-wise distribution with processor mapping specified in integer vectors. It can also handle data in compact lower triangular format, which is useful. This is done by pointers and dynamic memory allocation in the C routines. Scratch memory of about $\frac{4N^2}{p}$ is required.

The driver routine for the general eigenvalue problem is PDSPGV(). The problem is reduced to a standard one by Cholesky factorization of B, so $B = LL^T$ where L is a lower triangular matrix. This exists providing B is symmetric and positive definite, which guarantees real square roots. We have: $Cx = \lambda x$ where $C = L^{-1}AL^{-T}$.

The parallel routine CHOLESKI() uses a sub-matrix algorithm and instead of doing two solutions to form C the L is inverted using a routine INVERSEL() and a lower triangular and upper triangular matrix multiplication is performed. The back-transformation step must also include L^{-1} for the general case.

Adapting PeIGS to MOPAC

As described above, although the modified Gram-Schmidt procedure involves extra communication which reduces its performance, the performance is still comparable with other packages that can take advantage of block-cyclic data decomposition on Cray T3D. On distributed memory machines like the IBM SP2, it is the best choice. Especially when the problem size is much larger than the number of processors.

Parallel MOPAC uses the FORTRAN language and PeIGS supports both FORTRAN and C interfaces. Parallel MOPAC uses MPI to communicate between nodes and PeIGS supports MPI as well. PeIGS matches our environment well. At this stage, the specific work needed is to find the applicable functions in PeIGS and adapt them into parallel MOPAC.

PeIGS supports various eigensolvers for different uses. The eigensolver needed in HQRII is a general purpose one for real symmetric matrices, and the PeIGS function `pdspevx` matches this. Subroutine `pdspevx` is a very general purpose eigensolver and the calling interface of `pdspevx` is shown in Figure 5.13. To be able to cover most uses, `pdspevx` requires as many as 21 arguments. Not all arguments are useful for MOPAC, in fact only a few arguments are useful. Some arguments can be assigned constant values, while some arguments are not used. The arguments of `pdspevx` will be adjusted to make HQRII work as desired.

The useful `pdspevx` arguments are *n, vecA, mapA, meigval, vecZ, mapZ,* and *eval.*

- *n*: the number of rows and columns of the matrix A.
- *vecA*: lower triangular part of the i-th column of A which is owned by this processor. The columns of A owned by this processor are determined by *mapA*.
- *mapA*: the id of the processor which owns column i of the A matrix, $i = 0..n-1$.
- *meigval*: the total number of eigenvalues found.
- *vecZ*: i-th eigenvector (as determined by the exit values in *mapZ*) owned by this processor. The eigenvectors are normalized.

```
      subroutine pdspevx( ivector, irange, n, vecA, mapA,
     $   lb, ub, ilb, iub, abstol,
     $   meigval, vecZ, mapZ, eval, iscratch, iscsize,
     $   dblptr, ibuffsize, scratch, ssize, info)

      integer           ivector, irange, n, mapA, ilb, iub, meigval,
     $                  mapZ, iscratch, iscsize, ibuffsize,
     $                  ssize, info

      double precision  lb, ub, abstol, eval(*), scratch(*)
     $                  vecA(*), vecZ(*), dblptr(*)
```

Figure 5.13: The PeIGS subroutine pdspevx user interface

- *mapZ*: the id of a processor which owns the memory for storing the i-th eigenvector, $i = 0..n-1$.
- *eval*: the eigenvalues of the matrix in no particular order.

The input arguments *vecA* needs to be distributed before calling **pdsvex** and the result eigenvectors *vecZ* needs to be collected after calling **pdsvex**. An interesting problem is that the definition of the canonical form of the input matrix *vecA* in PeIGS is different from that in MOPAC. One uses row-major while the other uses column-major. It is revealing because that PeIGS was implemented in C and the FORTRAN user interface of PeIGS is just a wrapper of this C interface. An adjustment is needed before **pdsvex** is called. This adjustment takes $\frac{n\times(n+1)}{2}$ steps. It can be distributed to the p processors and takes $\frac{n\times(n+1)}{2\times p} \approx O(\frac{n^2}{p})$, which is smaller than the computational complexity of **pdsvex** and is negligible when the size of the input matrix n gets large.

