Journal of Computational Physics 229 (2010) 6804–6818

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Towards large-scale multi-socket, multicore parallel simulations: Performance of an MPI-only semiconductor device simulator ☆

Paul T. Lin *, John N. Shadid

Sandia National Laboratories, P.O. Box 5800, MS 0316, Albuquerque, NM 87185-0316, USA

ARTICLE INFO

Article history:
Received 19 October 2009
Received in revised form 12 May 2010
Accepted 20 May 2010
Available online 31 May 2010

Keywords:
Multicore
Multicore efficiency
MPI-only
Multilevel preconditioners
Newton–Krylov
Drift–diffusion
Semiconductor devices

ABSTRACT

This preliminary study considers the scaling and performance of a finite element (FE) semiconductor device simulator on a set of multi-socket, multicore architectures with nonuniform memory access (NUMA) compute nodes. These multicore architectures include two linux clusters with multicore processors: a quad-socket, quad-core AMD Opteron platform and a dual-socket, quad-core Intel Xeon Nehalem platform; and a dual-socket, six-core AMD Opteron workstation. These platforms have complex memory hierarchies that include local core-based cache, local socket-based memory, access to memory on the same mainboard from another socket, and then memory across network links to different nodes. The specific semiconductor device simulator used in this study employs a fully-coupled Newton–Krylov solver with domain decomposition and multilevel preconditioners. Scaling results presented include a large-scale problem of 100+ million unknowns on 4096 cores and a comparison with the Cray XT3/4 Red Storm capability platform. Although the MPI-only device simulator employed for this work can take advantage of all the cores of quad-core and six-core CPUs, the efficiency of the linear system solve is decreasing with increased core count and eventually a different programming paradigm will be needed.

1. Introduction

A current pathway to increase the peak performance of the next generation supercomputers is to utilize large numbers of powerful compute nodes comprised of multiple sockets with multiple cores on each socket. This is demonstrated by having nine out of the top ten supercomputers on the June 2009 Top 500 list (www.top500.org) using homogeneous multicore architectures (eight of them with quad-core processors). It has been forecasted that by around 2020 compute nodes could have up to a thousand cores [1]. This is a significant cause of concern for numerical methods and scientific application software developers who have typically had the performance of their codes limited by the bandwidth from the compute processor to local memory. Although there appears to be a belief that the single-level MPI-only programming model should be sufficient for the next 3–5 years, there appears to be no general consensus on the most effective programming model for larger core count multicore nodes [2].

An example of this trend to multicore nodes in our own research institution is the arrival of the Tri-Laboratory Linux Capacity Clusters (TLCC) at Sandia National Labs (SNL) in the fall of 2008. The SNL TLCC platforms consist of 272 compute nodes (4352 cores) connected by a 4x DDR InfiniBand switched network. Each compute node has four CPUs, each being a

☆ Partially supported by the DOE NNSA ASC program and the DOE Office of Science ASCR Applied Math Research program and ASCR IAA program at Sandia National Laboratory. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

* Corresponding author.
E-mail address: ptlin@sandia.gov (P.T. Lin).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.05.023

2.2 GHz AMD Opteron (Barcelona) quad-core processor, for a total of 16 cores per node, and 8 GB RAM local to each processor (32 GB RAM per compute node). A second example is the Red Sky machine, which is currently being built at SNL. An initial, much smaller, development platform has approximately 100 compute nodes connected by a 4x QDR InfiniBand network. Each compute node has two 2.9 GHz Intel Xeon X5570 (Nehalem-EP) quad-core processors, and 6 GB RAM local to each processor (12 GB RAM per compute node). Both AMD and Intel continue to increase the core count of their CPUs; AMD released the six-core Opteron Istanbul in June 2009.

This study is intended as an initial exploration of the performance of a memory bandwidth limited application, a finite element semiconductor device simulation code [3], on a few currently available homogeneous multicore architectures. The limited scope of this study is to examine the performance of a state-of-the-practice C++ object oriented transport reaction code on the TLCC machine, the Nehalem cluster, and an AMD Opteron Istanbul workstation. The device simulation code was originally developed to model semiconductor devices via the drift–diffusion equations, but it can also model magnetohydrodynamics (MHD) [4]. Currently this application code uses a single-level MPI-only programming paradigm. This study considers the performance of the device simulation code for solving a steady-state drift–diffusion problem. Steady-state problems have been selected since these cases, in general, require the most iterations of the underlying preconditioned Krylov linear solver kernel that stresses the memory bandwidth capability of a particular node architecture. The preconditioners that are considered include a reasonably common additive Schwarz domain decomposition preconditioner [5,6] and an algebraic multilevel preconditioner [7–9] that represents an advanced fully-coupled preconditioner for systems of partial differential equations (PDE). Device simulations will be run on these platforms to examine how efficiently the cores of these multicore processors can be utilized. First the efficiency of using the cores for a single compute node of TLCC and the Nehalem cluster and an Istanbul workstation will be examined. Then the effect of both the network and compute nodes for TLCC will be examined. As we are interested in how large-scale simulations with complex physics codes will perform on these multicore platforms, the performance of the TLCC machine will be compared with a Cray XT3/4 for a large-scale simulation. The Cray XT3/4 (Red Storm) [10] is intended for large-scale simulations and designed with a scalable custom high performance network that maximizes performance and scalability.

2. A brief description of the FE semiconductor device simulation code

2.1. Drift–diffusion model and the stabilized finite element discretization

A large number of advanced scientific and technology systems require the detailed analysis of electric potential and electron and hole concentrations in semiconductors. A base computational method for semiconductor devices is the drift–diffusion model. This coupled system of nonlinear partial differential equations (PDEs) relate the electric potential to the electron and hole concentrations in these devices. The equations governing the transport of charge carriers within a semiconductor device can be approximated using the standard drift–diffusion equations, in residual form, given by Kramer et al. [11] and Sze [12]:

$$R_\psi = -\lambda^2 \nabla \cdot (\epsilon_r \nabla \psi) - (p - n + C) = 0, \tag{1}$$

$$R_n = \frac{\partial n}{\partial t} + \nabla \cdot (\mu_n n \nabla \psi) - \nabla \cdot (D_n \nabla n) + G = 0, \tag{2}$$

$$R_p = \frac{\partial p}{\partial t} - \nabla \cdot (\mu_p p \nabla \psi) - \nabla \cdot (D_p \nabla p) + G = 0. \tag{3}$$

The scaled unknowns are: ψ, the electrostatic potential, n, the electron concentration (number of electrons per volume), and p, the hole concentration (number of holes per volume). The scaled parameters in this system are μ_n and μ_p, the electron and hole mobilities, respectively, D_n and D_p, the electron and hole diffusion coefficients, respectively, C the doping profile, G the generation/recombination source term, ϵ_r the relative permittivity, and λ is the minimal Debye length of the device [3].

In this study these governing PDEs are discretized by a stabilized finite element method. This discretization is associated with a simple extension of an SUPG-type approach to the drift–diffusion system [3]. The stabilized FE weak form is

$$F_\psi = \int_\Omega R_\psi \phi d\Omega = 0, \tag{4}$$

$$F_n = \int_\Omega R_n \phi d\Omega - \sum_e \int_{\Omega_e} \tau_n \left[\mu_n \mathbf{E} \cdot \nabla \phi\right] R_n d\Omega = 0, \tag{5}$$

$$F_p = \int_\Omega R_p \phi d\Omega + \sum_e \int_{\Omega_e} \tau_p \left[\mu_p \mathbf{E} \cdot \nabla \phi\right] R_p d\Omega = 0, \tag{6}$$

where

$$\mathbf{E} = -\nabla \psi.$$

The stabilized FE strategy is used to control instability in the Galerkin FE formulation for the drift–diffusion system. The terms in the weak form include the standard Galerkin term (first term in (4)–(6)) and an SUPG-type term (second term in (5) and (6)). This methodology is based on a variation of the streamline upwind Petrov–Galerkin (SUPG) type FE formulations

of Hughes et al. [13,14] and Shakib [15] for convection–diffusion systems and has similarities to the flux upwind Petrov–Galerkin method for the drift–diffusion equations of Carey and coworkers [16,17]. The stabilized FE method allows solution of convection–diffusion type systems by decreasing numerical oscillations due to convection effects. The specific definitions of the stabilization parameters (the τ's) can be found in [3].

2.2. Overview of the schwarz domain decomposition preconditioned Newton–Krylov methods

Discretization of the governing drift–diffusion equations produces a large sparse, strongly coupled nonlinear system. Currently a promising approach for the solution of these types of systems is the use of fully-coupled nonlinear solution methods based on Newton–Krylov techniques (see e.g. [4,18–20]). The robustness of these methods, however, comes at a high price, as these techniques generate very-large sparse systems of equations that must be solved at each nonlinear iteration. Therefore, these techniques place a heavy burden on the underlying linear solvers. Preconditioned Krylov iterative methods are among the most robust and fastest iterative linear solvers over a wide variety of applications (see e.g. [18,21,22], and references contained therein). For the parallel solution of very-large scale sparse linear systems, that can now range from 10^7 to 10^9 unknowns, the preconditioner is critical to the scalability, efficiency, and robustness of the linear solver.

