Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Magnetic Resonance 161 (2003) 183–190
JMR Journal of Magnetic Resonance
www.elsevier.com/locate/jmr

Efficient solid state NMR powder simulations using SMP and MPP parallel computation

Jørgen Holm Kristensen* and Ian Farnan

Department of Earth Sciences, University of Cambridge, Downing Street, Cambridge CB2 3EQ, UK

Received 7 August 2002; revised 31 December 2002

Abstract

Methods for parallel simulation of solid state NMR powder spectra are presented for both shared and distributed memory parallel supercomputers. For shared memory architectures the performance of simulation programs implementing the OpenMP application programming interface is evaluated. It is demonstrated that the design of correct and efficient shared memory parallel programs is difficult as the performance depends on data locality and cache memory effects. The distributed memory parallel programming model is examined for simulation programs using the MPI message passing interface. The results reveal that both shared and distributed memory parallel computation are very efficient with an almost perfect application speedup and may be applied to the most advanced powder simulations.

Keywords: Parallel simulation of solid state NMR powder spectra; Shared and distributed memory parallel supercomputers; Parallel programming in the OpenMP application programming interface and MPI message passing interface

1. Introduction

The lineshapes obtained in solid state NMR experiments on polycrystalline or amorphous powders are known to be sensitive to structural and motional details and may be used to characterize many different solid materials [1,2]. The structural and motional information can usually be extracted only by accurate simulation of the lineshapes. The calculation of powder spectra involves averaging the density operator over a large number of crystallite orientations. The literature presents a variety of numerical methods to perform the powder averaging. These include techniques to select the crystallite orientations and weights and methods to interpolate spectral frequencies and intensities [3–16]. The performance of these techniques is known to be comparable especially for simulations of broad lineshapes where the number of crystallites is substantial [17]. The evaluation of the density operator often requires very time consuming operations and reducing the number of these by averaging over a smaller but carefully chosen set of crystallites is a subject of continuing interest and importance.

Because of the nature of the problems being solved it is unlikely that further developments of powder integration methods will provide the orders of magnitude improvement in efficiency needed for the most advanced powder simulations. However, an alternative that has been relatively neglected is the method of parallel computation [18–22]. This is eminently suited to powder simulations because most of the computation time is spent on the repeated evaluation of the density operator and averaging of the result with a predictable pattern of communication between processors. The only practical purpose of parallel computation is to minimize the execution time of an application program. Because many powder simulations exceed the capabilities of single processors it is compelling to exploit the effects of multiple processors to provide sufficient computational power. It is noted that there are several programs available for powder simulations that may have various

* Corresponding author. Present address: Department of Chemistry, University of Cambridge, Lensfield Road, Cambridge CB2 1EW, UK.
E-mail address: jhk28@cam.ac.uk (J.H. Kristensen).

1090-7807/03/$ - see front matter © 2003 Elsevier Science (USA). All rights reserved.
doi:10.1016/S1090-7807(03)00007-7

advantages [23–25]. These programs have not been shown to provide any significant parallel capabilities and could possibly be improved considerably by introducing parallel powder techniques.

In this paper the principles of solid state NMR parallel powder simulations are discussed. The methods may be applied to any simulation program and are valid for any powder integration method. It is shown that efficient parallel programs may be developed for both shared and distributed memory parallel supercomputers. The programs have been implemented on different computer platforms and the performance has been investigated in detail. The results show that it is possible to obtain an almost perfect application speedup demonstrating the considerable advantage of parallel powder simulation.

2. Theory

The most advanced and comprehensive simulations of solid state NMR spectra are based on density operator algebra [26–28]. This formalism provides a useful mechanism to evaluate the different coherences and alignments of nuclear spin ensembles and describe coherence transfer processes and stochastic phenomena like molecular motion and relaxation. The density operator $\sigma(t,\Omega)$ is obtained as the solution to the stochastic Liouville–von Neumann equation

$$\frac{\partial}{\partial t}|\sigma(t,\Omega)\rangle = A(t,\Omega)|\sigma(t,\Omega)\rangle, \tag{1}$$

where the coefficient operator

$$A(t,\Omega) = -\mathrm{i}Ad(H(t,\Omega)) + \Delta(t,\Omega) + \Xi \tag{2}$$

involves the Hamiltonian $H(t,\Omega)$, the relaxation operator $\Delta(t,\Omega)$, and the stochastic operator Ξ. For a polycrystalline sample the anisotropy of the system implies that the nuclear spin interactions depend on the orientation Ω of the individual crystallites. The nuclear spin interactions are represented by the Hamiltonian, whereas the stochastic operator describes the effects of molecular motion. The motion induces random fluctuations in the nuclear spin interactions. These fluctuations may stimulate the relaxation of the system and are described by the relaxation operator.

