Monte Carlo and Molecular Dynamics Simulations Using p4

K. J. Runge,[1] P. Lee,[2] J. Correa,[1] R. T. Scalettar,[1] and V. Oklobdzija[2]
[1] *Physics Department, University of California, Davis, CA 95616*
[2] *Department of Electrical and Computer Engineering, University of California, Davis, CA 95616*

Monte Carlo (MC) and Molecular Dynamics (MD) simulations are powerful tools for understanding the properties of systems of interacting electrons and phonons in a solid. When mobile electrons are studied, these simulations are limited to a few hundred particles. More powerful machines and algorithms must be used to address many of the most important issues in the field. We present results from using the p4 parallel programming system on a variety of parallel architectures to conduct MC and MD simulations.

I. INTRODUCTION

Implementing parallel computations requires the distribution of a problem, undoable on a single cpu, over a network of coupled processors. Such a calculation involves interconnected questions concerning the underlying application and computer science issues of properly sequencing tasks, passing messages, etc. This requires significant rewriting of conventional serial codes. However, many applications do not need the full power of the parallel machine for a single run. Rather their complexity resides in conducting a large number of separate calculations. Most Monte Carlo applications fall into this class, since statistically independent configurations are needed which can be generated by launching simultaneous, non-communicating copies of the code with different random number seeds. Furthermore, it is frequently necessary to conduct a series of runs, for example sweeping the temperature, and different nodes can be assigned different parameter values. Finally, in cases where defects are present it is necessary to conduct a disorder average over hundreds or thousands of 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 platforms. These can range from a cluster of UNIX workstations to massively parallel supercomputers. The p4 package, available at `ftp://info.mcs.anl.gov/pub/p4`, is similar to PVM. For our work we use processor to processor message passing and timing library routines p4 provides. p4 also contains additional routines and interfaces. In the different environments (HP, Sparc, AIX, Linux, iPSC/860, and Meiko CS-2) in which we have installed p4, we have found (except for the HP) that the compilation, installation, and usage is straight forward.

In this paper we will 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 study a classical model of interacting lattice vibrations using Molecular Dynamics. We have written codes using the p4 library to produce portable parallelized programs. In the latter case of independent computations, we study by Quantum Monte Carlo algorithm the two-dimensional Hubbard Hamiltonian, a model which provides a computational challenge at the forefront of current solid state theory. Because of the complexity of the computations, our goal is simple parallelization with the least rewriting of application codes. We describe software tools we developed which enable the user easily to submit a set of independent simulations to a parallel computer or meta-computer. The software will also perform the necessary analysis of general data files, averaging data returned from individual nodes automatically.

The remainder of this paper is organized as follows: In Section II we review the basic ideas behind the models we are studying and key features of the Monte Carlo (MC) and Molecular Dynamics (MD) algorithms for serial codes. Section III will briefly describe the hardware platforms on which we test parallelized algorithms. Section IV contains details of our first application: a distributed MD calculation of interacting lattice vibrations. Section V 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. Section VI concludes with summarizing remarks.

II. REVIEW OF MOLECULAR DYNAMICS AND MONTE CARLO

Molecular Dynamics for a Lattice Vibration Model: In condensed matter physics one is interested in calculating properties of interacting electrons and ions in a crystal. This is a complicated task, and so 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 a *classical* model which consists of a "kinetic energy"

K which is a function of the particle velocities, and a "potential energy" V which depends on the particle positions. For realistic forms of V the models are not soluble analytically. Here one numerical approach is "Molecular Dynamics." One integrates Newton's equations of motion ($F = ma$) numerically by discretizing time and employing Runge-Kutta, leap-frog, or some equivalent finite difference approach. [3]

There is a large class of different problems which can be described in a similar fashion. A common feature is a rather local set of interactions between different coordinates. This simplifies things considerably. Nevertheless, the problems attempt to describe systems with a very large number of particles, so the most powerful computational resources are useful in their solution.

We are also interested in a second type of model in which *quantum mechanical* effects are crucial. The MC method we will use is too detailed to describe fully here. [8] In brief, the quantum mechanical problem of evaluating the partition function 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 a 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. Besides this added dimension, the primary difficulty is that, unlike many classical models which have simplified, short range interactions, here the equivalent classical model has interactions which are *very* non-local (the interactions are described by a determinant of a 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 each other frequently and in a complicated manner. [9] For this reason we explore only simpler parallelization schemes based on disorder or statistical averaging, which, we emphasize, are nevertheless useful and efficient ways to approach this class of problems.