A preconditioner defines a nonsingular matrix $\mathbf{M}$ that approximates the original matrix $\mathbf{A}$ and is easily inverted. The goal is to reduce the condition number of the preconditioned matrix, $\mathbf{A}\,\mathbf{M}^{-1}$ for right preconditioning, which effectively improves Krylov solver convergence. Ideally, this condition number should be independent of mesh spacing to guarantee that convergence does not deteriorate as the mesh is refined. Finally, Krylov methods do not require $\mathbf{A}$ or $\mathbf{M}^{-1}$ explicitly. Instead, procedures for applying $\mathbf{A}$ to a vector and efficient solution methods for $\mathbf{Mz} = r$ must be provided.

This paper considers Schwarz domain decomposition preconditioners as a common standard technique for parallel computation where the basic idea is to decompose the computational domain Ω into overlapping subdomains Ω_i and then assign each subdomain to a different processor [5,6]. One application of the algorithm consists of solving on subdomains and then combining these local solutions to construct a global approximation throughout Ω. In the minimal overlap case, the algebraic Schwarz method corresponds to block Jacobi where each block contains all degrees of freedom (DOFs) residing within a given subdomain. Convergence is typically improved by introducing overlap which can be done recursively [6,23,24]. A partition of the domain is obtained using the graph partitioning tool Chaco [25], producing a nonoverlapping subdomain decomposition. For this paper, Chaco used multilevel methods with Kernighan–Lin improvement. Each subdomain is then assigned to a single MPI task (one MPI task per processor core). We typically employ incomplete factorization (e.g. $ILU(k)$) techniques to approximate the solution of the local subdomain problems and avoid the large cost of direct factorization [21,26]. Typically, k is chosen so that the incomplete factorization requires a bit more than twice as much storage as the original matrix. We note that the one-level preconditioner is "black-box" in that the overlapping subdomain matrices are constructed completely algebraically.

A drawback of the one-level Schwarz method described above is its locality. Intuitively this method is slow to transfer information from disjoint subdomains in the partition since an application of the algorithm transfers information between neighboring subdomains only. This implies that many repeated applications are required to combine information across the entire domain. Thus, as the number of subdomains increases, the convergence rate deteriorates for standard elliptic problems due to this lack of global coupling [6]. The convergence rate also deteriorates as the number of unknowns per subdomain increases when $ILU(k)$ is used for a subdomain solver. To remedy this, coarse levels can be introduced to approximate global coupling [6,23,27] and produce a multilevel preconditioner. The use of a coarse mesh to accelerate the convergence of a one-level Schwarz preconditioner is similar to multigrid methods that use a sequence of coarser meshes [28,29].

Multigrid methods are among the most efficient techniques for solving large linear systems [28,29]. Their rapid convergence relies on an interplay between smoothing and coarse correction. Smoothing refers to an iterative procedure focused on reducing oscillatory error components. The coarse correction refers to the formation and projection of a residual equation onto a coarse space. The central premise is that once error has been smoothed, it can be represented at a coarser resolution. The projected system is then approximately solved (perhaps recursively invoking the multigrid idea) and the result is interpolated and added to the fine resolution iterate. The key is that basic iterative procedures are normally efficient smoothers as they rapidly reduce oscillatory error when applied to elliptic problems. Smooth errors appear more oscillatory at lower resolutions and so these modes are efficiently reduced by projecting them to a lower resolution and then applying a basic iterative procedure. Fig. 1 illustrates a multigrid V-cycle. A_ℓ is the discretization matrix on level ℓ. R_ℓ restricts residuals from level ℓ to level $\ell+1$, and P_ℓ prolongates from level $\ell+1$ to ℓ. $\mathcal{S}_\ell^{pre}()$ and $\mathcal{S}_\ell^{post}()$ are smoothing procedures. A W-cycle is obtained by adding a second $c \leftarrow \mathrm{MGV}()$ invocation immediately after the current one in Fig. 1 [29].

In the results presented in this study we consider a class of fully-coupled algebraic multilevel preconditioners [7–9,27,30,31] that are based on a variant of aggregation methods [32,33]. Aggregation type-methods implicitly define coarse meshes by grouping fine mesh vertices into aggregates so that each aggregate effectively represents a coarse mesh vertex. Standard smoothed aggregation typically utilizes aggregates with about 10 and 30 nodes, respectively, for 2D and 3D isotropic problems. The aggressive coarsening approach implemented in ML [24,34] applies an aggressive coarsening algorithm based on partitioning a graph representation of the nonzero block connectivity of the Jacobian matrix A_ℓ (here "block" is used to describe the collocated degrees of freedom at a particular FE node). Graph vertices correspond to matrix rows for scalar PDEs while for PDE systems it is natural to associated one vertex with each nodal block of unknowns, e.g. velocities and pressures at a particular grid point. A graph edge exists between vertex i and j if there is a nonzero in the block matrix which

$$
\begin{aligned}
&\mathsf{MGV}(A_\ell, u, b, \ell) : \\
&\quad \textit{if } \ell \neq \ell_{max} \\
&\qquad u \leftarrow \mathcal{S}_\ell^{pre}(A_\ell, u, b) \\
&\qquad r \leftarrow b - A_\ell u \\
&\qquad c \leftarrow 0 \\
&\qquad c \leftarrow \mathsf{MGV}(A_{\ell+1}, c, R_\ell r, \ell{+}1) \\
&\qquad u \leftarrow u + P_\ell c \\
&\qquad u \leftarrow \mathcal{S}_\ell^{post}(A_\ell, u, b) \\
&\quad \textit{else } u \leftarrow A_\ell^{-1} b
\end{aligned}
$$

Fig. 1. Multigrid V-cycle to solve $A_\ell u = b$.

couples *i*'s rows with *j*'s columns or *j*'s rows with *i*'s columns. In some situations it may be advantageous to omit edges if all entries within the coupling blocks are small [35]. Once defined, the graph is coarsened. In this paper, coarsening follows a smoothed aggregation philosophy [32,33] using METIS and ParMETIS [36] to define the aggregates (partitions). Fine mesh vertices are grouped into aggregates so that each aggregate effectively represents a coarse mesh vertex. We orient METIS/ParMETIS so that they generate somewhat larger aggregates than standard smoothed aggregation (e.g., >50 nodes per aggregate). This aggressive coarsening significantly reduces the total number of levels ($\leqslant 5$) which we find better suited for parallel computations [8,24]. Additionally, larger aggregates are consistent with the use of more substantial smoothing (i.e., Schwarz/*ILU*(*k*) compared to Gauss–Seidel). Once aggregates are chosen, the required projection operators can be defined by associating interpolation basis functions with each aggregate (see [8,9] for details on the choices of these interpolation operators).

In the context of the performance of the Krylov methods on distributed memory systems, we make the following brief comments. While a significant number of individual Krylov methods have been developed for specific problem classes (see e.g. [21,22]), these methods essentially rely on a small and well defined set of basic kernel routines. These kernel routines consist of vector–vector, vector inner product, matrix–vector, and preconditioning operations. Of these kernels the matrix–vector and preconditioning operations are the most sensitive to memory bandwidth limitations for information transfer between the multicore CPUs and their local RAM. In this study our intention is to focus on the relative performance of the one-level and multilevel Schwarz preconditioning methods (described above) on a number of recently available multi-socket, multicore architectures. In a later study we will consider the choice of matrix–vector operations that will include a comparison of a point entry distributed memory sparse format (such as the CRS format used in this study [37,38]) and alternate block entry formats that are designed to use directly addressed BLAS2 routines for the matrix–vector operation. The use of a block entry sparse format would also have an impact on the performance of preconditioning methods if details of the sparse format are taken into account in for example subdomain ILU factorizations. For an indication of how this type of choice of matrix format can affect the performance of the matrix–vector multiply on single-core CPUs see our earlier study [39] and recent work on multicore CPUs [40].

As described above the single-level and multilevel Schwarz methods employ essentially the same iterative solution algorithm, a domain decomposition additive Schwarz method with variable overlap and *ILU*(*k*) subdomain solver, as a preconditioner for the one-level method and as a smoother in the multilevel context. The essential difference in the preconditioners is the development of the projection operators to produce the coarse operators, the application of the Schwarz/*ILU*(*k*) smoother on intermediate levels and the direct sparse matrix factorization on the coarsest level. While the construction and solution of the intermediate scale problems corresponds reasonably closely to the parallel communication pattern that is required in the fine mesh smoother, the characteristics of the coarsest level problem are much different. At the coarsest level the communication that is required to produce the coarse operator requires information for each processor to be sent to a single (or a small subset of) processor(s) that is building the coarse matrix problem. This processor then solves the coarse sparse matrix problem (typically about 10,000 unknowns in size) with a serial direct sparse matrix solver. While the global communication and the serial work associated with the development and solution of the coarse operator (in this study solved on a single processor) constitutes a reasonably significant overhead, without the use of this global information, the nearly optimal algorithmic convergence of these methods as demonstrated in the results section of this study would not be obtained.

