Scaling Molecular Dynamics to 3000 Processors with Projections: A Performance Analysis Case Study

Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, Chee Wai Lee
{kale,skumar2,gzheng,cheelee}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract. Some of the most challenging applications to parallelize scalably are the ones that present a relatively small amount of computation per iteration. Multiple interacting performance challenges must be identified and solved to attain high parallel efficiency in such cases. We present a case study involving NAMD, a parallel molecular dynamics application, and efforts to scale it to run on 3000 processors with Tera-FLOPS level performance. NAMD is implemented in Charm++, and the performance analysis was carried out using "projections", the performance visualization/analysis tool associated with Charm++. We will showcase a series of optimizations facilitated by projections. The resultant performance of NAMD led to a Gordon Bell award at SC2002.

1 Introduction

How does one understand and optimize performance of complex applications running on very large parallel machines? The complexity of application means that multiple competing and interacting factors are responsible for its performance behavior. These factors often mask each other. Further, different factors may dominate as the number of processors used changes. Therefore, performance problems that manifest on the largest machine configuration cannot be debugged on smaller configurations — one must make most of the few runs one can get on the large configuration.

The approach we have pursued for this problem is based on application-level visual and analytical performance feedback. This is facilitated further by use of Charm++ [KK96] and more recently, Adaptive MPI [BKdSH01] which enables MPI programs to use the features of Charm++. The language runtime (Section 2), is capable of automatically recording substantial performance information at a level that makes sense to the application programmer.

We will illustrate this approach via a performance optimization case study. The application we have chosen is NAMD, a production-quality parallel molecular dynamics program developed using Charm++, and in routine use by biophysicists. The benchmark we describe runs each timestep in 25 seconds on a single processor of the PSC Lemieux machine, which consists of 3,000 processors! Given the amount of coordination and communication that must be done in each time-step, one can see the performance challenges involved in completing each timestep in 12 msecs, a result we have achieved.

We will first introduce Charm++ and Projections, the performance analysis tool associated with Charm++. We will then describe the parallel structure of NAMD. Utility of projections, and the performance optimization methodology it engenders is described via a series of examples. Each example shows how a problem is identified (and optimizations suggested) using projections, and the effect of resultant optimizations.

2 Charm++ and Projections

Virtualization is the driving force behind Charm++. The basic idea in virtualization is to let the programmer divide the program into a large number of parts independent of the number of processors. The parts may be objects, for example. The programmer does not think of processors explicitly (nor refer to processors in their programs). Instead they program only in terms of the interaction between these virtual entities. Under the hood, the Run Time System (RTS) is aware of processors and maps these virtual processors (VPs) to real processors, and remap them whenever necessary. Charm++ supports asynchronous method invocations, reductions and broadcasts which work correctly and efficiently in the presence of dynamic migrations of objects among processors. It uses the remapping capability to do automatic load balancing.

2.1 Performance Analysis Tools and Techniques in Projections

Projections is a performance analysis subsystem associated with the Charm++. It consists of an automatic instrumentation and tracing system, and an interactive graphic analysis system.

Efficient Automatic Tracing Since Charm is a message driven system, tracing can be fully automated. Specifically, the RTS knows when it is about to schedule the execution of a particular method of a particular object (in response to a message being picked up from the scheduler's queue).

Users can start and stop instrumentation during specific phases of the program. Different tracing modules can register themselves to the RTS via callbacks. The overhead on the application when NOT tracing is that of an if statement per event (which is typically associated with a message: so not much overhead in comparison). Even this overhead can be eliminated in production runs by linking the program with an optimized version of the RTS.

Unlike MPI, in Charm++ we can retrieve the idle time from the RTS. In MPI when one is waiting at a barrier or a recv, all the time spent in that gets called as communication overhead. However this often includes idle time, because another processor hasn't arrived at the barrier (or hasn't sent the message). Charm RTS can cleanly separate communication overhead from such idle time. This prevents users from erroneous conclusions that the performance is low due to "the slow barrier operations", whereas it may be due to load imbalances.

Two important tracing modules are *log* and *summary*. In the log mode each event is recorded in full detail (including timestamp) in an internal buffer.

