International Journal of High Speed Computing, Vol. 7, No. 3 (1995) 327–349
© World Scientific Publishing Company

SIMULATIONS OF INTERACTING MANY BODY SYSTEMS USING p4

R. T. SCALETTAR, K. J. RUNGE and J. CORREA
Physics Department
University of California, Davis, CA 95616, USA
E-mail: *runge@redhook.llnl.gov*

P. LEE and V. OKLOBDZIJA
Department of Electrical and Computer Engineering
University of California, Davis, CA 95616, USA

J. L. VUJIC
Department of Nuclear Engineering
University of California, Berkeley, CA 94720, USA

Received April 6, 1994

ABSTRACT
Monte Carlo (MC) and Molecular Dynamics (MD) simulations are powerful tools for understanding the low temperature properties of systems of interacting electrons and phonons in a solid, including the phenomena of magnetism and superconductivity. When mobile electrons are studied, these simulations are currently limited to a few hundred particles, and also largely to "clean" systems where no defects are present. Therefore, more powerful machines and algorithms must be used to address many of the most important issues in the field. In this paper, we present results from using some simple implementations of the p4 parallel programming system on a variety of parallel architectures to conduct MC and MD simulations of one and two dimensional electron-phonon models.

Keywords: Parallel monte carlo methods, message passing, p4

1. Introduction. Implementing parallel computations in the most interesting sense involves the distribution of a single, large problem over a network of coupled processors. Successful completion of such a calculation requires the addressing of an interconnected set of questions concerning the application, as well as computer science issues of sharing memory, properly sequencing tasks, passing messages, etc. Such an approach necessarily requires significant rewriting of conventional serial codes, with the attendant time required to debug them. For many applications, this process, though challenging, is unavoidable.

On the other hand, many applications do not need the full power of the parallel machine for a single run, but rather, their complexity resides in the necessity to conduct a large number of separate calculations. Most Monte Carlo applications fall naturally into this class, since statistically independent configurations are needed which can be generated by launching simultaneous, non-communicating copies of the code on each node with different random number seeds. Furthermore, it is frequently necessary to conduct a series of runs using different parameter values. For example, if we want to sweep the temperature down through the critical point of a phase transition, we can use more than one node and assign different temperature value to each node. Finally, in the field of condensed matter physics, there is increasing interest in the role of defects in interacting phonon and electron models. Here it is necessary to conduct a disorder average over hundreds or thousands of independent realizations.

The p4 parallel programming system is a library of routines [1] that enables one to write portable, machine-independent parallel code that can be used on a variety of parallel computers. These can range from a cluster of UNIX workstations to massively parallel supercomputers. The p4 package is available at `ftp://ftp.mcs.anl.gov/pub/p4`. For our work we use the processor to processor message passing and timing library routines which p4 provides. However, there are a number of additional routines and interfaces p4 contains that provide increased functionality. Among the six different environments (HP, Sparc, AIX, Linux, iPSC/860, and Meiko CS-2) in which we have installed p4, we find in five (there was some difficulty with the HP) that the compilation, installation, and usage is fairly straight forward and easy to implement.

In this paper we describe simulations using p4 with both types of parallelization mentioned above. In the former more challenging case of distributing a single run over many CPUs, we have chosen to study a classical model of interacting lattice vibrations using the Molecular Dynamics algorithm. We have written codes using the p4 library to produce portable parallelized programs which can run on a variety of machines and clusters. In the latter case of independent computations, we study by the Quantum Monte Carlo (QMC) algorithm the two-dimensional Hubbard Hamiltonian, a model which provides a computational challenge at the forefront of condensed matter physics. The essential algorithmic difference between the two problems is that the QMC problem involves an energy which couples degrees of freedom to each other in a highly complicated and nonlocal way. Knowledge of the state of the entire lattice is required to evaluate the change in energy of the system when a single lattice site is altered, because the energy is related to a determinant of a matrix involving *all* lattice sites. Furthermore, these QMC codes are rapidly evolving. This discourages efforts to write a parallelized code if the process involves a lot of machine specific details. Because of the complexity of these QMC computations, we use an approach which provides a way to accomplish simple parallelization with the minimum possible rewriting (in some cases none) of application codes. In particular, we describe a set of simple software tools we have developed which enable us to easily submit a set of independent Monte Carlo (or other) simulations to a parallel computer or meta-computer. The software will also perform the necessary data analysis of a rather general set of typical data files, averaging the data returned from individual nodes automatically.

The remainder of this paper is organized as follows: In Section 2 we review some of the basic ideas behind the models we are studying and key features of the Monte Carlo (MC) and Molecular Dynamics (MD) algorithms which have been used in the past for serial codes. Section 3 describes briefly the hardware platforms on which we test parallelized algorithms. Section 4 summarizes some features of p4 and gives some examples of typical calls to p4 library routines. Section 5 contains details of our first application: a distributed MD calculation of interacting lattice vibrations. Section 6 contains details of our second application: a Quantum MC calculation of an interacting electron model in which independent simulations are submitted to different nodes, differing either in the random number seeds or in the disorder potentials, and the software tools we have written for job submission and data averaging. Section 7 concludes with some brief summarizing remarks.

2. Review of molecular dynamics and Monte Carlo.

Molecular Dynamics for a Lattice Vibration Model: In condensed matter physics one is interested in calculating the properties of the interacting electrons and ions in a crystal. Since this is a tremendously complicated task, simple models are introduced to focus on the relevant degrees of freedom for particular problems. In order to understand the elastic and thermal properties of many crystals [2], for example, it is often sufficient to consider an energy function of the form,

$$H = \frac{1}{2}\sum_i |\mathbf{p}_i|^2/m_i + \sum_{i,j} V(\mathbf{x}_i, \mathbf{x}_j)\,. \tag{1}$$

Here the first term is the kinetic energy of a set of ions moving with momentum $\mathbf{p}_i$, while the second term, V, describes the potential energy between ions i, j at positions $\mathbf{x}_i$, $\mathbf{x}_j$. The electron degrees of freedom enter only indirectly, through the ion-ion potential V. If V is quadratic in the ionic coordinates, this model can be solved analytically, and the resulting wave-like excitations, termed "phonons", propagate independently in the crystal. For more realistic forms of V, the model is insoluble analytically, and therefore must be tackled by numerical means.

One such numerical approach is "Molecular Dynamics" in which Hamilton's equations of motion

$$
\begin{aligned}
\frac{d\mathbf{p}_i}{dt} &= -\frac{\partial H}{\partial \mathbf{x}_i} \\
\frac{d\mathbf{x}_i}{dt} &= +\frac{\partial H}{\partial \mathbf{p}_i}
\end{aligned}
\tag{2}
$$

are integrated numerically by discretizing time and employing Runge-Kutta, leap-frog, or some equivalent finite difference approach [3]. A central assumption of MD is the "ergodic hypothesis" [4], which states that averaging observables over the system's evolution in time is equivalent to other averaging formalisms, such as the canonical ensemble which weighs configurations according to their Boltzmann factors $\exp[-\beta H]$, where β is inversely proportional to the system temperature T.