Finally it should be noted that the Newton–Krylov solvers and multilevel preconditioners are available in the Trilinos framework [37]. Ifpack provides incomplete factorizations [41]. Krylov methods are implemented in AztecOO [42,43]. A serial sparse KLU factorization is used for the coarse direct solver [44]. The multigrid cycles and grid transfers are provided by ML [24]. ML also provides aggregation routines though METIS and ParMETIS [45] that are used in this paper.

3. Results and discussion

This section considers the efficiency of the device simulation code for multicore processors. First the efficiency of using the cores on a single mainboard will be considered, without the effect of the network. Three different configurations will be

considered: a single compute node of a quad-socket, quad-core 2.2 GHz AMD Opteron Barcelona processor linux cluster, a single compute node of a dual-socket, quad-core 2.9 GHz Intel Xeon X5570 (Nehalem-EP or Gainestown) processor linux cluster, and a workstation with dual-socket, six-core 2.6 GHz AMD Opteron Istanbul processor. Next the effect of the compute nodes and the network for the Barcelona platform will be considered. Finally, the performance of the Barcelona platform will be compared with a Cray XT3/4 machine for a large-scale simulation.

The numerical studies involve the steady-state solution of the two-dimensional drift–diffusion equations for a 2×1.5 μm silicon bipolar junction transistor (BJT) [3,46]. The steady-state calculation is performed with a voltage bias of 0.3 V. Fig. 2(a) shows the electric potential for the device. The maximum donor doping is 10^{19} and the maximum acceptor doping is 10^{16}. Fig. 2(b) shows the electric field.

Before considering how efficiently the device simulation code can use the cores on a multi-socket, multicore compute node, short discussion concerning accessing memory within a compute node and placing MPI tasks on cores would be pertinent. As an example, a TLCC compute node will be considered.

3.1. Memory and placement of MPI tasks on sockets

Each compute node of TLCC has four sockets, where each socket is a quad-core processor. Each core has its own L1 and L2 cache, but the four cores on a socket share a 2 MB L3 cache. In contrast to the traditional practice of having the memory controller separate from the CPU, various CPU designs such as the AMD Opteron and the Intel Nehalem place the memory controller on the CPU. Because of this, compute nodes with multiple sockets have shared memory, but nonuniform memory access (NUMA). For TLCC, the memory controller for each socket accesses 8 GB RAM, so each compute node has a total of 32 GB RAM. If a task on one socket needs to access memory handled by the memory controller of another socket, there will be a considerable performance penalty as the data will have to travel through the HyperTransport (HT) links connecting the sockets. By default, the operating system distributes the MPI tasks on the sockets and cores as well as distributing data for the shared memory. A user can use the `numactl` command to specify a certain placement of MPI tasks to sockets and cores as well as a specific memory placement policy. The main improvement is due to tying MPI tasks to specific sockets and using the corresponding on-chip memory controller.

For performance reasons, MPI tasks should access memory that is local to a socket. However, if the local memory for a socket fills up, rather than have the job crash, it may be better to allow an MPI task to use memory that is local to another socket so that the job will complete. To see the upper bound on how severe the performance penalty can be, one can force MPI tasks to use memory that is not local to its socket.

Consider an example for the BJT with a mesh with 27.9 million unknowns and run on 128 cores with a three-level multigrid preconditioner (W(1,1) cycle and 125 nodes per aggregate). All 16 cores per compute node are used, and each core has about 218,000 unknowns. Table 1 compares the time per Newton step for five choices of memory layout:

- "local": four MPI tasks tied to each socket and corresponding on-chip memory controller.
- "nonlocal 1-hop": four MPI tasks tied to each socket but each MPI task needs to perform one HT "hop" to access the memory controller on another socket.
- "nonlocal 1,2-hop": four MPI tasks tied to each socket but each MPI task needs to perform one or two HT "hops" to access the memory controller on another socket.
- "nonlocal 2-hop": four MPI tasks tied to each socket but each MPI task needs to perform two HT "hops" to access the memory controller on another socket.

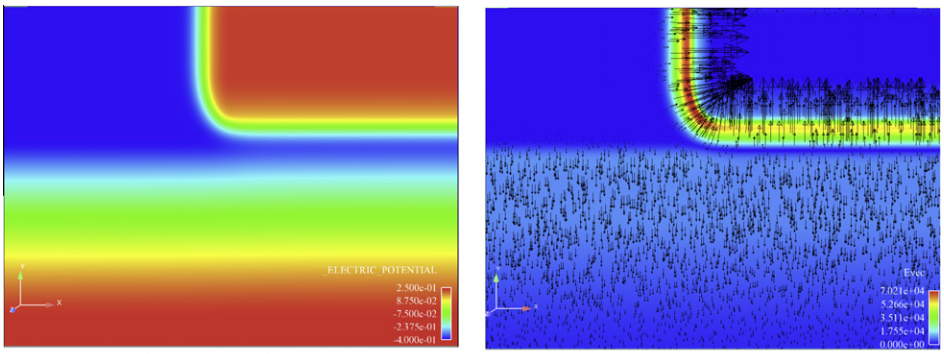

Fig. 2. (a) Steady-state electric potential at 0.3 V bias for 2×1.5 μm **2D BJT**. Red and blue denote high and low electric potential, respectively (for B&W case, high electric potential is region along the bottom and upper right region while low electric potential is upper left region). (b) Steady-state electric field: shading represents magnitude of electric field and the arrows represent a vector plot of the electric field. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 1
Effect on performance due to local vs. nonlocal memory access on a node.

`numactl` memory affinity	3-Level W(1,1) Per Newton step	
	Time (s)	Time/Ref
Local	225	Ref
Nonlocal 1-hops	409	1.81
Nonlocal 1,2-hops	459	2.04
Nonlocal 2-hops	436	1.93
None	256	1.14

- "none": `numactl` is not used; OS decides which sockets and memory controllers an MPI task will use and may allow MPI tasks on a socket to use nonlocal memory.

The third column in Table 1 is the ratio of the time to the reference time (local memory). These results show that in the worst-case scenario—when nonlocal memory is used—the run time can be doubled for this particular test case. However, the data also indicates that anytime nonlocal memory is used there will be a significant performance penalty. Of the four nonlocal memory usage cases, the smallest penalty is the "1-hop" case. Intuitively one would have expected the "2-hop" case to be around the same time as the two "1,2-hop" cases as synchronization causes all tasks to wait for the slowest task, but contrary to expectation it is a little faster. A few additional test cases with different sized meshes were used, and for those test cases, the ratio of the time for the "nonlocal 1-hop" and "nonlocal 1,2-hops" and the reference time (local memory controller case) were 1.4–2.2 and 1.5–2.5, respectively. It should be stated that the performance penalty is of course problem specific. The final row in Table 1 is the performance penalty when `numactl` is not used, and is 14% for this case, and for additional tests involving more than one compute node, this penalty was about 10–23%.

3.2. Parallel scaling and efficiency on a single compute node

For all three platforms (Barcelona, Nehalem, and Istanbul), the efficiency of using the different numbers of cores on the multicore processor will be evaluated by considering two weak scaling studies and one strong scaling study. The two weak scaling studies consider different amount of work per core (4–8 times as many degrees of freedom (DOF) per core). All these studies (Section 3.2) use the transpose-free quasi-minimal residual (TFQMR) Krylov subspace method for the linear solver [21] with three-level ML Petrov–Galerkin smoothed aggregation (PGSA) multigrid preconditioner [9,46]. A W(1,1) cycle was used (W-cycle with one presmoothing and one postsmoothing relaxation sweep). The two coarser levels were obtained using METIS then ParMETIS for the aggressive coarsening, with 125 nodes per aggregate. The smoothers on the fine and medium level are an incomplete lower upper factorization with fill of 1 (ILU (1)) with an overlap of one. The linear solve tolerance was set sufficiently low so that all Newton steps would require 100 Krylov iterations. Note this requirement is to assess the parallel efficiency of the algorithm and it would not be correct to conclude that the multilevel preconditioner, that costs more per iteration, is slower to solve a specific problem. For actual large scale simulations, the multilevel preconditioner will require significantly fewer iterations to meet a certain convergence tolerance than the one-level preconditioner. The Barcelona study additionally includes results with a one-level DD ILU preconditioner for the two weak scaling studies and one strong scaling study. Many application codes use a one-level DD ILU preconditioner, so these results are of relevance.

