A PC Cluster with Application-Quality MPI

M. Gołębiewski, A. Basermann, M. Baum, R. Hempel, H. Ritzdorf, J. L. Träff

C & C Research Laboratories, NEC Europe Ltd.,
Rathausallee 10, D-53757 Sankt Augustin, Germany
golebiewski@ccrl-nece.technopark.gmd.de

Abstract. This paper presents an implementation of MPI on a cluster of Linux-based, dual-processor PCs interconnected by a Myricom high speed network. The implementation uses MPICH for the high level protocol and FM/HPVM for the basic communications layer. It allows multiple processes and multiple users on the same PC, and passes an extensive test suite. Execution times for several application codes, ranging from simple communication kernels to large Fortran codes, show good performance. The result is a high-performance MPI interface with multi-user service for this PC cluster.

Keywords: Workstation cluster; Myrinet; MPI; FM; Numerical applications.

1 Introduction

Continuous price reductions for commodity PC equipment and increasing processor speeds have made PC clusters an attractive low-price alternative to MPP systems for parallel applications. We therefore decided to set up such a system to provide our applications programmers with the required compute power for a large variety of projects, most of them in the area of numerical simulation. Our machine is based on the Local Area MultiProcessor (LAMP) developed at the NEC Research Institute (NECI) [6]. With its symmetric multiprocessor (SMP) architecture, the LAMP was well-suited for our intended experiments with clustered shared memory machines, and the Myrinet [15] provided sufficient communication speed.

The cluster installed at our laboratory consists of 16 dual-processor SMP nodes interconnected by a Myrinet network. Dual 8-Port Myrinet-SAN switches, each serving 4 PCs, provide a tetrahedral network topology. Each SMP node has two Pentium Pro 200MHz CPUs installed on a Tyan S1668 motherboard with Intel's 440FX chip-set and 512 KB of L2 cache per CPU. The PCs have 128 MB of EDO RAM each, so that the cluster has a total of 32 CPUs and 2 GB of RAM. The nodes are equipped with two network interface cards: an Ethernet card and a LANai board for accessing the Myrinet network. The LANai cards are basic 32 bit PCI models equipped with only 256 KB of SRAM. The cluster runs under Linux (Red Hat 5.1 distribution) with kernel version 2.0.34 (SMP). By choosing Unix, we have full support for services such as remote debugging and

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 613–623, 1999.
© Springer-Verlag Berlin Heidelberg 1999

administration, and multiple users on each node. The cluster is complemented by a Pentium II PC (300 MHz, 64 MB RAM) with SCSI disk drives, which is used as a file and compilation server.

Most of our application codes are written in Fortran (some in C) and parallelized using the Message Passing Interface (MPI) [20] which thus had to be installed as the application programming interface on the LAMP. Since similar PC clusters were already in use we attempted to use existing software. The search was guided by the following requirements: reliable, error-free data transmission, multi-user support, support for multiple processes per node, a complete and correct implementation of MPI, including the Fortran77 binding, and good communication performance at the MPI level. As can be seen our goal was to use the LAMP as a reliable compute server for real applications, and not as an experimental system.

In this paper we briefly discuss our survey of existing software (Section 2), followed by a description of our own MPI development (Section 3). Finally, in Sections 4 and 5 we present application results, ranging from kernel benchmarks to real-world codes, and compare the performance of the LAMP cluster with that of MPP systems.

2 Survey of Existing Software

Most PC clusters so far use Fast-Ethernet networks and high-overhead protocols for communication, such as TCP or UDP. This results in a large performance gap between the processing speed of a single PC and the communication between them. So, till now only very coarse-grained parallel applications could use such clusters efficiently. However, recent developments in high-speed networking hardware as, for instance, the Scalable Coherent Interface (SCI) [11] or Myrinet [15] extend the application domain towards problems with medium-, or even fine-grained, parallelism.