In this paper, we study Eq. (1) with a potential V that has two pieces. The first consists of quadratic attractive pieces between neighboring masses only. If this were the only term in V, then when the coordinates are distributed on a set of nodes, very little data would need to be passed, only the positions of the masses at the boundaries. We report results for such a situation of very limited connectivity between the degrees of freedom. However, we are in fact primarily interested in a situation where a second, repulsive term in V, which is rather long ranged, is also present. In this case many more data need to be passed since the evolution of masses assigned to a particular node is influenced by the positions of a much larger number of masses on other nodes.

Although one can think about the model as a set of coupled masses and springs, the physics we are ultimately interested in is a description of the motion of flux lines penetrating into a type-II superconductor [5]. In order to do this, one chooses the first term in V so that it organizes the system into lines of attracting masses that would be appropriate for some discretized description of a set of elastic strings. It is believed that one approximate way of thinking about how a tube of magnetic flux traverses a type-II superconductor is as just such an elastic string [6]. The second term is then chosen to have the appropriate functional form to model the repulsive force between flux lines. Asymptotically, the force is a decaying exponential, with a length scale of the order of, or greater than, typical inter-line spacing [6]. There are also terms which model the attraction of the flux lines with various types of crystal defects, in particular with "point defects" which attract just a small portion of the line, and extended "columnar defects" which attract the entire line. Finally, we include a magnetic (Lorentz) force, which pushes on the lines. Although we do not describe them here, the detailed nature of the interactions plays an important role in determining suitability for parallelization, since non-local interactions might require significantly greater communication overhead than shorter range ones. One set of physics questions concerns the nature of the depinning transition as one increases the Lorentz force. How does the structure of the defects determine the critical value of the force required to set the flux lines in motion? This is an important issue since whether or not the flux lines get trapped on impurities is central to the material maintaining its superconducting properties [5].

It is remarkable that even though real systems contain roughly 10^{23} degrees of freedom, simulations of systems as small as a few thousand or tens of thousands of particles can often provide quite a realistic description of the bulk limit. An essential ingredient in using such modest size systems is to employ techniques of "finite size scaling", which allow one to extrapolate to the large system limit [7]. In the simulations we will describe a typical system size will consist of $10^4 - 10^5$ masses evolving for tens of thousands of time steps. One could study a single such system on a workstation or vector supercomputer. However, when the interactions become more long ranged, and disorder is present, more powerful parallel computers are necessary.

Quantum Monte Carlo for an Interacting Electron Model: The simplest model of interacting electrons on a lattice is the "Hubbard Hamiltonian"

$$H = -t \sum_{\langle ij \rangle \sigma} (c_{i\sigma}^{\dagger} c_{j\sigma} + c_{j\sigma}^{\dagger} c_{i\sigma}) + U \sum_{i} n_{i\uparrow} n_{i\downarrow} \,. \tag{3}$$

Here $c_{i\sigma}^{\dagger}$ is an operator which creates an electron on site i with spin σ, and $c_{j\sigma}$ destroys an electron on site j with spin σ. Thus the first term describes the destruction of an electron on one site and its creation on another. In other words, we are referring to the hopping of electrons between sites i and j. The second term describes the interaction of a spin up and spin down electron on the same site i. This term is nonzero when both occupations are nonzero. $n_{i\sigma}$ counts the number of electrons of spin σ on site i. For $U > 0$, the Hubbard Hamiltonian has been widely used to model magnetic transitions in transition metal oxides [8]. We will study the Hubbard model with $U < 0$, a form often used to study superconductivity [9]. We will also include some random site energies which represent imperfections in the crystal. Our interest is in understanding how superconductivity and disorder compete. How much disorder is necessary to destroy the superconducting phase? Does a metallic state exist after superconductivity is destroyed, or does the randomness immediately freeze the electrons into an insulator?

The detailed description of the QMC method is given in reference [10]. In brief, the quantum mechanical problem of evaluating the partition function of Eq. (3) is replaced by an equivalent classical problem in one higher dimension. The length of this added dimension is proportional to the inverse of the temperature it is desired to simulate. Typically, it is necessary to choose 100 or so lattice sites in this "inverse temperature" direction, so the study of an original two dimensional $8 \times 8 = 64$ *spatial* lattice of quantum mechanical electrons requires the evolution of an $8 \times 8 \times 100 = 6400$ classical lattice. Unlike many classical models (one would like to simulate) which have simple, short-ranged interactions, this equivalent classical model is characterized by long-ranged *non-local* interactions. Thus, the primary difficulty, besides the added dimension, is that the interactions are described by a determinant of a dense *full* matrix involving all of the coordinates. Naturally, this makes parallelization of a single simulation extremely challenging, since different pieces of the lattice must communicate with one another frequently and in a complicated manner [11]. For this reason we explore in this paper only a more straight forward parallelization scheme that is based on disorder or statistical averaging. We must, however, emphasize that this simplified approach, nevertheless, presents a useful and efficient way to solve this class of problems.

3. Testbed architectures.

Workstation Cluster: Our first test architecture consists of a set of approximately 40 HP 715 workstations linked by ethernet in the University of California at Davis (U. C. Davis) Department of Electrical and Computer Engineering. Each workstation has a HP PA-RISC 9000/715 CPU and 10MB of RAM memory. We have also worked on smaller clusters of Sun Sparc10 and IBM RS6000 550 workstations.

Intel IPSC860: Our second test architecture is a 32 node Intel iPSC/860 hypercube located in the College of Engineering at U. C. Davis. This machine, though now several years old, is still a rather a large–scale parallel computer [12] with a Multiple Instruction Multiple Data (MIMD) architecture where the computation is carried out by logically independent but cooperating processors. Each node is a 40MHz i860 processor with 8MB of RAM.

Meiko CS-2: The Meiko CS-2 at Lawrence Livermore National Laboratory that we use is a 128 node parallel supercomputer. Each node is a 70MHz Sparc CPU running the Solaris operating system with 64 to 128 MB of RAM. In addition, each node has an attached 90MHz TI 8847 based vector processing unit capable of up to 200 MFLOPS peak performance.

4. Implementation of p4. As we have discussed, p4 is a set of parallel programming libraries, written for C and Fortran, distributed by Argonne National Laboratory. In this work, we mainly use the p4 C library functions. If a problem can be parallelized, p4 can do the message passing of data between nodes in a parallel machine or between workstations in a cluster. An important feature of p4 is its portability. In principle, if p4 is installed on any kind of computer architecture (e.g., supercomputers, workstations, parallel machines), a program written using p4 will run on all of them without changes. However in practice, there are still problems with installing p4 on certain machines.

