RC 20436 (4/17/96)
Computer Science/Mathematics 18 pages

Research Report

Scalability of Sparse Solvers in the Context of Semiconductor Device Simulation Applications

Vijay K. Naik, Kalluri Eswar

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

MeiKei Ieong

IBM Microelectronics Division
Hopewell Junction, NY 12533

IBM **Research Division**
Almaden · T. J. Watson · Tokyo · Zurich

Scalability of Sparse Solvers in the Context of Semiconductor Device Simulation Applications*

Vijay K. Naik Kalluri Eswar MeiKei Ieong

IBM Research Division
P. O. Box 218
Yorktown Heights, NY 10598
{vkn, eswar}@watson.ibm.com

IBM Microelectronics Division
1580 Rt 52
Hopewell Junction, NY 12533
mkieong@vnet.ibm.com

Abstract

In this paper, we present our preliminary experience in parallelizing IBM's FIELDAY code which is a general purpose finite element code for simulating two- and three-dimensional semiconductor devices using the conventional drift-diffusion equations. We discuss some of the key features of our implementation, focusing primarily on the handling of the irregular grid computations and on the solution of the linear systems. In this implementation, the linearized systems are solved by preconditioned (using incomplete LU) conjugate gradient squared (CGS) type of iterative methods. Our preliminary performance data indicates that while scalability on a per iteration basis is within acceptable range for sparse, irregular problems, the overall scalability is poor. Our experiments so far indicate that the primary reason for the overall poor performance is the inherent tradeoff seen between the available parallelism and the convergence rate of the iterative methods used. By making the grid partitioner aware of the underlying physical processes involved in the device being simulated, we hope to improve the convergence behavior without compromising on the extractable parallelism.

*To appear in: Proceedings of the NASA Semiconductor Device Modeling Workshop, March 1996.

This page intentionally left blank.

Scalability of Sparse Solvers in the Context of Semiconductor Device Simulation Applications

Vijay K. Naik Kalluri Eswar MeiKei Ieong

IBM Research Division
P. O. Box 218
Yorktown Heights, NY 10598
{vkn, eswar}@watson.ibm.com

IBM Microelectronics Division
1580 Rt 52
Hopewell Junction, NY 12533
mkieong@vnet.ibm.com

Abstract

In this paper, we present our preliminary experience in parallelizing IBM's FIELDAY code which is a general purpose finite element code for simulating two- and three-dimensional semiconductor devices using the conventional drift-diffusion equations. We discuss some of the key features of our implementation, focusing primarily on the handling of the irregular grid computations and on the solution of the linear systems. In this implementation, the linearized systems are solved by preconditioned (using incomplete LU) conjugate gradient squared (CGS) type of iterative methods. Our preliminary performance data indicates that while scalability on a per iteration basis is within acceptable range for sparse, irregular problems, the overall scalability is poor. Our experiments so far indicate that the primary reason for the overall poor performance is the inherent tradeoff seen between the available parallelism and the convergence rate of the iterative methods used. By making the grid partitioner aware of the underlying physical processes involved in the device being simulated, we hope to improve the convergence behavior without compromising on the extractable parallelism.

1 Introduction

Computer-aided numerical modeling has become an indispensable tool in the design, development, and understanding of semiconductor devices [9, 5, 3, 10]. Traditionally, the laboratory development of semiconductor devices has involved a large number of empirical design processes, with many iterations of the fabrication stage to achieve the desired specification. Numerical device modeling and simulation can substantially reduce such costly iterations by allowing the device design engineer to alter relevant physical parameters prior to the fabrication stage.

In device simulations, the drift-diffusion (DD) model has been quite effective in modeling most carrier transport. However, as the dimensions of today's semiconductor devices continue to shrink, it becomes necessary to consider non-local and non-equilibrium effects that influence both device performance and reliability. Moreover, other effects such as gate and substrate currents in Metal Oxide Semiconductor Field Effect Transistor (MOSFET) devices require additional information about the carrier energy and impact ionization rate. Such information can be rigorously obtained from the solution of the Boltzmann Transport Equation (BTE):

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{r}} f + \frac{\vec{F}}{\hbar} \cdot \nabla_{\vec{k}} f = \left(\frac{\delta f}{\delta t}\right)_{coll} \qquad (1)$$

where $\vec{v}$ is the particle velocity and $\vec{F}(\vec{r})$ is the force exerted on the particle at location $\vec{r}$. The right hand side of equation (1) represents the total rate at which the charged carrier distribution function f is changing due to collisions.