5.6 Integration and Visualization

A parallelized legacy application contains at least two parts, the parallelized computation engine and the old sequential part. Besides the two parallel and sequential computation parts, our parallel MOPAC implementation adds a molecule structure rendering AVS module and an X-windows display module. The four parts are implemented as independent modules and can run nonetheless on networked heterogeneous machines, networked homogeneous machines, or on a single machine. AVS, as described in Appendix B, is a data visualization system that better provides easy user interface to show visual data for scientific applications. Besides the data visualization, AVS remote module can be used to integrate programs running on different platforms across networks. We will describe how we integrate the MOPAC modules using AVS remote modules.

MOPAC system configuration

The most important part is the parallel computational module. The parallel computational module contains the parallel versions of the computationally intensive subroutines. A powerful parallel computer with high speed, low latency communication system is recommended to serve as the Parallel Computation Host (PCH). An MPI communication library is required to supply the communication functions needed by the parallel computation module. The public domain mpich works for most parallel computers as well as workstation clusters. It is easy to obtain, configure, and install. However, a vendor supplied MPI implementation is preferred since the vendor supplied MPI implementation can take better advantage of the communication hardware and boost the communication performance. The SGI MPI runs as much as 1.8 times faster than mpich for some communication functions. Similarly, a vendor supplied BLAS is preferred since it is optimized for the maximum performance of the floating point processing unit.

The sequential computation module is the original MOPAC one with the ability to use the parallel computation module to speed up the computation time of the computationally intensive portions. The sequential versions of the parallelized subroutines are available to run small calculations. The main task of the sequential computation module is providing the user interface, file I/O, error handling, and other non-computationally intensive portions. A general purpose workstation with a local hard disk is good enough to serve as the Sequential Computation Host (SCH).

The output of MOPAC is only text files. Some results such as the final energy can be easily read from the output file. However, other results such as molecule structures can only be output in coordinates. Users needed to use their imagination to visualize the molecule structures from the coordinates. Parallel MOPAC adds the molecule structure rendering AVS module to provide a better compensated user interface. AVS is a very powerful graphic rendering system. It supports three dimensional viewpoints and animations. A fast workstation with the AVS system is required to serve as the AVS Host (AVSH).

The graphic data generated by AVS can be displayed on a number of graphic devices. The most commonly used graphic displaying tool is X-windows. An X-server with high resolution true color display is required for AVS to display pictures properly. We can use AVSH to run the X-server. However, AVS is licensed software which cannot run outside of registered machines. In that case, a separate X-windows Display Host (XDH) is needed to display the AVS output.

Figure 5.14 shows the integrated MOPAC system configuration. The four modules can run on four different hosts to improve the performance.

Coarse grained communication between MOPAC modules

Like the nodes of a parallel computer, the four MOPAC modules need to exchange information. The communication between modules is similar in many ways to the communication between the nodes of a parallel computer. The major difference between these two kinds of communication is that the communication between MOPAC modules is coarse grain while the communication between node programs is medium or fine grained.