For any crystallite orientation the observable quadrature signal $P(t,\Omega) = \langle I_+|\sigma(t,\Omega)\rangle$ is calculated from the corresponding density operator. The simulation of powder spectra involves evaluating the observable signal for all crystallite orientations to obtain the powder average

$$P(t) = \frac{\int_V P(t,\Omega)\,\mathrm{d}V}{\int_V \mathrm{d}V}, \tag{3}$$

where the integration domain V defines all possible crystallite orientations. For most systems it is impossible to evaluate the powder integral exactly and the solution is usually approximated using a numerical integration method. The parallel powder techniques introduced in this paper are valid for any integration method and the parallel performance is usually independent of the scheme. However, the evaluation of the density operator depends on the integration method and is often computationally expensive especially for systems exhibiting molecular motion. This demonstrates that although the parallel speedup may be independent of the integration method the absolute computation time depends explicitly on the particular implementation.

The evaluation of the powder average requires the calculation of the density operator for all crystallite orientations and is usually very expensive. In order to improve performance it is useful to distribute the powder integration over multiple processors and keep the evaluation of the density operator as local as possible to minimize communication requirements. The possibility of parallel powder integration becomes evident by partitioning the integration domain $V = V_1 \cup V_2 \cup \cdots \cup V_N$ into a set of subdomains $V_1,\ldots,V_N$ and rewriting the powder integral $P(t) = P_1(t) + P_2(t) + \cdots + P_N(t)$ as a sum of subintegrals $P_1(t),\ldots,P_N(t)$ where

$$P_n(t) = \frac{\int_{V_n} P(t,\Omega)\,\mathrm{d}V_n}{\int_V \mathrm{d}V} \tag{4}$$

defines the local powder average for each integration subdomain. In parallel powder integration it is most efficient to choose the number of subdomains N equal to the number of available processors. The parallel part of the algorithm occurs as each processor computes the powder subintegral for a different subset of crystallite orientations. At the end of the computation the local subintegrals are mapped onto the master processor and combined into a global powder integral. It is always possible to define the powder partition such that the integration subdomains have equal volume demonstrating that load balancing is perfect. The individual subintegrals can be evaluated independently of one another and communication requirements are consequently simple. An important advantage of this approach is that it is valid for any numerical integration method and usually the parallel performance is independent of the implementation. This implies that the approach may be used for any scheme to select the crystallite orientations and interpolate the spectral frequencies and intensities subject to the condition that the method is valid for the particular simulation. This is exemplified by the Alderman, Solum, and Grant interpolation method that may be applied successfully to simulations of many different powder spectra but is known to be invalid for simulations of molecular motion [23].

The only practical reason to write parallel powder simulation programs is to achieve improved and scalable

performance. Most sequential programs are tested for correctness by seeing whether they give the right answer. This is different from parallel programs that must be designed not only to provide the right answer but also to decrease the execution time. Therefore, measuring the speed of execution is part of testing parallel programs to see whether they perform as intended. It is important to understand that some problems lend themselves naturally to a programming style that will run efficiently on multiple processors while others are easy to code in ways that will run even slower in parallel than the original sequential program. Modern parallel multiprocessors are complicated systems and parallel programming involves additional design and coding difficulties. The variety of commercially available parallel machines is large and the performance characteristics vary widely. Tuning a parallel program for one system may not improve and may even reduce performance on another. However, in recent years many multiprocessor systems have shared the same fundamental characteristics. In particular, they have used standard processors connected together via either a bus system or a network and they have contained caches close to the processors to minimize the time spent accessing memory. Although the differences between these machines can be large the factors that affect performance on each one are remarkably similar.

In order to demonstrate the significance of parallel powder simulation it is necessary to define an index to measure how well parallel programs perform. In most applications what matters is the elapsed time relative to the original serial program. This is expressed in terms of the application speedup

$$S(N) = \frac{\text{Elapsed time of serial program}}{\text{Elapsed time of parallel program with N processors}}, \tag{5}$$

which depends explicitly on the number N of processors. In a perfect parallel program the application speeds up by a factor equal to the number of processors. In some cases where data locality and cache memory effects are important the application may speed up by a factor greater than the number of processors. However, depending on the nature of the application and the computer architecture parallel programs often exhibit less than perfect speedup.