We tested all available communication interfaces for Myrinet under Linux:

1. Myricom - Myricom's native API. Only TCP/IP interface provided as basis for third-party MPI [15].
2. Basic Interface for Parallelism (BIP) from Ecole Normale Supérieure de Lyon, France [16].
3. Virtual Memory Mapped Communication (VMMC) developed at NECI. A new, much improved, version requires LANai boards with 1 MB [5].
4. BullDog Myrinet Control Program (BDM) from Mississippi State Univ. [12].
5. High Performance Virtual Machine (HPVM) developed at Univ. of Illinois in Urbana-Champaign [4].

None of them fulfilled all our requirements. Some were optimized for performance, but didn't conform to the MPI standard and didn't pass our test programs. Others did not allow multiple users or check for transmission errors. Table 1 is a summary of our survey, details can be found in [7]. To meet our requirements we concluded that we had to develop our own MPI implementation.

Communication interface	Myricom	BIP	VMMC	BDM	HPVM
max. number of nodes	16	8	16	16	> 16
multiple users/cluster	yes	no	no	no	yes
multiple processes/node	yes	no	no	no	yes
MPI (availability)	third-party	yes	no	yes	yes
MPI (standard compliance)	n/a	low	n/a	low	low
MPI (performance)	slow	fast	n/a	fast	fast

Table 1. Comparison of existing communication interfaces for Myrinet.

3 Implementation

We decided to base our new MPI library on the HPVM low-level interface FM 2.1 which met our requirements at that level. We could thus save the substantial effort needed to develop a new low-level library. For the high-level part, we used the MPICH software [9] and connected it with FM by our own device driver at the generic channel interface level. This device driver should have as little overhead as possible. We developed two versions:

1. The single-threaded (ST) device driver is optimized for minimum latency. It implements only the blocking channel interface primitives.
2. In the multi-threaded (MT) version both blocking and non-blocking primitives are implemented by having a communication thread for each application thread. This also allows to overlap computation with communication.

Both drivers use different protocols for three message length regimes, called *short*, *eager* and *rendezvous*, and switch between protocols at the same message lengths. Details of these three MPICH standard protocols are described in [10].

Communication in a program using the FM library may block when the receiver does not extract incoming messages from the network. Since in the ST version all FM calls are issued directly from the application processes, this may block the sender when the receiver is busy in a long computation. In the MT version the application thread can continue to execute while the communication thread is blocked. We thus expected this version to be more efficient in applications with poor load balancing, in which corresponding send and receive operations are not well synchronized. For details of the driver implementations, see [7].

4 First Test Results

In this section we present the results of experiments performed to assess the maturity of our MPI implementation.

4.1 Completeness and Correctness

We tested the MPI library, using both our ST and MT device drivers. Both versions passed the whole MPICH test suite. These programs not only check the correct semantics of the MPI functions, but also test the behavior of the implementation under intense network traffic. ST additionaly passed a much more demanding test suite from the GMD Communications Library (CLIC) for block-structured grids [17].

4.2 Performance Measured with Kernel Benchmarks

The basic performance indicators, latency and bandwidth, as measured with the ping-pong benchmark *mpptest* from the MPICH distribution, are shown in Fig. 1. We compare our implementations with the MPI interface of HPVM, because, despite of the many failures detected by our test suite, it was the existing software package that came closest to the user requirements.

The ST driver has very low latency, for zero byte messages exceeding that of the HPVM/MPI library by only about 2 μs. With more than 100 μs, the latency is much larger for the MT driver. The reasons for this high latency are the use of mutual exclusion and synchronization primitives from the POSIX threads library [3], as well as the necessary request queue manipulations.

Both our versions and HPVM/MPI exhibit similar bandwidth. The maximum bandwidth reached by our implementations is higher than that obtained with HPVM/MPI. ST performance decreases for very long messages using the rendezvous protocol. MT is slow for shorter messages, but in the rendezvous protocol is the fastest. The decrease in bandwidth seen for all three implementations for very long messages is caused by cache effects, and our implementations seem to suffer more from them than HPVM/MPI. This suggests that there is still room for optimization in our device drivers for the rendezvous protocol.