The *summary* module reduces the size of output files and memory overhead. It produces (in the default mode) a few lines of output data per processor. For each entry-method it records standard profiling information such as total (max and average) time spent in it and the number of calls to it. It uses an adaptive strategy to limit the size of time-dependent data. This data is recorded in bins corresponding to intervals of size 1ms by default. If the number of intervals exceeds a threshold then the bin-size is increased to shrink the data into fewer bins. This way the size of recorded data is kept bounded.

Analysis and Visualization The visualization system supports multiple views: A *graph* view shows processor utilization as a function of time (using pre-set or user-specified bins of intervals) for a specified set of processors. One can also turn on a display of specific message types, to see when each phase of the computation started, and how dispersed over time its execution was. One of the simplest uses of this view is to identify the interval of times for a more detailed study.

The *profile* view shows a stacked column bar for each selected processor, for a selected time interval. The time spent by each processor in various activities is shown within each bar. This view clearly separates idle time and communication overhead. This is one of the most useful "summarizing" views in projections. One can identify overloaded processors, unexpected time-spent in specific methods, or high communication overhead from this.

The animation view shows time-varying behavior across processors, which can be arranged in multiple topologies (2D grid is most popular). Although this view is initially interesting, and can provide a "Gestalt" impression for the performance behavior, we find it to be not as useful as other static views, which one can stare at for insights..

The above views can be supported with both summary or log data. The timeline view, which is supported with the log data only, is essentially similar to other timeline tools such as Upshot, and Paragraph [HE91]. It is a highly sophisticated view which presents additional detailed information about events by simple mouse clicks.

3 Overview of NAMD

NAMD is a molecular dynamics program designed for high performance simulation of large biomolecular systems [PZKK02]. Each simulated timestep involves computing forces on each atom, and "integrating" them to update their positions. The forces are due to bonds, and electrostatic forces between atoms within a cut-off radius.

NAMD 2 is parallelized using Charm++ via a novel combination of force and spatial decomposition to generate enough parallelism for parallel machines with a large number of processors. Atoms are partitioned into cubes whose dimensions are slightly larger than the cutoff radius. For each pair of neighboring cubes, we assign a non-bonded force computation object, which can be independently mapped to any processor. The number of such objects is therefore 14 times (26/2 + 1 self-interaction) the number of cubes.

The cubes described above are represented in NAMD 2 by objects called *home patches*. Each home patch is responsible for distributing coordinate data, retrieving forces, and integrating the equations of motion for all of the atoms in the cube of space owned by the patch. The forces used by the patches are computed by a variety of *compute objects*. There are several varieties of compute objects, responsible for computing the different types of forces (bond, electrostatic, constraint, etc.). On a given processor, there may be multiple "compute objects" that all need the coordinates from the same home patch. To eliminate duplication of communication, a "proxy" of the home patch is created on every processor where its coordinates are needed. The parallel structure of NAMD is shown in Fig. 1.

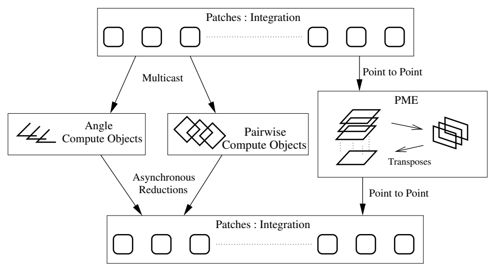


Fig. 1. Parallel structure of NAMD

NAMD employs Charm++'s measurement-based load balancing. When a simulation begins, patches are distributed according to a recursive coordinate bisection scheme, so that each processor receives a number of neighboring patches. All compute objects are then distributed to a processor owning at least one home patch. The framework measures the execution time of each compute object (the object loads), and records other (non-migratable) patch work as "background load." After the simulation runs for several time-steps (typically several seconds to several minutes), the program suspends the simulation to trigger the initial load balancing. The strategy retrieves the object times and background load from the framework, computes an improved load distribution, and redistributes the migratable compute objects.

The initial load balancer is aggressive, starting from the set of required proxies and assigning compute objects in order from larger to smaller, avoiding the need to create new proxies unless necessary. Once a good balance is achieved, atom migration changes very slowly. Another load balance is only needed after several thousand steps.