III. TESTBED ARCHITECTURES

Workstation Cluster: Our first test architecture consists of a set of approximately 40 HP 715 workstations linked by ethernet in the 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 [10] with a MIMD architecture where the computation is carried out by logically independent but cooperate 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.

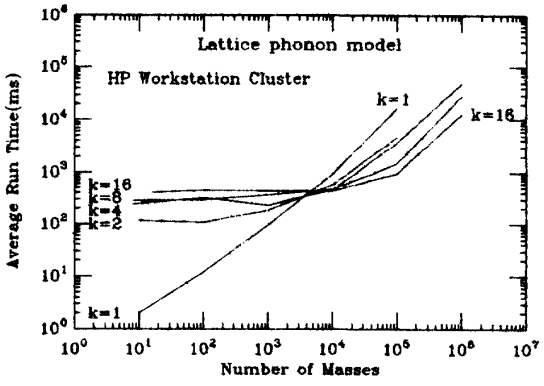

Fig. 1a: The run time T versus number of masses (problem size) N, for different numbers of nodes k. In each case we did 1000 MD time steps. For small N the efficiency is low, while for large N it approaches unity. Figs. 1a,b,c show results for a cluster of HP-715 workstations, a 32 node Intel iPSC/860, 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.

IV. RESULTS: INTERACTING LATTICE VIBRATIONS

In our MD simulations, we started with a conventional serial code which evolved the entire phonon lattice on a single CPU. We then parallelized the code by dividing the lattice into sections, and inserted appropriate p4 library commands, for example the processor to processor message passing routines (e.g. `p4_send()` and `p4_recv()`). These p4 commands pass the values of the ionic positions from processor to processor and also synchronized the calculation.

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 very 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 passed to neighboring nodes to continue with the MD evolution.

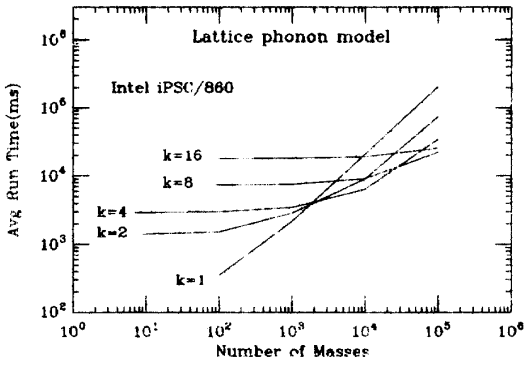

Fig. 1b:

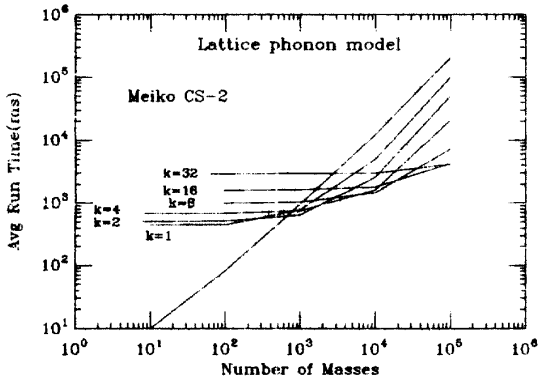

Fig. 1c:

The different curves correspond to different numbers of nodes in the calculation: $k = 1, 2, 4, 8$, and 16. Figs. 1a,b,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 [11] is close to unity.

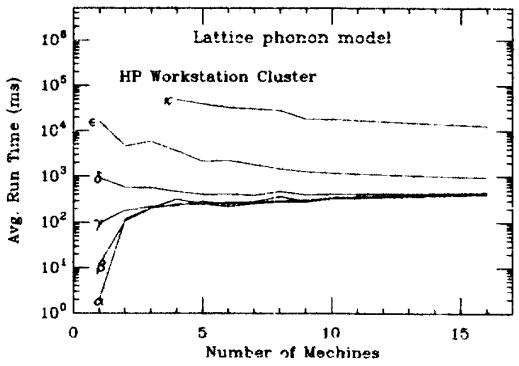

Fig. 2a: The run time T versus number of nodes k, for different numbers of masses N. In each case we did 1000 MD steps. For small N, a single cpu ($k = 1$) is optimal, while for larger N the run time T first decreases with k, then eventually saturates as the number of computations per node becomes small. α=10^1, β=10^2, γ=10^3, δ=10^4, ϵ=10^5, κ=10^6.

