Load-balanced drift-diffusion model simulation: cluster software performance evaluation

S. Bounanos, M. Fleury, S. Nicolas, and A. Vickers

Department of Electronic Systems Engineering
University of Essex, United Kingdom

Running head: Load-balanced drift-diffusion simulation

Authors:

- S. Bounanos
- M. Fleury (corresponding author) e-mail: fleum@essex.ac.uk
- S. Nicholas
- A. Vickers

The authors are all affiliated to:

Department of Electronic Systems Engineering
University of Essex,
Colchester, CO4 3SQ,
United Kingdom
{sbouna,fleum,snicol,vicka}@essex.ac.uk

Abstract

Design of an avalanche photodiode with high gain and low noise that can achieve single photon counting is a research application of the drift-diffusion model. System-level load-balancing when combined with application-level load-balancing is shown to improve the performance of simulation code on a Linux cluster supercomputer. The two forms of load balancing are required to approach a smooth increase in performance with scaling. Centralized and distributed organization of the adaptive simulation code reflected the choice of system software. Marked performance differences were observed when two contrasting cluster software parallelization systems, MOSIX and Charm++, were applied. The paper compares the two dynamically load-balanced systems to MPI implementations (LAM and MPICH), which are statically load-balanced. Also considered is AMPI, which is based on Charm++ and includes system-level load-balancing but additionally implements all MPI calls.

1 Introduction

In this paper, an example of an adaptive simulation for a single photon detector (Rarity, Wall, Ridley, Owens, and Tapster 2000) with an application in quantum cryptography is benchmarked. In fact, the simulated drift-diffusion transport model (Korman and Mayergoyz 1990) is applicable to a wide range of layered semi-conductor devices, whereas other models are limited to particular device geometries and bias conditions (Lin and Wu 1987). We have also used it for a number of devices including a back-gated Metal-semiconductor-metal (MSM)(Mahseyekhi 1999) and a hot electron barrier light emitter (HEBLE)(Aiyarak 2000).

Our experimental C++ code (Aiyarak 2000), written for uniprocessor simulations to find the profiles of the electric field and potential, is probably typical of that existing in other research laboratories. This code has the merit that: it is well-understood and trusted by us; and is easily modified to model exotic research devices. However, performance is slow on a uniprocessor, which seriously restricts: the size of models; their granularity; and their flexibility. Parallel processing on a cluster computer potentially provides a scalable solution by virtue of the distributed memory model. In this paper, the traditional drift-diffusion algorithm is parallelized, which allowed us to benchmark a variety of software packages. A more complex simulation algorithm (such as multigrid, discussed below) would make it more difficult to establish the relative merits of the support software.

The opportunity to convert the code to run on a Linux cluster supercomputer allowed dynamic load balancing to be introduced. The paper benchmarks this application using both the MOSIX (Barak, Guday, and Wheeler 1993) and Charm++ (Kalé and Krishnan 1993) load-balancing systems. The relative performance was further compared to that arising from the popular Message Passing Interface (MPI) (Gropp, Lusk, and Skjellum 1994) (both the LAM (Burns, Daoud, and Vaigl 1994) and MPICH (Gropp, Lusk, Doss, and Skjellum 1996) implementations). This study also investigates the performance of AMPI (Bhandarkar, Kalé, de Sturler, and Hoeflinger 2001), based upon Charm objects and including object migration. AMPI is a full implementation of MPI. While MPI applies a simple, static load-balancing system, MOSIX, Charm++, and AMPI dynamically migrate processes or threads between physical processors. Though case studies of adaptive simulations are available, these are normally confined to an implementation using just one system. Only system-level load-balancing is usually applied, whereas we have found that application-level load-balancing improves performance and scaling behavior, at a cost in development time.

The drift-diffusion transport model requires solution of non-linear Poisson equation and current continuity equations in the classical domain, together with a numerical solution to Schrödinger's equation if

extending to the quantum domain. (Quantum-level simulation allows a correct estimation of electron and hole density in the quantum well that lies between intrinsic or lightly-doped MSM regions.) A method of decoupling the computation of the constituent equations using block iteration was devised by Gummel (Gummel 1964). This method has a number of important advantages: it has global convergence in that convergence is guaranteed for any initial guess; the solution of simultaneous equations is avoided by updating the electrostatic and quasi-Fermi potentials at each mesh point by an explicit formula; and it only requires storage of the results of one previous iteration.

Because solution of Gummel's method is by a finite difference method, it is capable of parallelization, and indeed (Darling and Mayergoyz 1990) reports an implementation on NASA's Massively Parallel Processor (MPP). However, machines of this type are becoming a rarity, whereas clusters are becoming a common resource. Moreover, bit-wise processing of fixed-point code was needed to operate on the MPP. An alternative route to speed-up is use of a simulation algorithm with faster convergence but more complex data structures. For example, in (Mitchell 2001) in respect to multilevel algorithms for partial differential equations (PDEs) states "Effective parallelization is difficult because of the irregular nature of both adaptively refined grids and the multigrid process". For further discussion on multigrid methods refer to (Carey, Richardson, Reed, and Mulvaney 1996).

Linux clusters (Sterling 2002a) have been widely deployed to reduce run-times for scientific problems, thus allowing more tests to be performed, rather than be faced by long waits before new model parameters can be tried out. For the adaptive drift-diffusion model (DDM) simulation reported herein, the runtime was 54 min. on a single 2.8 GHz Athlon processor, even when the problem had been considerably simplified, compared to its original setting. Most cluster system software will result in a speedup up to a limit. However, this paper reports that *choice* of the type of parallel processing software has a considerable influence on wall-clock performance and scalability. Of the cluster system software considered in this paper, the MOSIX single system image software lends itself to a centralized organization. Boundary data are passed to a parent node before redistribution, after checking for convergence. The Charm++ asynchronous message-passing system, is more suited to distributed exchange of data and checks for convergence. Therefore, choice of parallelizing software influences the software architecture of the application.

Parallelization of the optoelectronic computation was affected by a temporal dependency within simulation iterations. It was found possible to by-pass the effect of sub-cycle dependencies with a gain in performance. However, the parallel version of the simulation was also slowed down by a severe spatial dependency across the simulation grid. Though MOSIX provides kernel-level load-balancing, this paper reports that the overhead from transparent process migration may be high. Therefore, an additional form of

load-balancing was introduced to overcome spatial dependency. Application profiling identified computational hot-spots. A curve was fitted to this profile using numerical analysis techniques, with the area under the curve integrated to make an equal division of work. Workload moulding improved performance and smoothed scalability as the number of cluster nodes was increased. Thus, the performance can be scaled without undue performance loss for particular cluster sizes. Results show that there is a synergy between application- and system-level load balancing. Whether application load-balancing is applied will be a trade-off between the performance time gained and the development time spent to investigate application-level load balancing.

There are a number of commercial semi-conductor modelling packages available, with well-known tools from Silvaco International and Synopsys Inc.. For example, DESSIS [1] simulates the electrical, thermal, and optical characteristics of semiconductor devices. It handles 1D, 2D, and 3D geometries, mixed-mode circuit simulation with compact models, and numeric devices and contains a comprehensive set of physical models that can be applied to relevant semiconductor devices and operating conditions. DESSIS is user extensible but its core models appear to be in 'black box format'. In the initial stages of research, a working knowledge of models is important and there is a need to experiment with those models. DESSIS does provide API calls allowing multi-threading, which implies a shared-memory model of computing as one would naturally expect with a tool that would normally be used in a desktop situation. However, a shared-memory model does imply limits to scalability, and the DESSIS v.9 user manual comments as such. There can now also be a cost threshold in using a software tool in research and within smaller organizations. This does not imply in any way that this is the case for DESSIS v.9., which is simply used as a convenient example. Provider/user agreements may also differ between each user and between tool versions.

The rest of the paper is organized as follows. Section 2 introduces the software and hardware computing environment utilized by us. Section 3 outlines the problem and the parallel decomposition strategies adopted to achieve sensible experiment turnaround times. Section 4 describes application level load-balancing, while Section 5 analyzes comparative performance in terms of convergence, scalability and run times, across the different computational configurations. Finally, Section 6 draws some general conclusions about the techniques employed.

[1] Refer to `http://www.synopsys.com/products/acmgr/ise/dessis_ds.html`.

2 Computing Environment

This section describes the two contrasting cluster parallel processing systems, MOSIX and Charm++, the MPI and AMPI implementations, and gives details of our cluster hardware.

2.1 MOSIX

The MOSIX *preemptive process migration* system (A., S., and R. 1993)(Barak and La'adan 1998) originated as a symmetrical distributed operating system and on a cluster acts as a 'single system image', performing automatic load balancing (Amir, Awerbuch, Barak, Borgstom, and Keren 2000) by migrating processes to faster or idle nodes, or 'memory ushering' (Barak and Braverman 1997) in which processes are migrated away from a node that is running out of memory to avoid heavy page swapping. Though MOSIX can be employed as a throughput engine, it has been recommended by its originators (Barak, La'adan, and Shiloh 1999) for parallel applications on cluster computers, for example molecular dynamics simulations. The software version considered is the openMosix (Bar 2002) version of MOSIX.

Migration transparency is achieved as follows. The process is separated into a user context (also known as 'the remote'), which can be located anywhere in the cluster, and a system context (also known as 'the deputy'), which remains on the home node. As a process must interact with its environment via its system context (in other words using kernel system calls), and the interface between system and user context is well defined, it is possible to capture this interaction and execute it at the present location of the user context (if the call is site-independent) or forward it over the network to the system context. This interaction is shown in Figure 1.

Instead of a spawn (creating a new process), MOSIX employs a Unix `fork()` to create a duplicate or child process. After the `fork`, the program counter of parent and child process is at the same position, and the two processes are only distinguished by the `fork` return value. `Fork`-style programming relies on the parent and child inheriting the same environment. In contrast, immediately 'execing' a new process does not assume a common environment, at a possible cost in set-up delay.

Given that under MOSIX file handles for filesystem and network based I/O are created on the home node and located in the system context, the separation imposes an overhead on communication. (Calls to `time()` are also relayed to the system context.) These issues are partially addressed by 'direct filesystem access' over a NFS-like filesystem known as the openMOSIX filesystem (oMFS) and ongoing work on 'migratable' network sockets, the aim being to reduce overhead by making some common I/O operations site independent. However, this did mean that migratable sockets were not available for this work.

2.2 Charm++

Charm++ (Kalé and Krishnan 1993) (Kale and Krishnan 1996) is an object-oriented C++ development of the Charm parallel programming environment from the University of Illinois. Charm employs a message-driven programming style known as "split-phase" or "continuation passing". Figure 2 gives a simple illustration of the continuation-passing model. In this model, computation and communication can overlap to a degree not attainable by other paradigms, *e.g.*, blocking remote procedure calls (RPCs), leading to a more "latency-tolerant" application. This does mean, however, that message queueing and scheduling are required at the system level, by linking in a runtime environment, and are possibly required at the application level.

The Charm system is logically divided in two components, the Charm language (Ramkumar and Kalé 1994a) and the Charm runtime (Ramkumar and Kalé 1994b). The Charm language, which conforms to the actor or active object model (Agha 1986), is designed around:

- concurrent objects (called *chares*) which encapsulate data and computation. Remotely accessible chare functions are explicitly declared as such and become *entry points*;
- prioritized messages, the primary communication mechanism, which are delivered to remote entry points asynchronously and non-preemptively;
- data abstractions such as distributed tables, which are modified at runtime using messages, and read-only global variables which are defined at compile time and can be accessed synchronously, as they are copied to all instances of the Charm runtime.

To support data-driven execution, the Charm runtime has been structured into the following modules:

- The machine interface module which implements machine-specific functionality.
- The language features module that supports Charm's data abstractions and a number of common parallel programming algorithms such as quiescence detection.
- The system 'strategies' module which is responsible for load balancing and memory and message queue management.
- The core kernel which interfaces to the queueing module and runs a 'pick and process' loop to deliver messages to local chares or remote runtimes.

While Charm uses 'C' as the base language, modules and chares are specified using non-standard syntax, requiring a pre-processing pass before compilation.

Charm++ has inherited the runtime architecture and retained most of the original language features, many of which have been rewritten to take advantage of C++ constructs. Two major additions to Charm are chare groups and chare arrays of arbitrary dimensions, intended for shared memory multiprocessors and networks of workstations respectively. Chare arrays are also the Charm++ collections of choice for operations such as barrier synchronization, reductions and message broadcasts. Figure 3 shows interaction between chare array elements via message passing, from the programmer's and system's point of view.

Charm++ provides measurement-based run-time load-balancing modules (Parallel Programming Laboratory, University of Illinois, Urbana 2001) for iterative applications (such as DDM simulations) as these generally display a computation and communication pattern that persists over time. (For applications with no or limited temporal behavior correlation, not applicable to DDM simulations, Charm++ employs 'seed' load balancers, which speculatively move seeds for new chares amongst processors.) Dynamic load-balancing is only available to chares in chare arrays. All chares are instrumented but there is a choice as to whether statistics are collected centrally or in distributed fashion. The centralized system employed (collection on processor 0) in the DDM simulation application incurs more overhead but in principle should allow more accurate decisions to be made.