3.2.1. Barcelona

We first consider scaling on a single compute node of a quad-socket, quad-core 2.2 GHz AMD Opteron Barcelona linux cluster. Table 2 presents a weak scaling study for the steady-state drift–diffusion simulation at voltage bias of 0.3 V for the BJT. The one-level DD ILU preconditioner is used. A "weak scaling study" denotes that the size of the problem is scaled up but the number of processors cores is scaled up in a manner so that the amount of work per core stays about the same.

Table 2
Weak scaling on quad-socket, quad-core Barcelona: 1-level preconditioner, 28 K DOF/core.

Core	Total DOF (K)	TFQMR 1-level DD ILU (1) 28 K DOF/core							
		Prec setup		Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	28	0.62	Ref	3.52	Ref	4.06	Ref	10.72	Ref
4	110	0.64	97	3.90	90	3.99	1.02	10.99	98
8	219	0.67	93	4.20	84	3.96	1.03	11.05	97
12	329	0.70	89	4.77	74	3.98	1.02	11.68	92
16	438	0.72	86	5.75	61	3.97	1.02	12.59	85

The problem is scaled up so that each core has about 28,000 DOFs. The columns in the table denote: number of processor cores, total number of DOFs over all the cores and time per Newton step in seconds and relative efficiency for the preconditioner setup, Aztec, Jacobian matrix construction, and total. "Aztec" is the time to solve the linear system once the preconditioner setup has been performed, and is sometimes referred to as the "preconditioned iteration time." It does not include the preconditioner setup time or the time to construct the Jacobian matrix. When the ML multigrid preconditioner is used, the linear system solve time will be denoted "ML/Aztec" as ML is applied every Krylov iteration. For the one-level DD ILU preconditioner, rather than writing "ILU/Aztec," we will simply write "Aztec." As mentioned earlier, all Newton steps took 100 Krylov iterations. η denotes an efficiency, i.e. ratio of time for that run and the reference time (run with one core). The 4-, 8-, 12-, 16-core cases denote 1, 2, 3, 4 MPI tasks per socket, respectively.

Table 3 presents a weak scaling study for the one-level preconditioner where the problem is scaled up so that each core has 110,000 unknowns (four times as much work per core as the previous study). Columns are the same as with the previous table. The reason for performing the two studies for different DOFs per core is to make sure that cache effects are not distorting the results for the smaller problem. Note that the efficiencies for the two studies are similar. The reader can observe that the time to construct the Jacobian matrix scales nearly perfectly, the time for the preconditioner setup does not scale as well, and the Aztec time scales the worst. The Jacobian matrix fill involves mostly element integration and other local operations, and mainly depends on cache and CPU performance. In contrast, Aztec involves substantial communication. Note that for large-scale problems, the Aztec time will dominate the total time, so the efficiency for the total time will be around the same as the Aztec time.

Table 4 presents a strong scaling study (fixed size problem) on a single compute node. The problem has 438,000 DOFs and is solved on a different number of cores. Both the parallel speedup and parallel efficiency are presented for the preconditioner setup, Aztec, Jacobian matrix construction, and total times.

The previous three tables are then repeated, but with the three-level PGSA ML preconditioner instead of the one-level preconditioner. The coarser levels are obtained by taking 125 nodes per aggregate. Tables 5 and 6 present weak scaling studies for the the 28,000 and 110,000 DOF/core cases, respectively. Table 7 presents a strong scaling study for the 438,000 DOF problem.

From Tables 5–7 one can see that the efficiencies obtained from using all 16 cores on a compute node show that it is advantageous to use all the cores with the MPI-only approach. The ML/Aztec operation drops most rapidly in efficiency as the number of MPI tasks per socket is increased. ML/Aztec requires both high bandwidth to main memory as well as significant communication between MPI tasks. The preconditioner setup requires a mixture of local operations, bandwidth to memory, and communication, so the efficiency drops more slowly. The Jacobian matrix consists of element integration (for the finite element method) which uses cache efficiently, and has optimal scaling. Note that for large-scale simulations, the ML/Aztec time will dominate the simulation time, so the "total" efficiency will be close to the ML/Aztec efficiency.

3.2.2. Nehalem

Next, scaling on a single compute node of a dual-socket, quad-core 2.9 GHz Intel Xeon X5570 (Nehalem) linux cluster will be considered. Similar weak and strong scaling studies for the Nehalem compute node as the Barcelona compute node will be

Table 3
Weak scaling on quad-socket, quad-core Barcelona: 1-level preconditioner, 110 K DOF/core.

Core	Total DOF	TFQMR 1-level DD ILU (1) 110 K DOF/core							
		Prec setup		Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	110 K	2.48	Ref	14.3	Ref	16.3	Ref	43.0	Ref
4	438 K	2.54	98	15.9	90	15.9	1.02	43.0	100
8	873 K	2.62	95	17.6	82	15.9	1.02	44.7	96
12	1.31 M	2.72	91	20.1	71	16.2	1.01	47.8	90
16	1.75 M	2.82	88	24.3	59	16.0	1.02	51.7	83

Table 4
Strong scaling on quad-socket, quad-core Barcelona with one-level preconditioner; "speedup" is abbreviated as "s.u." and η is parallel efficiency.

Core	DOF/core	TFQMR 1-level DD ILU; total 438 K DOF											
		Prec. setup			Aztec			Jacobian matrix			total		
		t (s)	s.u.	η	t (s)	s.u.	η	t (s)	s.u.	η	t (s)	s.u.	η
1	438 K	9.89	Ref	Ref	58.9	Ref	Ref	64.8	Ref	Ref	170	Ref	Ref
4	109 K	2.54	3.89	97	15.9	3.71	93	16.0	4.06	1.02	43.1	3.94	98
8	55 K	1.32	7.49	94	8.43	6.99	87	8.00	8.10	1.01	22.1	7.69	96
12	36 K	0.91	10.9	91	6.37	9.25	77	5.29	12.3	1.02	15.4	11.0	92
16	27 K	0.72	13.7	86	5.75	10.3	64	3.97	16.3	1.02	12.6	13.5	84

Table 5
Weak scaling on quad-socket, quad-core Barcelona: 3-level ML preconditioner, 28 K DOF/core.

Core	Total DOF (K)	TFQMR 3-level ML W(1,1) 28 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	28	1.10	Ref	8.61	Ref	3.52	Ref	14.6	Ref
4	110	1.14	96	9.54	90	3.48	1.01	15.4	94
8	219	1.19	92	10.4	83	3.45	1.02	16.3	89
12	329	1.25	88	12.0	72	3.46	1.02	17.9	81
16	438	1.30	85	14.5	59	3.13	1.12	20.1	73

Table 6
Weak scaling on quad-socket, quad-core Barcelona: 3-level ML preconditioner, 110 K DOF/core.

Core	Total DOF	TFQMR 3-level ML W(1,1) 110 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	110 K	4.44	Ref	34.8	Ref	14.2	Ref	58.6	Ref
4	438 K	4.60	97	38.5	90	13.8	1.03	62.0	95
8	873 K	4.79	93	41.9	83	13.8	1.03	65.4	90
12	1.31 M	5.06	88	48.9	71	14.1	1.01	72.9	80
16	1.75 M	5.27	84	59.7	58	13.7	1.03	83.4	70

Table 7
Strong scaling on quad-socket, quad-core Barcelona; "speedup" is abbreviated as "s.u." and η is parallel efficiency.

Core	DOF/core (K)	TFQMR 3-level ML W(1,1), total 438 K DOF											
		Prec setup			ML/Aztec			Jacobian matrix			Total		
		t (s)	s.u.	η	t (s)	s.u.	η	t (s)	s.u.	η	t (s)	s.u.	η
1	438	18.2	Ref	Ref	143	Ref	Ref	57.0	Ref	Ref	239	Ref	Ref
4	109	4.60	3.96	99	38.5	3.71	93	13.8	4.13	1.03	62.0	3.86	96
8	55	2.38	7.65	96	20.7	6.90	86	6.91	8.25	1.03	32.4	7.38	92
12	36	1.66	11.0	91	15.9	8.96	75	4.61	12.4	1.03	23.8	10.1	84
16	27	1.31	13.9	87	14.5	9.86	62	3.45	16.5	1.03	20.4	11.7	73

Table 8
Weak scaling on dual-socket, quad-core Nehalem: 3-level ML preconditioner, 28 K DOF/core.

Core	Total DOF (K)	TFQMR 3-level ML W(1,1) 28 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	28	0.57	Ref	2.44	Ref	2.12	Ref	5.92	Ref
2	55	0.59	97	2.60	94	2.14	99	6.14	96
4	110	0.58	98	3.07	79	1.95	109	6.35	93
6	165	0.60	95	3.71	66	2.00	106	7.10	83
8	219	0.61	93	4.42	55	2.00	106	7.82	76

performed. However, because the Nehalem node has only half the number of cores as the Barcelona node, a different sequence of meshes is used for the weak scaling studies. The two weak scaling studies used 28,000 DOF/core and 219,000 DOF/core. The same mesh with 438,000 DOF will be used for the strong scaling study, which means that except for the single core run, each core will have twice as many DOFs compared with the previous strong scaling study for the Barcelona.