The most accurate numerical method to solve the BTE is the Monte Carlo (MC) particle simulation performed in both momentum and real space [6]. An advantage of this technique is that one can have a complete picture of carrier dynamics with reference to microscopic material parameters such as energy band structure and coupling constants for scattering mechanisms. DAMOCLES (Device Analysis using Monte Carlo et Poisson Solver) [11] is a state-of-the-art Monte Carlo based device simulator developed at IBM. This program has been used to simulate two-dimensional device structures fabricated using both silicon and compound semiconductor materials. Devices simulated include MOSFET, bipolar junction transistor (BJT), hot electron mobility transistor (HEMT), and Metal Semiconductor Field Effect Transistor (MESFET). However, there currently are some practical difficulties in using the MC method for simulating the future generation devices. First of all, the MC method is ineffective for simulation under low voltage conditions due to the slow convergence to self-consistent solutions with Poisson's equation in the low-field and energy barrier regions. Also, since the treatment of the long time-scale SRH-like generation-recombination effect in MC simulation is difficult, there is no satisfactory MC-based approach to model device characteristics that require the inclusion of this effect. Moreover, this approach is computationally demanding and is still considered to be too time consuming and thus, uneconomical to be used by the day-to-day device designer.

An alternative is the hydrodynamic (HD) model which consists of the conservation equations for particle density, average momentum, average energy, and, in some advanced models, average energy flux [8]. For today's technology development, the use of the HD model can provide a reasonable compromise between the physical accuracy achieved and the computational resources required. The HD model has been adapted in many popular device simulators such as FIELDAY [2], MINIMOS [15], and PISCES [12].

Even when the HD models are used, the need for greater speed in technology development and for cost effectiveness necessitates the use of parallel and distributed computing systems. At the same time, semiconductor technology has become increasingly complex. New physical effects that have been ignored in previous generations of technology must now be considered. A practical device simulator must provide not only more physical accuracy but also shorter turnaround time. Improved physical accuracy can be achieved by upgrading the physical models, by adaptively refining grids, and by precisely describing the three-dimensional device geometry. Parallelization is the best strategy to reduce the total wall clock time without sacrificing physical accuracy. Using reliable algorithms to support improved physical models and developing efficient parallel implementations to tackle large problems faster are the key issues for future device simulators.

Keeping the above developments in mind, we are in the process of studying techniques for the efficient parallel implementation of two- and three-dimensional semiconductor device simulations. Our initial goal is to increase the understanding (from a parallel processing point of view) of the computations involved in the simulations based on the DD and the HD device models. For this we are using IBM's FIELDAY II code which is a general purpose finite element code for simulating two- and three-dimensional semiconductor devices. FIELDAY II incorporates both the DD and HD models mentioned above. In this paper, we describe some of our experience in parallelizing the computations in FIELDAY II that lead to a steady-state solution to the DD equations using DC analysis. We primarily focus on the solver part of these computations and describe in some detail how we handle the irregular computations and unstructured data structures. In the rest of the paper, we refer to FIELDAY II simply as FIELDAY.

For parallelizing FIELDAY we have used the message passing, single program multiple data (SPMD) model of parallel computation and have used the MPI protocol for interprocessor communication. At present our target architecture is IBM's SP2 system. In this paper, we report performance data that we have gathered recently on a 16-processor SP2 system with thin nodes. We note here that although our current work is targeted for the SP2 architecture, the implementation is portable to other parallel and distributed systems supporting the MPI message passing protocol.