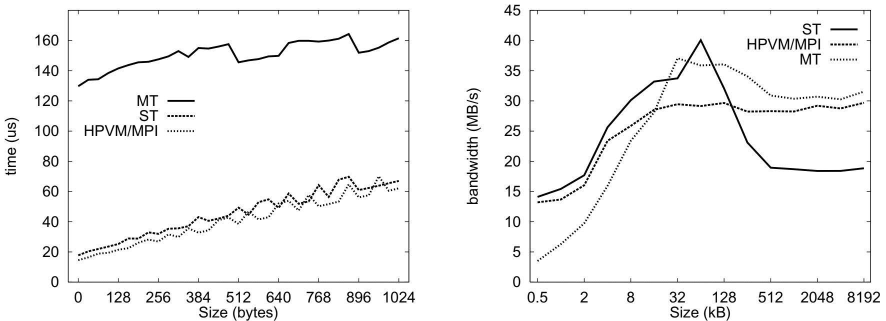

Fig. 1. Latency and bandwidth obtained with the ping-pong benchmark.

4.3 Effect of Load Imbalance on Performance

While basic test programs can be used to measure some basic performance characteristics of an MPI implementation, they cannot replace tests with full application codes. We have tested our implementations with the L_iSS package [18], a Fortran environment for solving partial differential equations, in our case the incompressible Navier-Stokes equations, on general two-dimensional block-structured domains. The CLIC library handles the inter-process communication for L_iSS. Together the packages contain some 85,000 lines of Fortran code.

By choosing the geometric domain appropriately, complicated communication patterns and various degrees of load imbalance can be generated. This latter feature was used to test our expectation that the MT device driver would improve the performance in the presence of load imbalance.

Table 2 gives parallel efficiencies and wall clock times for the ST and MT version of the MPI library for a problem with 66,240 grid points and good load-balance. Since HPVM/MPI does not have a usable Fortran binding, only our MPI libraries could be used for the L_iSS tests. For a comparison with an MPP computer, we included results achieved on a NEC Cenju-3 [14] for the same problem. In all test runs, 9 processes were used in a master-slave fashion. The tests were repeated with two slave processes per node to evaluate the performance loss caused by sharing the LANai boards, and the connection to a Myrinet switch, by two processes. Table 3 reports timings and efficiencies for a problem with 14,212 grid points and a high degree of load-imbalance. 30 processes were used for this set of tests, so either one or two application processes were assigned to each SMP node.

Our two implementations perform very well for the load-balanced test case, with ST always being the better choice. The efficiencies are well above 90%, except for MT with two processes per node. This is caused by the increase in CPU load by the additional threads. For ST we do not observe any significant performance drop by sharing the network interface between two processes. Apparently, there is not enough data traffic to saturate either the PCI bus or the network interface. The slower Cenju-3 performs worst in this test because its 75 MHz VR4400SC RISC CPUs are significantly slower than the 200 MHz PentiumPros of the PC cluster, and the interconnection network of the Cenju-3 is slower than the Myrinet. The unbalanced test case results in much lower performance than the well balanced problem. The parallel efficiencies for both ST and MT are only slightly above 22%. This shows that even for the situations for which the MT device driver was designed, it has no advantage over the ST version. We therefore selected the ST device driver for all further tests, and dropped further MT development.

5 Results for Complete Applications

This section summarizes some experiences with full application codes. The examples presented differ considerably in memory access patterns, communication

driver/ platform	procs per node	accumulated comm. time [s]	total wall clock time [s]	Efficiency [%]
ST	1	35.31	62.90	92.98
ST	2	34.43	62.97	96.16
MT	1	42.70	63.71	91.62
MT	2	145.65	77.04	76.37
Cenju-3	1	399.75	164.58	69.64

Table 2. Load-balanced problem with 66,240 grid points. Communication times are accumulated over all processes.

driver/ platform	accumulated comm. time [s]	total wall clock time [s]	Efficiency [%]
ST	919.45	40.89	22.45
MT	979.72	43.50	22.33
Cenju-3	1690.99	74.82	22.06

Table 3. Load-unbalanced problem, 14,212 grid points.

behavior, and computation to communication ratio, and illustrate well the performance of the LAMP for a wide range of potential applications and compared with other parallel computers. The SP-2 used for all three applications is configured with the High Performance Switch for MPI communication.