Tables 8 and 9 present the weak scaling studies for 28,000 DOF/core and 219,000 DOF/core, respectively, on the Nehalem. Table 10 is the strong scaling study for a problem size of 438,000 DOF. The 2-, 4-, 6-, 8-core cases denote 1, 2, 3, 4 MPI tasks per socket, respectively. Note that the efficiencies for the Nehalem are similar to the ones of the Barcelona, although the ML/Aztec efficiencies are dropping faster with Nehalem and that except for the weak scaling study with 28,000 DOF/core, Jacobian matrix efficiencies do not exceed 100%.

Table 9
Weak scaling on dual-socket, quad-core Nehalem: 3-level ML preconditioner, 219 K DOF/core.

Core	Total DOF	TFQMR 3-level ML W(1,1) 219 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	219 K	4.23	Ref	17.6	Ref	15.2	Ref	43.0	Ref
2	438 K	4.42	96	19.4	91	15.6	98	45.3	95
4	873 K	4.88	87	25.0	70	16.2	94	52.5	82
6	1.31 M	4.95	85	30.5	58	16.1	94	57.8	74
8	1.75 M	4.94	86	36.6	48	15.7	97	63.5	68

Table 10
Strong scaling on dual-socket, quad-core Nehalem; "speedup" is abbreviated as "s.u." and η is parallel efficiency.

Core	Total DOF (K)	TFQMR 3-level ML W(1,1) total 438 K DOF											
		Prec setup			ML/Aztec			Jacobian matrix			Total		
		Time	s.u.	η	Time	s.u.	η	Time	s.u.	η	Time	s.u.	η
1	438	8.62	Ref	Ref	35.8	Ref	Ref	30.7	Ref	Ref	86.4	Ref	Ref
2	219	4.42	1.95	98	19.4	1.85	92	15.6	1.97	98	45.3	1.91	95
4	109	2.23	3.87	97	11.9	2.99	75	7.75	3.95	99	24.9	3.48	87
6	73	1.59	5.42	90	9.64	3.71	62	5.20	5.89	98	18.4	4.70	78
8	55	1.23	7.00	88	8.51	4.20	53	3.90	7.86	98	15.1	5.71	71

3.2.3. Istanbul

Next, scaling on a workstation with dual-socket, six-core 2.6 GHz AMD Opteron Istanbul processors is considered. Similar weak and strong scaling studies for the Istanbul workstation as for the Nehalem compute node were performed. The Istanbul CPUs have six cores in contrast to the four cores on Nehalem. The weak scaling studies up to eight cores are the same for Istanbul and Nehalem, but for the Istanbul case, the problem is scaled further for the five and six-core cases.

Tables 11 and 12 are the weak scaling studies for 28,000 DOF/core and 219,000 DOF/core respectively on the Istanbul. Table 13 is the strong scaling study for a problem size of 438,000 DOF. The 2-, 4-, 6-, 8-, 10-, 12-core cases denote 1, 2, 3, 4, 5, 6 MPI tasks per socket, respectively.

Note that the efficiencies for Istanbul are similar to Barcelona for the same number of MPI tasks per socket. Naturally with 5 and 6 MPI tasks per socket the efficiency drops further. It is interesting to note that typically the Jacobian matrix construction scales extremely well when using more cores per socket. However for the 219,000 DOF/core Istanbul case, the Jacobian matrix efficiency drops from 97% to 91% when the number of cores used per socket is increased from 5 to 6. It is unclear what is the source of this efficiency drop, but it is likely contention for the L3 cache.

As the Nehalem and Istanbul are currently competing in the marketplace, it is natural for the reader to want to compare these CPUs. If one compares the 8-core case from Table 10 and the 12-core case from Table 13, it can be observed that for the 438,000 DOF test case, the Nehalem is about 35% and 20% faster for ML/Aztec and total time respectively. This result applies to this specific test case with our device simulation code only, and the authors do not make any claims for other test cases or other applications. Naturally for different applications, the results may be completely different. It should be noted that the Intel 11.0 compiler was used to build the device simulation executable for both the Nehalem and Istanbul, and the Intel compiler is designed to maximize performance on Intel processors and not AMD processors. For example, the Intel 11.0 compiler has an optimization flag to maximize SIMD performance on the Nehalem (-xsse4.2). Also, if one considers peak memory bandwidth, the Nehalem configuration used has a factor of 2.5 times greater memory bandwidth than the Istanbul.

Table 11
Weak scaling on dual-socket, six-core Istanbul: 3-level ML preconditioner, 28 K DOF/core.

Core	Total DOF (K)	TFQMR 3-level ML W(1,1) 28 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	28	0.78	Ref	4.60	Ref	2.46	Ref	8.72	Ref
2	55	0.81	96	5.02	92	2.46	100	9.19	95
4	110	0.86	91	5.92	78	2.50	98	10.2	86
6	165	0.89	88	6.76	68	2.55	96	11.1	78
8	219	0.93	84	7.85	59	2.52	98	12.2	71
10	273	0.95	82	8.83	52	2.52	98	13.2	66
12	329	0.99	79	9.98	46	2.55	96	14.5	60

Table 12
Weak scaling on dual-socket, six-core Istanbul: 3-level ML preconditioner, 219 K DOF/core.

Core	Total DOF	TFQMR 3-level ML W(1,1) 219 K DOF/core							
		Prec setup		ML/Aztec		Jacobian matrix		Total	
		Time/N (s)	η	Time/N (s)	η	Time/N (s)	η	Time/N (s)	η
1	219 K	6.59	Ref	37.61	Ref	19.73	Ref	71.06	Ref
2	438 K	6.82	97	40.57	93	19.99	99	74.46	95
4	873 K	7.08	93	47.72	79	20.05	98	82.07	87
6	1.31 M	7.35	90	55.69	68	20.02	99	90.35	79
8	1.75 M	7.58	87	65.19	58	20.13	98	100.3	71
10	2.19 M	7.87	84	74.74	50	20.37	97	110.5	64
12	2.62 M	8.21	80	85.33	44	21.72	91	123.2	58

Table 13
Strong scaling on dual-socket, six-core Istanbul; "speedup" is abbreviated as "s.u." and η is parallel efficiency.

Core	Total DOF (K)	TFQMR 3-level ML W(1,1) total 438 K DOF											
		Prec setup			ML/Aztec			Jacobian matrix			Total		
		Time	s.u.	η	Time	s.u.	η	Time	s.u.	η	Time	s.u.	η
1	438	13.4	Ref	Ref	76.4	Ref	Ref	40.3	Ref	Ref	144	Ref	Ref
2	219	6.82	1.96	98	40.6	1.88	94	20.0	2.02	101	74.5	1.94	97
4	109	3.46	3.86	97	23.5	3.25	81	10.2	3.97	99	40.7	3.55	89
6	73	2.40	5.57	93	18.2	4.20	70	6.77	5.96	99	29.8	4.84	81
8	55	1.84	7.27	91	15.5	4.92	61	5.05	7.99	100	24.3	5.96	74
10	44	1.53	8.74	87	14.2	5.38	54	4.06	9.94	99	21.3	6.79	68
12	36	1.33	10.1	84	13.3	5.73	48	3.38	11.9	99	19.3	7.49	62

3.3. Effects of network and compute nodes on scaling

Any reasonable size calculation needed to obtain sufficient fidelity will not fit on a single compute node, so one needs to consider how the network effects the performance of the calculations. If one needs to run a job with T MPI tasks on a machine with compute nodes with m cores, one can attempt to determine a rough idea of the penalty due to the contention for memory by considering the extreme case where one runs a single MPI task on T compute nodes, leaving the rest $m-1$ cores idle, and m MPI tasks on T/m compute nodes and comparing the run times. This choice involves a trade-off between network effects and memory contention effects.

Tables 14 and 15 presents results for this study involving 256 MPI tasks run on the SNL TLCC platform (Barcelona quad-socket, quad-core compute nodes with InfiniBand). A total of 256 cores were used for this test case, but with different numbers of MPI tasks per compute node ("ppn" denotes "processes per node"), which determines the number of compute nodes needed. For the "4ppn," "8ppn" and "12ppn" cases, each socket has one, two and three MPI tasks, respectively. Two different size meshes were used to show the effect of contention of bandwidth to memory among the cores. The 27,300 DOF/core (Table 14) case is more representative of a typical device simulation run. The 109,000 DOF/core (Table 15) case attempts to represent the worst case scenario that tries to use the majority of the RAM and therefore would maximize contention for memory. The 14% difference in efficiency for ML/Aztec between the 27,000 DOF/core and 109,000 DOF/core cases when using all 16 cores per compute node is due to this contention for memory. Note that the time to construct the Jacobian decreases as the number of compute nodes decreases for both cases. This is likely due to the fact that fewer network cards are involved in communication. The construction of the Jacobian involves two steps: first a local matrix fill, then a communication step for the global assembly. As in the previous section, the preconditioner is a three-level ML W(1,1) cycle with coarser

Table 14
Multicore network/node study starting with 256 compute nodes: 27 K DOF/core.