The organization of the rest of the paper is as follows. In the next section, we present some additional background information on the problem being solved and the overall structure of FIELDAY that is required to solve this problem. In Section 3, we describe, in some detail, our parallelization techniques to handle the irregular computations and unstructured data structures that characterize FIELDAY. We then present some preliminary performance data that we have gathered so far using a variety of 2D and 3D test cases representing N-FET devices. We observe not only the usual computation and communication tradeoff, but also the tradeoff in the quality of the convergence rate obtained and the degree of parallelism applied. We are in the process of understanding the complex relationships between the physical processes, their numerical representations, the convergence rate of the solution methods used, and the decomposition of work for parallel processing. We conclude the paper in Section 5 with observations on the effects of some of these parameters on the overall performance.

2 Some Preliminaries

In this section, we first briefly describe the drift-diffusion equations that are solved in a device simulation experiment. We then present a high-level overview of the computational structure incorporated in FIELDAY to conduct these simulations. As mentioned in the introductory section of this paper, we focus only on the steady state solution to the drift-diffusion equations using DC analysis. We refer to this as the DC solution phase. We further restrict our discussion to the steps relevant to the use of iterative solvers.

2.1 Problem Description

The drift-diffusion formulation used in FIELDAY for describing carrier transport is given by the following equations:

$$
\begin{aligned}
-\varepsilon \nabla^2 \psi &= q(p - n + N_D^+ - N_A^-) + \rho_f, &(2)\\
-q\frac{\partial}{\partial t}(n - N_D^+) &= -\nabla \cdot \vec{J_n} + R(\psi, n, p), &(3)\\
q\frac{\partial}{\partial t}(p - N_A^-) &= -\nabla \cdot \vec{J_p} - R(\psi, n, p). &(4)
\end{aligned}
$$

The quantities to be solved for are ψ, n, and p, the electrostatic potential and electron and hole carrier concentrations, respectively. N_D^+ and N_A^- are the donor and acceptor impurity ion concentrations. $\vec{J_n}$ and $\vec{J_p}$ are the electron and hole current densities. ρ_f is the fixed charge density, and $R(\psi, n, p)$ represents recombination-generation terms.

The electron and hole current density equations (not shown here) can be effectively discretized by the Scharfetter-Gummel (S.-G.) discretization scheme [14]. The above nonlinear system of equations are solved at several bias points where each bias point specifies the voltages or currents applied at the contacts. We note here that when the HD formulation is used, the energy flux equations are discretized using an extension of the S.-G. scheme proposed by Rudan and Odeh [13] and by Forghieri et al. [7].

2.2 Structure of the DC solution phase in FIELDAY

In the following, we briefly describe the setup steps leading to the DC solution phase as well as the structure of the computations in the DC solution phase of FIELDAY. The DC solution phase determines the steady state solution to the discretized DD equations described above at several bias points, where each bias point specifies the voltages or currents applied at the contacts. At each bias point, the Newton method is applied iteratively to solve the discretized system of nonlinear equations for the variables ψ, n, and p. In each Newton step, first the system is linearized, then a linear system representing an update to ψ, n, and p is solved, and finally the solution vector is updated. We denote the coefficient matrix of the linear system solved by A, and the right hand side by b. The most successful iterative solvers suitable for solving the unsymmetric linear systems arising from discretized device simulation equations are variants of biconjugate gradient methods. The most commonly used methods are CGS (conjugate gradient squared) and BiCGSTAB with preconditioning. In FIELDAY, the preconditioning is done by using incomplete LU (ILU) factorization of A to produce lower and upper triangular matrices L and U, respectively. The fill-in allowed in the preconditioning matrix is determined by a preset fill-level.

Prior to performing the DC solution, a few setup steps need to be performed. In addition, the symbolic structures of the sparse matrices used in the solution phase are also determined in the setup phase. Thus, the setup phase involves the following steps.

Input: This step involves setting up the geometrical mesh, doping, and contact information for the problem to be solved. Typically, this information is created separately and is stored in a database. In this step, this information is read in to setup the mesh and initialize various parameters of the problem. The Cartesian coordinates of each node in the mesh and the list of nodes constituting each finite element of the mesh are some of the main data read in this step.