Of the non-seed balancers, the `RefineCommLB` module was selected for the DDM simulation. Other balancers were experimented with but it was possible for the application to misbehave if a chare was registered as a sink for a collective communication operation. This included the reduction operation that was used in the Charm++ version of the DDM simulation, Section 3.3. Only load-balancers for which a chare could be registered as non-migratable were suitable. Fortunately, this includes the `RefineCommLB` module, which takes into account both communication and chare computational load, and, hence, was the most applicable of twelve available load-balancing strategies. Migration can be restricted to preset synchronization points, but the default option, periodic migration, was selected. Migration takes place unless the chare is completing computation of an entry point method.

2.3 MPI: MPICH and LAM Implementations

The structure of MPI is very well-known and, hence, we simply refer the reader to one of the books on MPI (Gropp, Lusk, and Skjellum 1994)(Snir, Otto, Huss-Lederman, Walker, and Dongarra 1998). For MPI-2 extensions, not considered herein, refer to (Gropp, Huss-Lederman, Lumsdains, Lusk, Nitzberg, Saphir, and Snir 1998). The application was ported to two implementations of MPI, MPICH v. 1.2.5.3 and LAM v. 7.1.1. MPICH was compiled and run with `ch_p4mpd` support for job startup, while LAM used its `rsh` boot method.

The LAM implementation of MPI (Burns, Daoud, and Vaigl 1994)(Squyres and Lumsdaine 2003) originally consisted only of the `lamd` Request Progression Interface (RPI) message-passing engine uses indirect communication among processes, via LAM daemons running on each node. As LAM is single threaded, this is currently the only RPI that provides truly asynchronous message passing. Subsequently, a variety of options have become available. The `sysv` RPI uses shared memory to communicate between MPI processes residing on the same node (with System V semaphores for locking), and TCP to reach remote MPI tasks. A similar module, `usysv`, uses spinlocks (with a backoff) rather than semaphores. This is intended for Symmetric Multiprocessor (SMP) nodes with more processors than running MPI processes (of the same job), for which busy waiting can improve performance by avoiding preemption while the shared memory is locked. In the DDM simulation application, in which potentially many MPI processes run on the same uniprocessor node, TCP is clearly inefficient, as it does not take advantage of shared memory. The `usysv` RPI would cause processes to compete for CPU time even while not doing useful work. Finally, in preliminary investigations `lamd` gave a net drop in performance, due to its increased message passing latency. Therefore, the default `sysv` RPI was the optimal communication method.

When TCP sockets are used for message passing, LAM employs different application layer protocols depending on the message size. Short messages are sent immediately and in a non-blocking manner, while for other sizes a *rendezvous* protocol ensures that a matching receive call has been posted on the remote side before the message body is sent. For the same efficiency reasons, similar distinctions are made by the `sysv` RPI (Open Systems Laboratory, Indiana University 2004). In the DDM simulation application, the message body size is always 624 bytes for boundaries (and much smaller for reductions), which is well below the short message threshold of all RPIs.

The MPICH implementation of MPI (Gropp, Lusk, Doss, and Skjellum 1996) is characterized by its wide range of native implementations, which it supports through its ‘abstract device interface’. For example, like LAM MPICH has a shared-memory interface (Gropp and Lusk 1997). MPICH has three different protocols depending on message length. The default ‘short’ message size is 1 Kbyte, which is small enough to be sent as part of a control message. Thus, again the DDM simulation’s boundary messages fall within this threshold. ’Eager’ messages of default length below 12 Kbyte are sent immediately, whereas larger ’rendezvous’ messages require a receive to be posted. In (Nevin 1996), benchmarking on a workstation cluster confirmed that MPICH’s large message threshold is larger than LAM’s. Various performance studies, *e.g.* (Nupairoj and Ni 1996), (S. Markus, Patazapoulos, A. L. Ocken, Wu, Weerawarana, and Maharry 1996) (also for a finite-element solver application),(Nevin 1996), give differing views of LAM and MPICHs’ relative performance. However, these studies also vary respectively by whether a custom paral-

lel processor, a (heterogeneous) network of workstation, or a Linux cluster was employed, which implies relative performance may vary according to target parallel processor.

2.4 AMPI

AMPI (Bhandarkar, Kalé, de Sturler, and Hoeflinger 2001)(Huang, Lawlor, and Kalé 2003) is an implementation of MPI v. 1.1 on top of the Charm++ system. AMPI was introduced to make it possible for existing parallel applications to use Charm++ features without an extensive rewrite. Load balancing, migration, and checkpointing calls are exported as MPI extensions. To support these, the `main` functions of MPI processes are renamed and run as user-level threads. All MPI peer and collective communication calls are mapped to wrappers and macros that in turn use the Charm++ runtime. This allows MPI processes to be migrated between Charm++ runtime instances as needed. Converted code must be made thread safe, principally by making global variables private (either manually — as used in this implementation — or through the ELF object code format for position-independent code). Code that modifies global system structures, such as standard stream redirection used for logging, also had to be adapted in our implementation.

Additional modifications are necessary to make MPI processes 'migratable'. These are mainly a set of pack/unpack routines which are registered with the Charm++ runtime and take care of the usual tasks of freeing heap-allocated memory and saving and restoring structures. These can also be used for checkpointing an MPI process by writing a serialized copy of its data on disk. Migration itself is apparently controlled by barrier synchronization, which is initiated by carefully placed `MPI_Migrate` calls.

The same load-balancing strategy and settings were employed in AMPI as in Charm++, and reference to AMPI henceforth assumes that AMPI is run with automatic load-balancing.

2.5 Cluster Hardware

The cluster employed consists of up to thirty-seven processing nodes connected with two Gigabit (Gb) Ethernet switches (Sterling 2002c), designated as the "Heap" (Ridge, Becker, Merkey, and Sterling 1997). Each node is a small form factor Shuttle box (Model XPC SN41G2) with an AMD Athlon XP 2800+ Barton core (CPU frequency 2 GHz) and dual channel 1 GB DDR333 RAM. In Figure 4, the nodes are connected via the two 24 port Gb Ethernet switches manufactured by D-Link (model DGS-1024T). Each switch is non-blocking and allows full-duplex Gb bandwidth between any pair of ports simultaneously. Each switch can be connected via independent network cards in the server, to allow the cluster to be partitioned into two, designated as "big" and "little heap" in Figure 4. Alternatively, the two switches can be linked together via

a Gb port, though obviously communication between nodes on different switches becomes blocking. The switches are unmanaged and, therefore, unable to carry ”jumbo” 9000 B Ethernet frames (which would lead to an increase in communication efficiency of about 1.5% and, more significantly, a considerable decrease in frame processing overhead(Breyer and Riley 1999)).

To make maintenance and cluster-wide propagation of configuration changes easier, at boot time a file server transfers using multicast a root file system to the local disc of all nodes, while other file systems are accessed via the Network File System (NFS) (Sterling 2002b).

3 Parallel Decomposition

This section details the test problem and its numerical form. Parallel decomposition of the problem by centralized and distributed algorithms is then described.

3.1 Modelling Optoelectronic Semi-conductors

Metal-semiconductor-metal (MSMs) photodiodes are finding a ready use in optical transmission (Vickers, Hassan, Mashakekhi, Griguoli, and Hopkinson 1996) and are attractive for other optoelectronic applications, such as single photon counting or detection of an incident radio frequency field. Because they can be of a large area without sacrificing device response, they offer low noise and high sensitivity (Anselm, Nie, Lenox, Hansing, Campbell, and Streetman 1998)(Lenox, Nie, Yan, Kinsey, Holmes, Streetman, and Campbell 1999). They also offer high bandwidth, and their planar structure is well suited to integrated circuits. Research is ongoing, as to how to optimize such devices. For example, there is a need to establish the best doping level for facilitating the jumping of particle holes between semi-conductor layers.

The example simulation in this paper models a Fabry-Perot resonant-cavity-enhanced avalanche photodiode (APD) based on a heterojunction formed between *InP* and *InGaAs* layers, of a type suitable for long-wavelength optical communication systems. The intention is to allow doping concentrations, percentage of aluminium, and size and position of the quantum well to be altered and simulated. A problem with these devices is that they have a slow response if holes are allowed to pile up as a result of the valence band discontinuity between the *InP* and *InGaAs* layers. To increase the transition rate of the holes, a layer of *InGaAsP* is inserted, separating absorption and multiplication (SAM) layers, with the intention of grading (forming a smoother transition) of the discontinuity, to allow the holes to pass the barrier more easily (Matsushima, Akiba, Sakai, Kushiro, Noda, and Utaka 1982)(Tagushi, Torikai, and Sugimoto 1988)(Parks, Smith, Brenman, and Tarof 1996). Further discussion of the operation of these devices is outside the scope

of this paper and the interested reader is referred to (Bae-Lev 1993)(Chuang 1995)(Wada and Hasegawa 1999).

An MSM consists of layers of semiconductor material each with differing doping consistencies. Figure 5 is a schematic diagram of a 2-D cross-section of the device that was simulated. The general form of simulation is by the iterative finite-element method. Figure 6 shows a 2-D mesh superimposed across the device, with Dirichlet boundary conditions (electrostatic potential is the sum of the applied voltage and the built-in potential in the region) where contact is made with a metal pad assumed to be Ohmic. Neumann conditions (reflection of values) were applied at the internal boundaries. Each mesh point contains information or properties of the material, *e.g.* doping density, electrostatic potential.

A 2-D simulation was used, as the device characteristics are the same for any parallel cross-section normal to the contacts at the n- and p-doped regions. For the APD SAM device under test, the mesh points were spaced 2.5 nm apart. Though a non-uniform mesh gives a quicker solution, for ease of writing the initial C++ program, the mesh was made uniform. Finer grid point separation at the heterojunction would have improved resolution of a voltage spike/notch at the *Inp* and *InGaAsP* layers, but this would cause difficulty in initializing the mesh points, and, if the separation was made finer across the device, simulation turnaround time on the sequential version would become impossibly long. Finally, even simulation of a 2-D surface covering the device was a lengthy process, and, hence, a narrow effectively 1-D strip was actually simulated. As is pointed out in Section 1, 1-D run times remain slow on current PCs with a high-end processor if a traditional algorithm is employed.

As mentioned in Section 1, though we are aware that faster convergence might be achieved by a reframing of the problem, in this paper the approach of (Korman and Mayergoyz 1990) is taken as a given. We briefly mention other ways of possibly improving the convergence rate. Gummel's method (Gummel 1964) is sometimes combined with Newton's coupled method (a generalization of the Newton-Raphson method for finding the roots of an equation) in conditions of high bias. Whereas Newton's method is not guaranteed to converge for all initial parameterizations, it may achieve faster convergence. It is entirely possible that a two or multigrid method, which has been applied in the non-linear case (Molenaar 1993)(Carey, Richardson, Reed, and Mulvaney 1996), will achieve faster convergence. However, this would require conversion of software to multigrid techniques, whereas at present we have more experience and confidence with the straightforward application of the equations. There has been considerable research activity on parallel multigrid methods for PDEs and related problems, examples being (Chan and Tuminaro 1987),(Frederickson and McBryan 1988),(Heise and Jung 1997)(Griebel and Zumbusch 1998).

Several equations (Poisson and continuity equations) govern the simulation model. Each equation is

solved within what we designated a sub-cycle of each simulation iteration. The order of solution is counter-intuitive in that an iteration of the governing equation is solved first, before the constituents forming that equation such as the current densities are found. This is because initial values are used for Poisson's equation in the very first iteration and then subsequent values for current density and so on are passed forward into the next iteration of the Poisson equation solver. If ϕ^k represents the value of the potential found in iteration k of the Poisson equation, then the value relies on the Slotboom (exponential quasi-Fermi potentials) variables (Korman and Mayergoyz 1990), u^k and v^k, derived from the continuity equations. In turn, the values of u^{k+1} and v^{k+1} are found using ϕ^k, u^k, and v^k. Convergence of the simulation is established by testing if the change in the potentials and the Slotboom variables across the mesh is within a given tolerance:

$$\sigma_{i,j}\phi_{i,j}^{k+1} - \sigma_{i,j}\phi_{i,j}^{k} \leq t \tag{1}$$

$$\sigma_{i,j}u_{i,j}^{k+1} - \sigma_{i,j}u_{i,j}^{k} \leq t \tag{2}$$

$$\sigma_{i,j}v_{i,j}^{k+1} - \sigma_{i,j}v_{i,j}^{k} \leq t, \tag{3}$$

where t is the tolerance, and the summation is across all of the mesh points.

It was possible to overlap the computation of different sub-cycles, *i.e.* the equations outlined previously, which reduces the synchronization dependency, but the presented solution avoids the dependency altogether. Sub-cycle decomposition is further discussed in Section 3.5. Tests showed that convergence still occurred. For all the parallel versions and runs, to verify correctness, a new grid was assembled and stored on disk for later comparison with the results of the sequential solver, which did use the four sub-cycle decomposition.

3.2 MOSIX Centralized Algorithm

To implement a parallel version of the simulation, a prototype was initially written in Perl using System V shared memory (Leach 1994) as the inter-process communication mechanism. With this method, the common 'slab' approach was used for the parallelization, as shown in Fig. 8. The grid was divided into segments of equal size and distributed to child processes by 'freezing' (serializing) and 'thawing' (deserializing) the grid, and copying to a shared memory region. Migration of the child processes, despite the use of shared memory, was possible with the shared memory migration patch, migSHM [2], for openMOSIX. However, using shared memory proved untenable, due to the immaturity (at that time) of the migSHM

[2] Available from `http://opensource.codito.com/migshm/` at 6.2.05.

add-on. The additional value of this exercise was that the correctness of the parallel decomposition was established, without the need to move outside a desktop machine.