Config.	3-level ML W(1,1), 27.3 K DOF/core							
	Prec setup		ML/Aztec		Jacobian		Total	
	Time/N	η	Time/N	η	Time/N	η	Time/N	η
256n 1ppn	2.07	Ref	7.85	Ref	3.26	Ref	14.6	Ref
64n 4ppn	2.06	100	8.48	93	3.13	104	15.0	97
32n 8ppn	2.08	100	9.10	86	3.11	105	15.5	94
21.3n 12ppn	2.15	96	10.9	72	3.06	107	17.4	84
16n 16ppn	2.23	93	13.5	58	3.06	107	20.1	73

Table 15
Multicore network/node study starting with 256 compute nodes: 109 K DOF/core.

Config.	3-level ML W(1,1), 109 K DOF/core							
	Prec setup		ML/Aztec		Jacobian		Total	
	Time/N	η	Time/N	η	Time/N	η	Time/N	η
256n 1ppn	6.01	Ref	38.5	Ref	15.8	Ref	66.9	Ref
64n 4ppn	5.87	102	40.6	95	15.4	103	68.1	98
32n 8ppn	6.06	99	50.3	77	15.2	104	77.6	86
21.3n 12ppn	6.16	98	68.8	55	14.7	107	95.9	70
16n 16ppn	6.30	95	88.4	44	14.3	111	115	58

levels generated by METIS/ParMETIS with 125 nodes per aggregate. The Krylov solver is GMRES. From Tables 14 and 15, one can see that the efficiencies obtained from using all 16 cores on a compute node show that for the device simulator it is definitely advantageous to use all 16 of the cores.

As mentioned earlier, the drop in ML/Aztec efficiency as the number of cores per socket is increased is not a good trend. Eventually it will reach an unacceptable level. Algorithms that efficiently use cache would be critical. For example, we plan to investigate variable block row (VBR) algorithms. VBR is a block entry format that can directly address the memory within a block and cut down the requirement for dereferencing pointers to get to memory. We also plan to look into other programming paradigms such as hybrid approaches that use MPI between compute nodes and threading within a node.

3.4. An illustration of large-scale computation and comparison with a Cray XT3/4

A weak scaling study that compares TLCC and the Nehalem InfiniBand cluster as the problem is scaled from 16 cores to 256 cores (and 7 million DOF) is presented in Table 16. The TLCC results in this section use all 16 cores per compute node, while the Nehalem cluster results use all 8 cores per compute node. The 16 core case for TLCC is just a single compute node and does not include the effect of the network. However, the 16 core case for the Nehalem cluster involves two compute nodes and does include network effects. As mentioned earlier, it should be noted that the Intel 11.0 compiler was used to build the device simulation executable for both the Barcelona and Nehalem, and the Intel compiler is designed to maximize performance on Intel processors and not AMD processors. Also, if one considers peak memory bandwidth, the Nehalem processors used has a factor of three times greater memory bandwidth than the Barcelona processors.

As we are interested in time to solution of large-scale simulations on multicore platforms, TLCC will be compared with a Cray XT3/4 (Red Storm) for a weak scaling study up to 4096 cores and 112 million DOF. The Cray XT3/4 (Red Storm) is intended for large-scale simulations and designed with a scalable custom high performance network that maximizes performance and scalability (3D mesh with custom Cray SeaStar chips). Each compute node has a single dual-core AMD Opteron. The Red Storm results in this section use one core of the 2.4 GHz dual-core CPUs, leaving the second core idle so that there will not be any contention for the shared L2 cache and memory controller, as well as the SeaStar. This limitation to one core per node has been chosen to allow comparison of the high performance communication interconnect, using the maximum number of communication links required (4096), of the capability Cray XT3/4 machine with the current performance of a capacity-type architecture that uses a multi-socket, multicore architecture with a less expensive interconnect.

It is of interest to compare the performance of a one-level DD ILU preconditioner and a multigrid preconditioner on the two platforms as the communication pattern is significantly different. The one-level DD ILU preconditioner involves subdomain boundary data exchange with its nearest neighbors. The multigrid preconditioner uses this same DD ILU preconditioner as smoothers on the fine and medium mesh and KLU direct solver on the coarse mesh. Not only does the multigrid preconditioner require boundary data exchange with its nearest neighbors for ILU on the fine and medium mesh, it also requires more complex communication for the grid transfer operators and the coarse mesh solve. Tables 17 and 18 illustrate the relative performance of the two machines for a weak scaling study with 27,000 DOF/core for the one-level and multigrid preconditioner, respectively.

As with earlier tables, under "Time/Newton step", "prec" denotes the time per Newton step to construct the preconditioner, "Aztec" or "ML/Aztec" denotes the time to perform the linear system solve (this does not include the time to construct the

Table 16
Weak scaling study upto 7 million unknowns comparing the TLCC and Nehalem cluster with 27 K DOF/core.

Proc.	Fine grid unk	GMRES 3-level ML W(1,1)									
		TLCC					Nehalem cluster				
		iter/Newt	Time/Newton step (s)				iter/Newt	Time/Newton step (s)			
			Prec	ML/Aztec	Jac	Total		Prec	ML/Aztec	Jac	Total
16	438 K	48	1.47	4.87	3.12	10.6	48	0.77	1.48	1.99	5.03
64	1.75 M	70	1.73	8.06	3.15	14.1	70	0.94	2.50	1.99	6.23
256	6.98 M	95	2.47	14.0	3.18	20.9	95	1.31	4.65	2.02	8.81

preconditioner), "Jac" denotes the time to construct the Jacobian, and "total" denotes the total time. For the 4096 core case, 256 compute nodes on TLCC are used while 4096 compute nodes on Red Storm are used.

All runs took seven Newton steps, except the 16-core three-level multigrid preconditioner case on TLCC which took six Newton steps. The GMRES Krylov method was used, preconditioned by a three-level multigrid preconditioner that uses a W(1,1) cycle with coarser levels generated using METIS/ParMETIS and 125 nodes per aggregate.

The comparison between TLCC and Red Storm for the one-level and three-level preconditioners are plotted in Fig. 3. The time plotted on the vertical axes is the total CPU time per Newton step in seconds. From the tables and figures a few issues should be pointed out to interpret the results. First the one-level DD ILU preconditioner, in general, has a very poor algorithmic scaling. This is apparent from the iteration count that increases significantly with problem size (and core count) in Table 17. This accounts for the significant increase in CPU time in Fig. 3(a) and is in contrast to the three-level method in Fig. 3(b) (note the change in CPU time magnitudes). This scaling illustrates why some type of multilevel technique is required to approach truly scalable simulation methods and therefore modern large-scale computing platforms need to run these methods efficiently. In Fig. 3 the increase in CPU times for the simulation have contributions from the increase in the number of iterations (an algorithmic issue) and in the inefficiency of the platform scaling (most likely due to network scaling). To attempt to more clearly show the effect of the scaling efficiency of the network, a plot of the ratio of the TLCC time over the Red Storm time is presented in Fig. 4 for the two different preconditioners. For the 4096 core case, as an example, for the one-level preconditioner this ratio is approximately 2.3, which means that the TLCC calculation took over twice as long as the Red Storm calculation. The horizontal dash-dot green and black lines are included to denote the case where the TLCC and Red Storm are scaling in a similar fashion for the one-level and three-level preconditioners. For the multigrid preconditioner, the fact that the ratio of TLCC over Red Storm time is rapidly growing as the problem size (and core count) is scaled indicates that the TLCC network does not scale as well as the interconnect on Red Storm, due to the increased communication required for the coarser levels and transfers between levels. TLCC was designed as a capacity machine, and was not intended to scale as well as a machine like Red Storm. The multigrid preconditioner, which is critical for large-scale simulations, definitely performed better on the Red Storm machine than TLCC. The performance degradation of TLCC for the largest-scale multilevel computations on 4096 cores, was apparently due to the use of a commodity network interconnect in contrast to the Cray XT3/4. This result indicates that the evaluation of solution methods and algorithms on TLCC, that are intended to elucidate the scaling of these algorithms on capability machines such as Red Storm, is problematic. For further comparisons between TLCC and Red Storm, please see Ref. [47].

3.5. Next steps to improve performance on multicore architectures

The studies demonstrate that for a fully-implicit Newton–Krylov approach, the construction of the Jacobian scales well with increasing numbers of cores per socket, while the efficiency of the linear solver (ML/Aztec) keeps decreasing. Eventually

Table 17
Weak scaling study up to 112 million unknowns comparing one-level preconditioner for TLCC and Red Storm.