Adjacency Matrix Structure: In this step, a sparse matrix representation of the mesh is determined. This step involves using the finite element information from the input step to construct an adjacency structure of the mesh. The adjacency structure determined in this step is used as the A matrix structure when solving for one or all of three variables ψ, n, or p in the DD equations.

"A" Matrix Structure: When a coupled system for all three variables (ψ, n, and p) is to be solved, the structure of the coefficient matrix A in the linear system is constructed from the adjacency structure of the mesh, but each element is now a 3×3 block matrix. The matrix A contains one row for each variable at each node.

Preconditioner Matrix Structure: In this step, the structure of the preconditioner is determined when incomplete LU factorization is used as a preconditioner in the solution of the linear system. L and U are triangular sparse matrices, whose structures typically do not change across the Newton steps. A symbolic incomplete LU factorization is performed on the A matrix structure. Fill-in is controlled by discarding nonzeros whenever the level of fill exceeds a preset threshold.

The first two steps of the setup phase are performed only once for each device to be simulated. The symbolic structure of A (computed in the third step above) also does not change from one Newton step to next nor does it change from one bias point computations to the next bias point computations. The symbolic structure of the ILU matrix, however, can change depending on the ill-conditioning of the problem. Although the actual magnitude of the entries in the ILU matrix are not taken into account in determining which fill to keep and which one to discard, depending on the nature of the problem, the fill-level threshold may be changed during the course of the execution. When the threshold is changed, the fourth step described above is performed again.

Each Newton iteration in the solution phase consists of the following three steps:

Matrix Assembly: In this step, the coefficient matrix A and the right hand side b are assembled. We refer to this step as the JCB-&-RHS step (JCB for Jacobian).

ILU Factorization: In this step, the preconditioner is determination by incomplete LU (ILU) factorization of A. We refer to this step as the ILU step.

Iterative Solve: In this step, an iterative solution of $Ax = b$ using a preconditioned conjugate gradient (PCG) method is obtained. We refer to this step as the PCG step.

In our parallelization effort, we have primarily focused on the parallelization of the solution phase which dominates the computations. In the following section, we describe our methodology in parallelizing the solution phase.

3 Parallelization of the Solution Phase

For parallelizing the solution phase, we distribute data and computations among the processors based on the partitioning of the vertices of the finite element mesh among the processors. In the following, we first describe the mesh partitioner we used. After that we describe the parallel implementation of the three steps (JCB-&-RHS, ILU, and PCG) in the solution phase.

3.1 Mesh partitioning and vertex reordering

To distribute work among multiple processors, it is necessary to partition the irregular mesh resulting from the finite element discretization. The partitioner should be able to divide work among processors as evenly as possible and at the same time it should minimize the number of edges across partitions so that communication cost during the parallel execution can be kept low. Determining the optimum partitioning of a general graph in this manner is an NP-complete problem. We use a heuristic called recursive spectral bisection (RSB) [1]. In the following, we briefly describe the algorithm used by this partitioning method.

Let G be the undirected graph representing the finite element mesh connectivity. The objective is to divide the vertices of G into P partitions, where P is the number of processors. Assuming that $P = 2^k$, the recursive spectral bisection technique performs k levels of recursive bisections of the graph. Each bisection subdivides a set of vertices into two partitions, thus doubling at each level the number of partitions created.

Each bisection is performed as follows. The discrete Laplacian Lap^G of the graph G to be partitioned is constructed. Letting $v_1, v_2, \ldots, v_n$ denote the vertices of G, Lap^G is an $n \times n$ matrix defined by:

$$\mathsf{Lap}^G_{ij} = \begin{cases} -\mathsf{degree}(v_i), & \text{if } i = j, \\ 1, & \text{if the edge } (v_i, v_j) \text{ exists}, \\ 0, & \text{otherwise.} \end{cases}$$

The eigenvector corresponding to the second largest eigenvalue of Lap^G is used by the algorithm to partition the vertices. The partitioning is done by sorting the list of nodes according to the weight of the corresponding value in the eigenvector, and splitting the sorted list into two parts of equal size.

