

Using a Beowulf Cluster to Study the Transitions to  
Superconductivity and Magnetism of  
One-Dimensional Systems

Robert James Sheerer  
University of Northern Iowa

September 30, 1999

# 1 Introduction

Due to the rapid growth of personal computer and networking technology, it has become possible to achieve supercomputer performance from a network of personal computers. These personal computers need not be expensive supercomputers or any computer not found in a college computer lab. By networking the computers together, the processing power of each single computer, or node, can be used in parallel to work on one task. The advantage of developing a cluster of machines is that current technology in the lab can be used for creating a machine which is able to solve computationally intense problems.

During the summer of 1999, the computers in room PYS 308 at the University of Northern Iowa were clustered together. The type of cluster constructed is called a Beowulf Cluster. The Linux operating system was installed on all the computers to facilitate the cluster. The advantages of Linux include its free cost, availability, and support for Beowulf clustering. Linux was installed along with Windows NT so that the lab computers could be used both for their original use (class exercises, student use, etc.) and as a cluster to solve computationally intense problems. A dual-boot mechanism allowed the computers to be switched from one system to the other at boot time.

After the Beowulf was built, the performance of the cluster was tested. Code that was previously written by Dr. John Deisz of the University of Northern Iowa and his collaborators[3] was run on the cluster for benchmarking. The initial results of this benchmark demonstrated that the physical network connecting the computers was too slow to make the Beowulf worthwhile

for this particular code. After upgrading the network, performance improved significantly and the cluster was ready to be used for numerical studies of physical properties.

After the cluster was constructed and tested, the cluster was used to study specific heat and density of state functions for nearly-magnetic and nearly-superconductive materials. Previous studies[3] have found a peak in the specific heat versus temperature curves that mark the temperature where materials become magnetic/superconductive. Materials are magnetic/superconductive at temperatures below this peak, and these materials lose their magnetic/superconductive properties above this peak. In the density of states function a gap is observed to mark this boundary. These peaks in the specific heat and gaps in the density of states have only been studied for three and two-dimensional material. The results for one-dimensional materials were studied using the processing power of the cluster. A break in the specific heat curve was found and a small pseudogap was found in the density of states function.

## **2 Beowulf Clustering**

### **2.1 History of the Beowulf**

In the past, universities, the government, and large companies have used supercomputers to model large numeric problems. These supercomputers have barely met the demand and have been very costly to operate. In the last few years, the personal computer market has become very competitive with prices dropping dramatically and processor speeds increasing. The Center of Excellence in Space Data and Information Sciences (CESDIS), a contractor of NASA, developed the Beowulf Project to harness the power of a large group, or cluster, of these

inexpensive PCs to achieve near supercomputer performance.

The development of the Beowulf Project began in 1994 at CESDIS. CESDIS is in part sponsored by the Earth and Space Sciences (ESS) project, which is within the High Performance Computing and Communications (HPCC) program. CESDIS's mission is to bring together the computer science and engineering research programs at universities and the studies conducted by NASA[2]. The first Beowulf built was to address large data set problems that occurred in ESS applications, such as the Shoemaker-Levy 9 comet collision with Jupiter.

The first Beowulf cluster consisted of 16 PCs with DX4 processors networked with channel bonded Ethernet. This cluster was built by Thomas Sterling and Don Becker. The cluster was a success, and the concept spread throughout NASA and eventually into academic and research institutions. From the start, the development of the Beowulf outside CESDIS grew for its own reasons and the types of Beowulf systems varied drastically from the original configuration. The HPCC now recognizes the Beowulf cluster as a category within the supercomputer community[2].

By the middle 1990's, it became apparent that it was the "right time" in history for Beowulfs. The mass-market competition of PCs and the development of the publicly available Linux OS and GNU compilers made the Beowulf very affordable. As PC prices kept dropping, building parallel computers with off-the-shelf components became more affordable than purchasing expensive supercomputers. Most noticeably, the price per performance ratio of a Beowulf cluster is three to ten times better than a supercomputer, such as a Cray model[6].

## 2.2 General Beowulf Architecture

A Beowulf machine is network of computers used for parallel computations. Unlike other clustering techniques or supercomputers, the Beowulf only uses commodities off the shelf (COTS) equipment, such as PC's running Linux (a free PC version of UNIX) connected with standard Ethernet adapters and cabling[2]. No custom components are required, so creating a Beowulf can easily be accomplished at an educational institute or small business. Also, many Beowulfs have been set up by people without much parallel computing experience.

Beowulf clusters are built in a variety of configurations. The typical Beowulf cluster contains one server computer, or node, and many client nodes. The server node serves files to the client nodes and is usually the computer that starts running a parallel program. Client nodes do not have to have monitors or keyboards. Ideally, they can have a "slim" installation of Linux and cannot be reached from outside the server; i.e., they are typically shielded from the outside network by the server node. Beowulf client nodes can be thought of as additional processors and memory to the server node [6].

Besides the Linux kernel on each node, there really is no special system software required for a Beowulf. A Network File System and remote shells could be set up to operate the parallel computer. However, there are many software packages, such as kernel patches and message passing libraries to facilitate a faster, more stable, and easier to use Beowulf system.