It is obviously necessary to parallelize a sufficiently large percentage of an application program to obtain good parallel performance. However, as the number of processors is increased the performance may become dominated by the serial parts of the program. It is only possible to achieve good parallel performance if the time spent in the serial parts and on communication is small compared to the time spent in the parallel portions. This can only be realized if the problem is inherently parallel. The two most important factors determining the efficiency of a parallel program are the time spent in the parallel parts compared to the time spent in the serial parts and the time spent on communication between the parallel parts of the program. The serial parts will execute in time $(1-F)T_S$, where F is the fraction of the program that is parallelizable and T_S is the serial execution time. The parallel parts will execute in time $S_P^{-1}(N)FT_S$, where $S_P(N)$ is the parallel speedup for N processors. The execution time for the parallel program is $T_P(N) = (1-F)T_S + S_P^{-1}(N)FT_S$, which is the sum of the serial and parallel execution times. The overall speedup is obtained from

$$S(N) = \frac{T_S}{T_P(N)} = \frac{S_P(N)}{S_P(N)(1-F)+F}, \tag{6}$$

which is known as Amdahl's law [18–22]. This reveals that no matter how successfully the program is parallelized and no matter how many processors are used eventually the performance will be limited by the proportion F of the code that is parallelizable. For small numbers of processors Amdahl's law has a moderate effect but as the number of processors increases the effect becomes surprisingly large.

The most important factors affecting parallel speedup are coverage, granularity, load balancing, locality, and syncronization. The first three are fundamental to parallel programming on any type of multiprocessor system while the latter two are related to the details of the computer architecture. The coverage refers to the percentage of a program that may be parallelized. The form of Amdahl's law demonstrates that it is important to have a high coverage although this by itself may not be sufficient to guarantee good parallel performance. Another factor that affects performance is granularity. This defines how much work is in each parallel region. The granularity is important because a program incurs an overhead each time it encounters a parallel region. If the coverage is perfect but the program has a large number of small parallel regions then performance may be limited by granularity. In most implementations parallel execution will not terminate until the last active processor has finished its assignments. If some processors are assigned more work than others performance may suffer because of improper load balance. The locality and syncronization refer to the cost of communication between processors. The locality is the property that if a program accesses a memory location there is a much higher than random probability that it will again access the same location soon. A second aspect of locality is that if a program accesses a memory location there is a much higher than random probability that it will access a nearby location soon. The first type of locality is called temporal and the second is called spatial. The locality is often the most critical factor affecting performance on

modern cache based systems. There is a limit to how much memory can be put on each processor chip and accessing memory that is farther away on some other chip is significantly slower. The cache memory provides a way to exploit locality and insure that a much larger percentage of memory references are to faster memory. Because caches are designed to exploit locality the performance of a parallel program can be improved if the code expresses high locality.

3. Experimental results

3.1. Shared memory parallel computation

There is currently no single software environment that absorbs the differences in the architecture of parallel computers and provides a single programming model. It is therefore necessary to adopt different programming models for different computer architectures in order to balance performance and the effort to program. It is often useful to classify parallel computational models according to whether memory is shared or distributed [29,30]. The symmetric multiprocessor (SMP) architecture illustrated in Fig. 1 implements shared system resources such as memory that can be accessed equally from all the processors. Each processor has its own cache memory that may have several layers. The connection between the caches and the memory is built as either a bus or a crossbar switch. A single operating system controls the SMP computer and schedules processes and threads on processors so that the load is balanced.