To picture how p4 works, imagine designating a node (or machine) as master, and many other nodes (or machines) as slaves. The master orchestrates and synchronizes the slaves. The slaves pass messages containing data to neighboring slaves and after calculations are done, send all the data to the master. The master receives these messages and completes the analysis. This is in general how our MD simulations are implemented.

A slice of our program is shown below:

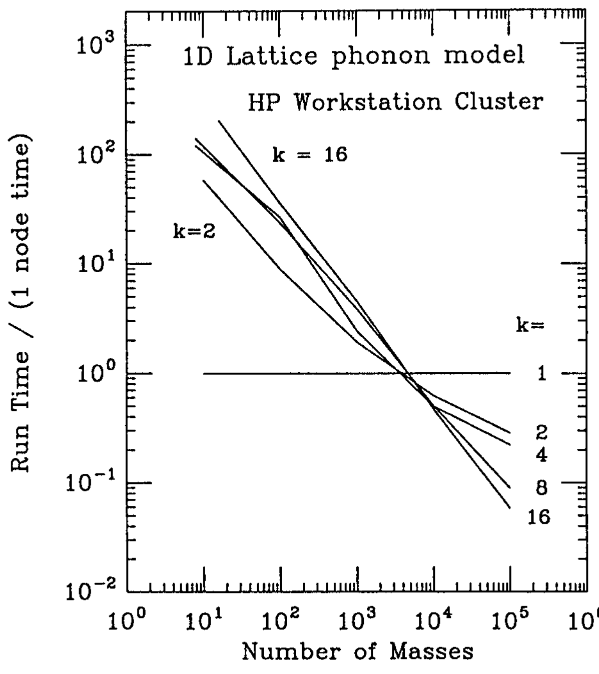
```
main(int argc, char **argv) {
    int start, stop, myid, time1, time2;

       /* Initialize p4 system */
    p4_initenv(&argc, argv);
    start = p4_clock();

       /* Read "process group" file */
    p4_create_procgroup();

       /* time1 = startup time */
    stop = p4_clock();
    time1 = stop - start;

    start = p4_clock();
    myid = p4_get_my_id();

       /* Go do the actual work */
    if (myid == 0) {
        master();       /* I am master */
    } else {
        slave();        /* I am slave */
    }

       /* Wait for end of p4 processes */
    p4_wait_for_end();

      /* time2 = time to do calculations */
    stop = p4_clock();
    time2 = stop - start;
}
```

The p4_initenv must always be called in the beginning of a p4 program. This function call initializes the p4 system. The function call p4_create_procgroup() reads a procgroup file that the user builds which lists all the slave machines one wants to use. p4_get_my_id() gets current node id number. Each machine is assigned a node number from $(0-n)$. Timings are taken using function p4_clock() which gives time in wallclock milliseconds. Within master and slave functions, commands such as p4_sendx and p4_recvx are called to send and receive data between nodes.

For example, here is a sample of code for the simplest case that involves a slave passing its right-most coordinate x[N] to its right neighbor slave, and receiving the right-most coordinate of its left neighbor slave into the variable x[0]:

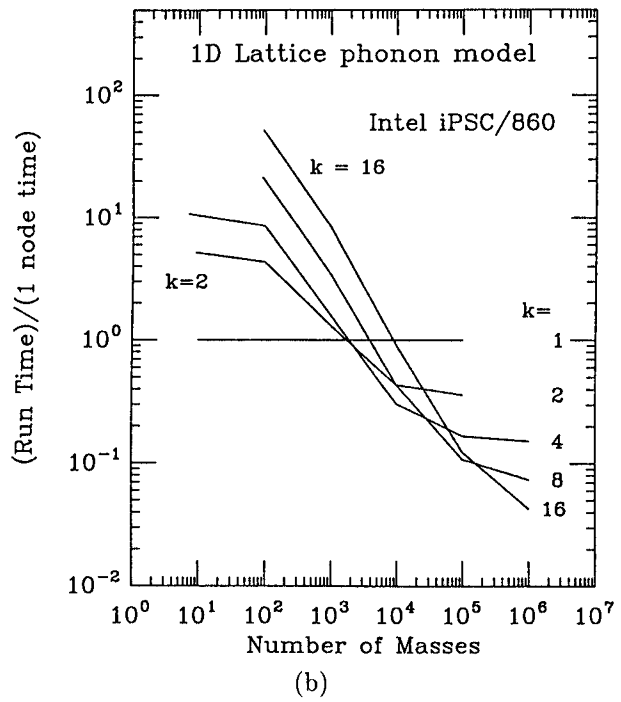
```
/*
 * Passes x[N] to RIGHT neighbor and
 * receives x[0] from LEFT neighbor
 */

pass(double x[], int Right, int Left) {

    double temp[1], *input;
    int type=-1, size, from;

       /*assign data to buffer*/
    temp[0] =  x[N];

       /*send to Right neighbor slave*/
    p4_sendx(100, Right, (char *)temp,
                   sizeof(temp), P4DBL);

       /*clear input buffer*/
    input = (double *) NULL;

       /*get from Left neighbor slave*/
    from = Left;
    p4_recv(&type, &from,
                  (char **) &input, &size);

    x[0]=input[0];

      /*release msg buffer from memory*/
    p4_msg_free((char*) input);
}
```

There is a similar routine that passes the information in the opposite direction. These passed coordinates are then used in the finite-difference updating via Hamilton's equations of motion. In the case of "long ranged" interaction

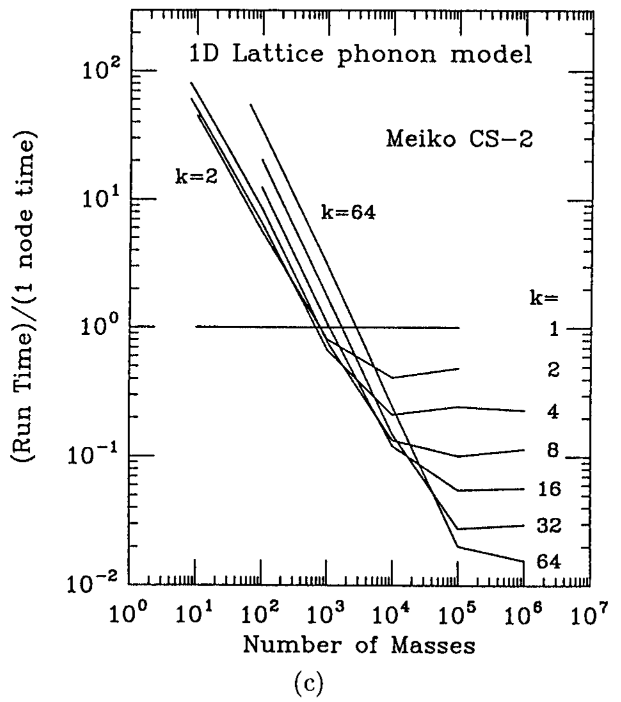


(a)