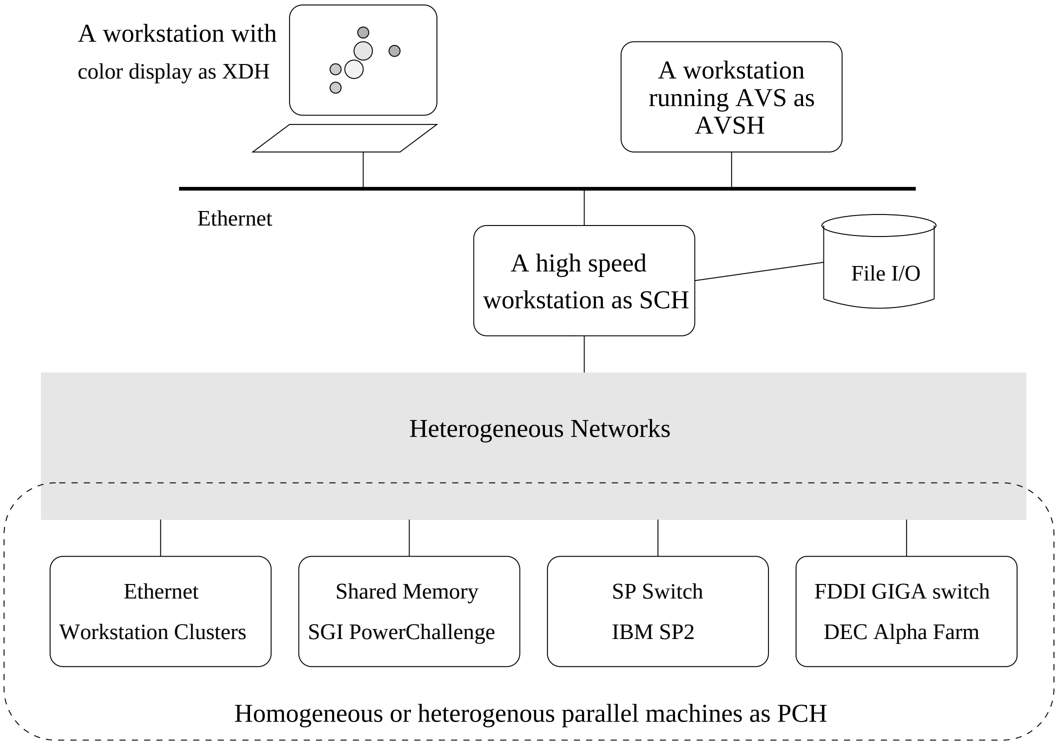


Figure 5.14: Parallel MOPAC system configuration

The frequency of coarse grained communication is lower than fine grained communication. The amount of data is often smaller, too. The performance of the communication subsystem does not affect overall performance of coarse grained programs as much as those of fine grained programs. However, due to the difference in functionality requirements of the four modules, it may be a desirable situation to run the MOPAC modules on different platforms. Heterogeneous machine configurations require a standard, widely supported communication system to pass data between coarse grained modules. Compatibility is the most important issue for coarse grained programs.

TCP/IP is the most popular and inter-operable communication protocol. A special virtue, it is supported on almost all modern computers. Many system calls and libraries are made available for communication programming. Using TCP/IP avoids incompatibility problems.

AVS not only supports visualization but also supports inter-processor communication. This means that AVS remote modules can be distributed on different machines working cooperatively. Each AVS module is represented by a rectangular icon with access points called ports. Ports are colored regarding their data types. Several AVS modules can be connected via ports to form a AVS network to provide the required functions. Making data a connection between two AVS modules is as easy as clicking on an output port of one module and dragging it to the input port of another module. AVS automatically creates a TCP connection between the two modules and everything written to the output port of the first module is transmitted through the TCP connection to the input port of the second module transparently. If the two modules are on the same machine, AVS is smart enough to upgrade the connection to use shared memory inter-process communication.

AVS eases the implementation of the communication for the coarse grained modules. Unfortunately, it is licensed software and mainly used on workstations. We can use it to implement the data connections between all MOPAC modules except for the one between the MOPAC sequential computation module and parallel computation module. We use a Berkeley socket to build TCP connections between the MOPAC sequential computation module and parallel computation module. The communication handling routines of these two modules are also smart enough to use faster shared memory inter-process communication if they detect the MOPAC sequential module runs on a common node of the parallel computer. The node is then assigned to be the communication host to distribute data and collect results.

Figure 5.15 shows the communication between the MOPAC coarse grained modules. Data flows between the MOPAC mopacavs module, the MOPAC geomcntl module, and the AVS geometry viewer module are unidirectional. Communications between these modules are handled by AVS. Communications between the MOPAC mopacavs module, the sequential processing, and the parallel computation engine are bidirectional. Berkeley sockets are used to transmit potential data between these modules. Faster shared memory inter-process communication is used automatically if possible.