Parallel decomposition of the MOSIX version of the simulation initially took advantage of the same 'slab' approach as was employed in the prototype. The grid was divided into segments of equal size and distributed to child processes. As the cluster consists of homogeneous processors (and for up to 24 nodes, links are homogeneous), it was initially thought that a simple division of the grid among the child processes would be sufficient. Figure 9 shows an example division of the 1820×1 grid used by this solver for six processes, with the resulting dataflow. Message sizes before addition of protocol headers were 1312 bytes, within the Ethernet frame size.

The parent process initializes the grid, prepares division offsets and bookkeeping structures, forks the required number of children and listens for connections. Each child uses the inherited information to select a segment out of the global grid, initialize its own structures and connect to the parent.

As MOSIX processes can potentially migrate a number of times, it is important to keep the memory footprint of each child process as small as possible. To help reduce migration overhead, the parent allocates the global grid and other initialization-only structures on the heap, where they can be freed by the children when no longer needed.

When all children have connected, the parent locks itself to the home node using the MOSIX `/proc` API and triggers the computation. The children begin to process their segments, increasing the load on the home node and creating an imbalance in the cluster. This is detected by the MOSIX load-balancing subsystem and child processes are migrated away to idle or under loaded nodes within the first few hundred iterations (typically less than 1% of the total number).

The MOSIX solver processes use Unix domain stream sockets. As MOSIX wraps all deputy-remote communication in its own UDP-based protocol, the encapsulated protocol should be kept as simple as possible to maximize efficiency. Unix domain stream sockets achieve low overhead and have semantics similar to their Internet domain counterparts, aiding code reusability.

Communication in the MOSIX solver reflects the MOSIX architecture and is done in a centralized fashion. At the end of each iteration, the parent receives each child's boundary rows as well as a value calculated from its local grid segment, the aggregate of which is checked against the convergence condition. Unless the algorithm has converged, the parent sends the boundaries to the owner's neighbors to resume the computation.

Although our cluster consists of little-endian machines only, boundary elements are serialized and encoded in network byte order using a lightweight, eXternal Data Representation (XDR) (Srinivasan 1995)

based library before they are transmitted. This is to ensure a degree of parity with the Charm++ solver, which packs message data in a platform independent format using its PUP interface (Kalé and Krishnan 1993).

When the algorithm converges, the child processes receive a signal by the parent to end processing and write out their local grids.

3.3 Charm++ Distributed Algorithm

The Charm++ version employs a distributed algorithm, with chares sending boundaries directly to neighbors.

In the initialization phase in the Charm++ version the global grid is declared as a read-only global variable and is initialized in the main chare's constructor. The main chare then determines grid division and instantiates a chare array with the required number of elements, which are distributed to the pool of available nodes by the Charm++ runtime. At construction time, each chare creates its local grid segment from the global copy and, having completed its initialization, notifies the main chare that it is ready and becomes idle. When all chares are ready, the main chare broadcasts a message to the array to begin the computation.

The communication style used in the Charm++ solver is significantly different to that of the MOSIX version. Figure 10 shows the exchange of boundary messages by chares, which periodically send their local maxima for convergence testing. This is done by calling `contribute()` with:

1. a callback object that is created at startup with the mainchare's `recv-lmax` entry point as an argument.

2. the size of the value, and the value itself, which is serialized and sent off to the runtime that has the mainchare.

3. the name of the method that will be called to aggregate the results, in this case the built-in `CkReduction::sum_double`.

Boundary message sizes (before addition of protocol headers) were 632 bytes, but unlike MOSIX the boundary data is not aggregated. Thus, the total message data exchanged is 1264 bytes, similar to MOSIX. Though choice of UDP or TCP transport layer protocol is possible, UDP was chosen to reduce latency. Messages in Charm++ are sent asynchronously and, therefore, to guard against using out-of-order boundaries, messages are sequenced by the sender and buffered at the receiver. Chares resume processing

when their receive-boundary entry point is called and when boundaries from all neighbors and with the correct iteration are found in the buffer.

Likewise, in the Charm++ implementation, the main chare broadcasts a message to the array, to which chares respond with a message containing their grid segments. Values used for convergence tests are not piggybacked on boundary messages, but are periodically aggregated using Charm++'s array reduction mechanism.

3.4 MPI Algorithm and Communication

Conversion of the solver to MPI was via the Charm++ version. This route has the advantage that comparison is made more direct. It also means that the same distributed algorithm was employed in the MPI versions.

The MPI processes use a combination of blocking and immediate calls (Gropp, Lusk, and Skjellum 1994) to maximize computation/communication overlap. At each iteration step:

1. The process blocks to complete an immediate receive request posted in the previous step.
2. An immediate, non-blocking, receive is posted for the next step, and the request stored.
3. The grid chunk with the new boundaries obtained from (1) is processed and the step number is incremented.
4. Boundaries for the current step are packed and sent to neighbors using a standard blocking send.
5. All processes perform a reduction and the result is checked for convergence.

Whereas contiguous data is most efficiently sent via standard MPI calls, derived datatypes (DDTs) are often used to send data scattered in memory to avoid copying. Current MPI implementations using DDTs are reported (Byna, Gropp, Sun, and Thakur 2003) to offer little advantage over judicious application-level packing. Hence, the DDM simulation currently uses MPI_PACKED datatypes, with generalized pack/unpack/sizers inspired by Charm++'s PUP framework (Kalé and Krishnan 1993) (refer to Section 3.2). However, testing after the main benchmarks were completed, shown in Fig. 11, suggests that, for the LAM implementation, at least DDTs offer a performance improvement and complexity reduction (from the user point-of-view). The data packed and sent in Fig. 11 is a DDM simulation boundary, *i.e.* three grid points with 26 doubles each, amounting to $3 \times 26 \times 8 = 624$ bytes.

3.5 Accounting for Sub-cycles

Because calculation of a sub-cycle, *i.e.* particular equation as outline in Section 3, is dependent on boundary values within the same cycle and sub-cycle, it is strictly necessary to stagger the calculation of adjacent segments of the simulated strip. Figure 12 shows, for a strip divided between (say) just three processors, then each must wait for the completion of the same sub-cycle on adjacent simulation strips.

The top half of Figure 13 details the computation order and data dependencies for each sub-cycle in the sequential version of the code. In the 'reset' sub-cycle, the values in the central column are copied to the 'halo' area (open circles) of the simulated linear strip. The order of processing follows the direction of the arrowed line, *i.e.* vertically downwards. In the 'phi' sub-cycle, each calculated value, *i.e.* the filled circles, require data from the four-nearest neighbors. For the top-left value, the reliance on two halo values is shown by dashed lines. Elsewhere, the reliance is implicit and not shown by dashed lines. Calculation proceeds in the order indicated by the arrowed heavy line. The 'u', 'v', and 'field' sub-cycles have similar dependencies on the halo region, though with differing calculation orders.

The bottom half of Figure 13 shows the parallel implementation, whereby, for a two node parallelization, the linear strip is simply bisected. Clearly, at the junction of the strips, in the sub-cycles following 'reset', the values taken from nearest neighbors are no longer updated ones, but values from a previous cycle. Only at the end of the cycle are the boundary values updated by exchange of the horizontal boundary strip.

4 Application-level Load Balancing

The separation of layers across the 1-D strip is shown in Table 1. Refer to Figure 5 and Section 1 for an explanation of the layers. Analysis at the sub-cycle level led to identification of the cause of load imbalance within the computation. Figure 14 shows a plot of iteration times for the four sub-cycles in the absorption (absorp.) layer of the photodetector, during the computation. (Two of the plots overlap.) In this Figure, three iterative sub-cycles maintain a constant time per iteration, while the first's iteration times increase in an exponential fashion, approximately halfway through the computation. All four sub-cycles in the rest of the grid are constant throughout the computation, with iteration times approximately constant. The cause of the load imbalance was found by instrumenting chares to record times on a per-sub-cycle basis with each of seven chares assigned to a particular layer.

In the sequential solver, this simply means that a single grid segment accounts for much of the total run time. In the parallel version, however, equal grid division results in many processors being idle waiting for

boundaries from those that have been allocated rows from the region with exponentially rising computation times. This can be mitigated in at least three ways, described below in decreasing order of complexity:

Redistribute grid rows at runtime as iteration time increases are detected. This would necessitate an additional channel over which to send load information, and a general solution that does not use *a priori* knowledge of run-time behavior would essentially be a custom load balancer with the complexity and overhead that that implies. Shedding excess load by dynamically creating new chares or processes would be undesirable for the same reasons and, in addition, would complicate scalability measurements. Load balancing techniques suffer if they are performed *ab initio* (as if starting again) at each check point. In (Aggarwal, Motwani, and Zhu 2003), this problem is discussed and a greedy algorithm is introduced for re-balancing load based on new computation and/or communication characteristics. The technique is implemented for Charm++ in (Agarwal 2004).

Add instrumentation calls to quantify the solver's non-linear behavior and use simple numerical analysis techniques to make a weighted grid division. This was straightforward to implement and resulted in significant improvements. The Charm++ solver was instrumented using the Charm++ Projections tool (Kalé, Kumar, Zheng, and Lee 2003), while the MOSIX version used the GRM library which is part of the PROVE grid monitoring and visualization toolkit (Balaton, Kacsuk, and Podhorszki 2001). The general load-balancing problem is akin to that of graph partitioning over parallel processors, which is explored in (Karypis and Kumar 1998). The `ParMetis` tool for MPI, originating from the work in (Karypis and Kumar 1998), collects information about graph node connectivity during a run and applies partitioning algorithms. In the context of Charm++, graph partitioning is discussed further in (Chakravorty 2002).

Decrease the granularity by increasing the number of chares or processes, and let the underlying system distribute them as needed. This technique is well exemplified in the molecular dynamics application of (Vadali, Shi, Kumar, Kale, Tuckerman, and Martyna 2004) in which a large number of virtual processors (chares) are created, specifically 12,800 chares in real space and a quarter of that in 'g-space'. In general, virtualization (Kalé 2002) under Charm allows parallel decomposition to be separated from mapping to a particular parallel machine.

This paper reports the effect of finer granularity by increasing the number of grid segments and using virtual nodes (*i.e.* when more than one node is mapped to a physical processor or cluster node) to host the additional segments.

Once the first subcycle timings (taken every 200 iterations) were available for the absorption layer, then the data was fitted[3] to a suitable function (in this case a 5th degree polynomial). With the functional form of the curve established then a numerical integrator such as Simpson's rule gives sufficient accuracy[4] to find the area under the constant part of the polynomial and the area under the "exponential" part (refer to Fig. 14). Dividing the latter area by the former area gives a factor by which to weight processing time of a row in the absorption layer, assuming a row outside this layer has unity weighting. The rows are then allocated so that each chare is allocated an equal share of the row workload.

This method of load balancing is generally applicable to problems of this sort, though in this example the fact that the unbalanced or data-dependent part of the workload was confined to a single sub-cycle and a single layer of the surface under test reduced the complexity of the task.

5 Performance Results

5.1 Charm++ and MOSIX Timings

The Section assumes that the sub-cycle dependencies mentioned in Section 1 can be ignored, as the results without sub-cycle dependency are the same after convergence as those with sub-cycle dependency. In other words, calculation in one iteration are only dependent on the boundary values from the previous iteration. Section 5.2 examines the effect of including the sub-cycle structure into the simulation, as it is recognized that not all simulations will have the convergence property.

Figure 15 shows comparative timings for the Charm++ and MOSIX version of the simulation. The graph is annotated with a line denoting the number of physical cluster nodes, beyond which more than one simulation process is placed on a cluster node or processor. Also shown is the effect of adding application-level load-balancing (annotated as `lb`) for both the MOSIX and the Charm++ versions. In both cases, load-balancing not only improves timings but also smoothes the performance curve between differing numbers of processes. Beyond fifteen nodes, the MOSIX load-balanced version begins to experience increasing overhead from process migration. In contrast, the Charm++ version, which had no system load-balancing applied in this instance, experiences a drop in performance immediately when more than two simulation processes share a node, with some perturbations thereafter. However, its performance curve is otherwise smooth as the number of nodes is increased. A slight improvement arises from finer segment granularity.

[3] A convenient way for to do this for load-balancing purposes is to use an on-line data fitting service such as that at `http//www.zunzun.com/`.

[4] Simpson's rule is, of course, completely accurate for 2nd degree polynomials.

In Figure 16, speed-up curves show almost linear speed-up for Charm++ up to the number of physical nodes. The workload is not evenly spread between nodes and as a result synchronization delays occur. However, a different situation occurs when more than one (virtual) node is placed on a processor. Then, while one node completes its work, the other(s) on that processor can receive messages and consequently continue with the next step's processing. In the Charm++ distributed algorithm this results in continued speed-up beyond the number of physical nodes, provided the process placement is propitious. The MOSIX version cannot benefit from asynchronous operation because of the centralized communication pattern, resulting in a reduction in speed-up beyond the critical point.

To understand more fully the impact of MOSIX load-balancing by process migration, Figure 17 plots the number of per-node migrations for increasing number of nodes. The total number of migrations for a run is found by multiplying the number of nodes by the average per-node number, when it will be observed that the number of migrations is considerable. In fact, the number rises significantly after the physical nodes are exhausted, and does not then always favor the load-balanced version.