FIG. 1. *The run time T versus number of masses (problem size) N, for different numbers of nodes k. In order to isolate features of p4 and our physics problem, rather than characteristics of the individual CPU, we have normalized the data to the run time on a single node. In each case we did 1000 MD time steps. For small N the efficiency is low, while for large N it approaches unity. Figures 1(a), 1(b) and 1(c) show results for a cluster of HP-715 workstations, a 32 node Intel iPSC/860, and a Meiko CS-2, respectively. In this problem the structure of the potential V requires communication only at the boundaries of the sublattices put on each node. Although not obvious on the log-log plot we noticed a definite cache overflow glitch on the Meiko at about 50,000 masses per node. At that point the MD vectors no longer fit into the Meiko's cache memory.*



(b)



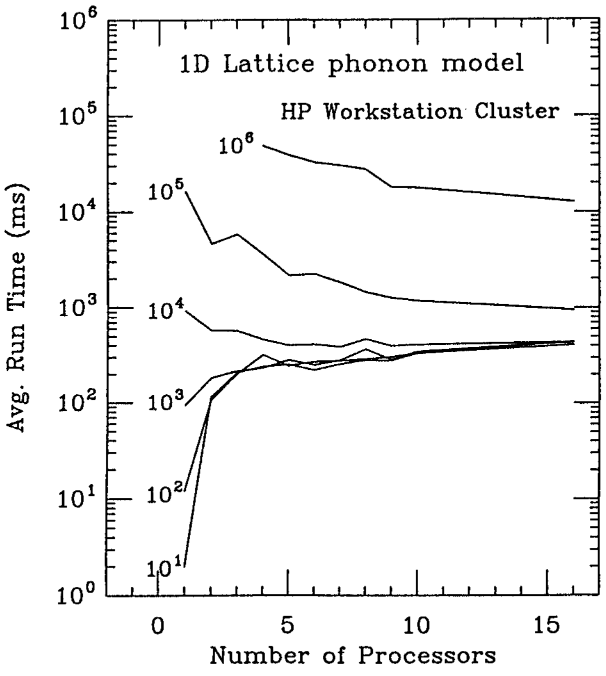
(c)

FIG. 1. (*Continued*)

discussed below, the whole body of coordinates, not just the endpoints x[1] and x[N], must be passed to neighboring nodes at each time step.

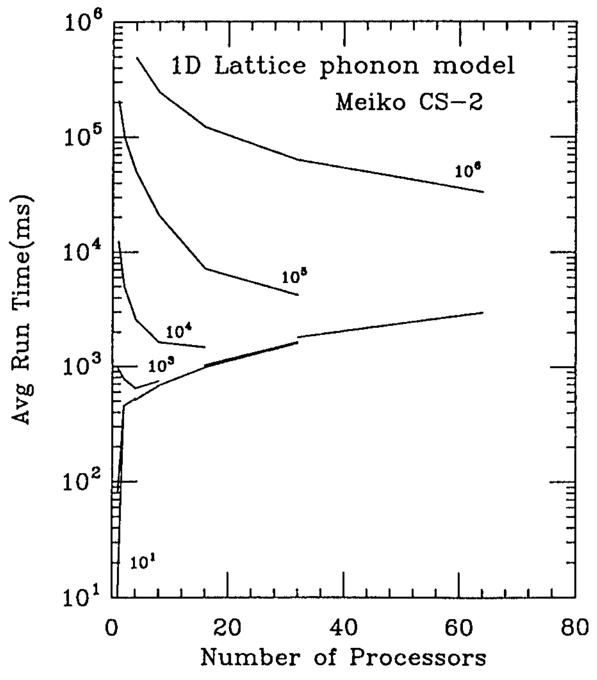
5. Results: Interacting lattice vibrations. In our MD simulations, we started with a conventional serial code that evolved the entire phonon lattice on a single CPU. We then parallelized the code by dividing the lattice into sections, and inserting appropriate p4 library commands, for example the processor to processor message passing routines (**p4_send** and **p4_recv**) described above. These p4 commands pass the values of the ionic positions from processor to processor and also synchronize calculations.

In Fig. 1 we show a plot of the run time as a function of the total number of masses for a situation when the interactions between masses is quite local. Specifically, if there are a total of N masses and k nodes, so that $n = N/k$ masses are assigned to each node, only 2 of the n mass positions need to be



(a)

FIG. 2. *The run time T versus number of nodes (i.e., processors) k, for different numbers of masses N. In each case we did 1000 MD steps. For small N, a single CPU ($k = 1$) is optimal, while for larger N the run time T first decreases with k, then eventually saturates as the number of computations per node becomes small. Curves lying on top of eachother indicate those simulations are in the communication dominated regime, while for large N the run time drops off with the expected $1/k$ dependence for the computational dominated regime.*



(b)

FIG. 2. (*Continued*)

passed to neighboring nodes to continue with the MD evolution. We refer to this case as the "1D" problem. The different curves correspond to different numbers of nodes in the calculation: $k = 1, 2, 4, 8$, and 16. Figures 1(a), 1(b) and 1(c) show results for the workstation cluster, the Intel iPSC/860, and the Meiko, respectively. We see that for small N, the computation is dominated by communication between the nodes. Running on a single node is substantially faster than distributing the calculation on any larger set of nodes. The run times for $k \geq 2$ are roughly the same, since the time to pass messages dominates and no advantage is gained by having fewer masses to update per node. Meanwhile, for large N a cross-over is seen to a computation dominated regime. Indeed the efficiency, the time spent for the serial case ($k = 1$) divided by the number of nodes k times the time spent in the parallel version [13] is close to unity.

If the nodes truly communicated simultaneously, one might expect the time T to be completely independent of k for $k \geq 2$ for small N. However, for all three architectures, we appear to observe roughly $T \propto k$ in the communication limited, small N regime. It is not clear to us at this stage whether this is a hardware or p4 software limitation.

We can present this data in a slightly different way by showing the run time as a function of the number of machines for different N. This is done in Fig. 2, where again data are shown for different parallel environments. We see that for computationally unintensive problems, small N, there is no gain by running on large k. Indeed, $k = 1$, where no communication is required, is optimal. However, as the problem becomes more demanding, we enter a regime where the run time falls off as $1/k$. We emphasize that in the problems of interest to us $N \approx 10^5$. We therefore want to simulate precisely this case where parallel computation is necessary and desirable.

