Parallel Computing in Chemistry

Rajarshi Guha
IIT Kharagpur

March, 2001

1 Introduction

This paper intends to discuss the role of parallel computing in computational chemistry. As the terminology implies computational chemistry is heavily dependent on the computational power available. Much of the field involved running programs and applications on single processor systems. Even though processor power has improved steadily (Moores Law [**1**]) over the years, the need for more CPU power has driven researchers to take advantage of the parallel paradigm.

When one thinks of parallel processing images of large, room sized supercomputers come to mind. Companies like Cray [**2**], IBM [**3**], SGI [**4**] provide such machines which are undoubtably hugely powerful. However these are not the only types of parallel machines available. In later sections the different types of parallel machine designs will be briefly discussed along with their advantages and disadvantages. However parallel machines are not the only component to a successful setup for computational chemistry (and for that matter, any field). The software techniques also play a vital role in such applications. In subsequent sections I discuss the available software and applications.

2 What is Parallel Processing?

Parallel processing refers to the concept of speeding-up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across

n processors *might* execute n times faster than it would using a single processor. Here I stress on *might* as there are several factors which can cause a given application to run more slowly when distributed among several processor s than when run on a single processor.

The design of a parallel program for a given problem depends on several factors. These include

- Type of parallel hardware available
- Nature of the problem
- Financial constraints

Coding a problem for a parallel machine is not a trivial task. As such there are no guides or cookbooks detailing how a problem may be parallelized. In general each problem must be studied on an individual basis and in some cases can require extensive recoding from the original serial implementation.

3 Types of Parallel Hardware

Parallel hardware has developed enormously. What started with a few companies developing highly specialized parallel hardware (typically referred to as supercomputers) costing hundreds of thousands, if not millions, of dollars has now burgeoned into a scenario whereby individuals can now assemble supercomputer class machines within a few thousand dollars.

The major difference between normal (serial) computers and parallel computers is that the latter will have more than one processor. The multiple processors may constitute a single machine or may be in the form of a distributed network. Parallel machines can be divided into two main types classified as *SIMD* and *MIMD*. These stand for *Single Instruction Multiple Data* and *Multiple Instruction Multiple Data* respectively. In the SIMD case all processors execute the same operation at the same time, but each processor is allowed to operate upon its own data. This model easily corresponds to the concept of performing the same operation on every element of an array, and is thus often associated with vector or array manipulation.

SIMD machines can again be grouped into two types:

- Processor array architecture

- Vector pipelines

Examples of the former machines are the Connection Machine (CM2), MasPar MP-1 (DEC mpp-12000) and the ICL DAP in which there are a very large array of processing elements usually handling only 1 or 4 bits of data at a time. These units are coordinated by a central dispatching unit

Vector machines have a fairly small number (usually less than 32) of very powerful execution units, called vectors because they are specially designed to be able to handle long strings ("vectors") of floating point numbers. The main CPU handles dispatching jobs and associated data to the vector units, and takes care of coordinating whatever has to be done with the results from each, while the vectors themselves concentrate on applying the program they've been loaded with to their own unique slice of the overall data. Examples of the such machines are the IBM 9000 and Cray YMP.

In the MIMD model on the other hand, each processor is essentially acting independently. This model corresponds to the concept of decomposing a program for parallel execution on a functional basis; This is a more flexible model than SIMD execution, but a downside is the presence debugging nightmares called *race conditions*, in which a program may intermittently fail due to timing variations reordering the operations of one processor relative to those of another. An example of this type of machine is the Beowulf clusters (also termed as *Network Of Clusters* or *NOW*). There is another model termed as *SPMD* or *Single Program, Multiple Data* and is a restricted version of MIMD in which all processors are running the same program. However unlike SIMD, each processor executing SPMD code may take a different control flow path through the program.

An important term that comes up when discussing parallel hardware is *SMP* or *Symmetric Multi Processing*. An SMP system is one that contains 2 or more processors on the motherboard. The main feature of such system is that it does not involve proprietary hardware CPU's or specialized network connections. If the motherboard and OS are SMP aware then the system can handle parallel applications.

4 Cluster Computing

The major disadvantage of using parallel supercomputers is that one is tied to a specific vendor. Any refinements or improvements to the hardware or the controlling software must be carried out with the vendors help. In

many cases only some specific applications are ported to the platform in question. A major point against such machines is the extremely high cost of the machine. This is not to say that such machine are unpopular. For certain problem areas and with the proper funding use of such machines can greatly speed up work. However for more modest applications and where funds are limited cluster computing is a very economical and efficient alternative.

Compute clusters basically consist of off the shelf PC's interconnected via a high speed network. There is usually a single node which acts as the controller, the remaining nodes acting as slaves. One of the most popular setups are Beowulf [**5**] clusters in which the nodes run a patched version of the Linux kernel. Some of the advantages in using clusters are noted below.

