Parallelization of Level Set Methods for Solving Solidification Problems *

Kai Wang[†] Anthony Chang[‡] Laxmikant V. Kale [§] and Jonathan A. Dantzig

Parallel Programming Lab
Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana, IL, 61801, USA

December 5, 2005

Abstract

Processor virtualization is a kind of parallelization technique which may be used to enhance the performance of parallel applications from the cache performance, overlapping of communication and computation. In this study, we use the processor virtualization technique to parallelize the level set method for solving solidification problems. Numerical results on a distributed memory machine are reported to show the performance of the resulting level set solver, and demonstrate the advantages of using processor virtualization.

Key words: Processor virtualization, Level set methods, MPI, AMPI, Solidification.

1 Introduction

Further understanding of the solidification process has become increasingly important in the development of advanced materials. Various solidification parameters

*This work was supported in part by the National Science Foundation (ITR 0205611, DMR 0121695)

†E-mail: wangkai@cs.uiuc.edu. URL: http://charm.cs.uiuc.edu/∼wangkai.
‡E-mail: anthony@solace.me.uiuc.edu.
§E-mail: kale@cs.uiuc.edu. URL: http://charm.cs.uiuc.edu
¶E-mail: dantzig@uiuc.edu.

can be controlled to produce the desired material microstructure and properties. Industrial casting processes are generally run under conditions that lead to a dendritic interface, a complex pattern that has been studied theoretically by numerous researchers. [1–6] Simulation of dendritic growth is a challenging problem that requires resolution of several different length scales, and the evolution of a complex interface. There are two classes of methods for problems of this type: fixed grid methods, in which the interface is recovered from the solution of a continuous field on a fixed mesh, and front-tracking methods, which somehow ensure that the interface coincides with the grid. In this work, we use a front tracking approach, called the level set method, described in detail below. [4, 7, 8]

Computations of dendrite growth require the simulation of the thermal and/or solutal diffusion field surrounding the dendrite tip, typically of order 1×10^{-4} m, while at the same time resolving dendritic pattern features that may be of the order of 1×10^{-7} m or smaller. The computational domain must be large enough to accommodate the large length scales, while the grid spacing must be sufficiently small to resolve the smaller length scales. Simulation of the solidification process usually requires several thousand time steps to obtain steady state. Hence, an enormous amount of computer memory and computing time are necessary to obtain an accurate result. Massively parallel high-performance computers can be used to alleviate some of the problems encountered when simulating solidification problems. The use of parallel computing allows the necessary calculations to occur on several processors to reduce computational times on problems that would otherwise be impractical.

Parallelization has been used to improve the performance of the level set method for a number of applications [9,10]. Most of the parallelization schemes use the Message Passing Interface (MPI) as the programming model. The MPI standard [11] is currently the most popular programming model for the development of parallel applications. In MPI, the programmer normally divides the computation into exactly P processes that run on P processors. For complex dynamic applications, significant effort is required to divide the computation into processes with good load balance characteristics and communication performance.

The processor virtualization concept has been proposed as a way to remedy these difficulties [12–16]. In this approach, the programmer decomposes the computation according to the nature of the problem rather than the number of physical processors available, dividing the problem into a large number of objects, which are called virtual processors, and the runtime system is responsible for mapping these virtual processors to different physical processors. This empowers the runtime system to do resource management, including automatic load balancing and communication optimization, by migrating the virtual processors across physical processors. It simplifies the programmer's task by substantially removing the constraint of physical processors from the algorithm design process [16].

The processor virtualization technique has been successfully employed and evaluated in many dynamic applications that are notoriously difficult to paral-

lelize [17–19]. In this study, we investigate the effect of processor virtualization on the parallel level set method, applied to simulate dendritic growth. In Section 2, we introduce some basic knowledge of the level set and solidification problem. In Section 3, we explain the idea of processor virtualization and its implementation in the level set tracking process. The numerical results are given in Section 5 to show the benefits of the processor virtualization brought to SAI computation.