In Fig. 3 we show a plot analogous to that of Fig. 1 for the case of longer range interactions when the position of every mass on every node needs to be passed to the neighboring node to continue the evolution. We refer to this case as the "2D" problem. Note that since each node still only needs to communicate with its two neighboring nodes (now, however, passing more data than the 1D case) and not with *all* of the nodes, we do not need to resort to any type of binary tree reduction summation to efficiently compute the force on a given ion. Despite the more complicated interactions, Fig. 3 has basically the same features as Fig. 1. For the HP cluster of workstation, the

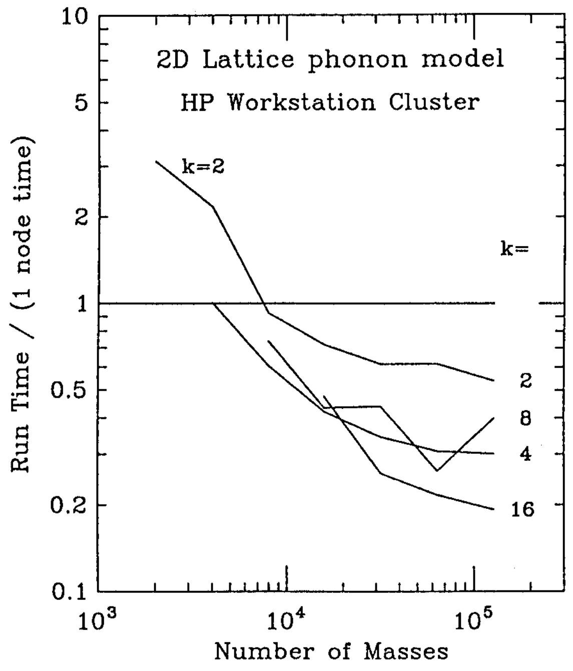


FIG. 3. *Same as for Fig. 1(a) (HP cluster) except V now requires long range communication between the degrees of freedom. Surprisingly, the crossover point between $k = 1$ and $k > 1$ was not evident for the iPSC/860. This is probably due to the fact that entire vectors of coordinates were passed at once, and the "chunk size" of the message passing library was sufficiently large to make the communications overhead actually negligible in this case. A cross-over point is observed for the HP workstation cluster.*

cross-over between the communication limited and the computation limited regimes is somewhat less well defined, and, as expected, occurs at a larger N since the communication required is greater. For the case of the Intel iPSC/860 *no* crossings were found, that is to say for the cases we could consider the data was essentially in the linear speedup regime (i.e., as in the right hand portions of Figs. 1 and 3). We attribute this to the message passing chunk size on the Intel iPSC/860 being sufficiently large so that the passing of the additional ionic coordinates did not significantly increase the amount of time passing messages.

6. Results: Interacting electron system.

Parallelized Monte Carlo Simulations: Since our group [14] does a very large number of Monte Carlo simulations (roughly hundreds of thousands of workstation hours per year) using our own Fortran and C programs, we decided that to be effective in taking advantage of the emerging parallel computing technologies, we should (1) write codes which are nearly machine independent and (2) exploit the statistical nature of the Monte Carlo algorithm as much as possible. The p4 parallel programming system supplies us with the solution to point (1). To address point (2) we have written a number of software tools, applied on top of the p4 environment, which we have found very effective in running simulations on parallel machines or clusters of workstations. This method will now be described.

Generally speaking, in Monte Carlo one has a "state" variable X that is a point in a large dimensional space (typically a 1000 to 10000 dimensional vector for our problems discussed in this paper). The state X is distributed via some complicated probability distribution $\mathcal{P}(X)$ that can nonetheless be sampled. For the problems discussed here the Metropolis random walk algorithm is used [15] and $\mathcal{P}(X)$ is the Boltzmann factor $\exp(-\beta H)$ mentioned above. In the following, however, the form of $\mathcal{P}(X)$ is immaterial. We wish to compute averages of a physical quantities $A(X)$ given by:

$$\langle A \rangle = \int A(X)\mathcal{P}(X)dX\,. \tag{4}$$

Where the integral indicates a quadrature in an extremely large dimensional space. Using a sampling algorithm for $\mathcal{P}$ (e.g., the Metropolis algorithm), one simply generates and collects a set $\{X_i\}$ of $\mathcal{N}$ sampled points, and uses

$$\langle A \rangle = \frac{1}{\mathcal{N}}\sum_{i=1}^{\mathcal{N}} A(X_i) \tag{5}$$

as an estimate for the desired value $\langle A \rangle$. The expected error will go as $\sigma/\sqrt{\mathcal{N}}$, with σ depending on the degree of correlation and fluctuation of $A(X)$ in the sampling.

The key point is that if one can use independent processors each to generate its own set of collected points $\{X_i\}$, then these sets can simply be combined at the end to form an overall estimate for $\langle A \rangle$ and also for its error. Hence the algorithm is very effectively parallelized, there being only trivial communication at the beginning and end of the calculation. In our implementation there is also a trivial communication during the calculation in order to "load balance" the processors: if the number of requested runs is larger than the number of processors, then the earlier finishing processors are given additional runs to perform.

We have implemented this strategy by trying to be as general and labor saving as possible. In fact, the parallel interface program we have written enables us to take any of our Monte Carlo (or similarly related) programs and run them on a parallel machine with essentially *no modification.* We have done this as follows. A program was written in C using the p4 message passing routines to set up linkage between a Master and many Slave processors. The Master coordinates the work, initializing and giving commands and input data (with a possible random seed) to be run by the Slaves. A Slave creates the necessary temporary directories and runs its command by the standard UNIX `system()` call. The Slave's command can be a Fortran or C executable, a pipe, or simply a shell script (that, say, in turn calls more complicated programs). When a Slave has finished its run, it sends a copy of each generated output file back to the Master which collects and labels them. If any more jobs are required the Slave gets the next command, otherwise it terminates. When all the runs have been completed, the Master averages all the data together and performs error estimates.

The only constraints our program imposes are the facts that (1) the application places all desired data required into ASCII files and (2) the data in those files can be simply averaged together to construct the overall estimates of quantities and their associated error bars. Occasionally an existing application program must be modified so that its output is suitable for our file averaging program. This is usually quite simple, and in general a good deal easier than explicitly parallelizing (say via p4 library calls) the application source code itself (perhaps by performing a global parallel reduction over variables one desires to average).

Although not essential, a "front end" to the program was written in the Perl scripting language. The Perl script provides the interface between the user at the command line or via a configuration file the user has constructed (it is essentially a superset of the p4 "procgroup" file format). The script also performs all the initializing, file averaging, and error estimation. The Perl script spawns and communicates with the Master processor via a two-way pipe communication provided by the Perl `open2()` subroutine. The reason for choosing Perl over C for these tasks is the convenient and robust string manipulation functions of Perl compared with using C regexp routines. The enhanced string manipulation was convenient for the parsing of the configuration file, and was nearly essential for the writing of the "file averaging" portion of the Perl program.