In today's supercomputers, multiple processors are often used instead of a single processor. There are many reasons why parallelism is more beneficial than using a single processor. Pro-

cessor speeds have been doubling every 18 months, while corresponding RAM, bus, and disk speeds have not. Running programs in parallel will decrease these limitations. Also, predictions indicate that processor speeds will not continue to increase at the rate that they have been lately[6].

### **2.3 Our Beowulf Architecture**

The Beowulf that was constructed at the University of Northern Iowa is composed of thirteen Pentium-III 550 MHz computers. Each has 1 gigabyte of hard drive disk space allocated for the Linux OS and 2 gigabytes allocated for data. Each computer has 128 megabytes of main memory. The nodes were first connected with a 10 MB hub. Each node has a 100 MB Ethernet card which was connected to the hub. As is later shown by the benchmarks, the 10 MB hub proved insufficient. The hub was replaced with a 100 MB smart switch.

RedHat Linux version 6.0 was installed on each node. The network file system (NFS) and remote shell (rsh) were also installed on each node. NFS enable file sharing and rsh enables processes to be started remotely on a node from another node. These networking systems allow sufficient communication (at the operating system level) to support parallel execution of code on the cluster. Before the Beowulf could be used for numerical studies, the performance had to be tested to ensure a correct setup.

## 3 Benchmarking

### 3.1 Fluctuation Exchange Approximation Code

The Beowulf Cluster was benchmarked with Fluctuation Exchange Approximation (FEA) code that was written by Dr. John Deisz[3]. This FEA code makes use of the Hubbard model[1]. This model describes one-dimensional ( $L_1$ ), two-dimensional ( $L_1 \times L_2$ ), or three-dimensional structures ( $L_1 \times L_2 \times L_3$ ) that hold electrons in a finite number of atomic sites. These atomic sites are equidistant from one another. Each atomic site in the lattice can hold up to two electrons: one with spin-up and the other with spin-down.

The Hamiltonian matrix can be used to describe the Hubbard lattice. The Hamiltonian matrix contains a term for each atomic site in the lattice. This term describes the tendency of the electron(s) to stay at the current site. The value for a term in the matrix is defined as

$$H(i, j) = \begin{cases} \epsilon_d & : & i = j \\ -t & : & i \text{ is a neighbor of } j \\ 0 & : & \text{otherwise} \end{cases} .$$

$t$  is the hopping frequency, or the rate at which an electron tries to “hop” to a neighboring site.  $\epsilon_d$  is the default value for  $H(i, j)$  when  $i = j$ . Another parameter associated with this model is the electron attraction energy  $U$ . This parameter describes the difference in potential energy of two electrons at the same atomic site.

When only 16 atomic sites are modeled, the Hamiltonian matrix may be computationally diagonalized and all properties of the model can be obtained. However, a 16-site lattice is not large enough to accurately model a real solid.

The FEA code attempts to use the Hubbard model with a different scheme that approximates the Green's function  $G(\vec{R}, \tau)$ . The Green's function is a function of distance  $\vec{R}$  and time  $\tau$ .  $G(\vec{R}, \tau)$  basically describes the properties of an electron removed at a distance  $\vec{R}$  from and time  $\tau$  after where another electron has been added to the system. The FEA code calculates the Green's function numerically by using Dyson's Equation[7]

$$G(\vec{R}, \tau) = G_0(\vec{R}, \tau) + \int_0^\beta \int_0^\beta \int \int G_0(\vec{R} - \vec{R}', \tau - \tau') \times \\ \Sigma(\vec{R}' - \vec{R}'', \tau' - \tau'') G(\vec{R}'', \tau'') d\tau' d\tau'' dr' dr''$$

where  $\beta = 1/(\text{temperature})$ . This relationship can be used to determine  $G(\vec{R}, \tau)$  because  $G_0(\vec{R}, \tau)$  can be determined exactly and  $\Sigma$  be approximated. Note that  $G_0(\vec{R}, \tau)$  is  $G(\vec{R}, \tau)$  when the electrons do not interact in the lattice and  $\Sigma(\vec{R}, \tau)$  describes the effects of the interactions. Once  $G(\vec{R}, \tau)$  is found, its value can be used to make a better approximation for  $\Sigma$ . Then Dyson's Equation can be used again to find a better approximation for  $G(\vec{R}, \tau)$ . This is the main cycle of the FEA code.

Solving the above equation for  $G(\vec{R}, \tau)$  is not a trivial task. In order to make this model numerically easier to calculate, periodic boundary conditions are imposed. With periodic boundary conditions, a position  $\vec{R} = 6\hat{x}$  with lattice dimension size  $nx = 5\hat{x}$  would be equivalent to  $\vec{R} = 1\hat{x}$ . Even though this is unphysical, using these conditions do not significantly affect calculations. Note that as soon as these conditions are imposed for every dimension in the lattice,  $G(\vec{R}, \tau)$  becomes a periodic function and Dyson's Equation can be solved more easily by using Fourier transformations. After performing the transformations, Dyson's Equation becomes

$$G^{-1}(\vec{k}, \epsilon_n) = G_0^{-1}(\vec{k}, \epsilon_n) - \Sigma(\vec{k}, \epsilon_n)$$

After  $G(\vec{k}, \epsilon_n)$  is determined,  $\Sigma(\vec{R}, \tau)$  can be evaluated using  $G(\vec{R}, \tau)$ , and then in turn,  $G^{-1}(\vec{k}, \epsilon_n)$  can be re-evaluated using  $\Sigma(\vec{k}, \epsilon_n)$ . This self-consistency loop involves many Fourier transformations. These transformations are implemented using the Fast Fourier Transformation (FFT) algorithm[5] in the FEA code.