4 Performance Analysis and Optimizations

We will present the performance optimizations we carried out with the help of Projections in a series of examples. The first two examples involve runs on the ASCI Red machine, while the rest are on PSC Lemieux.

Grainsize Analysis: The benchmark application we used on ASCI Red machine was a 92,000 atom simulation, which took 57 seconds on one processor. Although it scaled reasonable well for few hundred processors, initial performance improvements stalled beyond 1,000 processors. One of the analysis using projections logs we performed identified a cause. Most of the computation time was spent in force-computation objects. However, as shown in Figure 2, the execution time of computational objects was not uniform: it ranged from 1 to 41 msecs. The variation itself is not a problem (after all, Charm++'s load balancers are expected to handle that). However, having single objects with execution time of 40+ msecs, in a computation that should ideally run in 28 msecs on 2000 processors was clearly infeasible! This observation, and especially the bimodal distribution of execution times, led us to examine the set of computational objects. We found the culprits to be those objects that correspond to electrostatic force computations between cubes that have a common face. If cubes touch only at corners, only a small fraction of atom-pairs will be within the cut-off distance and need to be evaluated. In contrast, those touching at faces have most within-cutoff pairs. Splitting these objects into multiple pieces led to a much improved grainsize distribution as shown in Fig. 2b.

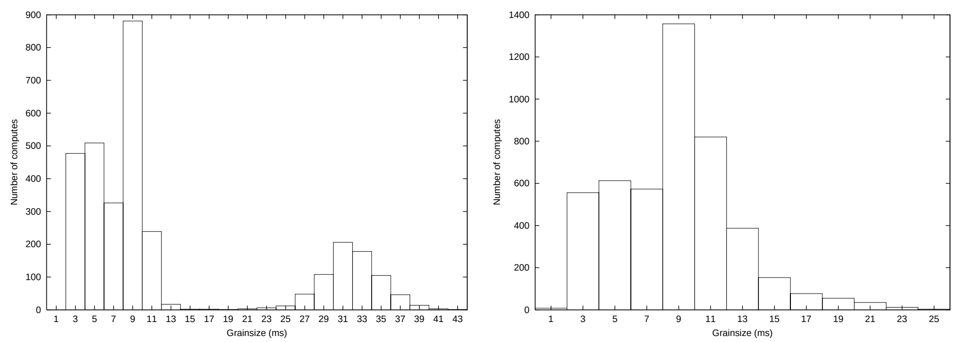


Fig. 2. Grainsize Distribution on ASCI Red

Message Packing Overhead and Multicast: Projections timeline is a very useful tool - it can easily tell what a parallel program is doing by showing the recorded events along the time axis. From a timeline view, one can easily identify what is bad or unexpected and try to optimize it. One of the analysis using projections logs in timeline tool we did for NAMD exemplified this.

Fig. 3 shows a timeline view of one 1024 processors run. The circled event shows the integration phase in NAMD. During the integration phase, each patch combines all incoming forces and calculates the new atom positions, etc. At end of integration, the patch sends the updated atom information to all the computes whose computation depends on it via a multicast. In timeline, the white little ticks at the bottom of the integration event show the message sending events. From Fig. 3, we observed the unusual long time spent in the multicast. The implementation of multicast in Charm++ was not efficient, it treated multicast as individual sends and each send paid the overhead of message copying and allocation. After reimplementing the multicast in Charm++, to avoid this overhead the integration time is greatly shortened as shown in the Fig. 3.