The file averaging script solves the practical problem of how one averages data embedded in files containing a mixture of text and numerical data. For example, if three output files from different processors contain the strings "`Energy(T)= 2.12`", "`Energy(T)= 2.22`", and "`Energy(T)= 2.26`", the first field will "average" to "`Energy(T)=`", while the second field will average to `2.20` along with an error estimate (in this case about `0.06`). The only constraint is that the files all possess the same "word pattern". That is to say, if nth line of file 1 contains m word fields (separated by white space), then the nth line of all the remaining files must also have m fields. Of course, the number of word fields need not be the same for each line inside a file. We find that this program handles nearly all of the output formats our programs generate. It can be easily generalized to treat new cases.

Roughly speaking, the startup time of 16-64 Slaves is rarely over a minute, and often much quicker, especially on the Meiko-CS2. The p4 software suite provides a startup server (as in PVM) to avoid most of the remote shell (`rsh`) startup overhead on a cluster of workstations. For the problems we have considered, however, the rsh startup time is small compared with the total run time. There is also a potential bottleneck in having the Perl script perform all of the output averaging. Typically, our output data files are less than 50KB in size, and it turns out this time also that it is acceptably small, except perhaps for very short test runs. Some large file outputs (500KB) tests were performed and took an average of 1 to 2 minutes for 16 node files on a typical UNIX workstation. If for some application this time becomes too large, the present program could be modified to begin the averaging as soon as an output file comes in, rather than wait for all files to be collected. Alternatively, the Perl script could be replaced by a (presumably faster) C program. As a last resort, the averaging tasks could be spread out over the processors.

The results presented below are Monte Carlo simulations, but they also include averaging over a random potential energy term (that is to say, a Monte Carlo simulation needs to be performed for each disorder realization). In this case a large number of statistically independent calculations are averaged together regardless of whether the simulations are done in parallel or serially. For pure systems, on the other hand, one may ask what the drawbacks to splitting a total of $T_{\rm total}$ Monte Carlo time steps over k processors with $T_{\rm proc} = T_{\rm total}/k$ Monte Carlos steps per processor are. What enters into this consideration is the correlation or relaxation time $T_{\rm relax}$, defined to be the Monte Carlo time steps t required for a sampled state X_{i+t} to lose memory of, or become independent of its state at time i: X_i. This time $t = T_{\rm relax}$ is dependent on the choice of Metropolis Monte Carlo *apriori* transition probability, the problem being investigated, and the physical quantities being measured. Roughly speaking, the initial T_{relax} Monte Carlo steps must be discarded from averaging. As k increases T_{proc} decreases down to T_{relax} and hence may provide *no* configurations acceptable for averaging.

From general statistical considerations, a rough estimate of the error for T_{total} Monte Carlo steps performed on a single processor when the initial T_{relax} steps are discarded is $\sim \sigma/\sqrt{T_{\text{total}} - T_{\text{relax}}}$. σ is the variance of the quantity being measured. In spreading the T_{total} Monte Carlo steps over k processors, we have to ensure that each independent run is properly equilibrated. Hence we must discard T_{relax} steps on *each* processor, for a total of $k \times T_{\text{relax}}$ steps thrown out as opposed to only T_{relax} for the single processor case. An analogous rough estimate for the k processor case is $\sim \sigma/\sqrt{T_{\text{total}} - k \times T_{\text{relax}}}$, and so when using k processors there is a danger that the error is much larger than for the single processor when $k \times T_{\text{relax}} \to T_{\text{total}}$, since the denominator for the k processor error estimate goes to zero in that limit.

The above rough estimates may be made more quantitative if we make a few assumptions that are nearly always satisfied in real Monte Carlo simulations. Let $x(t)$ be the value of an observable of interest at time t (e.g., the total energy of the system). We can usually take the autocorrelation function to be exponentially decaying in time:

$$\langle x(t)x(t')\rangle = \langle x\rangle^2 + \sigma^2 \exp(-|t-t'|/\tau)\,. \tag{6}$$

The angular brackets denote averages (here of a physical quantity measured at two different times), $\sigma^2 \equiv \langle x^2\rangle - \langle x\rangle^2$, and τ is the exponential decay constant, or autocorrelation time. Now, if the first T_{relax} out of T_{total} steps are discarded, then the average $\tilde{x}$, which we take for the measurement part of the run is clearly

$$\tilde{x} = \left(\frac{1}{T_{\text{total}} - T_{\text{relax}}}\right) \sum_{t=T_{\text{relax}}+1}^{T_{\text{total}}} x(t) \tag{7}$$

and its corresponding error is $\epsilon = \sqrt{\langle \tilde{x}^2\rangle - \langle \tilde{x}\rangle^2}$. ϵ may be calculated by squaring Eq. (7), averaging using Eq. (6), and summing the resulting geometrical series. We present the general result for k processors, where we can assume results from different processors are independent, but within a given processor run the quantities are correlated over times $\sim \tau$. The error ϵ_k for k processors is then

$$\epsilon_k^2 = \frac{\sigma^2}{\mathcal{T}_k k}\left[1 + \frac{2e^{-1/\tau}}{1-e^{-1/\tau}} - \frac{2e^{-1/\tau}}{\mathcal{T}_k}\frac{(1-e^{-\mathcal{T}_k/\tau})}{(1-e^{-1/\tau})^2}\right] \tag{8}$$

where $\mathcal{T}_k \equiv (T_{\text{total}} - kT_{\text{relax}})/k$, which is the number of measurements used on a single processor.

The prefactor in front of the square brackets, $\sigma^2/T_k k$ simply gives the rough estimate of the error mentioned earlier. The quantity inside the square brackets is due to autocorrelation (note that as $\tau \to 0$ the quantity in the brackets $\to 1$). The prefactor by itself suggests $\epsilon_k \geq \epsilon_1$, which is what we generally expect. However, consideration of the full expression in Eq. (8) shows that the ratio ϵ_k/ϵ_1 can actually be less than one, but only in the region where $T_{\text{relax}} \leq \tau$. This is due to the fact that with large autocorrelation it is better to have k independent runs with few measurements, rather than a single run with many correlated measurements. It is, however, dangerous to discard fewer than τ steps because one is then not guaranteed to obtain properly equilibrated results. In general, the equilibration time T_{relax} will be at least as big as τ, and in some cases a good deal larger. Thus, no matter how tempting, it is best to avoid the region $\epsilon_k/\epsilon_1 < 1$ and to take $T_{\text{relax}} > \tau$. Results for the various regimes taking typical values of $T_{\text{relax}} = 100$ and $T_{\text{total}} = 50000$ are displayed in Fig. 4. It is encouraging to note that (in this example) for a fairly wide range of processors number, say $1 \leq k < T_{\text{total}}/2T_{\text{relax}} = 250$, the k processor error is no more than 50% more than the $k = 1$ processor case. The discussion of error estimates in the