### 3.2 Parallelizing the FEA Code

Parallelizing a program is the process of enabling it to run on multiple computers. Determining how a program should run in parallel is a complex task that many third party compilers provide as a built-in function. In order to understand the process of developing a parallel program, the concepts of concurrency and parallelism must be given attention. Concurrency involves the parts of a program that can be computed independently, while parallelism refers to executing concurrent parts of a program in parallel. Deciding which concurrent parts of a program should be run in parallel is the task of the programmer.

Not all concurrent parts of a program should be run in parallel. For example, consider a program with a compute-bound loop that takes seconds to execute and requires one second for data to transfer during the loop. This program would not be a good candidate to parallelize. But consider a program with a compute-bound loop measured in minutes and only required a few seconds of data transfer. This program would be a good candidate to parallelize.

The rate of communication time versus processing time must be considered when determining whether or not concurrent parts should be run in parallel. Different programs have different communication versus processor time ratios, and running programs on different Be-

owulfs can alter the performance based on this ratio. For example, a compute-bound program would require a Beowulf with very fast CPUs and low speed network, while an input/output bound program would require a high-speed network and moderately fast CPUs.

In order for the Beowulf cluster to run a program in parallel, the program must be broken down in concurrent parts. This step is usually easier by leaving this to a parallelizing compiler. These compilers are usually for FORTRAN, but C compilers can also be obtained. FORTRAN is more commonly used because it is a language designed for scientific programs.

Concurrency in programs can be described in two primary ways: explicit parallelism methods and implicit parallelism methods. When converting a program using explicit parallel execution, the programmer must embed messages in the source code so that the Beowulf Cluster knows which parts of a program are concurrent and how to execute them in parallel. These embedded messages are usually used with the MPI (Message Passing Interface) standard. This library has functions included that enable easier parallel coding.

With implicit parallel methods, the compiler does all of the parallelization. High Performance FORTRAN (HPF) is the main programming language which supports this method with other languages under development. The user is required to provide some information about the concurrency of the program, but the compiler implements the parallelism itself. This method provides more portability than the explicit parallelism method[6]. The FEA code is implemented using HPF.

HPF parallelizes a program primarily by distributing the program's data across the nodes[4].

The FEA code maintains a multidimensional array that stores the current Green's function for each site in the lattice and for every time step. The array can be quite large. Most of the parallelization of this code is specified by defining this array to be distributed and notifying the compiler that operations done on this array are "safe" to be done independent of each other. For example, if an array contained 500 elements, then the array could be broken down into 5 blocks of 100 elements on each computer in a cluster with 5 nodes. When an operation is done on this array, each node performs that operation on the subset of the array which the node is storing. The FEA array is multidimensional, so this distribution is performed on the *time dimension only*.

### **3.3 Results of Parallelizing the FEA Code**

To test the performance of the 13-node Beowulf Cluster, the FEA code was run using arbitrary parameters for 5 iterations of its main loop. The FEA code was run on a varying number of processors in order to see how well the program would perform on different number of processors (in this case, nodes). The lattice dimension sizes were also varied. The result of the FEA benchmarks are shown in Figure 1.

The results for the 10 MB hub were disappointing. The cluster took up to five times longer to run code than a single processor computer. The sharp increase in program execution time from 1 to 2 processors shows that the code requires significant communication among the nodes. Viewing the processor usage of each node during execution of the code showed that each processor was not being used to its full capacity. Each processor was "starved" for information,