2 Level set methods for solving the solidification problem

We use the level set method to simulate directional solidification of binary alloys in two dimensions. We assume that the chemical diffusivity in the liquid D_ℓ is much smaller than the thermal diffusivity, and this allows us to apply the "frozen temperature approximation," wherein the thermal gradient is fixed in a frame that is translating at a constant velocity, V. The diffusivity in the solid phase, typically much less than D_ℓ is taken to be zero. Thus, we need to solve only for the concentration in the liquid phase C_ℓ, using the diffusion equation:

$$\partial_t C_\ell = D_\ell \nabla^2 C_\ell \tag{1}$$

At the liquid-solid interface, we must satisfy two conditions: conservation of solute, and thermodynamic equilibrium, which relates the interface temperature and composition for the solid and liquid phases.

$$\begin{aligned} -D_\ell \partial_n C_\ell &= V_n \left(C_\ell - C_s\right) && (2)\\ C_\ell^i &= \frac{1}{m_\ell}\left(T^i - T_m + \Gamma\kappa\right) && (3)\\ C_s^i &= \frac{1}{m_s}\left(T^i - T_m + \Gamma\kappa\right) && (4) \end{aligned}$$

where V_n is the normal velocity of the interface, C_ℓ^i and C_s^i are the concentration in the liquid and solid, respectively, T_m is the melting temperature of the pure material, T^i is the interface temperature, m_ℓ and m_s are the slopes of the liquidus and solidus lines on the equilibrium phase diagram, respectively, Γ is the interfacial energy and κ is the local interface curvature.

In the level set method, a function ϕ is introduced, representing a signed distance from the interface. The interface itself is defined as the level set $\phi = 0$. The concentration field and interface position are computed using a two step time-marching scheme. Assume that all variables are known at a given time. In the first step, the interface is fixed, and thermodynamic equilibrium, Eqn. (3), is applied as a boundary condition to solve Eqn. (1) for the concentration field. Next, V_n is computed from the concentration field using the solute balance, Eqn. (2). Once V_n

is known, the interface is advanced by solving a pure advection equation for the distance function, ϕ:

$$\partial_t \phi + v_x \partial_x \phi + v_y \partial_y \phi = 0 \tag{5}$$

where v_x and v_y are the components of V_n in the x and y direction respectively.

To avoid spurious oscillations, a fifth-order WENO (weighted essentially non-oscillatory) method is used to discretize space. A third-order Runge-Kutta method is used to discretize time. After advection, ϕ is reconstituted as a distance function by solving [7]

$$\partial_t \phi + S(\phi) [|\nabla \phi| - 1] = 0 \tag{6}$$

until steady state is reached. $S(\phi)$, is a smearing function defined as:

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + (\Delta x)^2}} \tag{7}$$

where Δx is the grid spacing. To solve Eqn. (6), we rewrite it in the form:

$$\partial_\tau \phi + \left(\frac{S(\phi)\phi_x}{\sqrt{\phi_x^2 + \phi_y^2}} \right) \phi_x + \left(\frac{S(\phi)\phi_y}{\sqrt{\phi_x^2 + \phi_y^2}} \right) \phi_y = S(\phi) \tag{8}$$

where τ is a fictitious time measurement. A modified Godunov's method (see Chen, et al. [7]) coupled with the WENO scheme is used to solve this equation.

We use the localized form of the level set method, in which ϕ is calculated only within a narrow region surrounding the interface. This significantly decreases the computation time. Once the distance function is reinitialized, the time step is complete. The curvature of the interface at the new time step can be calculated from the distance function. The concentration field for the new interface position can be determined separately in the solid and the liquid once the interface concentration is determined. This sequence continues until the specified end time is reached.

3 Processor virtualization

The goal of processor virtualization is to find an effective division of labor between the programmer and runtime system. Specifically, the programmer is best at finding and expressing the natural parallelism of the application, but the runtime system can efficiently carry out resource management and many performance optimizations [15, 16]. In the processor virtualization model, the programmer divides the computation into many virtual processors, and the runtime system assigns them to available physical processors. The management and inspection of the virtual processors are also controlled by the runtime system instead of the programmer.

Probably the most obvious advantage of processor virtualization is that the runtime system can do automatic dynamic load balancing, by moving the virtual

processors between physical processors. Suppose each physical processor houses many virtual processors. In the simplest setting, the runtime system can monitor the loads on all physical processors and their neighbors. When a physical processor goes idle, the run time system could request additional virtual processors from neighboring physical processors with high load, so that the loads are balanced. A powerful runtime system can perform this task without user supervision.

Processor virtualization has been applied in many different areas. Many of these applications benefit from the automatic load balancing mechanism [17–19]. When using the level set methods for solving the solidification problem, the load can be roughly balanced by partitioning the initial grid data evenly to different processors using domain decomposition. In this study, we do not focus on the automatic load balancing advantage of virtualization, but instead on its abilities to improve cache performance and optimize communication.