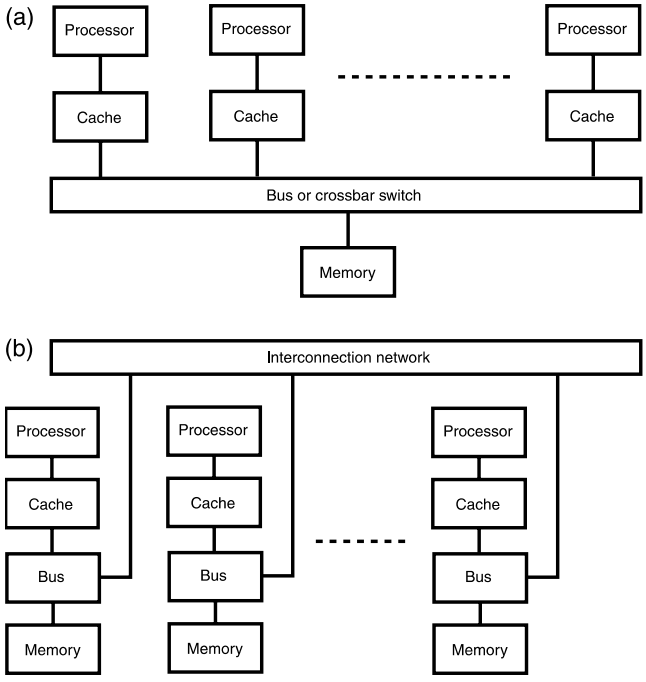

Fig. 1. The architecture of (a) shared and (b) distributed memory parallel computers. The shared memory system has multiple processors that can operate in parallel and cache memory modules associated with every processor. The cache modules are connected to the main memory through a bus or crossbar switch allowing the processors to cooperate and share data. The distributed memory system is designed with multiple computational nodes each containing a single processor and memory modules. Every processor can only directly access memory associated with its local node. In order to cooperate and share data the processors must transmit messages through an interconnection network.

For shared memory parallel architectures there are compilers that can parallelize an application program using explicit compiler directives provided by the programmer. These directives describe the parallelism in the source code and are usually supported by a library of subroutines and environment variables. The compiler directives along with the supporting subroutines and environment variables comprise an application programming interface as exemplified by the OpenMP and HPF parallel programming standards [31,32]. The executables generated by shared memory compilers run in parallel using multiple threads and these can communicate with each other by use of shared memory without explicit message passing statements. The individual threads represent different program controls and execution stacks and are usually associated with different physical processors. During the execution of a program the behavior of each thread is controlled exclusively by its thread number. The shared memory execution model is illustrated in Fig. 2 where a single master thread executes all statements until a parallel region is encountered. At the entrance to the parallel region the master thread forks a set of parallel slave threads. The master and slave threads execute all statements in the parallel region redundantly. The parallel speedup comes from each thread operating on a different part of the data or the code concurrently. At the end of the parallel region the slave threads are joined to the master thread that executes all statements in the sequential regions. In the case of parallel powder simulations each parallel region corresponds to the evaluation of a local powder subintegral.

The last decade has seen a tremendous increase in the widespread availability and capability of shared memory parallel computers. These have not only become much more prevalent but also contain increasing numbers of processors. However, the physical limitation on the memory bandwidth makes it difficult to develop shared memory multiprocessors with more than a few tens of processors. In order to achieve the SMP model with a large number of processors one must allow some memory references to take longer than others. This is the fundamental principle in shared memory architectures that provide nonuniform memory access (NUMA). In this model each processor has its own memory and cache and is located inside a computational