# Beowulf Cluster Performance

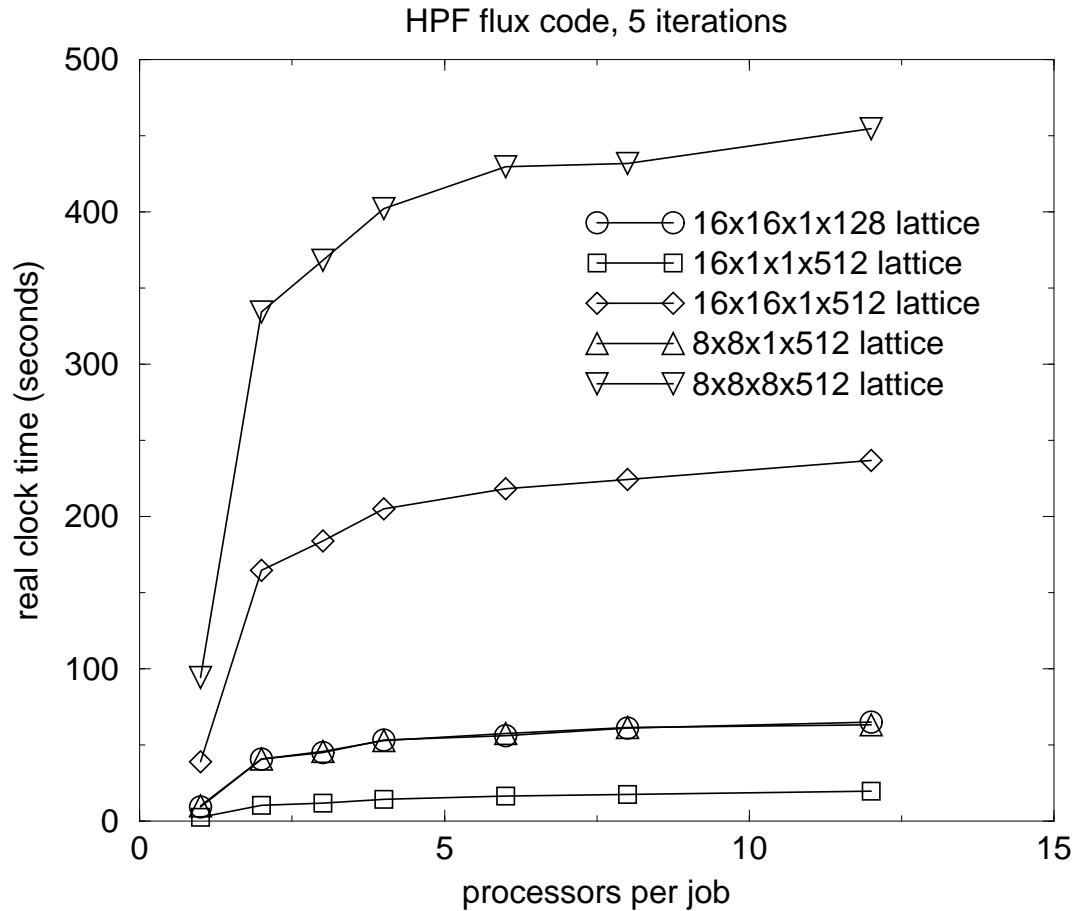


Figure 1: This graph shows the performance of the Beowulf cluster using the 10 MB hub. Network speed appears to be the limiting factor due to the great increase in process time observed from going from 1 processor to 2 processors. After 2-5 processors are reached, the process time slowly increases with number of processors used.

which was not located locally but on another node, required to make calculations. As shown in Figure 2, the Beowulf performance improved dramatically when the network was upgraded from a 10 MB hub to a 100 MB smart switch.

The results of these benchmarks conclude that the FEA code requires much communication, but what would be the reason for this? Fortunately, the HPF software that was used to compile the FEA code has a profiling mechanism where the execution time for each part of the