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 $T \propto k$ in the communication limited, small N regime. It is not clear to us whether this is a hardware or p4 software limitation.

We can present this data in a different way by showing 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, $k = 1$ (no communication) 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. Despite the more complicated interactions, Fig. 3 has basically the same features as Fig. 1. For the HP cluster of workstation, the 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.

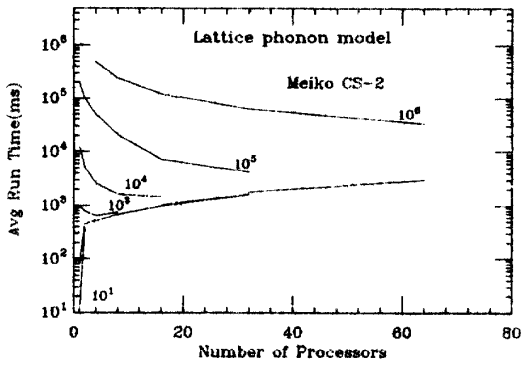

Fig. 2b:

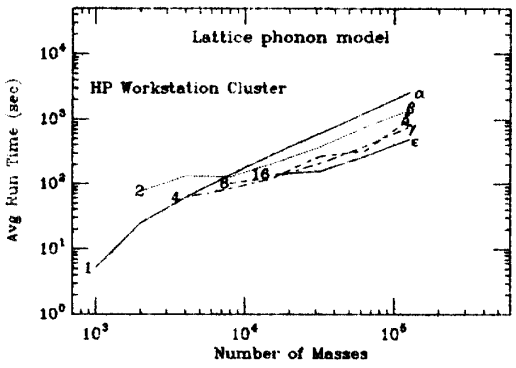

Fig. 3a: Same as for Fig. 1 except V now reqnires long range communication between the degrees of freedom. Surprisingly, the crossover point between $k = 1$ and $k > 1$ was not evident for the iPSC/860. This is probably due to the fact that entire vectors of coordinates were passed at once, and the "chunk size" of the message passing library was sufficiently large to make the communications overhead small in this case. A cross-over is observed for the HP workstation cluster. α=2597, β=1400, γ=783, δ=1037, ϵ=500.

V. RESULTS: INTERACTING ELECTRON SYSTEM

Parallelized Monte Carlo Simulations: Since our group [12] does a very large number of MC simulations ($\sim 10^4$ workstation hours per year) using our own Fortran and C programs, we decided, to be effective in taking advantage of the emerging parallel computing technologies, to 1) write code that is nearly machine independent and to 2) exploit the statistical nature of the MC 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, that we have found very effective in running simulations on parallel machines or clusters of workstations. This method will now be described.

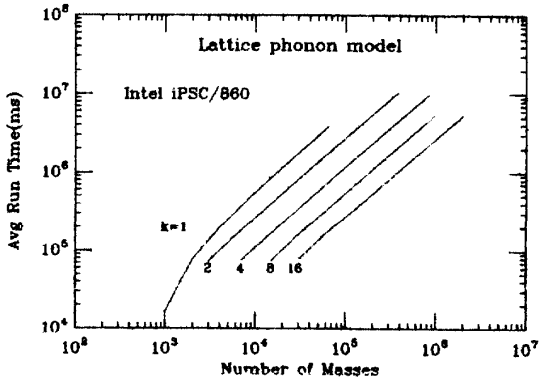

Fig. 3b:

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 be sampled. [13] We wish to compute averages of physical quantities $A(X)$ given by: $\overline{A} = \int A(X)\mathcal{P}(X)dX$. Where the integral indicates a quadrature in an extremely large dimensional space. Using a sampling algorithm for $\mathcal{P}$, one simply generates and collects a set $\{X_i\}$ of $\mathcal{N}$ sampled points, and uses $\hat{A} = (1/\mathcal{N})\sum_{i=1}^{\mathcal{N}} A(X_i)$ as an estimate for the desired value $\overline{A}$. 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 one can use independent processors each to generate its own set of collected points $\{X_i\}$, and then these sets are combined to form an overall estimate for $\overline{A}$ and 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 simple communication in order to "load balance" the processors: if the number of requested runs is larger than the

number of processors, the earlier finishing processors are given more 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 to 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. When a Slave has finished its run, it sends a copy of each output file generated back to the Master who collects and labels them. If more jobs are required the Slave gets the next one, otherwise it terminates. When all the runs are done, the Master averages the data together and performs error estimates. The only constraint our program imposes is that the application places all desired data into ascii files and that data in those files can be simply averaged together to construct the overall estimates of quantities and error bars.