Better cache performance The parallelization scheme for a level set method on virtual processors is the same as for physical processors—the grid data are distributed to the virtual processors by the domain decomposition. When the number of virtual processors is larger than the number of physical processors, each virtual processor handles a smaller set of data than each physical processor. A virtual processor may thus have better memory locality during both communication and computation. This blocking effect is the same strategy employed by many sequential cache optimization techniques.

Adaptive overlap of computation and communication The communication in the level set method for solidification involves only ghost layer data exchange. For example, for the decomposition of a 2D domain, each processor needs to communicate with four neighbors. Therefore, the time spent on an iteration for one processor does not only depend not only on itself, but also on its slowest neighbor.

Typical parallel programming models such as MPI support only one process per physical processor. Therefore, if this single process is blocked on a receive, the whole physical processor blocks and becomes idle. This communication idle time can be traced to two distinct causes. First, a processor B may have to wait for processor A to complete its work, for example because A is a slower processor or more heavily loaded (load imbalance). Second, even after A sends its data, processor B still must wait for the data to arrive across the network (message delay). This is illustrated in Fig. 1, where processor B has finished its first phase computation, but it cannot go to its second phase computation without the message from processor A. Processor B remains idle until the message from A arrives. Both load imbalance and message delay prevent us from taking full advantage of machine's power.

Allowing each physical processor to contain many virtual processors can decrease the amount of time wasted. When one virtual processor is blocked, the runtime system can keep the CPU working by picking up another virtual processor to

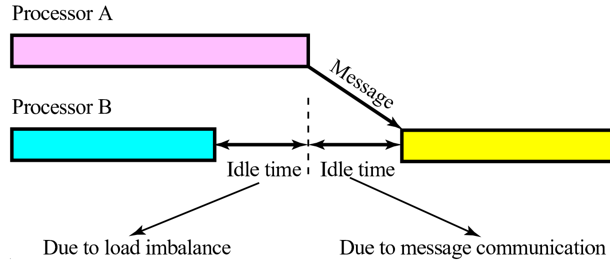

Figure 1: Processor idle time caused by load imbalance and message delay.

take the control of the CPU. This behavior is illustrated in Fig. 2, where the physical processors A and B now contain two virtual processors $A1$, $A2$, $B1$, and $B2$. When the first phase computation of $B1$ and $B2$ is finished, the message from $A1$ arrives, hence $B1$ can start its next phase computation immediately. Compared with Fig. 1, the idle time in Fig. 2 is reduced because the computation and communication in Fig. 2 are overlapped.

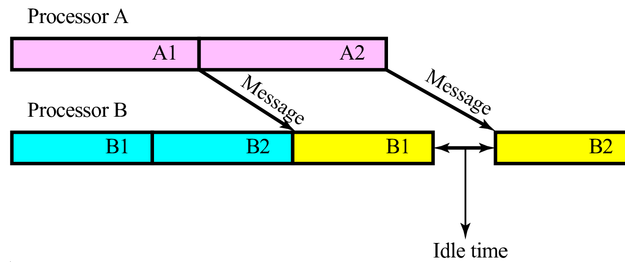

Figure 2: Processor idle time reduction via processor virtualization.

4 Implementation issues

Currently there are at least two parallel programming systems supporting the processor virtualization technique. Charm++ is one of the earliest. It is C++ based and uses parallel objects called Chares to express each virtual processor. Chares communicate via asynchronous invocation of each other's special "remote" methods. For some types of applications, Charm++ has better performance and modularity properties than MPI [14–16]. However the asynchronous message-passing mechanism is unfamiliar to many programmers, in Science and Engineering or those using languages other than C++.

Adaptive MPI (AMPI) was developed to provide processor virtualization within the popular MPI programming interface. [20] AMPI is built on Charm++, but provides the familiar programming model of MPI. Details of the Charm++ and AMPI programming systems can be obtained online [21], or publications [12–14,20].

In this study, we first use MPI to parallelize the level set method for solving

the solidification problem on a two dimensional domain. The initial grid data is distributed evenly to a two dimensional processor array. Parallelization of the WENO scheme is straightforward, as it needs only to communicate among neighboring processors. The concentration field is solved in parallel using a red-black Gauss-Seidel iteration scheme. Virtualization is achieved by replacing its MPI commands with corresponding AMPI commands and compiling it with the AMPI complier.