Proc.	Fine grid unk	GMRES 1-level DD ILU									
		TLCC					Red Storm				
		iter/Newt	Time/Newton step (s)				iter/Newt	Time/Newton step (s)			
			Prec	Aztec	Jac	Total		Prec	Aztec	Jac	Total
16	438 K	142	0.88	13.0	2.88	17.8	142	0.91	6.45	7.00	16.7
64	1.75 M	287	0.91	43.3	2.93	48.3	287	0.93	20.6	7.00	37.4
256	6.98 M	571	0.95	156	2.96	161	571	0.96	72.0	7.14	82.6
1024	27.9 M	1145	1.00	639	3.01	644	1145	1.00	271	7.57	282
4096	112 M	2267	1.05	2440	3.13	2447	2264	1.01	1072	7.57	1083

Table 18
Weak scaling study up to 112 million unknowns comparing three-level preconditioner for TLCC and Red Storm.

Proc.	Fine grid unk	GMRES 3-level PGSA W(1,1)									
		TLCC					Red Storm				
		iter/Newt	Time/Newton step (s)				iter/	Time/Newton step (s)			
			Prec	ML/Aztec	Jac	Total	Newt	Prec	ML/Aztec	Jac	Total
16	438 K	48	1.50	4.73	2.89	10.2	49	1.90	2.71	7.00	14.0
64	1.75 M	70	1.68	7.69	2.94	13.4	72	2.03	4.46	7.00	15.9
256	6.98 M	95	2.16	12.8	2.95	19.0	97	2.45	7.42	7.57	19.6
1024	27.9 M	125	3.38	25.6	3.02	33.2	125	2.66	11.0	7.57	24.0
4096	112 M	152	10.4	64.0	3.15	78.9	153	4.23	17.7	7.00	31.4

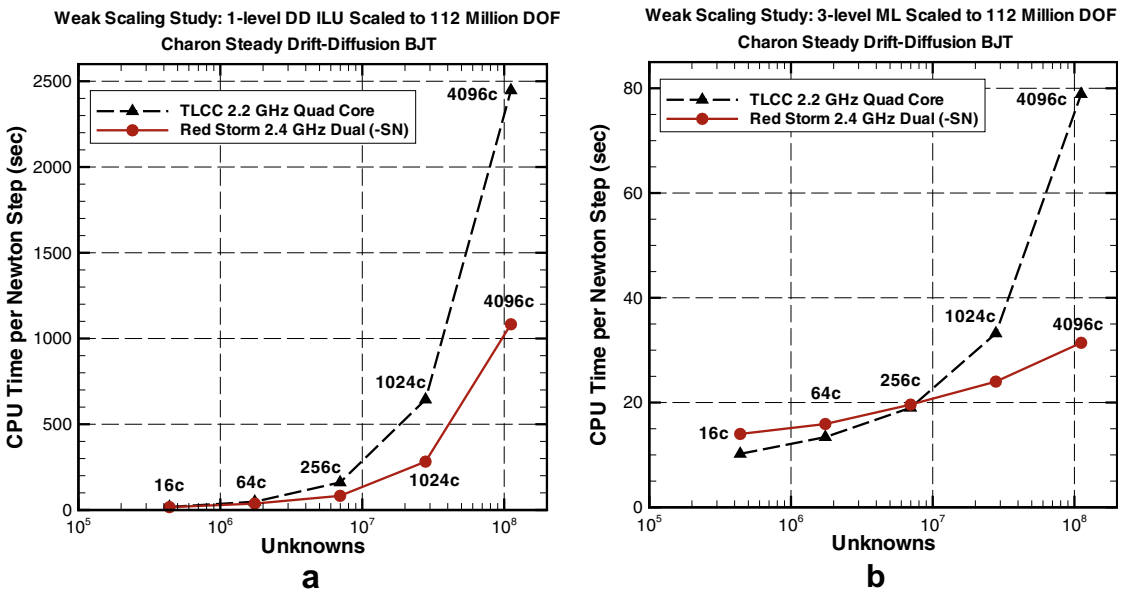

Fig. 3. (a) Weak scaling study up to 112 million unknowns for 1-level DD ILU preconditioner. (b) Weak scaling study up to 112 million unknowns for 3-level ML W(1,1) preconditioner.

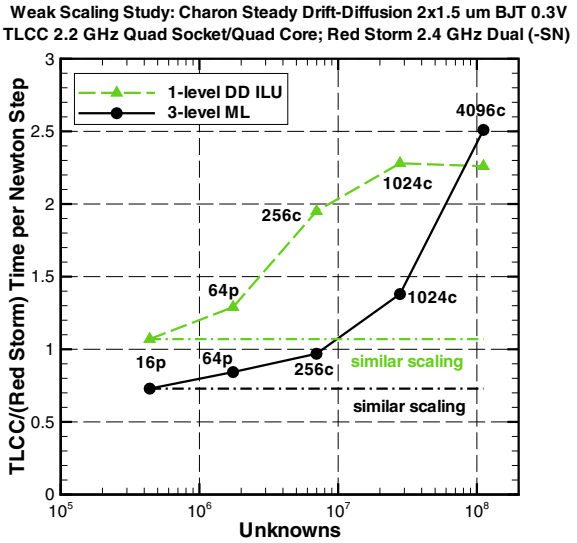

Fig. 4. Weak scaling study up to 112 million unknowns: ratio of times TLCC/RS.

with sufficient number of cores per socket, a hybrid MPI/threaded execution model will be needed for the latter. The plan for the near-term and next few years would be to stay with the MPI-only approach for the Jacobian construction, but employ a hybrid MPI/threaded approach for the linear solver.

The following are some of the ideas we plan to explore to improve the linear solver efficiency. First we will implement a variable block row (VBR) approach for matrix storage. Currently, the matrices are stored in sparse compressed row (CRS) matrix format, which requires indirect addressing to access any element. For the VBR approach, once the beginning of the block is located, one uses direct addressing to access different elements of the block rather than always having to use indirect addressing. This substantially reduces access time. One can implement VBR methods through the Trilinos Epetra_VbrMatrix API [38]. Block Krylov methods can be used to construct a Krylov subspace with a set of vectors, rather than a single vector and can sometimes achieve faster convergence rates. In addition, block Krylov methods can exploit optimized BLAS 3 routines for matrix-multivector multiply, and allow better reuse of memory. Finally, for each compute node, one could have a single subdomain and perform a parallel ILU or direct solver for domain decomposition smoothers and solvers. This will

significantly decrease the total number of subdomains, and decreasing the number of subdomains will decrease the iteration count for the Krylov solver. This will also more effectively use local resources on the compute node. The Trilinos developers are pursuing a hybrid MPI/threaded approach through a compute node API [48].

4. Conclusions

This preliminary study concerned the performance of the finite element semiconductor device simulation code on three current multicore platforms for strong and weak scaling, and multicore performance. In the context of multicore performance this study also demonstrated that the device simulation code could take advantage of all the cores of quad-core and six-core CPUs. Although the efficiency of the linear system solve is dropping as the number of cores is increasing, so far it has not dropped enough to overcome the substantial inertia for application developers to invest in the significant effort to rewrite their codes for multicore architectures. Currently the majority of application developers just run existing codes designed for single-core processors on multicore processors without many changes. However, as the number of cores per CPU keeps increasing, eventually the number of cores will be great enough that the efficiency of the application code drops to an unacceptably low level that application developers will be forced to perform a major revision of their codes.

Although the device simulation code currently performs with satisfactory efficiency for the multi-socket, multicore architectures with an MPI-only programming paradigm, in the future the MPI-only programming paradigm will not be satisfactory and an alternative approach such as a hybrid approach, for example MPI between compute nodes and threading within a compute node, will probably be needed. Clearly algorithms that exploit cache performance will also be important.

Acknowledgments

The authors are grateful to Douglas Doerfler, Jeffrey Ogden, Michael Heroux, Kevin Pedretti, Brian Barrett, and Tommy Minyard for helpful discussions. They thank Douglas Doerfler for access to the Istanbul workstation and Sophia Corwell and the rest of the TLCC and Red Sky teams for their help.