# Beowulf Cluster Performance

HPF flux code, 5 iterations, 100MB smart switch

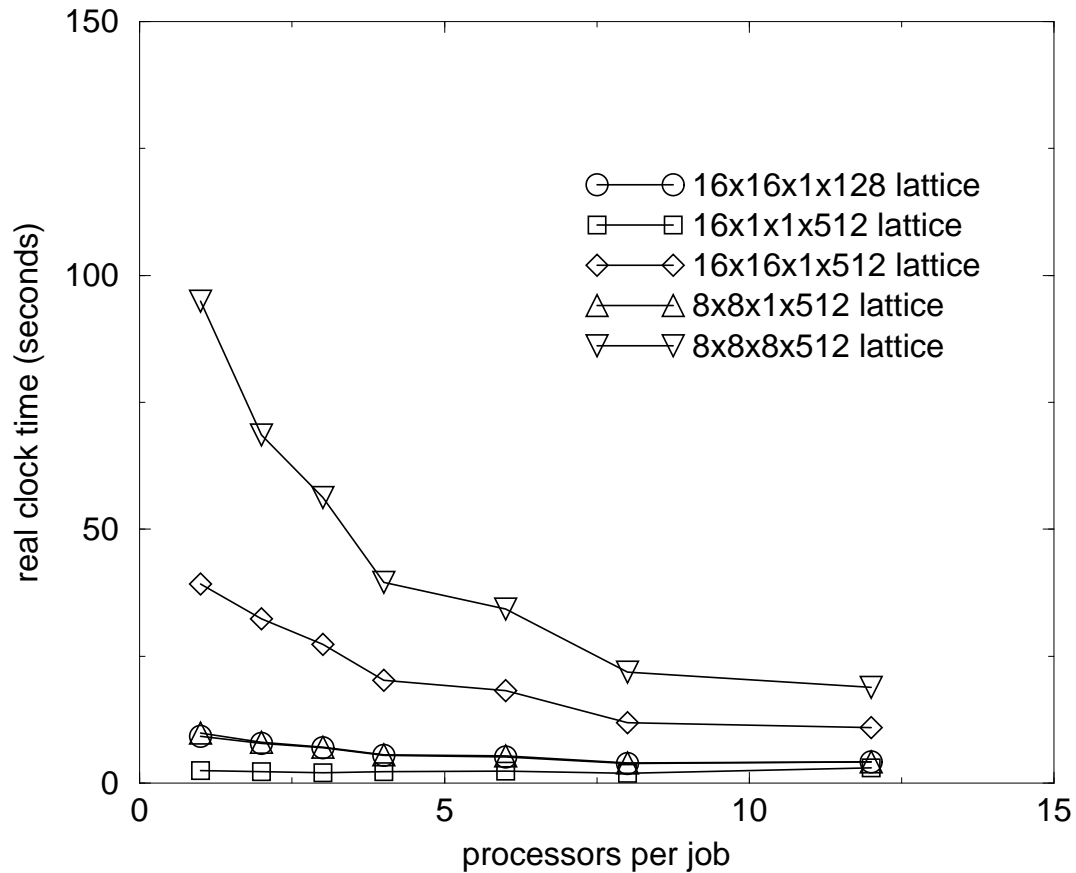


Figure 2: This graph shows the performance with the 100 MB smart switch. Considering the performance observed with the hub, the most obvious part of this graph is the large drop in process time from 1 to 2 processors. The code with the largest array  $8 \times 8 \times 8 \times 512$  appeared to have the most benefits from the cluster. Not much performance is gained on more than 8 processors.

code can be analyzed. Two profiling screen dumps are shown in Figure 3.

The `fft_4d` function is responsible for converting the Green's function and  $\Sigma$  to their different forms via the FFT. In order for the Green's function and  $\Sigma$  to be converted to functions of different variables, the FFT function must perform a separate summation of each dimension of the array storing  $G$ . Note that one dimension, the time dimension, is distributed across the

<b>Profile of FEA code on hub – 8x8x8x512 lattice, 4 processors</b>				
<i>File Name</i>	<i>Line</i>	<i>Function</i>	<i>Calls</i>	<i>Time</i>
4D_fft.hpf	1	fft_4d	48	372.907
sigma_pp.hpf	1	sigma_pp	6	52.735
sigma_ph.hpf	1	sigma_ph	6	44.672
main.hpf	1	hubbard	7	17.915
chi_pp_om.hpf	1	chi_pp_om	6	13.634
t_pp_r_tau.hpf	1	t_pp_r_tau	1	12.095
chi_ph.hpf	1	chi_ph_om	7	8.422260
g_tau.hpf	1	g_tau	6	8.023600
g0_komega.hpf	1	g0_komega	1	7.580720
g0_tau.hpf	1	g0_tau	1	5.112040
sigma_init.hpf	1	sigma_init	3	4.541970
phi_fea.hpf	1	phi_fea	2	3.475250
cor_energy.hpf	1	cor_energy	7	2.031020
bc_find.hpf	1	bc_find	6	1.743420
x_find.hpf	1	x_find	1	0.163754
var_init.hpf	1	var_init	1	0.032496
wv_find.hpf	1	wv_find	6	0.003014
poly_init.hpf	1	poly_init	1	7.81e-05
pghpf.prelink.f	0	pghpf\$static\$init	1	3.11e-05

<b>Profile of FEA code on switch – 8x8x8x512 lattice, 4 processors</b>				
<i>File Name</i>	<i>Line</i>	<i>Function</i>	<i>Calls</i>	<i>Time</i>
4D_fft.hpf	1	fft_4d	48	107.384
sigma_pp.hpf	1	sigma_pp	6	27.573
sigma_ph.hpf	1	sigma_ph	6	18.816
main.hpf	1	hubbard	1	18.051
chi_pp_om.hpf	1	chi_pp_om	7	17.934
t_pp_r_tau.hpf	1	t_pp_r_tau	6	13.639
chi_ph.hpf	1	chi_ph_om	7	8.423920
g_tau.hpf	1	g_tau	6	7.764820
g0_komega.hpf	1	g0_komega	3	4.517040
g0_tau.hpf	1	g0_tau	2	3.292500
sigma_init.hpf	1	sigma_init	1	3.046050
phi_fea.hpf	1	phi_fea	1	2.649080
cor_energy.hpf	1	cor_energy	1	1.706130
bc_find.hpf	1	bc_find	6	1.212070
x_find.hpf	1	x_find	7	0.835585
var_init.hpf	1	var_init	1	0.032174
wv_find.hpf	1	wv_find	6	0.003018
poly_init.hpf	1	poly_init	1	6.20e-05
pghpf.prelink.f	0	pghpf\$static\$init	1	3.19e-05

Figure 3: These two charts show the number of calls and processing time for each section, or function, of the FEA code for a sample run. The fft\_4d function dominated the total processing time for the code running on both the hub and the switch. The switch improved the performance of the fft\_4d function by the greatest amount.

machines. The presents a problem with the parallelized FEA code. Before and after the FFT on the time dimension, the whole array storing G must be redistributed across the nodes in order for each node perform the FFT on the time dimension (the array is simply redistributed across the space dimension for this interval). Since the FEA code is data distributed, each node would

want this data distributed at the same time. This causes an enormous network bottleneck which is shown by Figure 1 and the profiling results.

## 4 Specific Heats for 1-D Superconductors and Magnets

In some three-dimensional material, previous results have indicated that there is a discontinuity, or peak, in the temperature-dependent specific heat function. This peak distinguishes whether the material being modeled is magnetic/superconductive or not. Above this peak, the material is thought to generally lose its magnetic/superconductive properties. For two dimensions, a small hump is observed in the  $C_v(t)$  function.  $C_v(t)$  has not previously been studied in detail for one dimension using the Fluctuation Exchange Approximation. Using the FEA code,  $C_v(t)$  was found for the single dimension lattice.