In Figure 18, the effect on speed-up of adding the `RefineCommLB` (rc) Charm++ load-balancing module (Section 2.2) to application load-balancing is shown. In fact, adding system load-balancing brings a relatively small improvement over the 'plain-vanilla' Charm++ version. Additionally, for high numbers of (virtual) nodes, the measured performance oscillates according to node number. We were able to establish that no background process on the cluster was responsible for the oscillation and nor was migration thrashing. Compared to MOSIX, the Charm++ performance continues to improve as more than one chare is placed on a processor. The Charm++ load-balancing module applies a small number of migrations (two or less) for most number of nodes, which is the main reason for the superior performance. However, it is when application level load balancing is added to system level load balancing that the main gains are made. Before about seventy nodes, combining both forms of load-balancing results in an almost 'symbiotic' performance improvement. The resulting speed-up curve is smoothed out, making performance easier to predict.

5.2 Sub-cycle Timings

Figure 19 shows the differing number of iterations needed to achieve convergence, with and without the inclusion of sub-cycles. (In the tests in this Section the cluster was enlarged by just one physical node.) Comparing Charm++ and MOSIX versions, the number of iterations to convergence is actually the same, with the stepped MOSIX curve simply reflecting the fact that measurements were only taken every five nodes. Another feature of the plots is the difference in the number of iterations needed to converge ac-

cording to whether load-balancing is applied or not and indeed according to the number of nodes. This applies equally to the results including sub-cycles, showing a spatial dependency affecting boundary exchanges. However, the main point that Figure 19 illustrates is that the introduction of sub-cycles leads to swifter convergence. Unfortunately, faster convergence, though clearly of benefit in the sequential version of the code, does not lead to faster run-times, as each iteration is that much slower in the sub-cycle version. In Figure 20, below about 25 nodes, the sub-cycle inclusive version under-performs all parallel versions, whereas after about 25 nodes, other effects slow down the MOSIX version.

5.3 Comparing MPI Systems

Figure 21 compares the speed-up performance of MPI systems with system-level load-balancing for Charm++ and MOSIX versions. The MPI-based systems behave very similarly to each other and to Charm++ until the number of physical nodes is exhausted. Given that AMPI uses the same load-balancing system as Charm++, the consistent drop in performance after 55 nodes indicates a weakness in the current implementation of the AMPI software. The addition of system load balancing to Charm++ does bring gains in performance if virtual nodes are available for migration. However, again the erratic Charm++ performance for large numbers of nodes makes the circumstances when load-balancing is advantageous compared to an MPI implementation difficult to predict. There is little to choose between the two implementations of MPI. As in the Charm++ 'plain-vanilla' version of the DDM simulation, the MPI systems benefit from the reduction in latency from placing more than one node on a processor. Hence, speed-up continues to rise beyond the number of physical nodes. Again, this can be ascribed to parallel slackness allowing processing on one virtual node to proceed while another node on the same processor is temporarily stalled.

Application-level load balancing was added to all versions, with the results shown in Figure 22. Charm++ and the MPI implementations were best able to take advantage of this type of load-balancing, though again AMPI's performance is weaker. Performance of these versions gradually rises, as more and more virtual nodes are added. Again, there is a thresholding affect as the number of nodes on any one processor exceeds two and then three. Adding both application load-balancing and system-level loading results in the best speed-ups. However, MPICH's performance is very competitive and exceeds that of the LAM implementation for large numbers of nodes.

5.4 Discussion

Speed-up graphs were chosen in the Sections 5.1 – 5.3 as a way of easily discriminating the relative performance of the various DDM simulation versions. If absolute timings are employed across the range of timings then it becomes difficult to discriminate across the range of times. Figures 23 and 24 shows the absolute timings for the versions respectively without and with application-level load balancing. These graphs allow one to judge the gains to be made from developing application-level load-balancing software for an application of this type. While converting an application to a message-passing form might be reasonably expected of a support programmer, load-balancing software adds another level of difficulty and the possibility of introducing errors. Equally, the graphs presented also allow one to consider whether moving from the highly-portable MPI to Charm++ system load-balancing is justified. This judgement will also take into account other factors, such as Charm++'s object orientation. Unfortunately, AMPI cannot currently be said to offer a middle path for this type of application. Tentatively, AMPI's weaker performance may be ascribed to its more recent provenance, without time to refine the software, or to the overhead of multiple indirections when passing from MPI to Charm++. However, the drop in performance as virtual nodes are used is unexplained.

For the DDM simulation application, adding virtual nodes can bring a performance advantage whether load-balanced or non-load-balanced code is applied. Without the benefit of a set of performance curves then the regime giving optimal performance is not easy to predict. For best performance this study found that a combination of application-level and system-level load-balancing was preferable. The efficiency was 68% on a thirty-seven node cluster.

6 Conclusion

Though the advent of clusters have been widely adopted for scientific computing, to reduce turnarounds further, rather than suffer repeated wait times, requires attention to which parallelizing software system is applied.

Single system image software appears to offer the convenience of transparent process placement and load balancing.convenience. However, for the somewhat typical finite-element problem reported in this paper, a restriction to centralized algorithms incurs a performance penalty, as essentially it does not allow the application to take account of parallel slackness. Message-passing software, based on asynchronous message delivery, recorded better and more consistent wall-clock times (scaling over a range of cluster sizes). A distributed rather than centralized exchange of data and convergence test was found to be preferable.

Automatic process migration imposed a considerable overhead, especially when the problem was decomposed beyond the number of physical cluster nodes. The obverse side to this story is that sequential code parallelized by way of Charm++ requires careful design because of the non-blocking message semantics. The same applies to the various MPI implementations tested.

An application-layer load-balancing scheme, based on elementary numerical analysis techniques, was found to improve all versions of the simulation, in terms of times and performance predictability when scaling the cluster size. Having characterized the paper's problem as 'somewhat' typical, in fact, computation times varied widely within the spatial sampling grid, requiring load balancing, and was also hampered by the need to synchronize between sub-cycles, when transferring from sequential to parallel version. The latter obstacle was overcome by taking values from a previous cycle, which prolonged the number of iterations to convergence but reduced the time for each iteration, leading to a net gain. This may be an unwelcome result to those who seek to parallelize applications without detailed knowledge of the application's characteristics.

System-level load-balancing has a disappointing effect. Applied alone it sometimes serves to degrade speed-up rather than improve it. Only when combined with application-level speedup did we notice an improvement from including a load-balancing module in the Charm++ run-time in terms *both* of a significant increase in speed-up and predictability of performance for a given configuration. In many cases, one of the MPI implementations (other than AMPI) without dynamic load-balancing gave very respectable performance in practical terms for jobs of this computational scale.

References

Agarwal, T. (2004). Strategies for toplogy-aware task mapping and for rebalancing with bounded migrations. Master's thesis, University of Illinois.

Aggarwal, G., R. Motwani, and A. Zhu (2003). The load rebalancing problem. In *15th annual ACM symposium on Parallel algorithms and architectures*, pp. 258–265.

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems"*. MIT, Cambridge, MA.

Aiyarak, P. (2000). *Theoretical and Experimental Studies of the Hot Electron Barrier Light Emitter*. Ph. D. thesis, University of Essex.

Amir, Y., B. Awerbuch, A. Barak, R. S. Borgstom, and A. Keren (2000). An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Trans. on Parallel and Dist. Systems 14*(7), 760–768.

Anselm, K. A., H. Nie, C. Lenox, C. Hansing, J. C. Campbell, and B. G. Streetman (1998). Resonant-cavity-enhanced avalanche photodiodes grown by molecular beam epitaxy on InP for detection near 155μm. *Journal of Vacuum Science Tecnology 16*(3), 1426–1429.

Bae-Lev, A. (1993). *Semiconductors and Electronic Devices* (3rd ed.). Prentice Hall, London.

Balaton, Z., P. Kacsuk, and N. Podhorszki (2001). Application monitoring in the grid with GRM and PROVE. In *International Conference on Computational Science*, pp. 253–262.

Bar, M. (2002). openMOSIX, an open source Linux cluster project, at `http://www.openmosix.org/`.

Barak, A. and A. Braverman (1997). Memory ushering in a scalable computing cluster. In *IEEE 3$^{\underline{rd}}$ Int. Conf. on Algorithms and Architectures for Parallel Processing*, pp. 211–224.

Barak, A., S. Guday, and R. G. Wheeler (1993). *The MOSIX Distributed Operating System, Load Balancing for UNIX*. LNCS no. 672. Springer, Berlin.

Barak, A. and O. La'adan (1998). The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems 13*(4–5), 361–372.

Barak, A., O. La'adan, and A. Shiloh (1999). Scalable cluster computer with MOSIX on LINUX. In *Linux Expo'99*, pp. 95–100.

Bhandarkar, M., L. V. Kalé, E. de Sturler, and J. Hoeflinger (2001). Object-based adaptive load balancing for MPI programs. In *International Conference on Computational Science*, pp. 108–117. LNCS # 2074.

Breyer, R. and S. Riley (1999). *Switched, Fast, and Gigabit Ethernet*. Macmillan, San Francisco, CA.

Burns, G., R. Daoud, and J. Vaigl (1994). LAM: An open cluster environment for MPI. In *Supercomputing Symposium*, pp. 379–386.

Byna, S., W. Gropp, X. Sun, and R. Thakur (2003). Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *$5^{\underline{th}}$ IEEE International Conference on Cluster Computing (CLUSTER'03)*, pp. 412–419.

Carey, F., W. B. Richardson, C. S. Reed, and B. Mulvaney (1996). *Circuit, Devices, and Process Simulation*. Wiley, Chichester.

Chakravorty, S. (2002). Implementation of parallel mesh partition and ghost generation for the finite element mesh framework. Master's thesis, University of Illinois.

Chan, T. F. and R. S. Tuminaro (1987). Design and implementation of parallel multigrid algorithms. In S. F. McCormick (Ed.), *Third Copper Mountain Conference on Multigrid Methods*, pp. 101–115.

Chuang, S. L. (1995). *Physics of Optoelectronic Devices*. Wiley Interscience.

Darling, J. P. and I. D. Mayergoyz (1990). Parallel algorithm for the solution of nonlinear Poisson equation of semiconductor device theory and its implementation on the MPP. *Journal of Parallel and Distributed Computing 8*(2), 161–168.

Frederickson, P. O. and O. A. McBryan (1988). Parallel superconvergent multigrid. In S. F. McCormick (Ed.), *Multigrid Methods: Theory, Applications, and Supercomputing*, pp. 195–210. Springer, Berlin. Lecture Notes in Pure and Applied Mathematics,# 110.

Griebel, M. and G. Zumbusch (1998). Parallel multigrid in an adaptive PDE solver based on hashing. In E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg (Eds.), *Parallel Computing Conference, ParCo '97*, pp. 589–599.

Gropp, W., S. Huss-Lederman, A. Lumsdains, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir (1998). *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions* (2nd ed.). MIT, Cambridge, MA.

Gropp, W. and E. Lusk (1997). A high-performance MPI implementation on a shared-mmeory vector supercomputer. *Parallel Computing 22*, 1513–1526.

Gropp, W., E. Lusk, N. Doss, and A. Skjellum (1996, September). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*(6), 789–828.

Gropp, W., E. Lusk, and A. Skjellum (1994). *Using MPI Portable Parallel Programming with the Message-Passing Interface*. MIT, Cambridge, MA.

Gummel, H. K. (1964). A self consistent iterative scheme for one-dimensional steady-state transistor calculations. *IEEE Transactions on Electron Devices 11*, 455–456.

Heise, B. and M. Jung (1997). Parallel solvers for nonlinear elliptic problems on domain decomposition ideas. *Parallel Computing 22*(11), 1527–1544.

Huang, C., O. Lawlor, and L. V. Kalé (2003). Adaptive MPI. In *16$\underline{th}$ International Workshop on Languages and Compilers for Parallel Computing*. Paper # 03-07.

Kalé, L. and S. Krishnan (1993). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, pp. 91–108.

Kalé, L. V. (2002). The virtualization model of parallel programming : Runtime optimizations and the state of art.

Kale, L. V. and S. Krishnan (1996). Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu (Eds.), *Parallel Programming using C++*, pp. 175–213. MIT Press, Cambridge, MA.

Kalé, L. V., S. Kumar, G. Zheng, and C. W. Lee (2003). Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, Int. Conf. on Computational Science (ICCS)*.

Karypis, G. and V. Kumar (1998). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing 48*(1), 96–129.

Korman, C. E. and I. D. Mayergoyz (1990). A globally convergent algorithm for the solution of the steady-state semiconductor device equations. *Journal of Applied Physics 68*(3), 1324–1334.

Leach, R. (1994). *Advanced Topics in UNIX: Processes, Files, and Systems*. Wiley, New York.

Lenox, C., H. Nie, P. Yan, G. Kinsey, A. L. Holmes, B. G. Streetman, and J. C. Campbell (1999). Resonant-cavity InGaAs/InAlAs avalanche photodiodes with gain-bandwidth of 290 GHz. *IEEE Photonic Technology Letters 11*(9), 1162–1164.

Lin, P. and C. Wu (1987). A new approach to analytically solving the two-dimensional Poisson's equa-

tion and its applications in short-channel MOSFET modeling. *IEEE Transactions on Electron Devices 34*(9), 1947–1956.

Mahseyekhi, H. R. (1999). *Theoretical and Experimental Studies of Back-Gated Metal-Semiconductor-Metal Photodetectors*. Ph. D. thesis, University of Essex.