5 Experimental results

In this section, we show the parallel performance of the level set method based on the processor virtualization technique introduced in the previous section. We performed numerous experiments using different grid sizes, degrees of virtualization and numbers of processors. The results we report here are selected to illustrate the individual effects of cache performance improvement and scalability. All tests were run on the Tungsten Xeon machine at NCSA using up to 32 nodes. Each node has 3GB memory and dual Intel Xeon 3.06 processors, each having a 512 KB L2 cache and 1MB L3 cache.

5.1 Virtualization overhead

We first compare the performance of the resulting program using different number of virtual processors on only one physical processor. The purpose of the test is to study the overhead of virtualization. It is important to point out that using a single virtual processor on one physical processor is equivalent to doing a serial computation.

Fig. 3 shows the result of simulating a solidification process on a grid size of 100×100. The computational times reported here are recorded after 100 seconds of simulated solidification time. We see that the CPU time increases when more virtual processors are used, as more communication and computation overhead is incurred. The total CPU time increases by almost 50% when the degree of virtualization is equal to 8. It should be noted, however, that the grid size for this test is relatively small, which tends to emphasize the relative cost of virtualization.

5.2 Cache performance

In this section, we demonstrate the effect of virtualization on cache performance. This time we used a larger grid size (150×150) for a simulation of 100 seconds solidification process. The results are reported in Fig. 4, where we see that the CPU time decreased drastically when assigning 2 virtual processors to each physical processor. This can be explained as improved cache performance. The data on the 150×150 grid may be difficult to be fit in the cache. However, when using two virtual processors in the computation, each virtual processor handles less data,

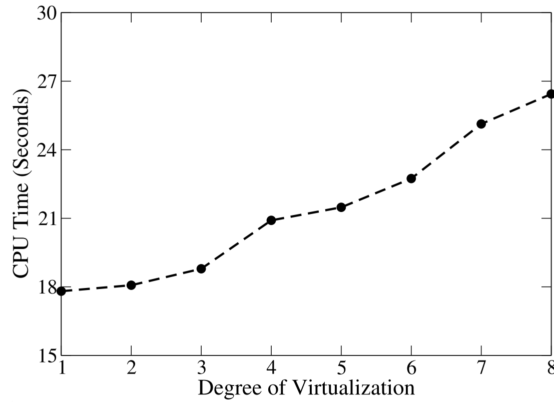

Figure 3: Virtualization overhead on one physical processor case. Grid size = 100×100. 1 physical processor.

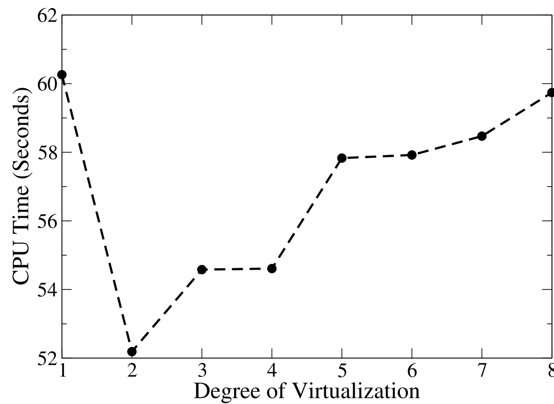

Figure 4: Cache performance on one physical processor case. Number of Unknowns = 150×150. 1 physical processor.

hence the cache performance can be improved.

We also observe a CPU time increase when using more virtual processors in the computation because of the increased virtualization overhead. These losses offset any further cache performance improvement. Note that when these same tests were run using a smaller 100×100 grid, the CPU time increased for all degrees of virtualization, indicating that the computations for the smaller grid fit entirely within the cache. We use this fact in the next section to isolate the effect of adaptive overlap on program performance.

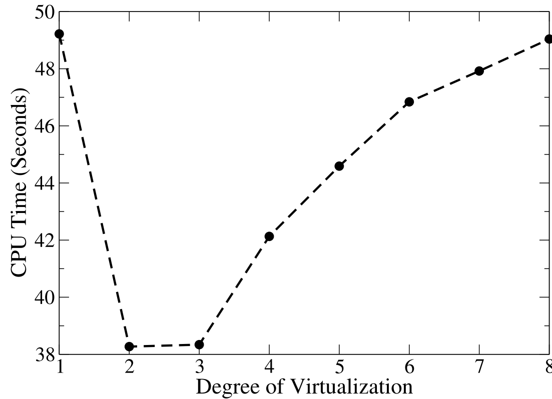

Figure 5: Performance of adaptive overlapping. Number of Unknowns = 500×500. 32 physical processors.