The partitioner returns a color for each vertex of the graph G, where the color ranges from 0 to $P-1$. The current parallel version uses the color value to reorder the mesh nodes so that all nodes with color 0 are numbered first, followed by nodes with color 1, and so on.

3.2 Parallelization of the assembly step

The assembly step deals with a number of arrays containing values of various physical quantities associated with the vertices and edges in the finite element mesh. The values in arrays containing one value per mesh vertex (*vertex-arrays*) are distributed among the processors based on how the vertices are assigned to the processors by the partitioner. Arrays containing values associated with the edges in the mesh (*edge-arrays*) are distributed based on the distribution of the vertices connected by the edges. Each mesh vertex has a set of rows in the matrix A that correspond to it. These rows are stored on the processor assigned to the corresponding vertex. The right hand side vector b is distributed similarly. In all cases, the array subscripts on each processor start from 1 (the same as in the sequential code) and range up to the number of elements of the array that need to be stored on the processor. The local indices that a processor uses to refer to values will, in general, be different from the global indices used in the sequential program. Maps between these two index spaces have to be maintained on each processor.

Some of the vertex-arrays are completely distributed, in the sense that the sum of the sizes of each processor's local portion of the array would equal the size of the original sequential array. This is because each processor stores the values corresponding only to vertices assigned to it. Some vertex-arrays, however, have the property that a processor would need the values at vertices that are neighbors in the mesh of the vertices assigned to that processor. Since the vertices are partitioned among the processors, some of these neighboring vertices may be assigned to other processors. The approach taken in the parallel version is to allocate space in the nodal arrays for keeping copies of such "alien" values. The sum of the sizes of such a vertex-array over all processors would, in general, be more than the size of the original array in the sequential program.

An example will help illustrate the correspondence between the original vertex-array in the sequential version and the arrays stored on each processor in the parallel version. Figure 1 shows a mesh with six vertices, which is partitioned among two processors. Associated with each vertex is an index of the form $i(\mathbf{i_0},\ i_1)$. The number i which is outside the parenthesis, gives the global index associated with that vertex. It is global in the sense that this index is assigned to that vertex when all vertices in the mesh are ordered using some global numbering scheme. We also refer to the global index as the original index or as the natural index. The sequential program uses this index to access array values corresponding to that vertex. The number $\mathbf{i_0}$ is the index of that vertex as per the local indexing scheme used by processor P_0. Similarly, the number i_1 is the index of the same vertex as per the local indexing scheme used by processor P_1. A '-' indicates that the vertex is not considered by the indexing scheme on the corresponding processor.