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) and the script also performs all the initializing, file averaging, and error estimation. The Perl script spawns and communicates with the Master processor via the two-way pipe communication provided by the Perl **open2()** subroutine. The reason for choosing Perl over C for these tasks is its extremely convenient string manipulation capabilities over the standard C regexp libraries.

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 contains 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", *i.e.*, if n−th line of file 1 contains m_n word fields (separated by white space), then the n−th line of all the remaining files must also have m_n fields.

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 to avoid most of the remote shell (**rsh**) startup overhead, but for the problems we considered this startup time is small compared to the total run time. There is also a potential bottleneck in having the Perl script perform all of the output averaging. Typically, our output datafiles are less than 50KB in size, and it turns out this time also is acceptably small, except perhaps for very short test runs.

The results presented below are MC simulations, but they also include averaging over a random potential energy term (that is to say, a MC 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 are 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. What enters into this consideration is the correlation or relaxation time $T_{\rm relax}$, defined to be the number of Monte Carlo times 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 MC algorithm and on the problem being investigated. The initial $T_{\rm relax}$ or so Monte Carlo steps must be discarded from averaging. As k increases $T_{\rm proc} \to T_{\rm relax}$ and hence may provide *no* configurations acceptable for averaging.

From general statistical considerations, a rough estimate of the error for $T_{\rm total}$ MC steps performed on a single processor with $T_{\rm relax}$ steps discarded is $\sim \sigma/\sqrt{T_{\rm total} - T_{\rm relax}}$. In spreading the $T_{\rm total}$ MC steps over k processors, we have to be concerned that each independent run is properly equilibrated, hence we must discard $T_{\rm relax}$ steps on *each* processor, for a total of $k \times T_{\rm relax}$ steps thrown out as opposed to only $T_{\rm relax}$ for the single processor case. The estimate for the k processor case is $\sim \sigma/\sqrt{T_{\rm total} - k \times T_{\rm relax}}$. So when using k processors there is a danger the error is much larger than for the single processor as $k \times T_{\rm relax} \to T_{\rm total}$.

These estimates may be made more quantitative if we make a few assumptions about the form of the correlation. Let $x(t)$ be the value of an observable of interest at time t, *e.g.* the total energy of the system. We can nearly always take the autocorrelation to decay exponentially with time: $\langle x(t)x(t')\rangle = \langle x\rangle^2 + \sigma^2 \exp(-|t-t'|/\tau)$ 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 $T_{\rm relax}$ steps are discarded, the average, $\tilde{x}$, of the used portion of the run is

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

and its error is $\epsilon = \sqrt{\langle \tilde{x}^2 \rangle - \langle \tilde{x} \rangle^2}$. ϵ may be calculated from the above equations 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

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

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

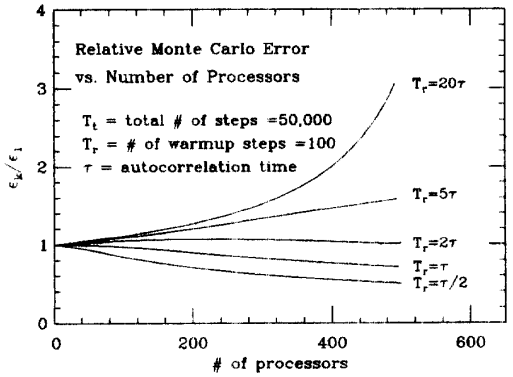

Fig. 4: The ratio ϵ_k/ϵ_1 of the error for a k processor run to a single processor run. As k becomes large, an increasingly large fraction of the run time on the k processors is spent in a wasted, duplicate effort at equilibration.