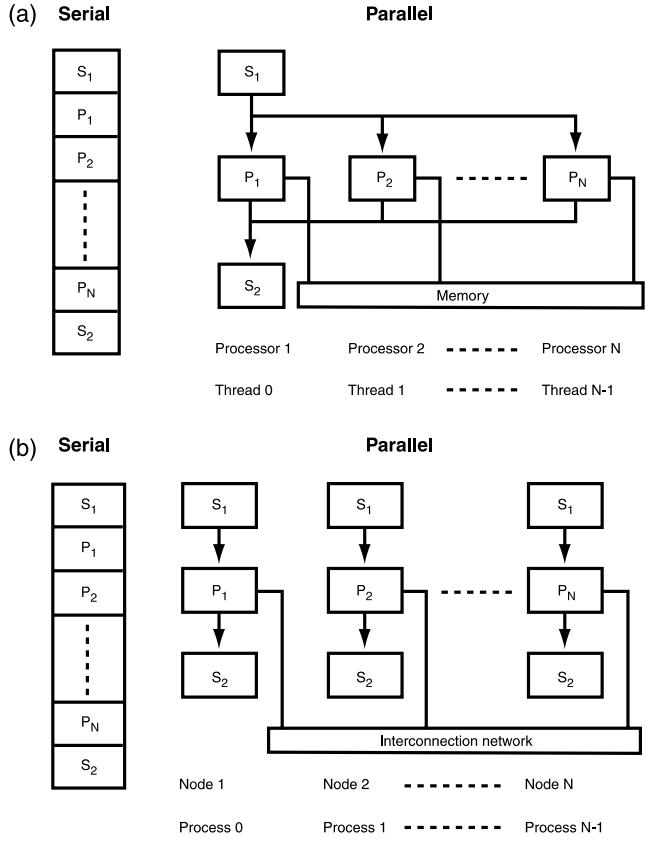

Fig. 2. Diagrams illustrating the execution of (a) shared and (b) distributed memory parallel programs. The serial program proceeds sequentially from the region S_1 to S_2 through the regions P_1 through P_N that can be processed in parallel. The shared memory parallel program first processes serially with a single master thread in the sequential region S_1. When the master thread encounters the parallel region it forks some additional threads often referred to as the slave threads. Each thread represents an independent program counter that executes within the shared address space with direct access to all of its variables. The team of threads processes the regions P_1 through P_N in parallel and when finished the slave threads are joined to the master thread that resumes execution in the sequential region S_2. In the distributed memory parallel program one process runs on each node and the processes communicate with each other during the execution of the parallelizable part P_1 through P_N. The processes have only local memory but are able to communicate via the interconnection network.

node. The nodes are connected to each other through an interconnection network and the system is designed such that any processor can access data in any memory. However, accessing memory that is in the local node or a nearby node may be faster than accessing memory that is in a remote node. This architecture combines the logical view of a shared memory machine with physically distributed memory and may support hundreds and even thousands of processors. The NUMA model is implemented in the SGI Origin 2000/3000 series of computers.

In order to give a representative example of the performance of a shared memory multiprocessor system we have chosen the SGI Origin 2000 to generate sample numbers. Although the exact numbers will differ it is expected that other cache based shared memory parallel computers will have similar performance characteristics. The SGI Origin 2000 is known to be the most scalable of existing cache based shared memory machines and provides a nearly uniform memory access model to all of its main memory. The system configuration involves 64 MIPS R12000 300 MHz processors each with 32 kB data cache, 32 kB instruction cache, and 8 MB secondary cache. The system has 32 computational nodes each containing 2 processors and 4 GB memory. Although the system implements the NUMA architecture the differences between local and remote memory accesses are usually much smaller than cache effects and data locality problems are usually related entirely to cache memory.

Any performance and scalability analysis involves estimating the computation and communication requirements of a particular problem and the study of how these requirements change as the problem size or the number of processors change. In the case of shared memory parallel architectures we have parallelized a Fortran 95 simulation program using OpenMP compiler directives. The program is designed to simulate effects of finite pulse lengths and molecular motion on central and satellite transition powder spectra of any quadrupole nucleus [27,28]. The result of an experimental simulation is shown in Fig. 3 which includes the effects of finite pulse lengths and molecular motion. The details of the integration method determine the computation time but are irrelevant when evaluating the parallel performance. This depends on the coverage and granularity of the program that are determined by the total number of

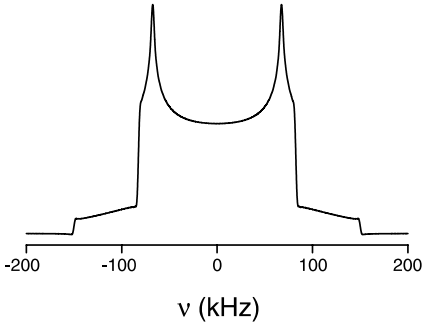

Fig. 3. Result of parallel simulation of a quadrupole echo ^{2}H NMR powder spectrum. The system is specified by a quadrupole tensor with quadrupole coupling constant $C_Q = 200$ kHz and quadrupole asymmetry parameter $\eta_Q = 0.10$. The quadrupole tensor is assumed to reorient between three equally probable orientations $\Omega(\xi_m) = \{0, \pi - 2\arctan(\sqrt{2}), \frac{2\pi[m-1]}{3}\}$ with rate constants $\log(k_{mn}) = \log(k_{nm}) = 2$ corresponding to ultraslow motion of a methyl group. The simulation implemented the rf field strength $\nu_{rf} = 100$ kHz, pulse lengths $\tau_{p1} = \tau_{p2} = 2.5\,\mu$s, and pulse delays $\tau_{d1} = \tau_{d2} + \frac{1}{2}\tau_{p1} = 50\,\mu$s. The number of crystallites depends on the accuracy of the calculation and the powder integration method and determines the granularity of the parallel simulation.

crystallites and the cost of evaluating the density operator. For most powder simulations the coverage and granularity are large indicating that the parallel speedup may be substantial. The powder iterations have been assigned in large blocks and distributed evenly among the processors. This improves the granularity and eliminates any load imbalance problems. In parallel powder simulations the coverage is almost perfect because a large part of the computation time is spent on the powder iterations. This suggests in accordance with Amdahl's law that the parallel performance of powder simulations may be very high.