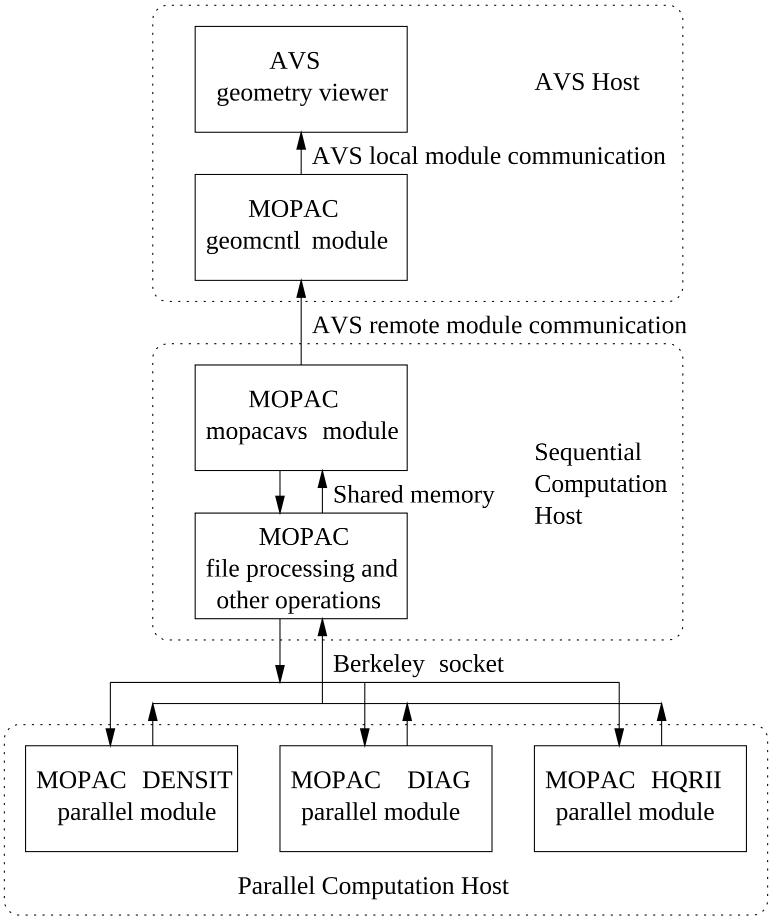


Figure 5.15: Parallel MOPAC communications between coarse grained modules

MOPAC Visual Data

The most important results in MOPAC are the energy and molecule structures. Energy can be specified by numbers while molecule structures are better viewed in graph. The parallel MOPAC graphic user interface consists a control panel and several display windows.

The control panel is implemented by using AVS widgets. Users can click on the buttons to input data file, specify the number of processors to use, select molecule to display, control the viewpoint of molecules and make a slide show if the molecule structures is a series of change of a molecule.

Molecule structures are specified in internal and Cartesian coordinates. Cartesian coordinates are easier for specifying graphic data. MOPAC has subroutines to convert internal coordinates to Cartesian coordinates. Cartesian coordinates are re-organized into AVS 1-dimensional 3-vector uniform double fields. A corresponding atom name is specified as the order in the periodic table. They are kept in AVS 1-dimensional scalar uniform integer fields. The bonds between atoms are specified in AVS 1-dimensional 2-vector uniform integer fields.

AVS module geomcntl converts the molecule structures into a list of graphic objects of type GEOMedit_list which can be drawn on the screen by AVS build-in geometry viewer. Atoms are shown in different sizes and colors to indicate their weights and the column in periodic table. AVS module, geomcntl, converts the molecule structures based on the parameters specified by the MOPAC graphic control panel. Molecule structures can be rotated, moved and enlarged. The user can change the viewpoint to see the molecule structures from different angles or take a close-up view. The light sources can also be changed to make the molecule structures look much clearer to users.

The parallel MOPAC graphic user interface and sample visual data can be found in Section 6.1.