References

[1] H. Simon, T. Zacharia, R. Stevens, Modeling and simulation at the exascale for energy and the environment. Technical Report <http://www.er.doe.gov/ascr/ProgramDocuments/ProgDocs.html>, DOE Office of Science, Advanced Scientific Computing Research, 2007.
[2] M.A. Heroux, Design issues for numerical libraries on scalable multicore architectures, Journal of Physics: Conference Series 125 (2008) 1–11.
[3] P.T. Lin, J.N. Shadid, M. Sala, R.S. Tuminaro, G.L. Hennigan, R.J. Hoekstra, Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling, Journal of Computational Physics 228 (2009) 6250–6267.
[4] J.N. Shadid, R.P. Pawlowski, J.W. Banks, L. Chacon, P.T. Lin, R.S. Tuminaro, Towards a scalable fully-implicit fully-coupled resistive MHD formulation with stabilized FE methods, Journal of Computational Physics, submitted for publication.
[5] A. Quarteroni, A. Valli, Domain Decomposition Methods for Partial Differential Equations, Oxford University Press, Oxford, 1999.
[6] B. Smith, P. Bjorstad, W. Gropp, Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations, Cambridge University Press, 1996.
[7] M. Sala, L. Formaggia, Algebraic coarse grid operators for domain decomposition based preconditioners, in: P. Wilders, A. Ecer, J. Periaux, N. Satofuka, P. Fox (Eds.), Parallel Computational Fluid Dynamics – Practice and Theory, Elsevier Science, The Netherlands, 2002, pp. 119–126.
[8] M. Sala, J.N. Shadid, R.S. Tuminaro, An improved convergence bound for aggregation-based domain decomposition preconditioners, SIAM Journal of Matrix Analysis 27 (3) (2006) 744–756.
[9] M. Sala, R.S. Tuminaro, A new Petrov–Galerkin smoothed aggregation preconditioner for nonsymmetric linear systems, SIAM Journal of Science and Statistics 31 (2008) 143–166.
[10] J.L. Tomkins, R. Brightwell, W.J. Camp, S. Dosanjh, S.M. Kelly, P.T. Lin, C.T. Vaughan, J. Levesque, V. Tipparaju, The Red Storm architecture and early experiences with multi-core processors, International Journal of Distributed Systems and Technologies 1 (2) (2010) 74–93.
[11] Kevin M. Kramer, W. Nicholas, G. Hitchon, Semiconductor Devices, A Simulation Approach, Prentice Hall PTR, 1997.
[12] S.M. Sze, Physics of Semiconductor Devices, second ed., John Wiley & Sons, 1981.
[13] T.J.R. Hughes, A. Brooks, A theoretical framework for Petrov–Galerkin methods with discontinuous weighting functions: application to the streamline-upwind procedure, in: R.H. Gallagher et al. (Eds.), Finite Elements in Fluids, vol. 4, John Wiley & Sons, 1982, pp. 47–65.
[14] T.J.R. Hughes, M. Mallet, A. Mizukami, A new finite element formulation for computational fluid dynamics: II. Beyond SUPG, Computer Methods in Applied Mechanics and Engineering 54 (1986) 341–355.
[15] F. Shakib, Finite element analysis of the compressible Euler and Navier–Stokes equations. Ph.D. Thesis, Division of Applied Mathematics, Stanford University, 1989.
[16] M. Sharma, G.F. Carey, Semiconductor device simulation using adaptive refinement and flux upwinding, IEEE Transactions on Computer-Aided Design 8 (6) (1989) 590–598.
[17] G.F. Carey, A.L. Pardhanani, S.W. Bova. Advanced numerical methods and software approaches for semiconductor device simulation, Technical Report SAND2000-0763J, Sandia National Laboratories, 2000.
[18] D.A. Knoll, D.E. Keyes, Jacobian-free Newton–Krylov methods: a survey of approaches and applications, Journal of Computational Physics 193 (2004) 357–397.
[19] J.N. Shadid, A.G. Salinger, R.P. Pawlowski, P.T. Lin, G.L. Hennigan, R.S. Tuminaro, R.B. Lehoucq, Large-scale stabilized FE computational analysis of nonlinear steady-state transport/reaction systems, Computer Methods in Applied Mechanics and Engineering 195 (2006) 1846–1871.
[20] P.T. Lin, M. Sala, J.N. Shadid, R.S. Tuminaro, Performance of fully coupled algebraic domain decomposition preconditioners for incompressible flow and transport, International Journal for Numerical Methods in Engineering 67 (2006) 208–225.
[21] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, 2003.
[22] V. Simoncini, D.B. Szyld, Recent computational developments in Krylov subspace methods for linear systems, Numerical Linear Algebra with Applications 14 (1) (2007) 1–59.
[23] R.S. Tuminaro, C.H. Tong, J.N. Shadid, K.D. Devine, D.M. Day, On a multilevel preconditioning module for unstructured mesh Krylov solvers: two-level Schwarz, Communications in Numerical Methods in Engineering 18 (2002) 383–389.
[24] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, M.G. Sala, ML 5.0 smoothed aggregation user's guide, Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[25] B. Hendrickson, R. Leland, The Chaco user's guide–version 1.0, Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque NM, 87185, 1993.
[26] O. Axelsson, Iterative Solution Methods, Cambridge University Press, New York, 1994.
[27] J.N. Shadid, R.S. Tuminaro, K.D. Devine, G.L. Henningan, P.T. Lin, Performance of fully-coupled domain decomposition preconditioners for finite element transport/reaction simulations, Journal of Computational Physics 205 (1) (2005) 24–47.
[28] U. Trottenberg, C. Oosterlee, A. Schüller, Multigrid, Academic Press, London, 2001.
[29] William L. Briggs, Van Emden Henson, Steve McCormick, A Multigrid Tutorial, second ed., SIAM, Philadelphia, 2000.
[30] M. Sala. Domain Decomposition Preconditioners: Theoretical Properties, Application to the Compressible Euler Equations, Parallel Aspects. Ph.D. Thesis, EPFL, Lausanne, Switzerland, 2003.
[31] M. Sala, P.T. Lin, J.N. Shadid, R.S. Tuminaro, Algebraic multilevel preconditioners for non-symmetric PDEs on stretched grids, in: Olof B. Widlund, David E. Keyes (Eds.), Domain Decomposition Methods in Science and Engineering XVI, Lecture Notes in Computational Science and Engineering, vol. 55, Springer-Verlag, 2006, pp. 741–748.
[32] P. Vaněk, J. Mandel, M. Brezina, Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems, Computing 56 (1996) 179–196.
[33] P. Vaněk, M. Brezina, J. Mandel, Convergence of algebraic multigrid based on smoothed aggregation, Numerische Mathematik 88 (2001) 559–579.
[34] R. Tuminaro, J. Hu, M. Gee, C. Siefert, M. Sala, ML: Multilevel preconditioning package: Web page, 2009. <http://trilinos.sandia.gov/packages/ml/>.
[35] J. Ruge, K. Stüben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, vol. 3, SIAM, Philadelphia, PA, 1987, pp. 73–130.
[36] G. Karypis, V. Kumar, Parallel multilevel *k*-way partitioning scheme for irregular graphs, in: ACM/IEEE Proceedings of SC96: High Performance Networking and Computing, 1996.
[37] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, An Overview of Trilinos, Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
[38] M. Heroux, Epetra linear algebra services package, Web site: <http://trilinos.sandia.gov/packages/epetra>, April 2008.
[39] J.N. Shadid, S.A. Hutchinson, G.L. Hennigan, H.K. Moffet, K.D. Devine, A.G. Salinger, Efficient parallel computation of unstructured finite element reacting flow solutions, Parallel Computing 23 (1997) 1307–1325.
[40] H. Tadano, Acceleration of convergence by block Krylov subspace methods on multi-core processors, in: Talk at International Conference on Finite Elements in Flow Problems (FEF09), Tokyo, Japan, April 1–3, 2009.
[41] M. Sala, M. Heroux, Robust algebraic preconditioners with IFPACK 3.0, Technical Report SAND2005-0662, Sandia National Laboratories, Albuquerque, NM, 87185, 2005.
[42] R.S. Tuminaro, M. Heroux, S.A. Hutchinson, J.N. Shadid, Aztec user's guide-version 2.1, Technical Report SAND99-8801J, Sandia National Laboratories, Albuquerque NM, 87185, Nov. 1999.
[43] M. Heroux, AztecOO user guide, Technical Report SAND2007-3796, Sandia National Laboratories, 2007.
[44] T.A. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006.
[45] G. Karypis, V. Kumar, ParMETIS: Parallel graph partitioning and sparse matrix ordering library, Technical Report 97-060, Department of Computer Science, University of Minnesota, 1997.
[46] P.T. Lin, J.N. Shadid, R.S. Tuminaro, M. Sala. Performance of a Petrov-Galerkin algebraic multilevel preconditioner for finite element modeling of the semiconductor device drift–diffusion equations. in press IJNME, published online in Wiley InterScience (<www.interscience.wiley.com>) Doi:10.1002/nme.2902, 2010.
[47] P.T. Lin, J.N. Shadid, Performance of an MPI-only semiconductor device simulator on a quad socket/quad core InfiniBand platform, Technical Report SAND2009-0179, Sandia National Laboratories, January 2009.
[48] C.G. Baker, M.A. Heroux, H.C. Edwards, A.B. Williams, A light-weight API for portable multicore programming, in: Proceeding of PDP2010, IEEE, 2010.