The results of an experimental performance analysis are shown in Fig. 4 for the SGI Origin 2000 shared memory parallel multiprocessor. It is evident that the parallel speedup is almost perfect for this application. The results can be generated N times faster when using N processors compared to using only a single processor. This performance level cannot be supported by any single processor system. Even the fastest single processors currently available deliver a performance equivalent to only about ten of the Origin 2000 processors. Because of these significant performance gains it becomes possible to provide much more detailed and accurate simulations of polycrystalline or amorphous solids.

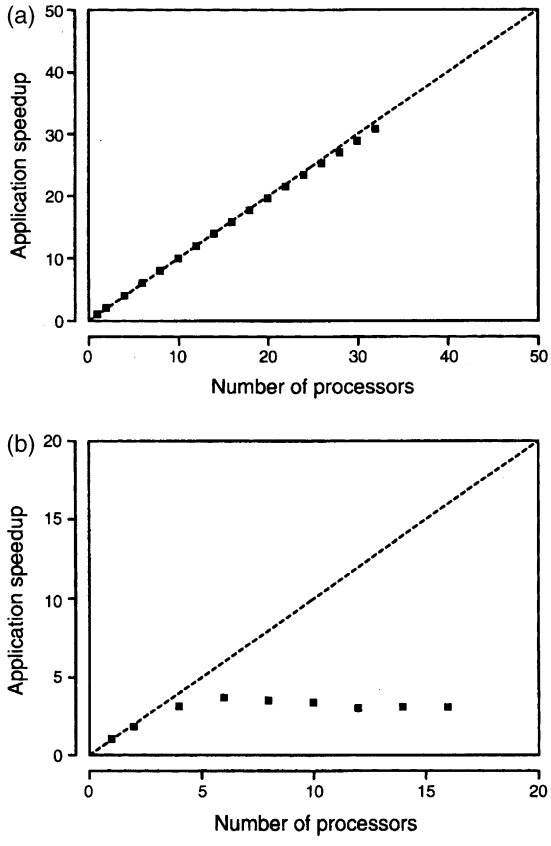

Fig. 4. Experimental results illustrating the performance of OpenMP shared memory parallel powder simulation programs. The results demonstrate (a) the possibility of an almost perfect application speedup and (b) the adverse effects of communication overhead in parallel programs that have not been modified to exhibit locality. The programs were executed on a SGI Origin 2000 shared memory parallel supercomputer.

There are considerable design and coding difficulties involved in writing correct and efficient shared memory parallel programs. In order to demonstrate this we have produced a parallelized version of the Fortran 95 simulation program using OpenMP compiler directives without any concern about data locality and cache effects. The results of an experimental calculation are shown in Fig. 4 for the SGI Origin 2000 shared memory parallel supercomputer. It is seen that the parallel performance is nearly perfect for small numbers of processors. However, when the program executes on a large number of processors the parallel performance suffers dramatically. This version of the parallel program involves only modest code modifications and has not been designed to match the memory architecture of the parallel machine. This causes significant data locality and cache memory effects that reduce the parallel performance. In order to get sufficient parallel speedup it is necessary to modify the code fundamentally. This is a difficult problem that depends on the programming style and program design. However, the additional programming effort is justified by the significant parallel speedup that is possible for most powder simulations.

3.2. Distributed memory message passing computation

The alternative to shared memory is the distributed memory architecture where each processor is only capable of directly addressing memory physically associated with it. The distributed memory parallel programming model is targeted for the massively parallel processors (MPP) architecture. This architecture is illustrated in Fig. 1 where each node has its own processor and memory. The operating system is running on each node that can be considered as a separate workstation. Despite the term massively the number of nodes is not necessarily large and usually there is no limitation. In the MPP architecture the memory is not physically shared among nodes and parallel processes have to transmit messages over an interconnection network in order to access data that other processes have updated. This is different from the shared memory model where individual threads access memory without knowing whether they are triggering remote communication at the hardware level.

For distributed memory parallel systems the message passing programming model is a useful and complete framework in which to express parallelism [33,34]. The message passing programming model has been standardized by the message passing interface (MPI) which defines an effective and portable application programming interface for writing parallel programs. The mes-

sage passing execution model is shown in Fig. 2 for distributed memory systems. The message passing interface is not a computer language but a library of subroutines to express communication. The message passing programs are compiled with ordinary compilers and linked with the MPI library to produce the parallel executables. In the message passing model a computation remains a collection of parallel processes communicating with messages transmitted over the interconnection network. Every process represents an address space and one or more threads. The processes are associated with different processors where a processor represents a central processing unit capable of executing a program. Every process has an identifier called rank and although each process may execute the same program the behavior of each process can be made different by using the value of the rank. This is similar to shared memory programming where the behavior of each thread is defined by its thread number.