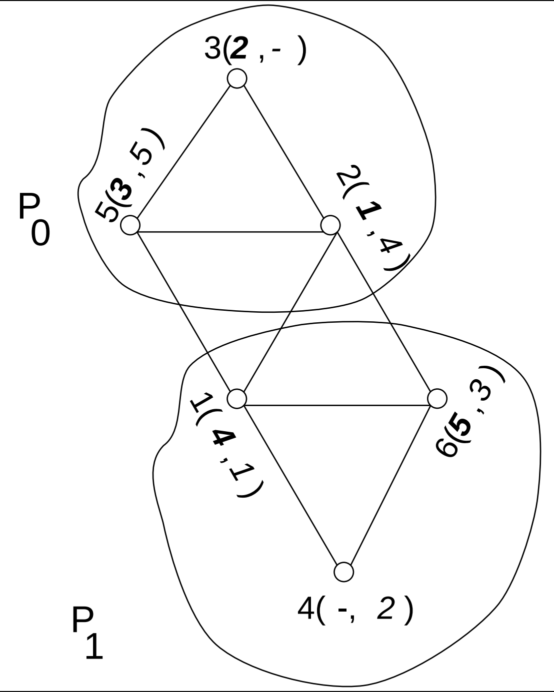


Figure 1: Sample mesh

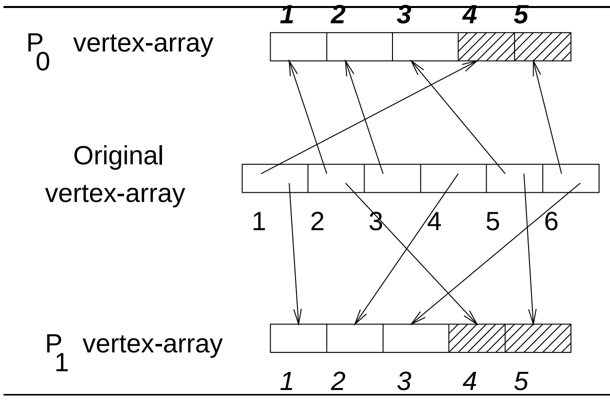


Figure 2: Vertex-arrays

It can be seen that processors P_0 and P_1 both own three vertex values each, and that each needs two values from vertices assigned to the other processor. Figure 2 shows the vertex-arrays on each of the processors. The first three locations of the array on each processor are used for storing the values of vertices assigned to that processor. For example, processor P_0 uses locations 1 through 3 for storing the values of the vertices originally numbered 2, 3, and 5, respectively. Therefore, the local index 1 on P_0 corresponds to the global index 2. This is illustrated by the arrows going from the locations in the original array to the vertex-array on processor P_0. The alien values needed by each processor are stored at locations following the owned values. For example, processor P_0 needs values for vertices whose original indices are 1 and 6. Locations 4 and 5 of its vertex-array are used to store these values. These indices (4 and 5) are therefore the local indices by which processor P_0 refers to the alien values.

Edge arrays are distributed in the following manner. Such arrays actually have two values corresponding to each undirected edge in the mesh. With respect to a vertex on which an edge is incident, one value may be considered to be outgoing and the other value incoming. The outgoing value may be associated with this vertex and the incoming value may be associated with the neighboring vertex (it would be considered an outgoing value of the neighboring vertex). With this view, the edge arrays are partitioned among the processors according to the way the vertices are partitioned: each processor stores the "outgoing" edge values of vertices that have been assigned to it. As mentioned above, the subscripts are made to start from 1 for all the distributed arrays, so that the edge values relevant to a processor will be stored in sequence in the distributed version. The nature of the code is such that, with very few exceptions, the edge values of neighboring vertices are not required directly during computations involving vertices. This means that there is no need for storing "alien" values in the case of edge-arrays, making them fully distributed with no memory overhead.

```
for each vertex i do
    ...
end for
```

```
for each vertex i do
    for each outgoing edge of vertex i do
        ...
    end for
end for
```

Figure 3: Common loop types

The work involved in the assembly step is distributed among the processors according to the distribution of the mesh vertices. The iterations of the loops in the program are divided among the processors such that each processor executes iterations relevant to data that it owns. The data parallelism in the assembly step is exploited in this manner. Two of the common types of loops present in the routines constituting the assembly step are shown in Figure 3. The singly-nested loop involves a pass through all the vertices in the mesh. The doubly-nested loop involves a pass through all the edges in the mesh. The latter is usually done by scanning the neighbor lists of all vertices in the mesh. The parallel version distributes the i loop in both cases. The structure of the parallel loops, and how the array subscripts within the loops are made to be consistent with the local indices on each processor, can be illustrated as follows.

```
for i ← 1 to num_vertices do
    vertex_array2(i) ← f1(vertex_array1(i))
    for j ← nei_start(i) to nei_start(i+1) − 1 do
        edge_array(j) ←
            f2(vertex_array1(i), vertex_array1(nei_list(j)))
    end for
end for
```

Figure 4: Sample loop nest

Consider the sample sequential loop nest shown in Figure 4. This loop nest's structure contains features of both the kinds of loops mentioned above. vertex_array1 and vertex_array2, as their names indicate, are vertex-arrays, with vertex_array2's values being set using the corresponding values in vertex_array1. edge_array is an edge-array, whose values are set using the two neighboring nodal values in vertex_array1. The neighbors of each vertex are listed in the array nei_list, whose size is the same as that of edge_array. The array nei_start indicates the index in nei_list where the neighbor list of each vertex begins.