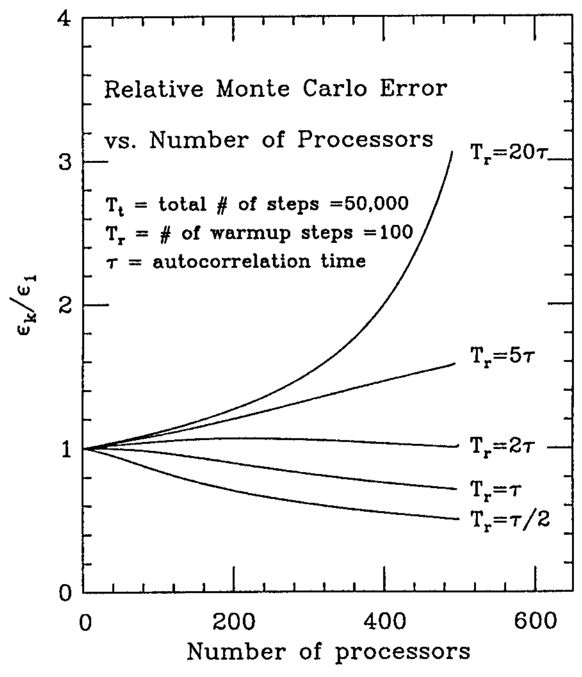


FIG. 4. *An example of the ratio ϵ_k/ϵ_1 of the Monte Carlo error for a k processor run to a single processor run. As k becomes large, an increasingly large fraction of the run time on the k processors is spent in a wasted, duplicate effort at equilibration.*

present paragraph ignores, of course, the large benefit that the k processor case finishes in nearly $1/k$ of the amount of wallclock time for the single processor case. This is the basic reason why one considers parallel processing in the first place.

If τ is large ($\tau > 5$) and yet $\mathcal{T}_k \gg \tau$ (i.e., each run is not overwhelmed by the autocorrelation), then Eq. (8) reduces to $\epsilon_k^2 \approx (2\tau/\mathcal{T}_k)(\sigma^2/k)$. That is to say, the effective number of independent measurements is $\mathcal{T}_k/2\tau$ and correlated blocks are roughly 2τ long. Going in the other direction, when there is only one measurement per processor ($\mathcal{T}_k = 1$) one finds that $\epsilon_k/\epsilon_1 \approx \sqrt{T_{\text{relax}}/2\tau}$, and this form describes the endpoints of the curves in Fig. 4 rather closely.

Hubbard Model Results: Clearly, the parallelization of codes in the way we had described above side-steps the deepest issues of parallel computation. We therefore do not report on any timing benchmarks as we did in the case of the genuinely distributed problem. We can easily increase the Monte Carlo run time to amortize the cost of starting up the Slave processes and performing the final averaging. This method has the virtue of being rapidly applicable to existing codes, which for quantum MC tend to be quite elaborate [16]. Instead, we will report on some new results for the physics of the $-U$ Hubbard model in the presence of disorder.

Figure 5 displays the off diagonal long-range order-parameter, or superconducting pair-pair correlation function P_s for 4×4 and 6×6 lattices as a function of disorder Δ. For large enough Δ, P_s is driven to zero as the system size increases, indicating a localized, rather than superconducting, state. In Fig. 6 we show a set of quantities which measure the transport properties of the electrons in the $-U$ Hubbard model, as a function of the amount of disorder Δ put into the site energies. These quantities have hitherto only been reported in the clean system [17,18]. The required averaging over many disorder realizations makes this problem of "Grand Challenge" calibre. The solid triangles show the kinetic energy of the electrons. This is a local quantity, which varies fairly smoothly as Δ is increased. The open circles are the superfluid density D_s, as determined by analyzing the momentum and frequency dependence of the current-current correlation function [18]. The full circles are an alternate measure of this quantity, obtained by looking at the asymptotic behavior of the equal time pair-pair correlations. When these are nonzero, some fraction of the electrons in the system can flow with zero resistance. Finally, the full squares measure the "Drude weight" D, an ingredient in the normal conductivity. When D becomes zero the system cannot conduct. For a clean system where the transitions are driven by interactions rather than disorder, three phases are possible: a superconductor where D and D_s are both nonzero; a normal metal where only D is nonzero; and an insulator where both vanish. In our disordered system, the criteria are in fact more complicated.

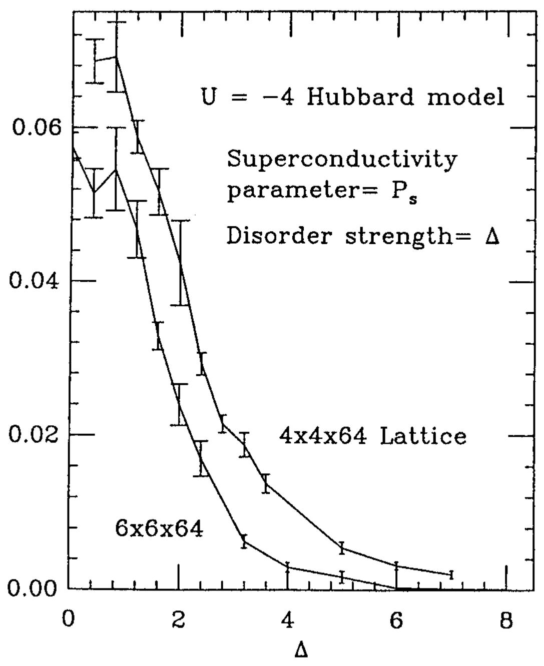


FIG. 5. *The pair-pair correlation function P_s for different strengths of randomness is shown. Data is given for 4×4 and 6×6 spatial lattices with $L = 64$ imaginary time points to begin to assess finite size effects. When P_s is nonzero, the system is in a superconducting state.*

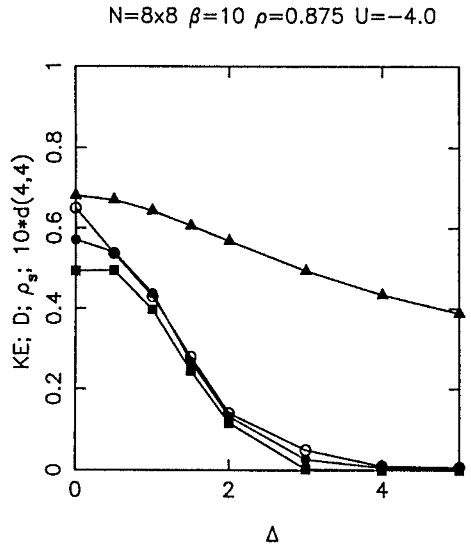


FIG. 6. *The Kinetic energy (solid triangles), transport properties (D and ρ_s, (solid squares and open circles, respectively)), and pair-pair correlation function (P_s = d(4, 4)) (solid circles) for an 8×8 disordered Hubbard lattice.*

These last three quantities are all global measurements of the mobility of the electrons in the model, and therefore unlike the kinetic energy they all are capable of showing indications of a transition between these phases as the disorder strength Δ is increased. The results we show suggest that there are superconducting and insulating phases, with no intervening metallic region. We will need to do considerably more data analysis in order to determine if indeed there is a finite value Δ_c and the numerical location of this transition point. In particular, it will be necessary to do a finite size scaling study of the behavior of these quantities with lattice size.