5.3 Adaptive overlap

Adaptive overlap of communication and computation occurs when one virtual processor blocks for a receive, and the runtime system switches to another virtual processor. Theoretically, adaptive overlap can save CPU time, since better overall processor utilization is to be expected. To isolate this effect, we must first eliminate the influence of the cache effect. Based on the results of the experiments in the previous subsection, we ensure that no processor is assigned more than a 100×100 section of the grid.

Fig. 5 reports the results of simulations using 32 physical processors and a total grid size of 500×500. Each physical processor has no more than 100×100 grid data. Using 2 virtual processors per physical processor decreases the cpu time by 11 seconds, a 20% reduction compared to the non-virtualizatized case. Since improved cache performance has been eliminated as a possible explanation for the improved performance, the speedup here can be only be due to the adaptive overlap of communication and computation.

5.4 Interleaved performance

With a larger problem size, we show the performance of the program with both improved cache performance and adaptive overlapping of communication and computation. The result in Fig. 6 is from a test on a grid size of 1000×1000 on 32 processors. The simulation time is still 100 seconds.

The data in Fig. 6 illustrate that when the degree of virtualization is 3, the CPU time is around 150 seconds, which corresponds to a 25% improvement compared with the non-virtualized case. The total CPU time that can be saved is more than 55 seconds. The speedups here are both from the improved cache performance and

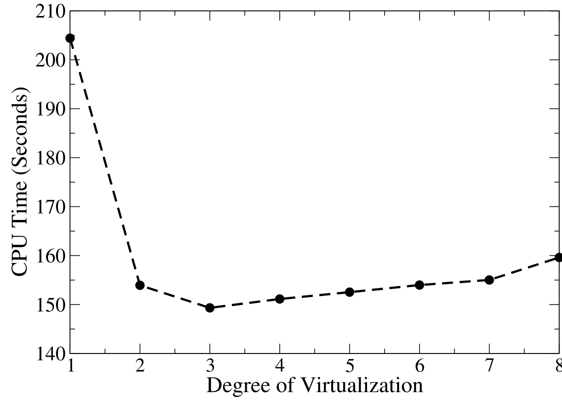

Figure 6: Interleaved performance. Number of Unknowns = 1000×1000. 32 physical processors.

Table 1: Raw data of Fig. 7. size=1000 × 1000.

# Processors	degree=1	degree=2	degree=3	degree=4
4	9713.10	8769.38	7402.09	7398.27
8	5618.45	4167.17	3610.77	3677.76
16	2093.14	1945.00	1818.83	1613.69
32	1052.84	948.87	907.63	991.85
64	789.36	651.97	585.34	622.55

adaptive overlapping of communication and computation.

5.5 Scalability test

In this section, we do the test using different numbers of physical processors. The problem size we choose here is 1000 × 1000. The result is shown in Fig. 7, and for convenience of discussion, the raw data are also listed in Table 1. The picture on the left of Fig. 7 shows the relationship between the degree of virtualization and the speed up when using different numbers of physical processors. The best speedup (1.56) is found when using 8 physical processors. We can also see that the best performance is obtained using a degree of virtualization equal to 3. The picture on the right of Fig. 7 is actually a scalability test. It is not surprising to see that there are not to much difference when using different degrees of virtualization. The whole program scales well, even the best cases is when the degree of virtualization equals to 2.