Since the FEA code is able to output an energy value,  $E(n)$  can be differentiated with respect to temperature to obtain the specific heat function. The FEA code must, however, be run for each temperature being used to produce the function. To model magnetism, interaction potentials of +1 and +3 are used as parameters to the FEA code. Note that these values are dimensionless since the potentials are defined with the hopping frequency  $\frac{1}{t}$ . The  $C_v(t)$  functions for  $U = +1$  and  $U = +3$  are shown in Figure 4. The functions for lattice sizes of 64 and 256 are plotted in order to show that a 64-site lattice is significant enough to produce accurate results (for later calculations, 256 sites will be too large to compute).

The  $C_v(t)$  is also shown for superconductivity in Figure 4. Superconductivity is studied by just changing the potentials to -1 and -3, and by changing the electron density parameter to

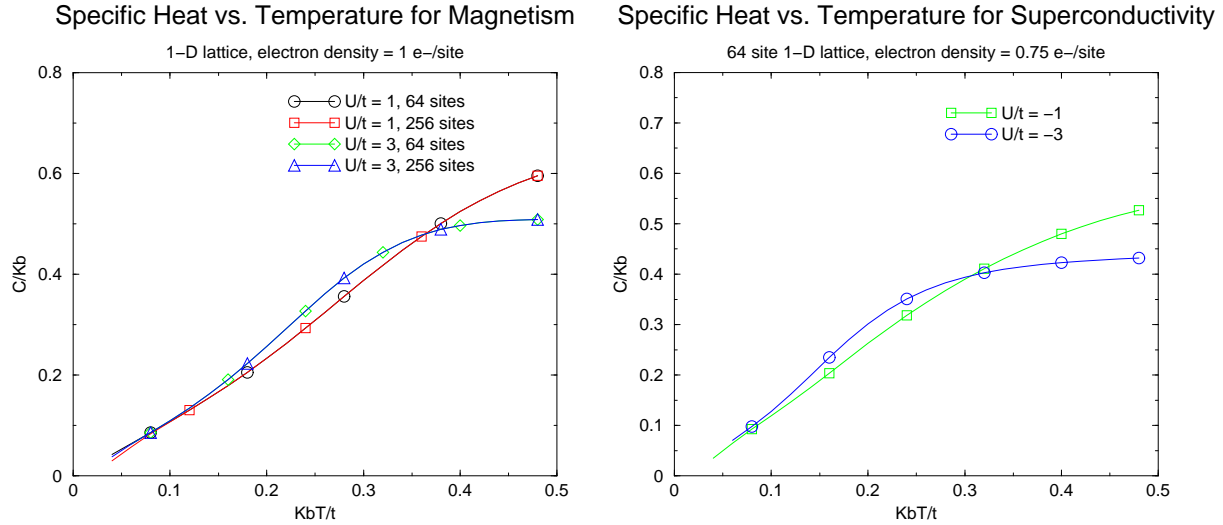


Figure 4: These two graphs show the temperature-dependent specific heat functions for superconductivity and for magnetism. The FEA code was run for temperatures of  $0.5 \frac{K_b T}{t}$  to  $0.02 \frac{K_b T}{t}$  values with a step size of  $0.02 \frac{K_b T}{t}$ . Functions for only 64-site lattices are shown for superconductivity since the magnetism calculations proved 64 sites were enough to avoid errors with the model. Breaks are shown in the  $C_v(t)$  functions for  $U = \pm 3$ .

0.75 e-/site instead of 1.0 e-/site, which was used for the magnetism calculations.

Instead of peaks, changes in slope are observed in the specific heat functions. These changes are only noticed for potentials of  $U = \pm 3$ . These results follow the pattern of diminishing structure in the specific heat for lattices of fewer dimensions.

## 5 Density of States Function for 1-D Superconductors and Magnets

Many experiments and calculations for three-dimensions show a gap in the density of states function when magnetism/superconductivity develops, similar to the discontinuity in the specific heat versus temperature function. Previous research has also established that pseudogaps appear in two-dimensions. The FEA code was used on the Beowulf Cluster to determine the

density of states for one-dimension.

The density of states for electrons experiencing interactions can be determined by using the analytic continuation method. The FEA considers electron interactions through the quantity  $\Sigma$ . The  $G(\epsilon_n)$  quantity is returned from the FEA code. To find the spectral density for a given frequency, the following equation can be used

$$\rho(\epsilon) = -\frac{1}{\pi} \text{Im} G(\epsilon + i\delta).$$

In order to find  $G(\epsilon + i\delta)$ , a fit must be made to the  $G(\epsilon)$  data. The Pudé Approximation was used for this fit. This approximation uses the first M points from the data to fit a curve.