In order to generate this data, we have used the above mentioned software tools to generate input files, submit programs, and average data. Each point in Fig. 5 represents an average over 20-64 independent configurations of random site energies. This was done by running on 10-16 HP/715 workstations and assigning 2-4 disorder realizations to each node. The total wall clock time for these simulations was about 2500 workstation hours. In order to undertake the detailed scaling analysis, we will run this program, using the parallel interfaces tools we have developed, on the 128 node Meiko CS-2 or possibly an IBM SP1.

7. Conclusions.

We have first described results for Molecular Dynamics simulations for a set of coupled masses and springs, which is set up to model of interacting magnetic flux lines in a superconductor. We have focussed on the efficiency when a single such task is distributed (using p4) over a set of communicating processors, and have given timing results for different numbers of CPUs, problem sizes, and types of platforms. We have also tested two different types of potentials between the masses, one short-ranged and one long-ranged, which require different levels of internodal communication.

The Molecular Dynamics simulation of lattice vibrations has a much simpler algorithmic structure than quantum Monte Carlo simulations of interacting electrons. For these Monte Carlo problems, we have explored a less elegant, but nevertheless effective approach which distributes largely independent runs on a set of processors. This is an attractive procedure for quantum MC simulations, both because it lends itself to efficient parallelization, and preserves existing, complicated, codes. Indeed, we have developed a set of simple software tools which allow the easy construction of generic input files, the submission of multiple independent jobs to different nodes, and the subsequent averaging of data files [19]. However, we do emphasize that the approach could have significant drawbacks. An important one is that each node must then have sufficient memory to store all the data for its own run, a memory requirement which in some of our applications goes as $1/T^2$ where T is the temperature. Since we are often interested in the $T \rightarrow 0$ limit, this could be a problem. In contrast, in a "distributed calculation" method, the memory requirements themselves can also be distributed.

Acknowledgement. This work was supported by the National Science Foundation under grant No. ASC-9405041. We thank Jeff Collins for bringing up and installing p4 on the ECE HP cluster at U. C. Davis. We thank Steve Slater of U. C. Berkeley for useful information about p4 and parallel Monte Carlo calculations, and to Nandini Trivedi and Mohit Randeria for illuminating discussions on the physics of the $-U$ Hubbard model. We thank the Academic Computing Service in the College of Engineering for the use of the Intel iPSC/860 hypercube and the Lawrence Livermore National Laboratory for the use of the Meiko CS-2.

REFERENCES

[1] R. Butler and E. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. Parallel Computing 20, April 1994. Also Argonne National Laboratory preprint MCS-P362-0493. p4 has its origins from the work of J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson and R. Stevens. Portable Programs for Parallel Processors, Holt, Rinehart, and Winston, 1987.

[2] N. W. Ashcroft and N. D. Mermin. Solid State Physics. W. B. Saunders, 1976.

[3] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. Numerical Recipes. Cambridge University Press, 1986.

[4] R. Kubo. Statistical Mechanics. North-Holland, 1965.

[5] See the articles by D. R. Nelson and D. S. Fisher. In *Phenomenology and Applications of High Temperature Superconductors.* K. S. Bedell *et al.* (ed), Addison-Wesley, 1992.

[6] P. G. De Gennes and J. Matricon. Rev. Mod. Phys. 36, 1964: 45.

[7] J. L. Cardy (ed). Finite Size Scaling. North-Holland, 1988.

[8] A. Montorsi (ed). The Hubbard Model. World Scientific, 1992.

[9] R. T. Scalettar, E. Y. Loh, Jr., J. E. Gubernatis, A. Moreo, S. R. White, D. J. Scalapino, R. L. Sugar and E. Dagotto. Phys. Rev. Lett. 62, 1989: 1407.

[10] R. Blankenbecler, D. J. Scalapino and R. L. Sugar. Phys. Rev. D24, 1981: 2278.

[11] D. W. Heermann and A. N. Burkitt. Parallel Algorithms in Computational Science. Springer-Verlag, 1991.

[12] A 64 node Intel Gamma has been found to perform at roughly the same speed as a 4 processor CRAY-XMP for some MD applications. See, for example, S. J. Plimpton. In Proceedings of 5th Distributed Memory Computing Conference (published by IEEE), Charleston, SC, April 1990; S. J. Plimpton and G. Heffelfinger. In Proceedings of Scalable High Performance Computing Conference (published by IEEE), Williamsburg, VA, April 1992.

[13] D. P. Bertsekas and J. N. Tsitsiklis. Parallel and Distributed Computation. Prentice Hall, 1989.

[14] A feature of quantum MC in the condensed matter community, as opposed to, for example, lattice gauge theory calculations in high energy physics, is that there is no single, underlying model whose study is central to the entire community for highly extended periods of time. In such a situation where models evolve rapidly and a set of groups are interested in rather diverse phenomena, it is clearly less sensible to spend large amounts of time optimizing codes, and the ability to take

advantage easily of serial codes on parallel platforms becomes more valuable.

[15] P. ALTEVOGT AND A. LINKE. Parallel Computing 19, 1993: 1041.

[16] We emphasize that for nearly all problems we consider there is no loss in efficiency in parallelizing the Monte Carlo program in the manner that we have. Indeed, one may argue that any parallelization scheme based on message passing would run the risk of being significantly slower. A case where our method would fail would be for a system so large that one was unable to even equilibrate one independent system in the available CPU time. In that case some method of breaking up the large system into subsystems would have to be attempted. Fortunately, very few of the applications we work on in fall into this category.

[17] A. MOREO AND D. J. SCALAPINO. Phys. Rev. Lett. 66, 1991: 946; and M. RANDERIA, N. TRIVEDI, A. MOREO AND R. T. SCALETTAR. Phys. Rev. Lett. 69, 1992: 2001.

[18] D. J. SCALAPINO, S. R. WHITE AND S. C. ZHANG. Phys. Rev. B47, 1993: 7995.

[19] Anyone interested in obtaining the p4 parallel independent run interface tools we have developed are welcome to try them. Contact `runge@solid.ucdavis.edu` for more information. We are interested in thinking of ways of extending these tools to treat different types of problems not as straight forward as the independent data averaging we have performed.