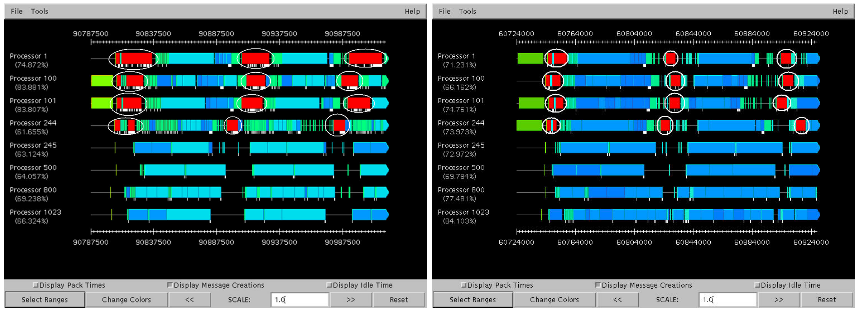


Fig. 3. Effect of Multicast Optimization on Integration Overhead

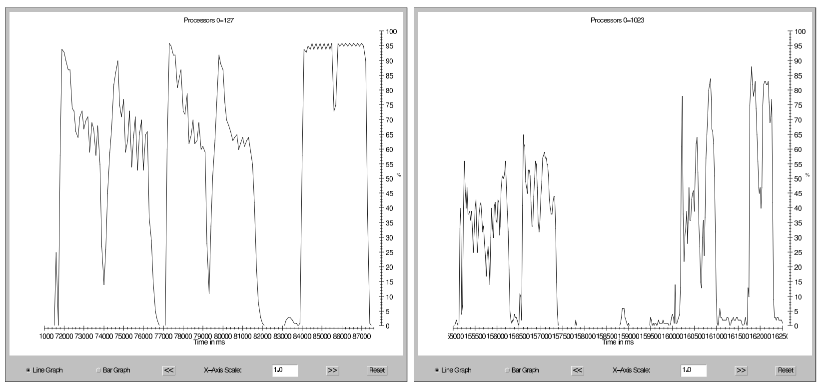


Fig. 4. Processor Utilization against Time on (a) 128 (b) 1024 processors

Load Balancing: Dynamic load balancing was an important performance challenge for this application. The distribution of atoms over space is relatively non-uniform, and (as seen in the grainsize analysis above) the computational work is distributed quite non-uniformly among the objects. We used a measurement-based load balancing framework, which supports runtime load and communication tracing. The RTS admits different strategies (even during a single run) as plug-ins, which use the trace data. We used a specific greedy strategy[KSB+99]. For a 128-processor run, Projections visualization of the utilization graph (Fig. 4(a)) confirmed that the load balancer worked very well: Prior to load balancing (at 82 seconds) relatively bad load imbalance led to utilization averaging to 65-70% in each cycle. However after load balancing, the next 16 steps ran at over 95% utilization.

However, when the same strategy was used on 1024 processors, the results were not as satisfying (Fig. 4 (a)). In particular, (via a profile view not shown here) it became clear that the load on many processors was substantially different than what the load balancer had predicted. Since the greedy strategy used ignored existing placements of objects entirely (in order to create an unconstrained close-to-optimal mapping), it was surmised that the assumptions about background load (due to communication, for ex-

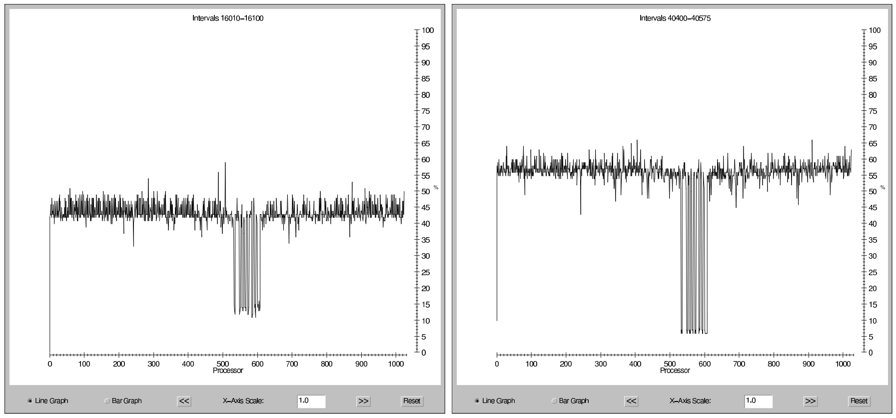
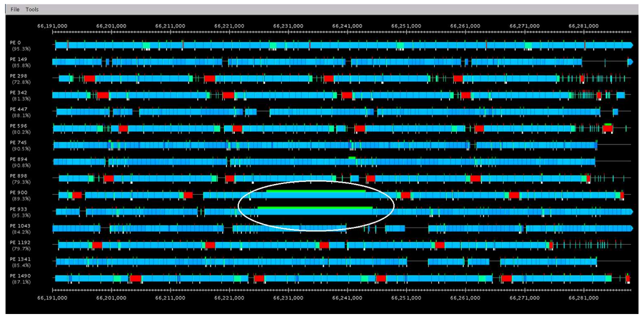