Matsushima, Y., S. Akiba, K. Sakai, Y. Kushiro, Y. Noda, and K. Utaka (1982). High-speed response InGaAs/InP heterostructure avalanache photodidode with InGaAsP buffer layers. *IEE Electronic Letters 18*(22), 945–946.

Mitchell, W. F. (2001). Adaptive grid refinement and multigrid on cluster computer. In *IEEE 15$\frac{th}{}$ International Parallel and Distributed Processing Symposium*.

Molenaar, J. (1993). *Multigrid Methods for Semiconductor Device Simulation*. Centre for Mathematics and Computer Science, Amsterdam. CWI Tract, Volume 100.

Nevin, N. J. (1996). The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio.

Nupairoj, N. and L. Ni (1996). Performance evaluation of some MPI implementations on workstation clusters. Technical report, Dept. of Computer Science, Michigan State University.

Open Systems Laboratory, Indiana University (2004). *The LAM/MPI User Guide v. 7.1.1*. Open Systems Laboratory, Indiana University.

Parallel Programming Laboratory, University of Illinois, Urbana (2001). *The Charm++ Programming Manual Version 5.4*. Parallel Programming Laboratory, University of Illinois, Urbana.

Parks, J. W., A. W. Smith, K. F. Brenman, and L. E. Tarof (1996). Theoretical study of device sensitivity and gain saturation of separate absorption, grading, charge, and multiplication InP/InGaAs avalanche photodiodes. *IEEE Electronic Devices 43*(12), 2113–2120.

Ramkumar, B. and L. V. Kalé (1994a). Machine independent AND and OR parallel execution of logic programs: Part I-The binding environment. *IEEE Transactions on Parallel and Distributed Systems 5*(2), 170–180.

Ramkumar, B. and L. V. Kalé (1994b). Machine independent AND and OR parallel execution of logic programs: Part II-Compiled execution. *IEEE Transactions on Parallel and Distributed Systems 5*(2), 181–192.

Rarity, J. G., T. E. Wall, K. D. Ridley, P. C. M. Owens, and P. R. Tapster (2000). Single-photon counting for the 1300-1600-nm range by use of Peltier-cooled and passively quenched InGaAs avalanche

photodiodes. *Applied Optics 39*(36), 6746–6753.

Ridge, D., D. Becker, P. Merkey, and T. Sterling (1997). Beowulf: Harnessing the power of parallelism in a pile-of-PCs. In *IEEE Aerospace*, Volume 2, pp. 79–91.

S. Markus, S. B. K., K. Patazapoulos, E. N. H. A. L. Ocken, P. Wu, S. Weerawarana, and D. Maharry (1996). Performance evaluation of MPI implementations and MPI based parallel ELLPACK solvers. In *$2^{\underline{nd}}$ MPI Developers Coneference*, pp. 162–168.

Snir, M., S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra (1998). *MPI—The Complete Reference: Volume 1, The MPI Core* (2nd ed.). MIT, Cambridge, MA.

Squyres, J. M. and A. Lumsdaine (2003). A component architecture for LAM/MPI. In *$10^{\underline{th}}$ PVM/MPI Users' Group Meeting*, pp. 379–387. LNCS # 2840.

Srinivasan, S. (1995, August). XDR: External data representation standard. *RFC 1832*.

Sterling, T. (Ed.) (2002a). *Beowulf Cluster Computing with Linux*. MIT, Cambridge, MA.

Sterling, T. (2002b). Network hardware. In *Beowulf Cluster Computing with Linux*, pp. 113–130. MIT, Cambridge, MA.

Sterling, T. (2002c). Node hardware. In *Beowulf Cluster Computing with Linux*, pp. 31–60. MIT, Cambridge, MA.

Tagushi, K., T. Torikai, and Y. Sugimoto (1988). Planar-structure InP/InGaAsP/InGaAs avalanche photodiodes with preferential lateral extended guard ring for 1.0-1.6μm wavelength optical communication use. *Journal of Lightwave Technology 6*(11), 1643–1655.

Vadali, R. V., Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna (2004). Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Computational Chemistry 2004*(16), 2006–2022.

Vickers, A., M. A. Hassan, H. R. Mashakekhi, A. Griguoli, and M. Hopkinson (1996). Study of a backgated metal-semiconductor-metal photodetector. *Applied Physics Letters 68*(6), 815–817.

Wada, A. and H. Hasegawa (1999). *InP Materials and Devices*. Wiley Interscience.

Layer:	Inp1.	Mult.	Charge	Hole	Absorp.	Doping	Inp2.
Range	1–400	401–800	801–1100	1101–1200	1201–1320	1321–1420	1421–1820

Table 1: Distribution of the linear strip rows from 1–1820 according to semi-conductor layer.

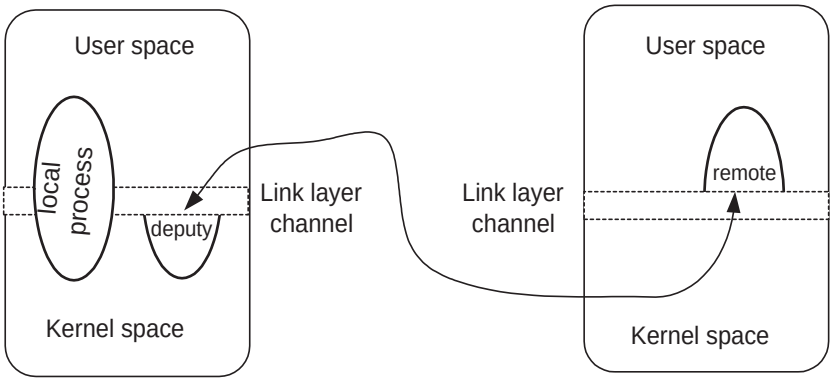

Figure 1: Local, and migrated MOSIX process, showing the data link layer channel for deputy to remote communication.

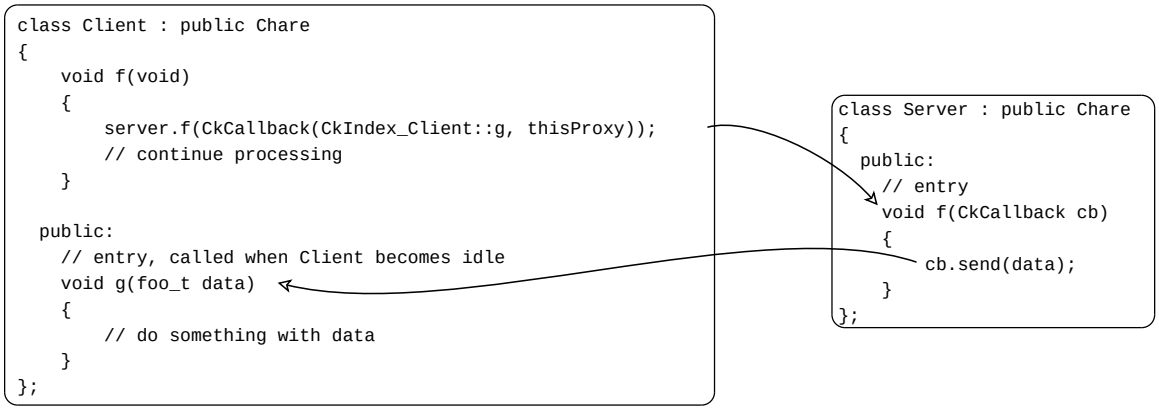

Figure 2: Continuation-passing in Charm++

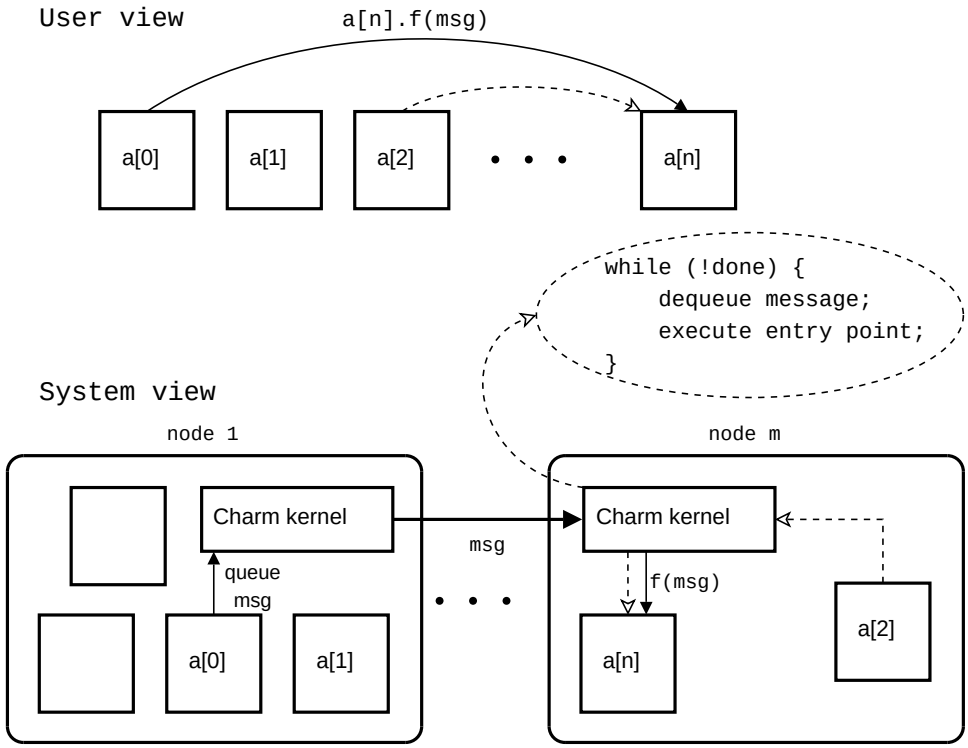

Figure 3: Charm++ system-user interaction

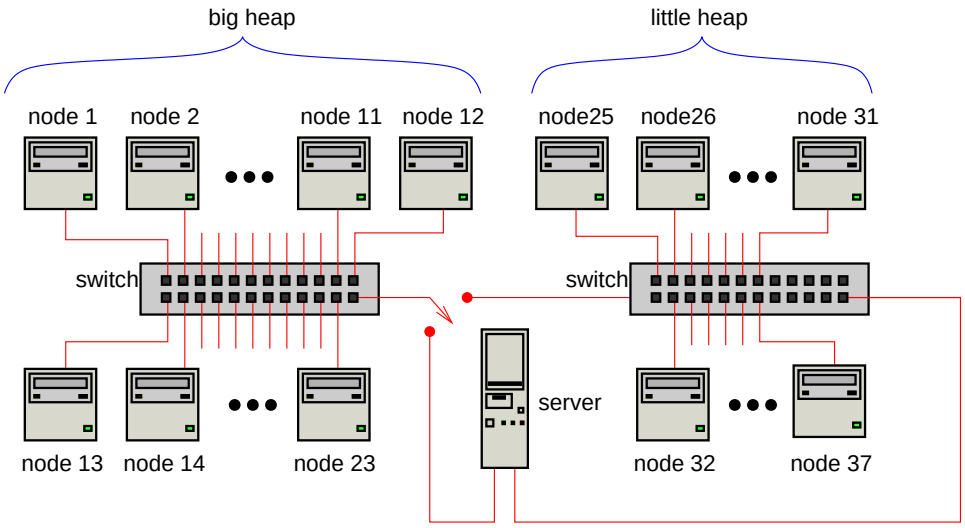

Figure 4: Schematic diagram of the Linux cluster, showing system partitioning and fileserver

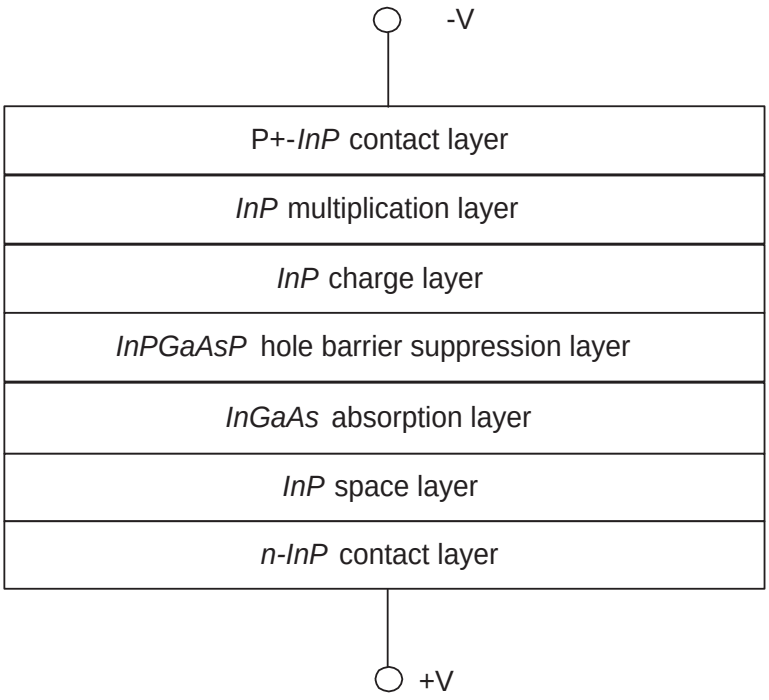

Figure 5: Schematic cross-section through an *InGaAs/InP* SAM APD (layer widths are not representative)