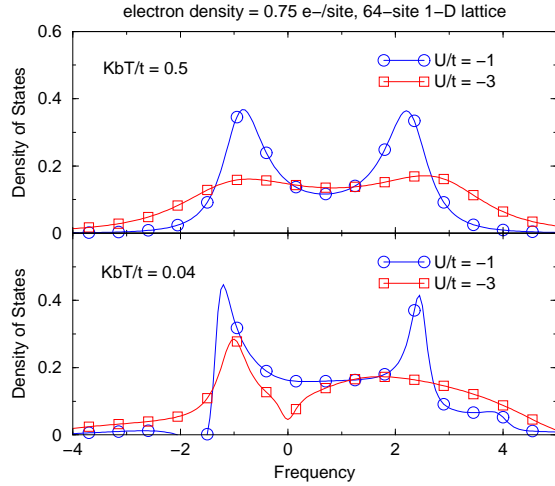
$$\tilde{G}_M(\epsilon) = \frac{(i\epsilon_n)^M + P_{M-1}(i\epsilon_n)^{M-1} + \dots + P_0}{(i\epsilon_n)^{M+1} + Q_M(i\epsilon_n)^M + \dots + Q_0}$$

Note that M must be smaller than the total number of points to fit. The code used for this fit is listed in Appendix A.

The FEA code was run for the case of magnetism (electron density = 1.0 e-/site and U/t = 1,3) and superconductivity (electron density = 0.75 e-/site and U/t = -1,-3). The FEA code was run for temperatures  $\frac{K_b T}{\tau} = 0.5, 0.48, 0.46, \dots, 0.02$ . The data was collected for 256 sites and 64 sites for the magnetism calculations to ensure that 64 sites were enough for an accurate model. The results are displayed in Figure 5.

Figure 6 shows the formation of the pseudogap at low temperatures for magnetism and superconductivity.

Density of States vs. Frequency for Superconductivity



Density of States vs. Frequency for Magnetism

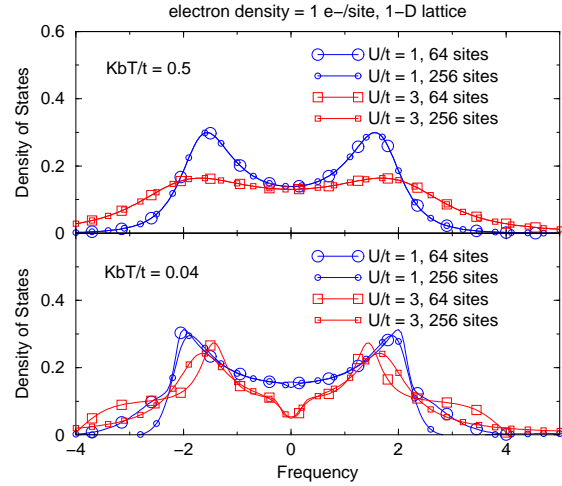
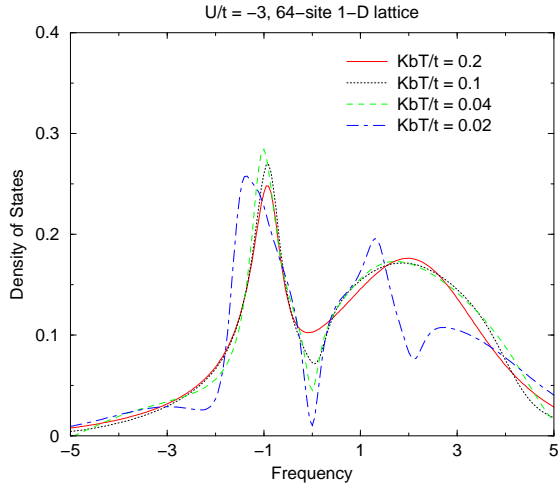


Figure 5: The density of states functions are shown for magnetism and superconductivity. A graph for high temperature  $\frac{K_bT}{\tau} = 0.5$  and low temperature  $\frac{K_bT}{\tau} = 0.02$  are shown for magnetism and superconductivity. Two different potentials  $\frac{U}{t} = \pm 1, \pm 3$  are graphed together on each plot. The magnetism density of states appear symmetric, while the superconductivity density of states appear shifted to the right with a higher peak in the negative frequency range. When potential approaches zero, the functions generally flatten. Notice that for both magnetism and superconductivity a pseudogap occurs at the lower temperature.

Formation of Pseudogap for Superconductivity



Formation of Pseudogap for Magnetism

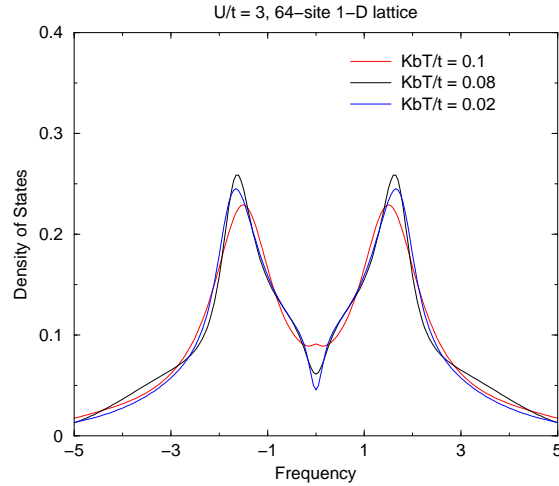


Figure 6: Graphed are the density of states functions at low temperatures for magnetism and superconductivity. The graphs show the formation of the pseudogap. The pseudogap for magnetism appears to form earlier than for superconductivity. The bizarre humps in one of the superconductivity graphs is only the result of a difficult fit.