Fig. 5. Processor Utilization for each processor after (a) greedy load balancing and (b) refining

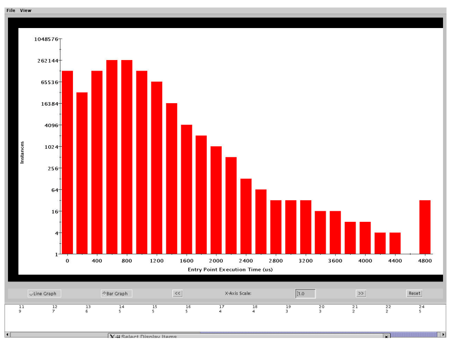
ample) as well as cache performance were substantially different in the new context after the massive object migration induced by load balancer. Since the new mapping was expected to be close to optimal, we didn't want to discard it. Instead, we added another load balancing phase immediately after the greedy reallocation, which used a simpler "refinement" strategy: objects were moved only from the processors that were well above (say 5%) the average load. This ensured that the overall performance context (and communication behavior) was not perturbed significantly after refinement, and so the load-balancer predictions were in line with what happened. In Fig. 4(b), initial greedy balancer works from 157 through 160 seconds, leading to some increase in average utilization. Further, after the refinement strategy finished (within about .7 seconds) at around 161.6 seconds, we can see that utilization is significantly improved. Another view in Projections (Fig. 5), showing utilization as a function of processors for the time intervals before and after refinement, shows this effect clearly.

Note that due to some quirks in the background load, several processors in the range between 500 and 600 were left underloaded by the greedy algorithm. The refinement algorithm did not change the load on those, since it focuses (correctly) only on overloaded processors: having a few underloaded processors doesn't impact the performance much, but having even one overloaded processor slows the overall execution time. Here, we see that 4 overloaded processors (e.g, processor 508) were significantly overloaded before the refinement step, whereas the load is much much closer to the average after refinement. As a result, overall utilization across all processor rises from 45 to 60%.

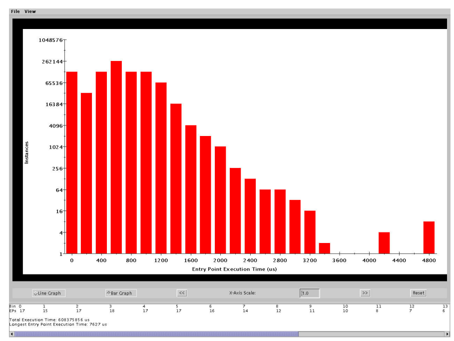
Using 4 processors on each node and "Stretches": The Lemieux machine has 3,000 processors. However, when we tested NAMD on more than 2,000 processors, the program performed relatively poorly. Several separate runs, observed with the timeline view, showed that this was due to several object methods (seemingly) randomly "stretching" to take a much longer time than they took during other time-steps. (For example, see Fig. 6(a)). The two computations highlighted by a green line take over 10 msecs, in comparison with a couple of msecs taken by the same object-method in the other



(a) “Stretched” computations



(b) Before:Large Number of Stretches



(c) After:Fewer Stretches

Fig. 6. “Stretched” executions, and their resolution

timesteps. Such stretching was seen to be more pronounced (stretches of over 50msecs were observed) when we used all 4 processors on each node, in a bid to utilize all 3,000 processors.

Eliminating stretches involved fine tuning the elan communication library provided by Quadrics through significant trial-and-error experimentation. It also involved the use of blocking receives to reduce operating system interference which was causing the stretches. Although we cannot show all the steps involved, the evaluation process was assisted significantly by Projections. Specifically useful was the grainsize analysis histogram that we discussed in an earlier section. Fig. 6(b) shows that about 30 objects were stretched in an earlier 2,250 processor run, beyond 5 msecs, with the largest one well over 20 msecs. After all the fixes were applied, the resultant histogram on 3,000 processors shows (Fig. 6(c)) only 5 stretched executions, with the largest being only 7.5 msecs, in spite of the fact that we are now using all 4 processors on each node.