- Each of the machines in a cluster can be a complete system, usable for a wide range of other computing applications.
- Most of the hardware for building a cluster is being sold in high volume, with correspond dingly low *commodity* prices as the result.
- Cluster computing can scale to very large systems.With a little work, hundreds or even thousands of machines can be networked to form a cluster whose performance approaches that of conventional supercomputers.
- The fact that replacing a bad (faulty) machine within a cluster is trivial compared to fixing a partly faulty SMP yields much higher availability.
- There is quite a lot of software support that will help achieve good performance for programs that are well suited to this environment. Furthermore much of this software can be obtained freely and is open source in nature.

5 Programming in Parallel

Up to this point I've described the hardware that is involved in parallel applications. But hardware alone doesn't get you results. Software for parallel applications can make or break a parallel project. In this section I'll be restricting the discussion to cluster computing on Linux - other platforms are quite proprietary and can more information can be obtained from their web sites.

When designing parallel algorithms the implementation usually involves the use of libraries which provide parallel constructs. Parallel programs are different from serial programs in that the code must be *parallelized*. Essentially, this means that various parts of the problem which may be computed independently should be split up into independent sections. These sections can then be distributed to the compute nodes. There should be a controlling code which coordinates the activities of the individual nodes and synchronizes communications between them. In addition there should be code which will collect the data from the individual nodes and combine them to give the final result. In the Linux world there are two main libraries which allow the programmer to implement parallel algorithms: *PVM* (Parallel Virtual Machine) [**6**] and *MPI* (Message Passing Interface) [**7**] [**8**]. Both these libraries provide fundamental operations for parallel programming. The major difference between the two libraries is that the latter is based on the concept of virtual machines while the latter is based on a message passing mechanism between cooperating processes.

However even with underlying libraries, the program must be parallelized. This job should be done by the compiler; however compiler technology has not yet reached the stage where a compiler can generate truly efficient parallel code. In general explicitly implementing parallel constructs by hand (using the various constructs provided by the libraries) is still the best way to go. However there several efforts to produce good quality parallelizing compilers. A few are given below:

- HPF (High-Performance Fortran) [**9**], is essentially the enhanced, standardized, version of what many of us used to know as CM Fortran or Fortran D; it extends Fortran 90 with a variety of parallel processing enhancements, largely focussed on specifying data layouts.

- GLU (Granular Lucid) [**10**] is a very high-level programming system based on a hybrid programming model that combines intensional (Lucid) and imperative models. It supports both PVM and TCP sockets.

- Jade [**11**] is a parallel programming language that extends C to exploit coarse-grain concurrency in sequential, imperative programs. It assumes a distributed shared memory model.

- pC++/Sage++ [**12**] is a language extension to C++ that permits data-parallel style operations using "collections of objects" from some base

"element" class. It is a preprocessor generating C++ code that can run under PVM

Of course these are not the only compilers available. A number of commercial compilers especially for HPF and C based languages are available (notably the Portland Group [**13**] compilers).

6 Parallel Performance

One of the major questions that arise is whether a problem is parallelizable or not. The answer to this question is heavily dependent on the nature of the problem and also the efficiency to be gained from such parallelization. Below I look at some of the factors that can affect the expected speedup of parallelized programs.

Software Overhead

Even with a completely equivalent algorithm, software overhead arises in the concurrent (parallel) implementation. (e.g. there may be additional index calculations necessitated by the manner in which data are "split up" among processors) i.e. there is generally more lines of code to be executed in the parallel program than the sequential program.

Load Balancing

Speedup is generally limited by the speed of the slowest node. So an important consideration is to ensure that each node performs the same amount of work. i.e. the system is load balanced.

Communication Overhead

Assuming that communication and calculation cannot be overlapped, then any time spent communicating the data between processors directly degrades the speedup. (because the processors are not calculating).

Nature of the Interconnect

When the code may be parallelized there may be large amounts of communication (data transfer, control messages etc) between individual units. In such cases the speedup to be obtained from parallelization becomes limited by the nature of the network connecting the individual processing units. Such problems can be alleviated to some extent by using high speed interconnects (such as Myrinet for Beowulf clusters).

An important law of parallel programming known as Amdahl's Law [**14**] states that

> the speedup of a parallel algorithm is effectively limited by the number of operations which must be performed sequentially, i.e its Serial Fraction

As a result of this Law even problems which have apparently been parallelized may not exhibit the expected speedups due to the presence of serial code. In many cases this severely limits the speedup obtainable from parallelization.