The specific details of the communication network are not a part of the message passing computational model. The precise connection topology is irrelevant to the programmer and the model may be implemented on a wide variety of hardware architectures. The message passing interface provides the control missing from the compiler based models in dealing with data locality. As modern processors become faster management of their caches and the memory hierarchy has become the key to achieving maximum performance. The message passing interface provides a mechanism for the programmer to explicitly associate specific data with processes and thus allow the compiler and cache management hardware to function fully. This explains why message passing has emerged as one of the more widely used models for expressing parallel algorithms. The message passing model matches the hardware of most modern parallel supercomputers. Representative systems include the IBM SP2/SP3 parallel computers and Beowulf clusters of workstations.

As an example of a massively parallel processors architecture we have chosen the IBM SP2 parallel supercomputer. The system configuration involves 160 Power3-II 375 MHz processors each with 128 kB data cache, 128 kB instruction cache, and 8 MB secondary cache. The system is designed with 10 SMP nodes each with 16 processors and 12 GB memory. The SMP nodes are connected via a high performance IBM crossbar switch.

In order to demonstrate the usefulness of distributed memory message passing parallel powder computation we have parallelized the Fortran 95 simulation program described above using the MPI message passing programming model. The performance was monitored by measuring the application speedup for the powder simulation illustrated in Fig. 3. The results are shown in Fig. 5 as function of the number of processors. The

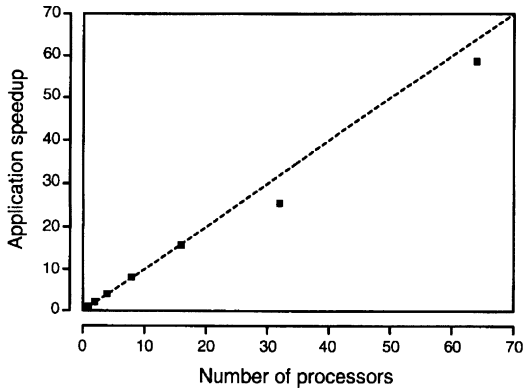

Fig. 5. Experimental results demonstrating the performance of an MPI distributed memory message passing parallel powder simulation program. The program was executed on an IBM SP2 massively parallel processors supercomputer.

parallel speedup is seen to increase by a factor of N as we apply N processors to the same simulation indicating high parallel efficiency. The computation time differs slightly between 2–32 and 32–64 processors as a result of increasing communication overhead. Because many powder simulations are very expensive it is clear that parallel computation can have an enormous impact on performance. The advantage of distributed memory programming is that data locality effects are less important making it easier to produce efficient parallel programs.

4. Summary

In this paper numerical methods for parallel computation of solid state NMR powder spectra have been described and investigated in detail. The approach involves distributing the powder integration over multiple processors and keeping the evaluation of the density operator as local as possible. This technique has a substantial parallel efficiency as a result of high coverage and granularity as well as modest communication requirements. The methods have been implemented to parallelize a simulation program designed to calculate effects of finite pulse lengths and molecular motion for both central and satellite transition spectra of any quadrupole nucleus.

The programming model depends on the computer architecture and is different for shared and distributed memory multiprocessors. In the case of shared memory architectures we have parallelized the simulation program using the OpenMP application programming interface. The computation and communication requirements were estimated by evaluating the parallel performance for different numbers of processors on an SGI Origin 2000 shared memory parallel supercomputer. The results indicate a high parallel efficiency with

an almost perfect parallel speedup. However, the results show that data locality and cache memory effects reduce parallel performance and the programs must be substantially modified in order to exhibit high locality.

The distributed memory parallel programming model was evaluated by parallelizing the simulation program using the MPI message passing interface. The message passing programming model provides the control needed to express data locality and for efficient cache management. The parallel performance was evaluated by measuring the parallel speedup for different numbers of processors on an IBM SP2 massively parallel processors supercomputer. The results reveal a high parallel efficiency and show that distributed memory parallel computation can have an enormous impact on performance. An important advantage of the distributed memory programming model is that it matches the architecture of many modern supercomputers.