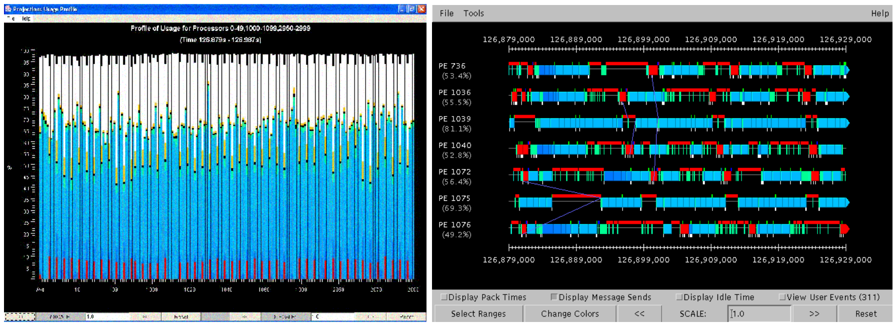


Fig. 7. Profile (a) and Timeline (b) view of a 3000 processor run.

Triumph! Tera-Flop using all 3000 Processors: After using elan based communication with several optimizations, and a few more relatively minor optimizations, we were able to run the application on all 3,000 processors. The time per step was 12 msecs, leading to 1.04 TF performance! Within these 12 msecs, many processors send 30-50 messages containing atoms coordinates (8-10KB each), receive results from the 30-50 force computations objects, carry out asynchronous reductions for total kinetic and potential energies, and migrate atoms to neighboring (26) boxes every 8 steps. For molecular dynamics for biophysics, this is an unprecedented level of performance, exceeding the previous state of art by almost an order of magnitude.

The profile view of selected 200 processors (0-49, 1000-1100, 2950-2999) is shown in Figure 7(a). The white area at the top represents idle time, which is quite substantial (25% or so). Timeline views (Figure 7(b)) show that load balancing is still a possible issue (See processor 1039, identified from the profile view), while communication subsystem still is showing minor, but significant hiccups (a message sent from one of the processor 1076 is not available on processor 1074 for over 10 msec after it is sent). These observations indicate that further performance improvement may be possible!

5 Conclusion and Future Work

We introduced *Projections*, a performance analysis tool used in conjunction with the Charm++ parallel programming system. Projection tracing system automatically creates execution traces, in brief "summary" mode, or detailed "log" mode. We showed how the analysis system, and various views it presents, were used in scaling a production quality application to 3,000 processors and 1 TF.

This experience has helped us identify several new capabilities for Projections. As the problem at hand demanded new analysis, we had to add new capabilities to Projections. Now, we plan to extend Projections, so that users can add such capabilities by expressing simple queries or predicates they want evaluated. The relatively large number and size of trace files in the *log* mode has led us to extend the *summary* mode so that more insightful information can be captured with fewer bytes of summary data. Linking performance visualization system to the source, as done in Pablo, will also be another useful extension. We have already added preliminary features that use performance counters which will be used in an integrated automatic analysis system. We are currently also in the process of extending these features to AMPI.

Acknowledgements

NAMD was developed at the Theoretical Biophysics Group (Beckman Institute, University of Illinois) and funded by the National Institutes of Health(NIH PHS 5 P41 RR05969-04). Projections and Charm++ are supported by several projects funded by the Department of Energy (subcontract B523819) and the National Science Foundation (NSF DMR 0121695). The parallel runs were carried out primarily at the Pittsburgh Supercomputing Center(PSC) and the National Center for Supercomputing Applications (NCSA). We are thankful to these organizations and their staff for their continued assistance and for the early access and computer time that we were provided for this work. In particular we would like to thank Jim Phillips(TBG), David O'Neal, Sergiu Sanielevici, John Kochmar and Chad Vizino from PSC and Richard Foster(Hewlett Packard) for helping us make the runs at PSC Lemieux and providing us with technical support.

References

[AMCA+95] V.S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D.A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing'95*, December 1995.

[BKdSH01] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[HE91] M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, September 1991.

[KK96] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[KSB+99] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[LK01] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

[PZKK02] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[San96] Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, NM, February 1996.

[SK96] Amitabh Sinha and L. V. Kale. Towards Automatic Peformance Analysis. In *Proceedings of International Conference on Parallel Processing*, volume III, pages 53–60, August 1996.