5.1 Large Sparse Eigenvalue Problems

The simulation of quantum chemistry and structural mechanics problems requires the solution of computation intensive, large sparse real symmetric or complex Hermitian eigenvalue problems. The JADA package [1] uses the JAcobi-DAvidson method [19] and iterative preconditioners for convergence acceleration to solve them.

JADA's parallelization strategy is matrix and vector partitioning with a data distribution and a communication scheme exploiting the sparsity of the matrix. Grouping of inner products and norm computations within the preconditioners and the basic Jacobi-Davidson iteration reduces the synchronization overhead. In a preprocessing phase, data distribution and communication scheme are automatically derived from the sparsity pattern of the matrix. This provides load-balance and makes it possible to overlap computations with data transfers. The JADA code is written in Fortran77 and C with MPI for message passing. It uses non-blocking communications in the multiplication of sparse matrices with dense vectors and MPI reduce operations for vector reductions.

Figure 2 illustrates the timing and speedup behavior of JADA in determining the four smallest eigenvalues and -vectors for an electron-phonon coupling simulation problem [22] on three different platforms: NEC Cenju-3, LAMP, and

IBM 9076 SP2. The matrix of order 98,800 with 966,254 nonzeros is real symmetric; the sparsity pattern is highly irregular. The result is a marked variation in message lengths and numbers of communication partners per processor. This relatively small problem was chosen because a sizable fraction of the execution time is spent on communication, and thus the effect of the MPI performance is more pronounced.

In Figure 2 (left) JADA has the best performance on the SP2 and the worst on the Cenju-3, with the LAMP results somewhere in between. For 1 processor, the SP2 is 1.6 times faster than the LAMP, which in turn is 1.9 times faster than the Cenju-3. The scaling behavior, as shown in Figure 2 (right), is best for the Cenju-3, which has the best communication to computation ratio. The reason for the marked efficiency loss between 16 and 32 processors for the LAMP is the use of both processors of the SMP nodes. Currently, internode communication is not handled optimally. Due to the good scaling, the LAMP times approach those of the SP2 for up to 16 processors, whereas on 32 processors the execution time on the LAMP is only slightly slower than on the Cenju-3.

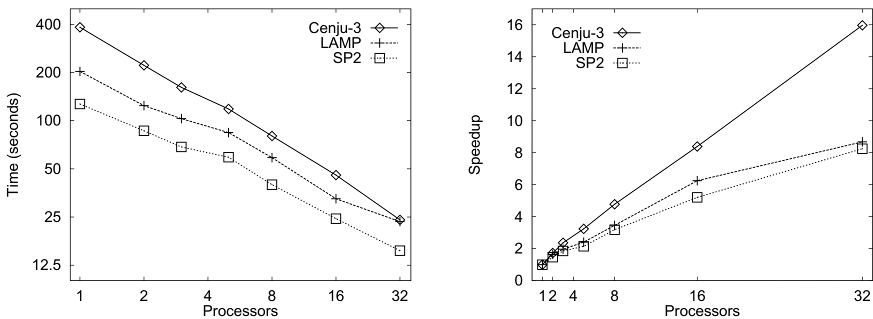

Fig. 2. Wall clock time (note the log-log scale) and parallel speedup for JADA.

5.2 Numerical Simulations of Complex Flows

Cellular automata methods represent an alternative approach to finite volume or finite element techniques for the numerical prediction of fluid flows. The basic idea of these methods is the numerical simulation of simplified molecular dynamics derived from the microscopic description of the fluid, instead of solving macroscopic governing equations. This is done by evaluating a time and space discrete Boltzmann equation, the so called *lattice Boltzmann equation* [2] describing the dynamics of the particle density function by two basic mechanisms, (1) *particle propagation* and (2) *particle collision*. The collision step of this procedure enables the specification of boundary conditions at fluid/solid interfaces in a way that overcomes the usual difficulties of classical CFD-methods in generating suitable grids. A simple bounce-back procedure fulfills the velocity boundary conditions for this type of interface and makes it possible to deal with