7 Applications to Chemistry

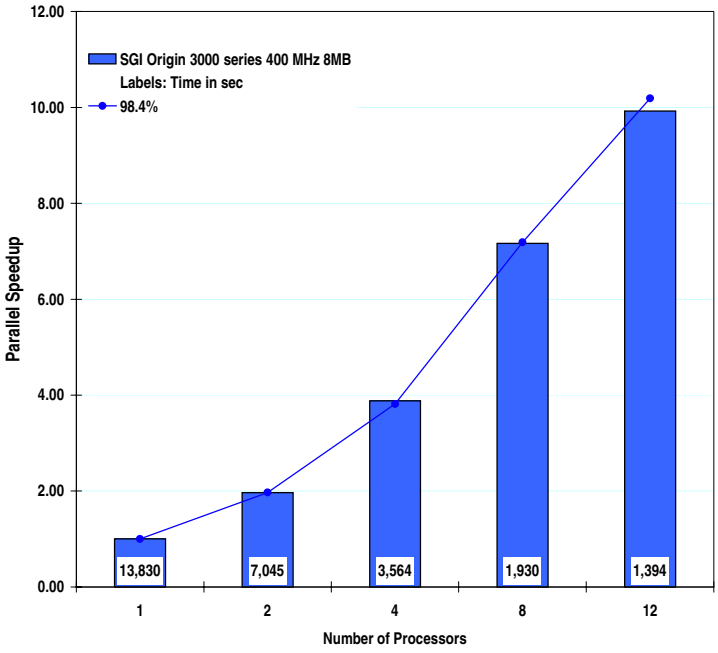
There are a large number of centers around the world using parallel techniques for computational chemistry applications. Both university and commercial concerns have projects and applications that are designed for parallel processing. Among the commercial companies a notable example is SGI [**4**] which produces parallel processing hardware as well as applications for their hardware. These applications cover quantum chemistry, drug design and computational biology [**15**].

There are quite a few examples of computational chemistry software that is designed for parallel processing. Notable examples include Gaussian [**16**], GamessUS [**17**], MOPAC [**18**]. An example of the speed ups obtained from an increase in processor for a Gaussian calculation can be seen in the adjoining graphic taken from a SGI report [**22**]

Gaussian 98 Parallel Performance (1)

α-pinene $C_{10}H_{16}$
B3-LYP Frequency Calculation
basis: 6-31G(d)
182 basis functions

On an SGI 3000 series using 12 MIPS R12000 400 MHz processors, the execution runs close to 10 times faster than on one processor, resulting in a significant reduction of execution time, from close to four hours to just over 23 minutes. The case followed the 98.4% Amdahl's Law curve.



MPQC (Massively Parallel Quantum Chemistry) [**21**] is a quantum chemical application designed from the ground up to be highly parallelized. It is based on MPI and is highly configurable and extendable.

The parallel version of Gaussian uses the Linda batch distribution software. Most universities use Beowulf clusters as their underlying hardware (due to cheapness, low maintainence hassles and availability). MPI or PVM is used as the underlying communication protocol. Apart from quantum chemistry applications molecular dynamics simulations is an area which is usually highly amenable to parallel encoding. Examples of parallelized MD codes include NAMD [**19**] and Moldy [**20**]

An example of the of the speed up obtained on a Beowulf cluster can be seen from the table below which compares times for a 0.2 ps MD simulation of dppc (a phospholipid membrane system) in water solution (7616 atoms, Rcut=14.5A, Ewald summation, double time-step algorithm 2/0.2 fs) carried out at the Arrhenius Laboratory, University of Stockholm. (All times are in seconds)

Number of Processors	IBM SP2	Cray TE3	Beowulf
1	1807	3630	1488
2	931	1676	775
4	492	863	437
8	269	476	273
10	190	337	196

8 Conclusion

The field of parallel programming is not a standardized one. Many factors affect whether and how a particular application can be parallelized. Both hardware and software aspects of the problem must be considered and for both there are a large number of options available. As of now there is no cookbook or 'Parallel Programming for Dummies' available. A lot of the work is thus based on rules of thumb and experience. As a result one must carefully weigh the pros and cons of going on for a parallel solution for a given problem. However, with the advent of Beowulf clusters, using off the shelf hardware and freely available software libraries, languages and applications the barrier to investment has been considerably lowered. In the field of chemistry the extreme computational requirements will always lead researchers to explore the possibilities provided by parallel techniques for improved speed

up and efficiency. The presence of both freely available software as well as commercial software for nearly every aspect of computational chemistry and biology allows one to experiment and explore the avenues opened up by parallel techniques.

Internet Resources

[1] tp://www.intel.com/intel/museum/25anniv/hof/moore.htm
[2] http://www.cray.com/
[3] http://www.ibm.com/
[4] http://www.sgi.com/
[5] http://www.beowulf.org/
[6] http://www.epm.ornl.gov/pvm/pvm_home.html
[7] http://www.mcs.anl.gov/mpi/
[8] http://www.mpi-forum.org/
[9] http://www.crpc.rice.edu/HPFF/home.html
[10] http://www.csl.sri.com/GLU.html
[11] http://suif.stanford.edu/ scales/sam.html
[12] http://www.extreme.indiana.edu/sage/
[13] http://www.pgroup.com/
[14] www-jics.cs.utk.edu/I2PP/I2PP-0698/sld065.htm
[15] http://www.sgi.com/solutions/sciences/chembio/comp_chem.html
[16] http://www.gaussian.com
[17] http://www.msg.ameslab.gov/GAMESS/ or http://lacebark.ntu.edu.au/gamess/
[18] http://sal.kachinatech.com/Z/2/MOPAC.html
[19] http://www.ks.uiuc.edu/Research/namd/
[20] http://www.earth.ox.ac.uk/keith/moldy.html
[21] http://aros.ca.sandia.gov/cljanss/mpqc/
[22] http://www.sgi.com/solutions/sciences/chembio/pdf/comp_chem_00.pdf