Acknowledgments

The UK Joint Infrastructure Fund (JIF) and Joint Research Equipment Initiative (JREI) supported this research. The Cambridge-Cranfield High Performance Computing Facility (HPCF) is acknowledged for the use of the SGI Origin 2000 and IBM SP2 parallel supercomputers.

References

[1] M. Mehring, Principles of High Resolution NMR in Solids, Springer-Verlag, Berlin, 1983.
[2] K. Schmidt-Rohr, H.W. Spiess, Multidimensional Solid State NMR and Polymers, Academic Press, London, 1994.
[3] D.L. VanderHart, H.S. Gutowsky, T.C. Farrar, J. Am. Chem. Soc. 89 (1967) 5056.
[4] G.V. Veen, J. Magn. Reson. 30 (1978) 91.
[5] K. Balasubramanian, L.R. Dalton, J. Magn. Reson. 33 (1979) 245.
[6] K.W. Zilm, R.T. Conlin, D.M. Grant, J. Michl, J. Am. Chem. Soc. 102 (1980) 6672.
[7] K.W. Zilm, D.M. Grant, J. Am. Chem. Soc. 103 (1981) 2913.
[8] J.G. Hexam, M.H. Frey, S.J. Opella, J. Chem. Phys. 77 (1982) 3847.
[9] S. Ganapathy, V.P. Chacko, R.G. Bryant, J. Magn. Reson. 57 (1984) 239.
[10] D.W. Alderman, M.S. Solum, D.M. Grant, J. Chem. Phys. 84 (1986) 3717.
[11] A. Kreiter, J. Hütterman, J. Magn. Reson. 93 (1991) 12.
[12] M.J. Mombourquette, J.A. Weil, J. Magn. Reson. 99 (1992) 37.
[13] J.M. Koons, E. Hughes, H.M. Cho, P.D. Ellis, J. Magn. Reson. 114 (1995) 12.
[14] D. Wang, G.R. Hanson, J. Magn. Reson. 117 (1995) 1.
[15] M. Bak, N.C. Nielsen, J. Magn. Reson. 125 (1997) 132.
[16] M. Edén, M.H. Levitt, J. Magn. Reson. 132 (1998) 220.
[17] A. Ponti, J. Magn. Reson. 138 (1999) 288.
[18] R.W. Hockney, C.R. Jesshope, Parallel Computers, Adam-Hilger, Philadelphia, 1988.
[19] S.G. Akl, The Design and Analysis of Parallel Algorithms, Prentice-Hall, New Jersey, 1989.
[20] K.M. Chandy, S. Taylor, An Introduction to Parallel Programming, Jones and Bartlett, Boston, 1992.
[21] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing. Design and Analysis of Algorithms, Addison-Wesley, Boston, 1994.
[22] I. Foster, Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering, Addison-Wesley, Boston, 1995.
[23] M.S. Greenfield, A.D. Ronemus, R.L. Vold, R.R. Vold, P.D. Ellis, T.E. Raidy, J. Magn. Reson. 72 (1987) 89.
[24] S.A. Smith, T.O. Levante, B.H. Meier, R.R. Ernst, J. Magn. Reson. 106 (1994) 75.
[25] M. Bak, J.T. Rasmussen, N.C. Nielsen, J. Magn. Reson. 147 (2000) 296.
[26] U. Fano, Rev. Mod. Phys. 29 (1957) 74.
[27] J.H. Kristensen, G.L. Hoatson, R.L. Vold, Solid State Nucl. Magn. Reson. 13 (1998) 1.
[28] J.H. Kristensen, I. Farnan, J. Chem. Phys. 114 (2001) 9608.
[29] J. Hennessey, D. Patterson, Computer Architecture. A Quantitative Approach, Morgan-Kaufmann, San Fransisco, 1996.
[30] D. Culler, J. Singh, A. Gupta, Parallel Computer Architecture. A Hardware/Software Approach, Morgan-Kaufmann, San Francisco, 1999.
[31] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele, M.E. Zosel, The High Performance Fortran Handbook, MIT Press, Cambridge, MA, 1994.
[32] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP, Morgan-Kaufmann, San Francisco, 2001.
[33] P. Pacheco, Parallel Programming with MPI, Morgan-Kaufmann, San Francisco, 1996.
[34] W. Gropp, E. Lusk, A. Skjellum, Using MPI. Portable Parallel Programming with the Message Passing Interface, MIT Press, Cambridge, MA, 1999.