arbitrarily complex geometries of these interfaces, while ensuring unconditional stability for the overall procedure. This capability to overcome the complex and time-consuming grid-specification procedure of classical approaches make the *Lattice Boltzmann Automata* (LBA) techniques especially appealing for many industrial applications. LBA techniques have been implemented in the Fortran program BEST [2] with MPI for data communication during parallel execution of the code. While particle collision is performed locally on each lattice, particle propagation implies transfer of data between neighboring points. The combination of an explicit time marching method and the restriction to next-neighbor dependencies simplifies the development of an efficient parallel code. However, the very small number of integer and floating point operations per lattice (about 150 flops per lattice per iteration) often leads to situations where the execution time is dominated by communication. The code is parallelized using the domain partitioning technique, with neighboring partitions being stored with an overlap of one lattice layer. The values in the overlap areas are updated once per iteration with all data for each destination process combined into a single message.

Since BEST is communication rather than computation bound for mid-size problems, it is an excellent benchmark for the communication network of parallel platforms under production conditions. Figure 3 shows the performance of BEST for a problem with 32 lattices in each space direction on the LAMP, an IBM SP2, and a NEC Cenju-3. Additionally for the LAMP, results are given for the largest problem that fits into a single node memory, with $256 \times 32 \times 32$ lattices.

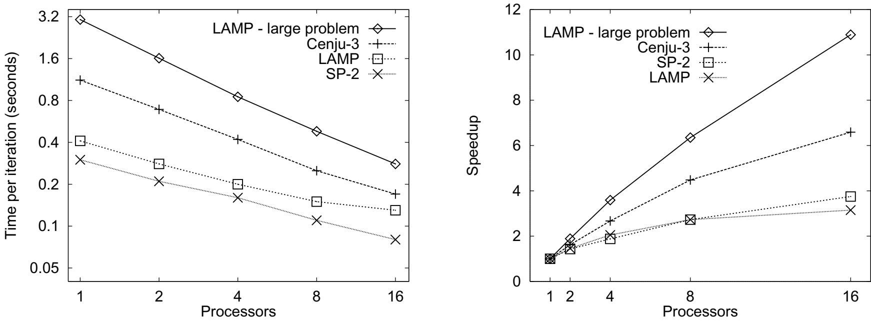

Fig. 3. BEST performance.

5.3 Distributed List Ranking

The last example is a kernel problem in parallel graph algorithms. Given a singly linked list, the *list ranking problem* consists in computing for each list element the number of elements that follow it by a traversal of the links until the last element is reached. It is assumed that the list elements are stored in an array but not necessarily in the order in which they are encountered during traversal of the list. The problem is important in a parallel setting because it makes it possible to perform reduction operations on lists [13].

The sequential problem is trivial, and a careful implementation requires only two scans of the list. The small amount of computation makes it difficult to achieve any speed-up in a parallel program, and thus provides us with another hard test case for evaluation of the LAMP system. We consider the two algorithms discussed in [21]. The list is assumed to be evenly distributed among the processors in a random fashion, with N being the total number of list elements, p the number of processors, and n the number of elements per processor. The first algorithm is the standard pointer jumping algorithm [13]. Depending on the implementation of the all-to-all communication operation, the execution time per processor can be decomposed into $3(p-1)\lceil \log N \rceil$ start-ups, $4n\lceil \log N \rceil$ words communicated, and $O(n \log N)$ local computations (with a very small constant factor). The other, fold-unfold, algorithm works without all-to-all communication. Here the execution time can be decomposed into $2(p-1)$ start-ups per processor, $4n \log p$ words communicated, and $O(n \log p)$ local computations (again with a very small constant factor).

In Figure 4 we give results obtained on an IBM SP2 and the LAMP for lists of length $N = pn$ with n fixed to 1 000 000 elements. The total execution time is compared with the sequential time for the corresponding problem size, optimistically estimated as p times the time to rank the sublists residing at one processor. The two systems show comparable performance. The fold-unfold algorithm performs significantly better than the pointer jumping algorithm, but in terms of achieved speed-up the results are unsatisfactory.