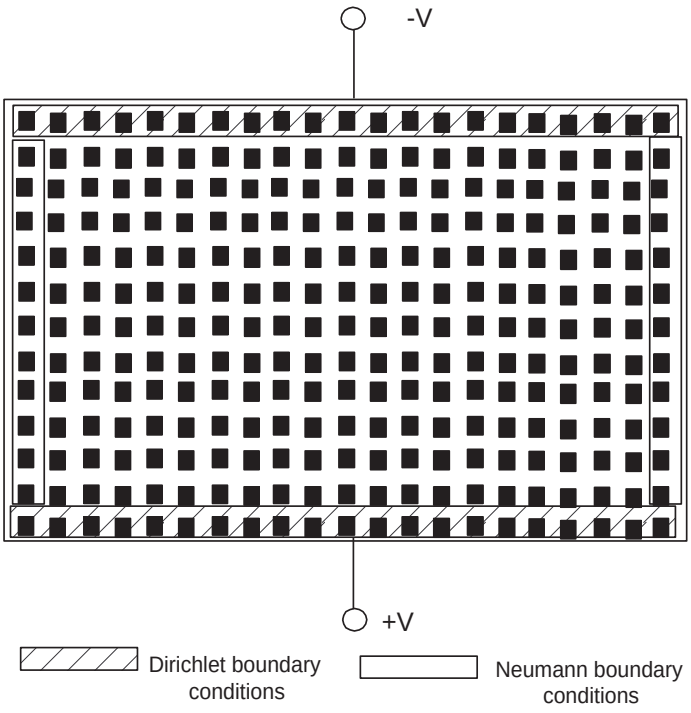

Figure 6: Device cross-section with simulation grid superimposed, showing treatment of boundary conditions

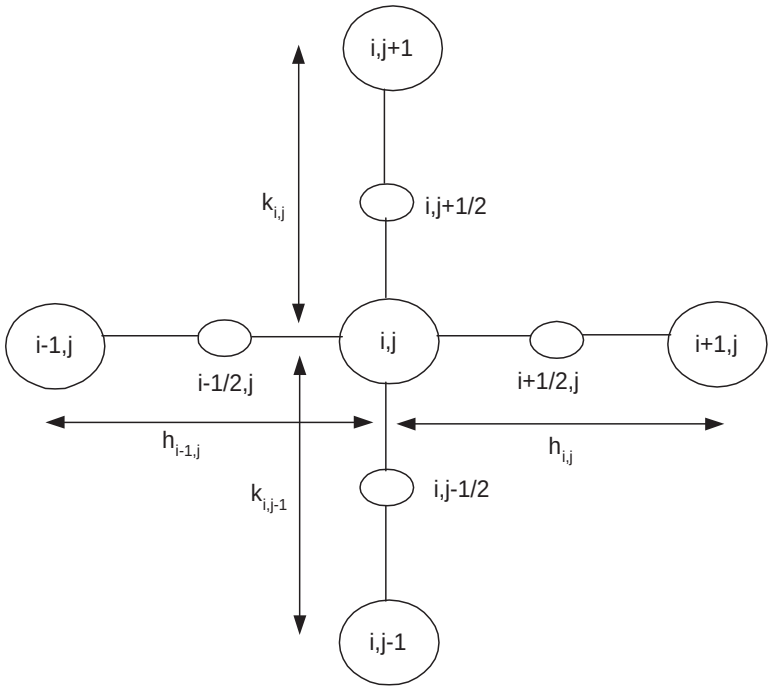

Figure 7: Calculation of mesh point (i, j) for $n = 7$

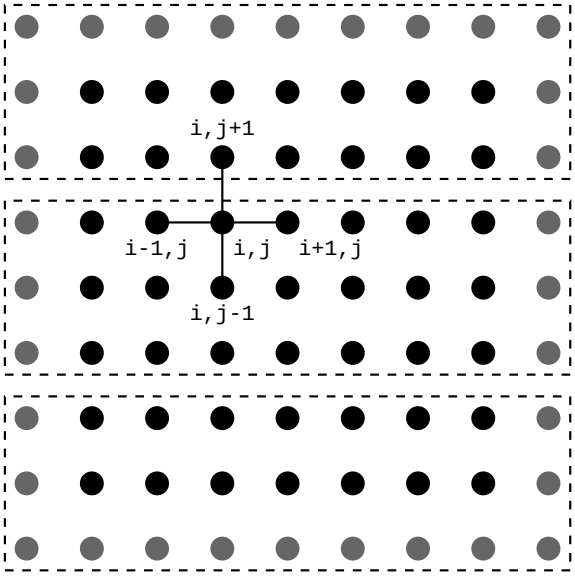

Figure 8: 1-D decomposition of the domain into three slabs

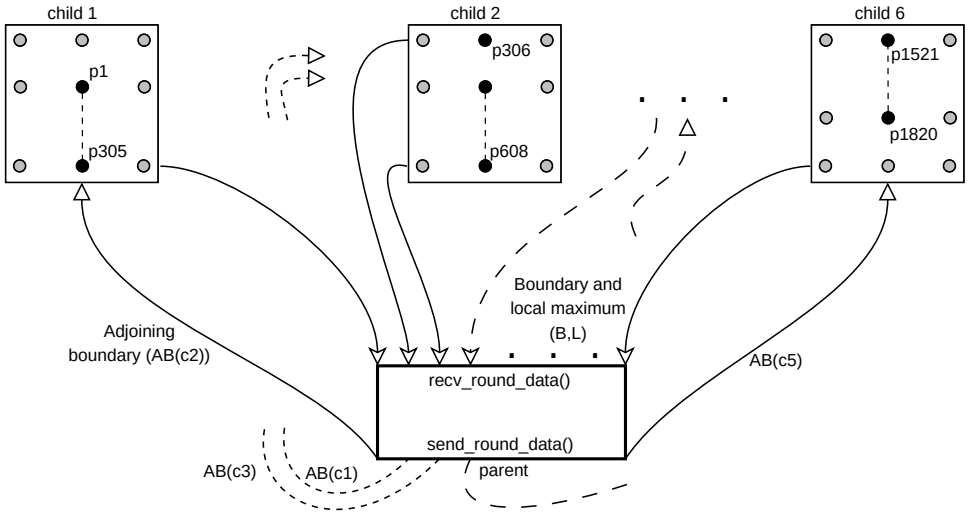

Figure 9: Grid division for six processes in the centralized MOSIX scheme

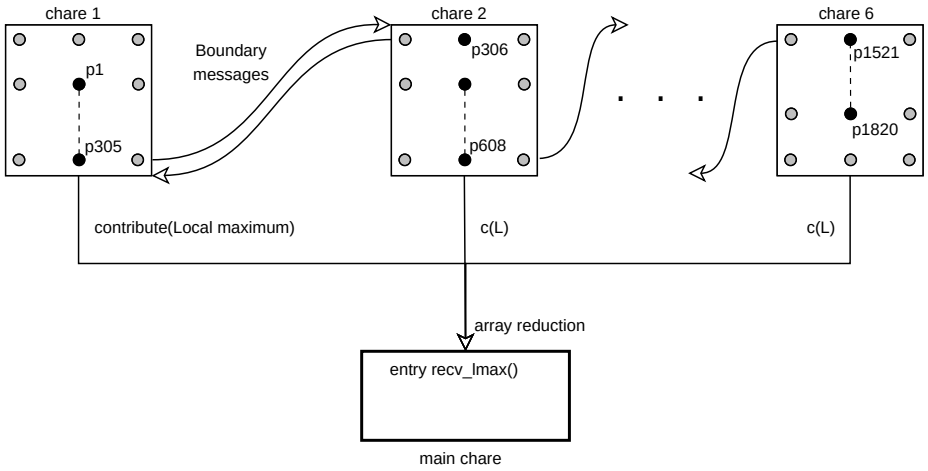

Figure 10: Boundary exchanges and reductions between Charm++ chares

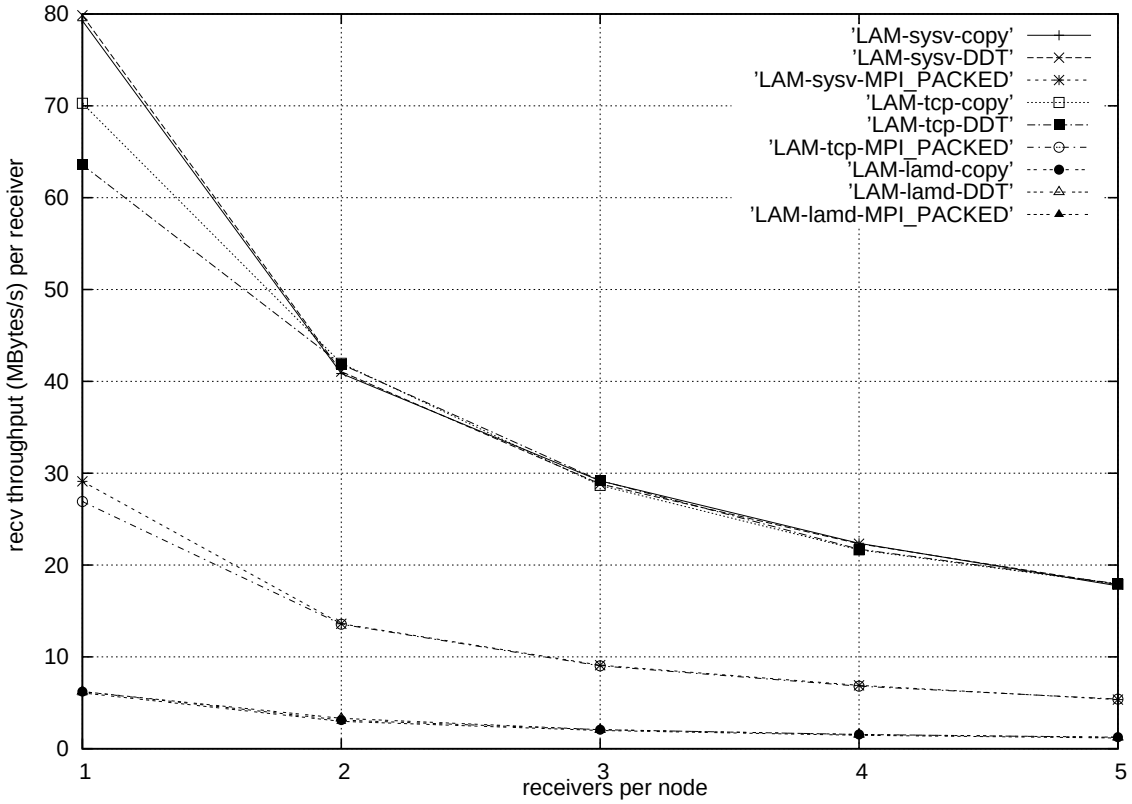

Figure 11: Comparison between application-level packing (MPI_PACKED) and derived datatype (DDT) for copy-in-buffer-and-send performance using LAM with either `sysv`, `tcp`, or `lamd` communication modes.

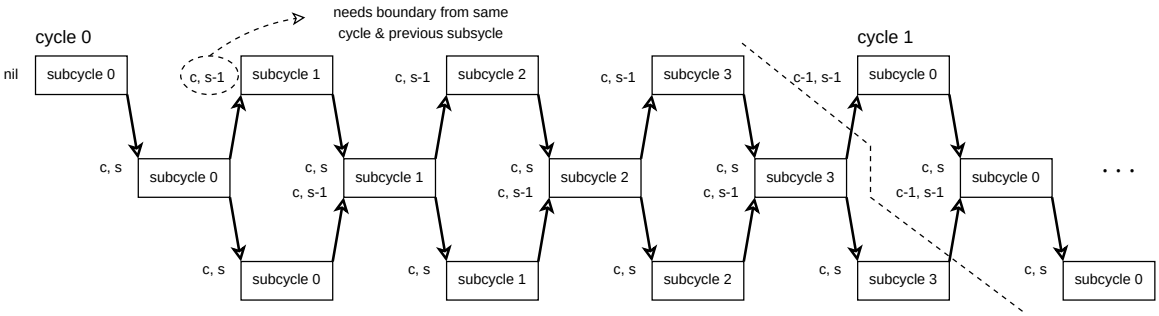

Figure 12: Data dependencies between sub-cycles and resulting synchronizations within a sub-cycle observant parallel version