The prefactor in front of the square brackets, $\sigma^2/\mathcal{T}_k k$ simply gives the rough estimate of the error mentioned above. The quantity inside the square brackets is due to autocorrelation The prefactor alone itself suggests $\epsilon_k \geq \epsilon_1$, however, 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 $T_{\rm relax} \leq \tau$. This is due to the fact that with large autocorrelation is it 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 $T_{\rm relax}$ will be at least as big as τ, and in some cases a good deal larger. Thus it is best to avoid the region $\epsilon_k/\epsilon_1 < 1$ and to take $T_{\rm relax} > \tau$. Results for the various regimes taking typical values of $T_{\rm relax} = 100$ and $T_{\rm total} = 50000$ are displayed in Fig. 4. It is encouraging to note that (in this example) for a fairly wide range of processors number (*i.e.* up to 250) the ϵ_k is no more than 50% greater than the $k = 1$ processor case. The discussion of error estimates in the 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.

Hubbard Model Results: Clearly the parallelization of codes in the way we have 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 MC run time to amortize the cost of starting up the Slave processes and performing the final averaging. This method does has the virtue of being rapidly applicable to existing codes, which for quantum MC tend to be quite elaborate. Instead, we will report on some new results for the physics of the $-U$ Hubbard model in the presence of disorder.

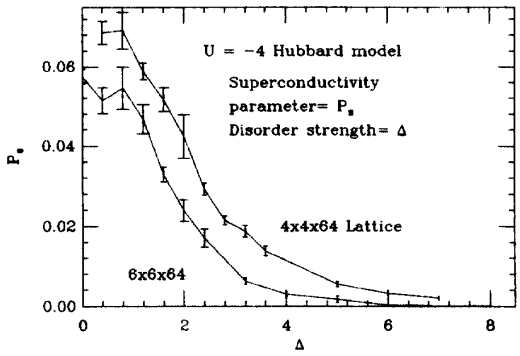

Fig. 5: The pair correlation function P_s for different strengths of randomness is shown. Data is given for 4×4 and 6×6 spatial lattices with $L = 64$ imaginary time points to begin to assess finite size effects. When P_s is nonzero, the system is in a superconducting state.

In Fig. 5 we show a measure of the tendency towards superconductivity of the system, the "pair-pair correlation function," as a function of the amount of disorder put into the lattice. As the disorder strength Δ is increased, the superconductivity pair-pair correlation function is driven to zero. 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 represents an average over 64 independent configurations of random site energies. This was done by running on 16 HP/715 workstations and assigning 4 disorder realizations to each node. Results for 4×4 and 6×6 are presented. The 4×4 sweep took 8 hours of wall clock time, hence representing a total of 128 workstation hours. The 6×6 data took 24 wall clock hours and represent only 32 independent

random site realizations per point. We intend to run this program, using our parallel tools we have developed, on a 128 node Meiko CS-2 or possibly an IBM SP1 to obtain a more accurate picture of the possible phase transition.

VI. CONCLUSIONS

We have first described results for Molecular Dynamics simulations for a set of coupled masses and springs, which, in actual fact, is really a model of interacting magnetic flux lines in a superconductor. We have focused 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 types of potentials between the masses, one short- and one long- ranged, which require rather quite 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, but also because it preserves existing, complicated, codes. Indeed, we have developed a set of simple software tools [14] 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.

ACKNOWLEDGMENTS

This work was supported by the NSF under grant No ASC-9405041. We thank Jeff Collins for bringing up and installing p4 on the ECE HP cluster at U.C. Davis. Thanks also goes to Prof Jasmina Vukic and Steve Slater of U.C. Berkeley for useful information about p4 and parallel Monte Carlo calculations. 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.

[1] Ralph Butler and Ewing 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 and R. Butler and T. Disz and B. Glickfeld and E. Lusk and R. Overbeek and J. Patterson and R. Stevens, *Portable Programs for Parallel Processors*; Holt, Rinehart, and Winston, 1987.

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

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

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

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

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

[7] 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**, 1407 (1989).

[8] R. Blankenbecler, D.J. Scalapino, and R.L. Sugar, Phys. Rev. **D24**, 2278 (1981).

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

[10] A 64 node Intel Gamma performs at roughly the same speed as a 4 proc 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.

[11] *Parallel and Distributed Computation*, D.P. Bertsekas and J.N. Tsitsiklis, Prentice Hall (1989).

[12] 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 long periods of time. In such a situation where models evolve rapidly and a set of groups are interested in 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.

[13] P. Altevogt and A. Linke, Parallel Computing **19**, 1041 (1993).

[14] Anyone interested in our p4 parallel independent run interface tools is welcome to try them. Contact `runge@solid.ucdavis.edu` for more information.