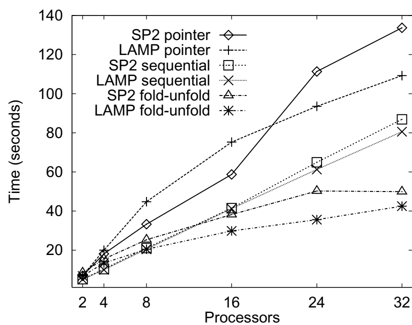

Fig. 4. List ranking: lists with fixed local length $n = 1\,000\,000$.

6 Outlook

The work presented in the paper is an ongoing project. We recently completed a multi-device version of the MPI library using shared memory communications between processes running on the same node. Another important enhancement for the nearest future is to implement an ADI2 device that will replace the current driver for the generic channel device. We also plan to migrate to the new MPICH 1.1.1 version, which integrates the MPI-2 [8] parallel I/O functions. The

main objective of the project, however, has been reached already: providing our researches with an efficient and reliable platform for development of complex MPI applications on our PC cluster in a multi-user-environment.

References

[1] A. Basermann, Parallel Preconditioned Solvers for Large Sparse Hermitian Eigenvalue Problems, *Proceedings of the Third International Meeting on Vector and Parallel Processing (VECPAR'98)*, (1998) 31–44.

[2] J. Bernsdorf, F. Durst and M. Schäfer, *Comparison of Cellular Automata and Finite Volume Techniques*, International Journal for Numerical Methods in Fluids **29**, (1999) 251–264.

[3] D.R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley (1997).

[4] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini and J. Prusakova, High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance, *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (PP97)* (1997).

[5] C. Dubnicki, A. Bilas, K. Li and J. Philbin, *Design and Implementation of Virtual Memory-Mapped Communication on Myrinet*, NECI Technical Report (1996).

[6] J. Edler, A. Gottlieb and J. Philbin, The NECI LAMP: What, Why, and How, *Heterogeneous Computing and Multi-Disciplinary Applications, Proceedings of the 8th NEC Research Symposium.*

[7] M. Gołębiewski, M. Baum and R. Hempel, High Performance Implementation of MPI for Myrinet, *ACPC'99*, LNCS 1557, (1999) 510–521.

[8] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI –The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[9] W. Gropp and E. Lusk, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Parallel Computing, **22(6)**, (1996) 789–828.

[10] W. Gropp and E. Lusk, *MPICH Working Note: The implementation of the second generation MPICH ADI*, Argonne National Laboratory internal report.

[11] D. Gustavson, The Scalable Coherent Interface and related standards projects, *IEEE Micro*, **12(1)** (1992) 10–22.

[12] G. Henley, N. Doss, T. McMahon and A. Skjellum, *BDM: A Multiprotocol Myrinet Control Program and Host Application Programmer Interface*, Technical Report (Mississippi State University, 1997).

[13] Joseph JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley (1992).

[14] N. Koike, NEC Cenju-3: A Microprocessor-Based Parallel Computer, *Proceedings of the 8th IPPS* (1994) 396–403.

[15] Myricom, *Myrinet Documentation*, `http://www.myri.com/scs/documentation`

[16] L. Prylli and B. Tourancheau, BIP: new protocol designed for high performance networking on Myrinet, *IPPS/SPDP'98 Workshops*, Lecture Notes in Computer Science **1388** (1998), 472–485.

[17] H. Ritzdorf and R. Hempel, *CLIC - The Communications Library for Industrial Codes*, `http://www.gmd.de/SCAI/num/clic/clic.html`

[18] H. Ritzdorf, A. Schüller, B. Steckel and K. Stüben, L_iSS - An environment for the parallel multigrid solution of partial differential equations on general 2D domains, *Parallel Computing* **20** (1994) 1559–1570.

[19] G. L. G. Sleijpen and H. A. van der Vorst, A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems, *SIAM J. Matrix Anal. Appl.* **17** (1996) 401–425.
[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI –The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
[21] J. L. Träff, Portable randomized list ranking on multiprocessors using MPI, *5th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science **1497** (1998) 395–402.
[22] G. Wellein, H. Röder and H. Fehske, Polarons and Bipolarons in Strongly Interacting Electron-Phonon Systems, *Phys. Rev.* **B 53** (1996) 9666–9675.