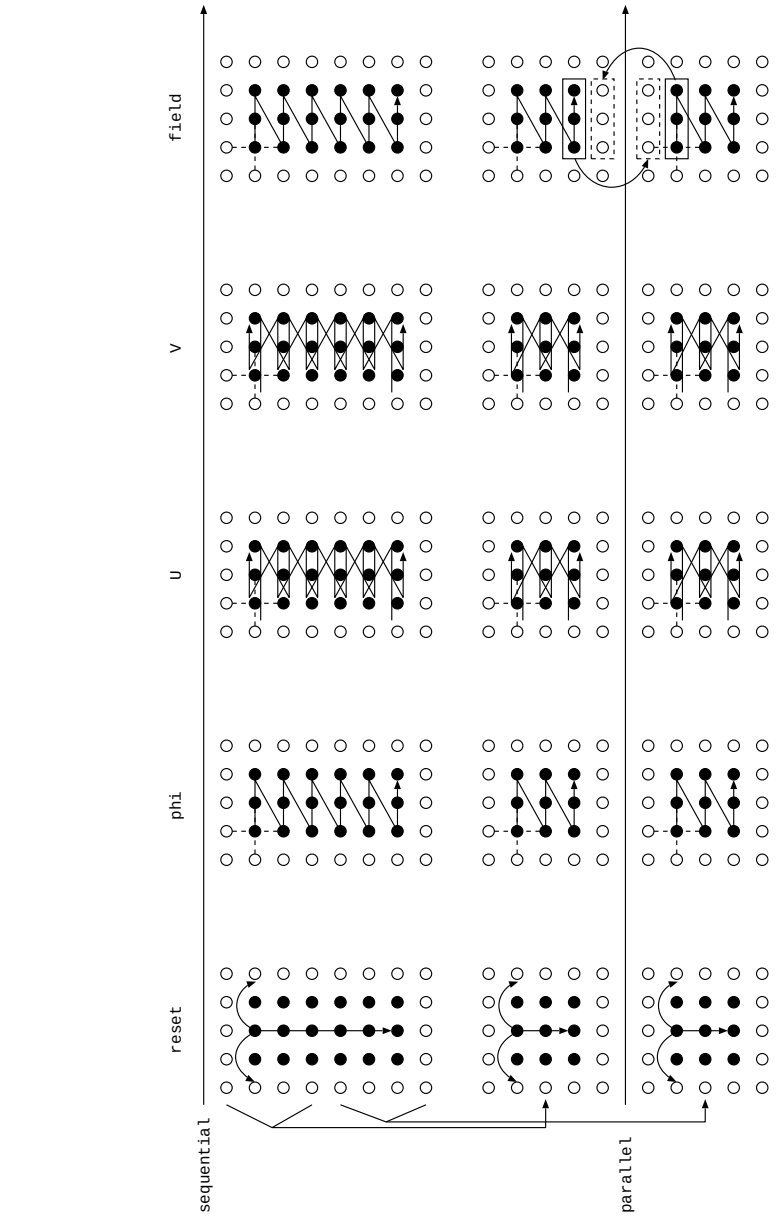

Figure 13: Detailed view of sub-cycles and the implemented parallel deomposition

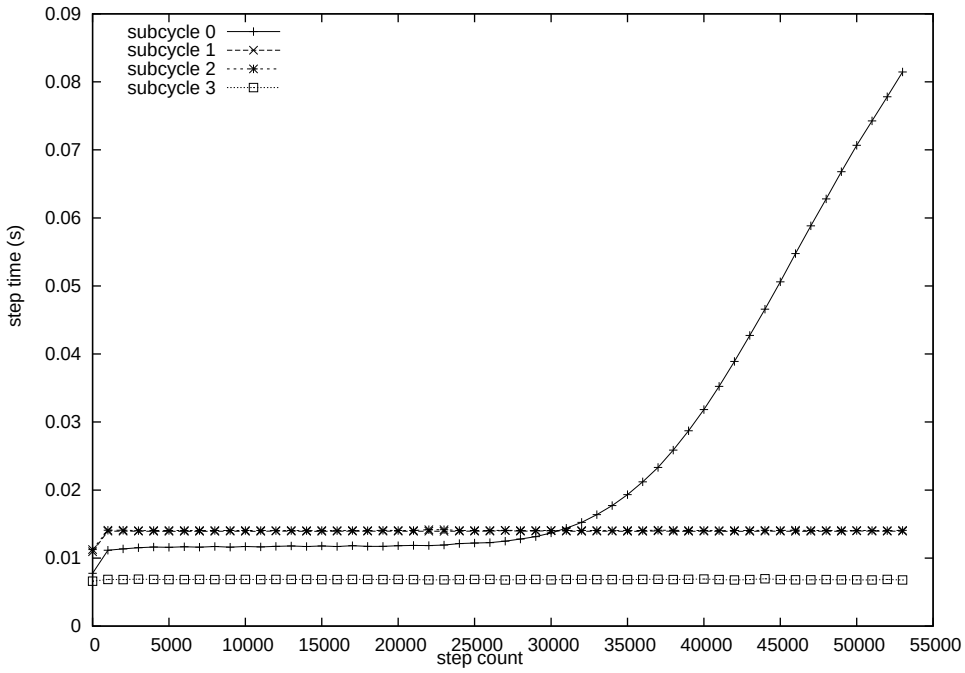

Figure 14: Iteration timings in the absorption layer for the first sub-cycle.

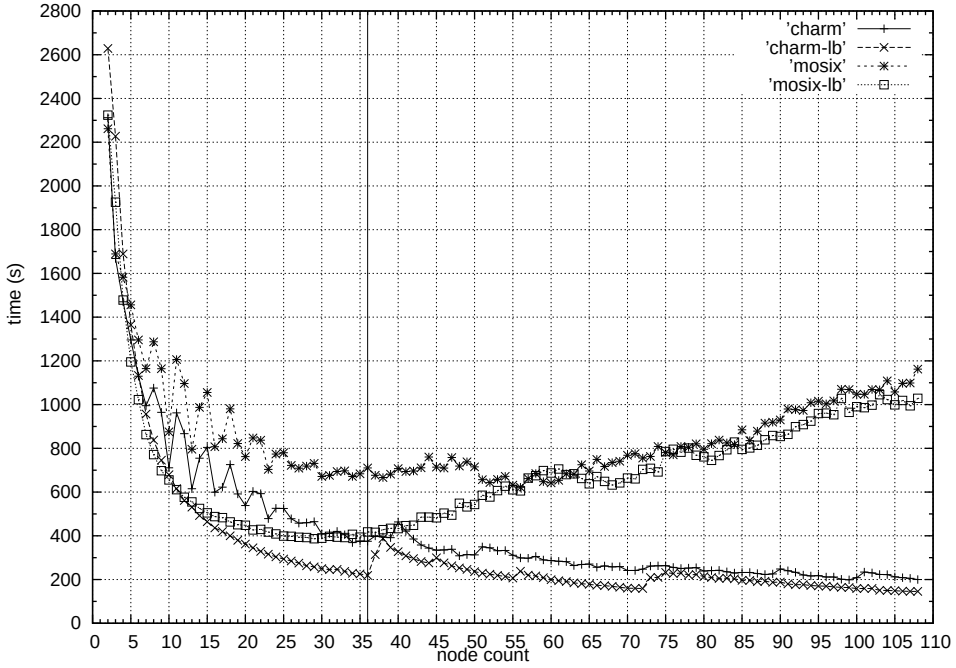

Figure 15: DDM simulation run times, showing application-level load-balanced and non-balanced performance

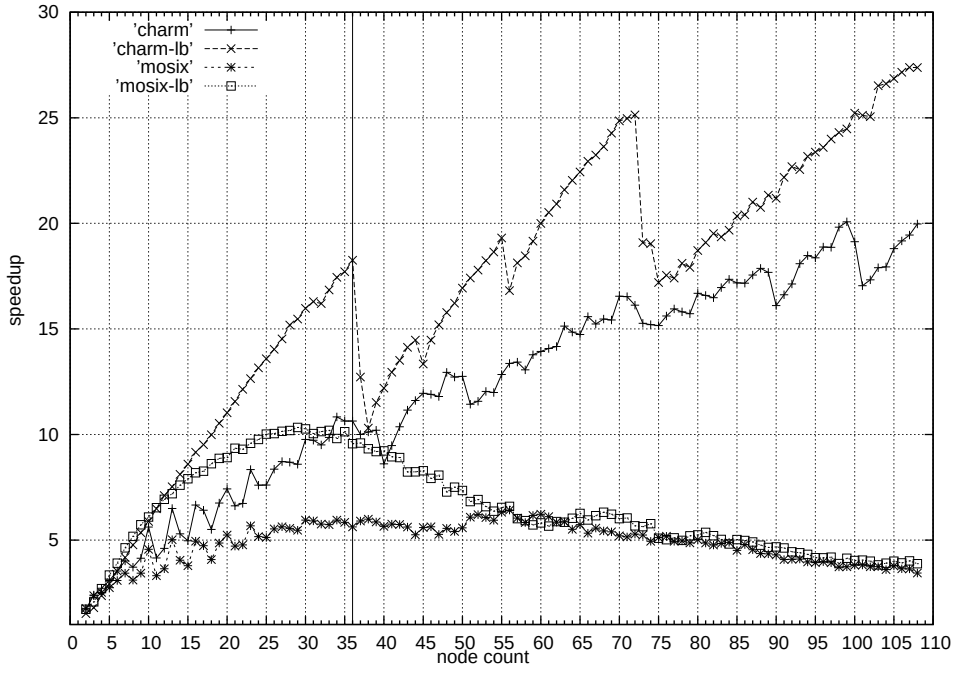

Figure 16: DDM simulation speedup curves, showing application-level load-balanced and non-balanced performance

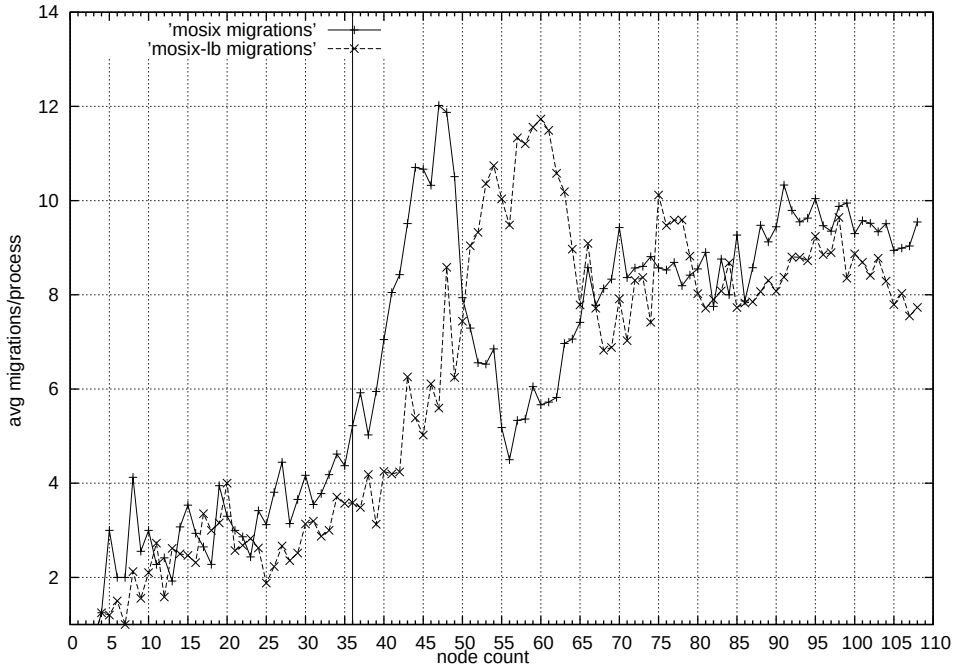

Figure 17: Per node MOSIX migration numbers

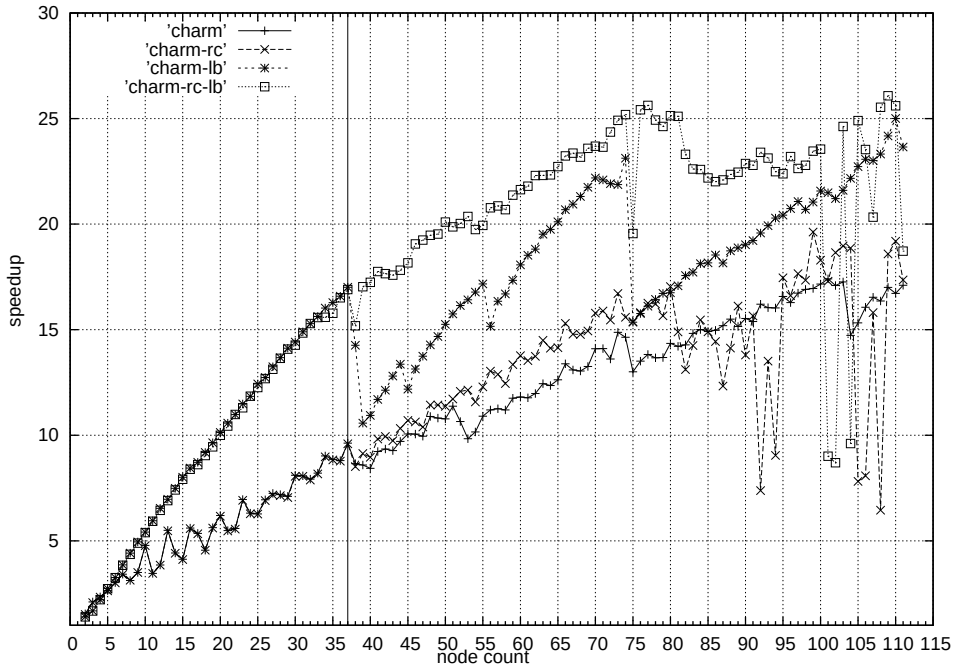

Figure 18: DDM simulation speedup curves, showing the effect of Charm++ system-level load-balancing