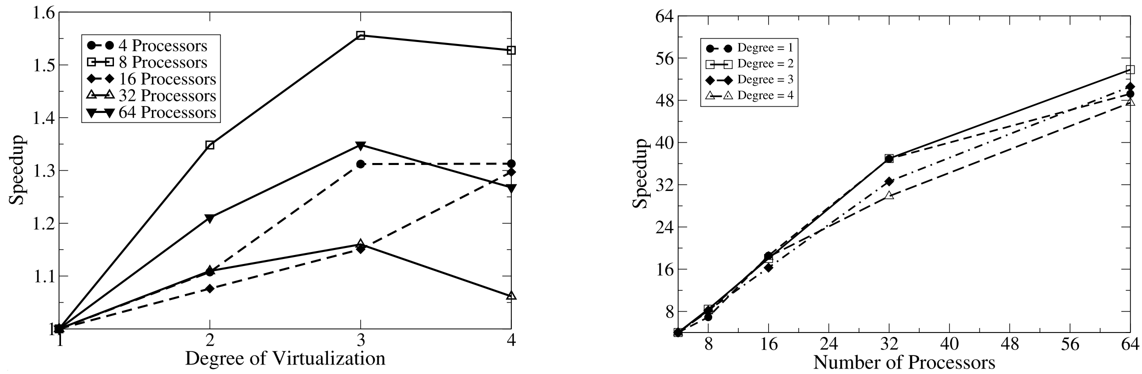

Figure 7: Speedup test. Left: Relationship between the degree of virtualizations and speedup. Right: Relationship between the number of processors and the speedup. Grid size = 1000×1000. 500 seconds simulation.

6 Conclusion

In this paper, we talk about using the processor virtualization method to parallelize the level set method for solving solidification problems. We give the performance data which compare the different degrees of virtualization. We can see that a good degree of virtualization can improve the performance of the resulting program from the cache performance and adaptive overlapping of communication and computation. However, it also brings virtualization overhead which may finally make the program run slower. In this implementation, we observed that assigning 2 or 3 virtual processors to each physical processor is usually a tradeoff between the virtualization overhead and virtualization benefits.

Automatic load balancing is another advantage of using processor virtualization. The solidification process is a moving boundary problem. Most computation happens on the boundary, which may move from one processor to another processor in some cases. Therefore load imbalance may occur. We will investigate the load balancing problem in our ongoing implementation of the level set method with adaptive gridding.

References

[1] B. Grossmann, K. Elder, M. Grant and M. Kosterlitz. Phys. Rev. Lett. **71**(20), 3323 (1993).

[2] B. Echebarria, R. Folch, A. Karma and M. Plapp. Phys. Rev. E **70**, 061604 (2004).

[3] N. Provatas, Q. Wang, M. Haataja and M. Grant. Phys. Rev. Lett. **91**(15), 155502 (2003).

[4] Y.-T. Kim, N. Goldenfeld and J. A. Dantzig. Physical Review E **62**(2), 2471 (2000).

[5] H. Udaykumar, R. Mittal and W. Shyy. J. of Comp. Physics **153**, 535 (1999).

[6] J.-H. Jeong, N. Goldenfeld and J. A. Dantzig. Physical Review E **64**, 041602 (2001).

[7] S. Chen, B. Merrimann, S. Osher and P. Smereka. J. of Comp. Physics **135**(1), 8 (1997).

[8] F. Gibou, R. Fedkiw, R. Caflisch and S. Osher. J. of Sci. Comp. **19**(1–3), 183 (2003).

[9] X. Li. Phys. Fluids A **5**(8), 1904 (1993).

[10] M. Sussman. Comp. and Struct. **83**, 435 (2005).

[11] MPI website: http://www-unix.mcs.anl.gov/mpi/.

[12] L. Kale, B. Ramkumar, A. B. Sinha and A. Gursoy. IEEE Transactions on Parallel and Distributed Systems (1994).

[13] L. Kale, B. Ramkumar, A. B. Sinha and A. Gursoy. IEEE Transactions on Parallel and Distributed Systems (1994).

[14] L. Kale and S. Krishnan. In *Parallel Programming Using C++*, edited by G. V. Wilson and P. Lu, 175–213 (MIT Press, 1996).

[15] LACSI. *The Virtualization Model of Parallel Programming: Runtime Optimizations and the State of Art*, Albuquerque (October 2004).

[16] HPCA 10. *Performance and Productivity in Parallel Programming via Processor Virtualization*, Madrid, Spain (February 2004).

[17] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan and K. Schulten. J. of Comp. Physics **151**, 283 (1999).

[18] M. Nelson, W. Humphrey, F. Gursoy, A. Dalke, L. Kale, R. Skeel and K. Schulten. Inter. J. Super. App. and High Perform. Computing **10**(4) (1996).

[19] IPDPS. *BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines*, Santa Fe, New Mexico (April 2004).

[20] LCPC 03. *Adaptive MPI*, College Station, Texas (October 2003).

[21] Homepage of Parallel Programming Lab at Department of Computer Science, University of Illinois at Urbana-Champaign: http://charm.cs.uiuc.edu.