## **6 Conclusion**

The Beowulf parallel computer cluster is a versatile, cost-effective solution for computationally intense projects. The COTS technology and desire for more flexibility has driven the development of the Beowulf. Starting from a need of CESDIS within NASA, the Beowulf project has quickly grown. The development of parallelization methods of programs and third party software support such as MPI have increased the performance necessary to put Beowulf systems in the supercomputer realm. The do-it-yourself attitude associated with the Beowulf systems increases the possibilities of the architecture and enables more institutions to build such a system inexpensively (compared with expensive supercomputers).

After upgrading the network, the Beowulf Cluster constructed at the University of Northern Iowa performed very well, despite the data distribution problems with the multidimensional FFT. The specific heat and density of states function were determined for nearly-magnetic and nearly-superconductive materials by using the Fluctuation Exchange Approximation code and the Hubbard model. The specific heat as a function a temperature had a break and a pseudogap was found in the density of states function for both magnetism and superconductivity.

## **7 Acknowledgements**

I would like to thank the University of Northern Iowa Physics department and Dr. John Deisz for providing me with the opportunity for this project. I felt that it was a very successfully project and learning experience, and hope that I am able to participate in further research.

## References

- [1] Ashcroft, Neil W., and N. David Mermin. *Solid State Physics*. Philadelphia: W.B. Saunders, 1976: 685.
- [2] *Beowulf Project at CESDIS*: n. pag. Online. Internet. 10 April 1999. Available <http://www.beowulf.org/>
- [3] Deisz, J.J., D.W. Hess, and J.W. Serene. "Incipient antiferromagnetism and low-energy excitations in the half-filled two-dimensional Hubbard model." *Physical Review Letters*. 76 (1996): 1312.
- [4] Koelbel, Charles H., et al. *The High Performance Handbook*. Cambridge: MIT, 1994: 26.
- [5] Press, William H., et al. *Numerical Recipes in Fortran 77*. Cambridge: Cambridge UP, 1992: 498.
- [6] Radajewski, Jacek, and Douglas Eadline. "Beowulf HOWTO." *Linux Documentation Project* 11 November 1998: n. pag. Online. Internet. 10 April 1999. Available <http://metalab.unc.edu/LDP/HOWTO/Beowulf-HOWTO.html>
- [7] Serene, J.W., and D.W. Hess. "Massively-Parallel Realizations of Self-Consistent Perturbation Theories." *Recent Progress in Many-Body Theories*. 3 (1992): 469-479.

## A Code for Pudé Approximation to $G(E_n)$

```
C      Robert Sheerer
C      July 29, 1999
C      pude.F

C      preprocessor constants
#ifdef M
# define M 30
#endif
#ifdef NMAX
# define NMAX 1024
#endif
#ifdef S
# define S 0.01
#endif

C      declarations
character*80 filename
integer n, l, input, row, col, info
real E(NMAX)
real spectral_density
complex q(2*(M+1)), p(2*(M+1)), G(NMAX), c_vector(2*(M+1),1)
complex c_vector_orig(2*(M+1),1), c_vector_computed
integer x_vector(2*(M+1))
complex matrix(2*(M+1),2*(M+1))
complex greens, greens_num, greens_den

C      initializations
call getarg(1,filename)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC MAIN CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

C      read Ei,G(Ei) data
input = 1
open (input, FILE=filename, FORM='FORMATTED')
do n=1,NMAX
  read (input,*) E(n), G(n)
end do
close (input)

C      create matrix
do row=1,2*(M+1)
  do col=1,M+1
    matrix(row,col) =
&      G(row)*(CMPLX(0.0,E(row)))**((M+1)-col)
    matrix(row,(M+1)+col) =
&      -(CMPLX(0.0,E(row)))**((M+1)-col)
  end do
end do

C      create c_vector
do l=1,2*(M+1)
  c_vector(l,1) = -(CMPLX(0.0,E(l)))**(M+1) * G(l)
  c_vector_orig(l,1) = -(CMPLX(0.0,E(l)))**(M+1) * G(l)
end do

C      solve linear equation: "matrix*x_vector = c_vector" for x_vector
call CGESV(2*(M+1),1,matrix,2*(M+1),x_vector,
&      c_vector,2*(M+1),info)
if( info .ne. 0 ) then
  print *, 'info = ', info
  print *, 'Unable to solve linear equation. Exiting.'
  stop
end if
```

```

do i=1,2*(M+1)
  c_vector_computed = CMPLX(0.0,0.0)
  do j=1,2*(M+1)
    c_vector_computed = c_vector_computed +
&      matrix(i,j) * c_vector(j,1)
  end do
end do

C  assign q's and p's
do l=1,M+1
  q(l) = c_vector(M+2-l,1)
  p(l) = c_vector(2*M+3-l,1)
end do

C  output E, spectral_density(E) data
do n=-800,800

  omega = float(n)/20.0

  greens_num = CMPLX(0.0,0.0)
  do l=0,M
    greens_num = greens_num + p(l+1)*(CMPLX(omega,S))**l
  end do
  greens_den = q(1) + (CMPLX(omega,S))**(M+1)
  do l=1,M
    greens_den = greens_den + q(l+1)*(CMPLX(omega,S))**l
  end do
  greens = greens_num / greens_den

  spectral_density = (-1.0/3.14159)*aimag(greens)
  print *, omega, "\t", spectral_density

end do

end

```