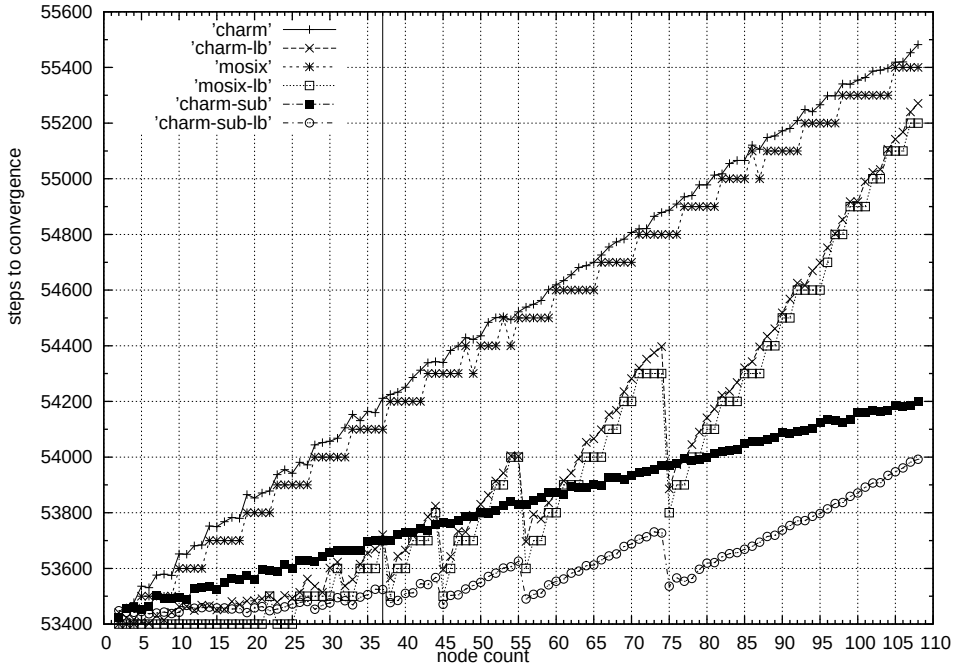

Figure 19: Differing convergence rates according to parallel decomposition, node, and load-balancing

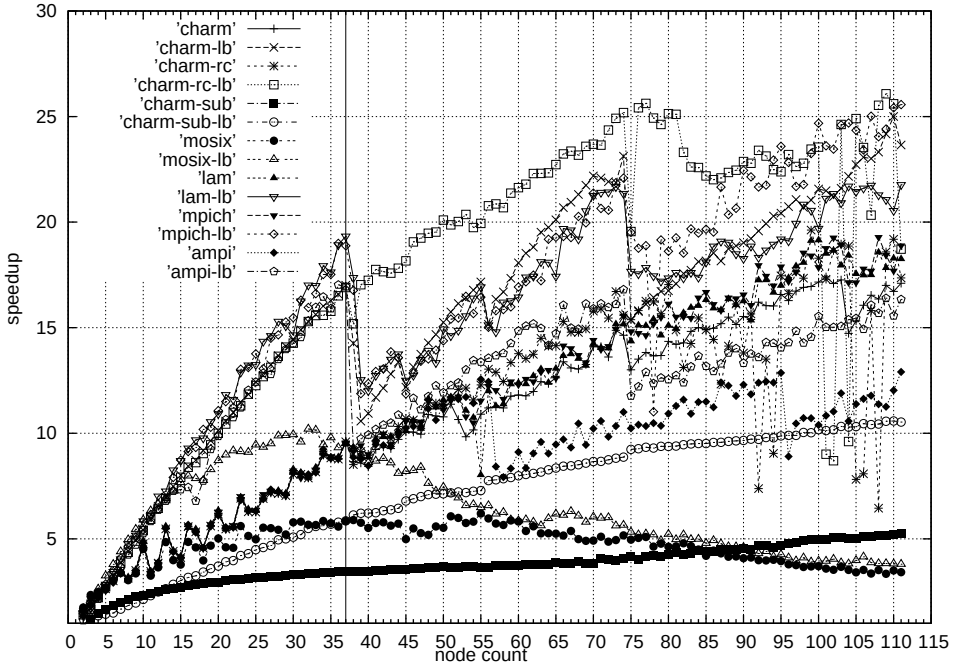

Figure 20: Speed-ups with sub-cycles introduced

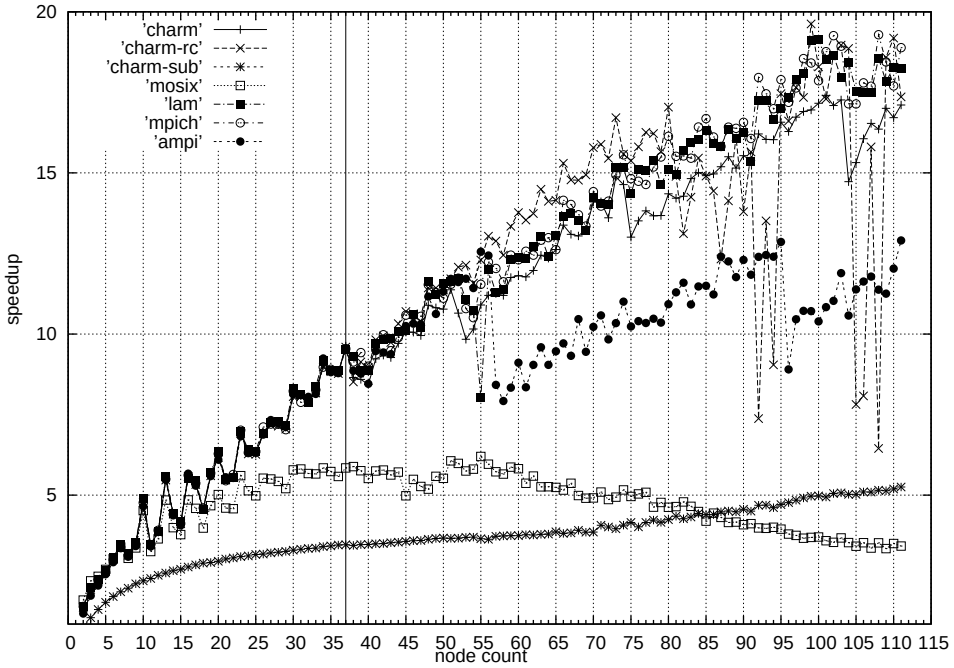

Figure 21: Speed-ups for load-balanced and non-load balanced systems

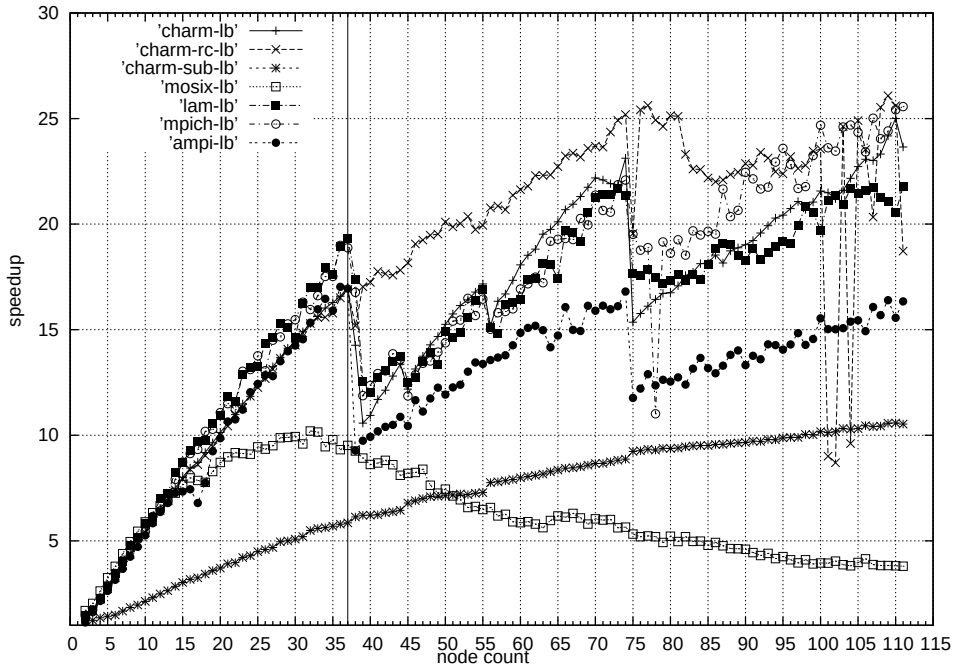

Figure 22: Speed-ups with application-level load-balancing for all systems

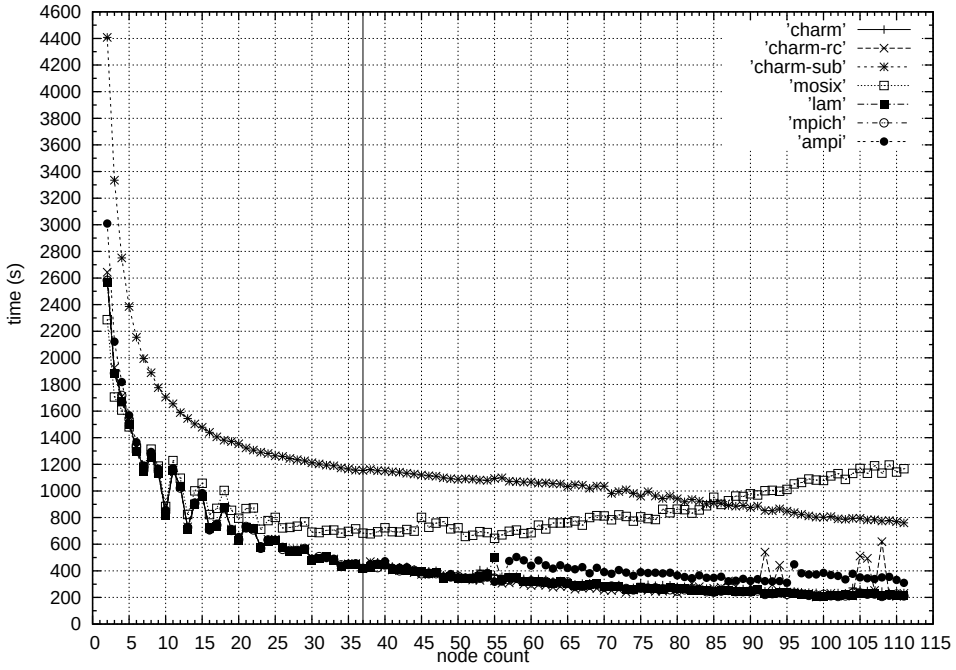

Figure 23: Timings without application-level load-balancing for all systems

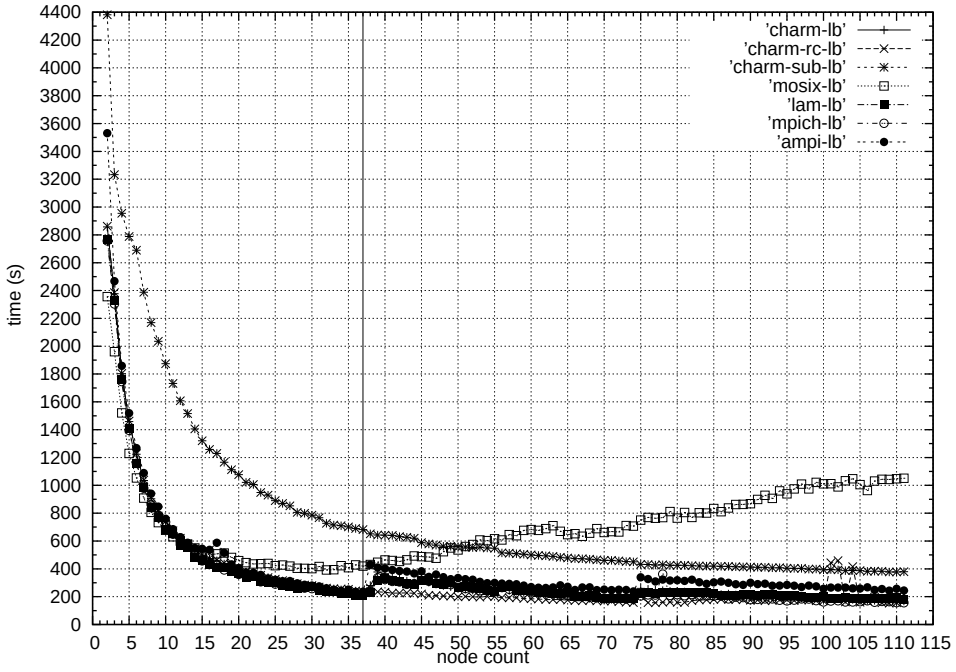

Figure 24: Timings with application-level load-balancing for all systems

Biographies

Stylianos Bounanos is pursuing PhD studies at the University of Essex, UK. He holds an MSc in Computer and Information Networks from the same institution. Stylios' research interests are cluster computers, parallel and mobile computing languages, and performance benchmarking.

Martin Fleury is a Senior Lecturer at the University of Essex, UK, where he was also awarded a PhD in Parallel Image Processing. His first degree was from Oxford University, and he holds an MSc in Astrophysics from the University of London. He is the co-author of a book on parallel computing for embedded systems. He has authored forty journal papers in the last ten years on parallel image and vision processing, performance prediction, real-time systems, reconfigurable computing, software engineering, document compression, and video networking.

Sebastien Nicholas is pursuing PhD studies at the University of Essex, UK. Sebastien's research interests are in the design and modelling of novel optoelectronic devices.

Anthony Vickers is a Reader at the University of Essex, UK, where he is also Head of the High-Speed Optoelectronics Lab. and Head of Department of the Electronic Systems Engineering Dept. Anthony has published widely in the physics literature including studies on the fundamental properties of new materials, and the development of novel photonic devices and semiconductor laser systems. His current research interests are in optoelectronic devices; high-speed lasers and photodetectors; ultrafast measurement techniques; THz devices; and advanced RF measurement techniques.