The parallel version of the sample loop nest differs from the sequential version only in the extent of the outer loop. The range is limited to the number of vertices assigned to each processor (num_own_vertices). The outer loop index gives the local index of the owned vertex under consideration. Since vertex_array1 and vertex_array2 are distributed and have their subscripts range from 1 to the number of owned vertices, the assignment in the outer loop involving these two arrays does not have to be altered in the parallelization process. The key to making the array subscripts in the inner loop consistent with the distributed arrays is to distribute the integer arrays nei_start and nei_list, and to have the appropriate set of indices in them. Figure 5 illustrates this using the same ex-

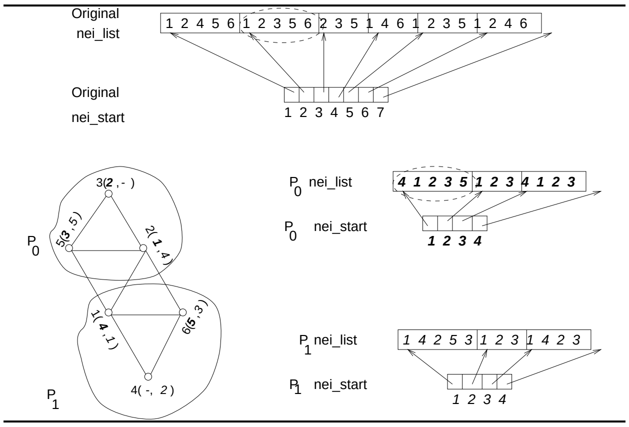


Figure 5: Neighbor lists

ample used earlier. The original nei_list and nei_start arrays are shown, as are the distributed versions of these arrays on processors P_0 and P_1. The distributed nei_start's subscripts are made to range from 1 to num_own_vertices+1, making them refer to local indices of owned vertices. The values in the distributed nei_start range from 1 to the number of edges going out of owned vertices. This is consistent with the manner in which edge_array is distributed, as described above, so that edge_array(j) means the same thing in both sequential and parallel versions. Finally, in order to make the reference vertex_array1(nei_list(j)) consistent with vertex_array's distribution, the distributed nei_list contains local rather than global indices of neighboring vertices. Moreover, owned neighboring values and alien neighboring values are accessed in the same manner, using the same subscript as the sequential version. For example, on processor P_0, when considering local vertex 1 (which is global vertex 2), its neighbor that is globally numbered 3 (locally numbered 2) is owned by the same processor. The local index 2 is put in P_0's nei_list since that is the local position in vertex-arrays such as vertex_array1 where that vertex's values are stored. Global vertex 1 is not owned by P_0, and has been given the position 4 in vertex-arrays for storing a copy of its value got from P_1. The local index 4 is therefore put in P_0's nei_list. In Figure 5, the global and local versions of the neighboring vertex list of global vertex 2 are circled.

The above discussion used edge-arrays to illustrate the parallelization of loops in the assembly step. Loops that deal with edge-arrays occur, for example, in mobility calculations. These and other arrays are intermediate arrays used to produce the final output of the assembly step: the A matrix and b vector. Loops where the A matrix values are assigned values often have a structure similar to that shown earlier in Figure 4. The techniques described above, distribution of work by partitioning the loop iterations and the use of local indices in indexing arrays, are also used for the loops involving the A matrix.

```
for i ← 1 to num_vertices do
    vertex_array(i) ← ···
end for
···
for i ← 1 to num_vertices do
    for j ← nei_start(i) to nei_start(i + 1) − 1 do
        ··· ← ··· vertex_array(nei_list(j)) ···
    end for
end for
```

Figure 6: Communication between loops

An important issue in the parallelization of the assembly step is ensuring that alien values used in the vertex-arrays are current. Communication is needed between the processors in order to maintain correctness. This is illustrated by the example in Figure 6, which shows two sequential loops. The second loop uses neighboring values of vertex_array, and since the first loop writes into vertex_array, fresh copies of alien values have to be obtained by each processor before their use in the second loop. Dependence analysis was done by hand in order to determine at what places and which vertex-arrays need to take part in the communication. The pattern of communication is determined once during the setup phase of the run, since the mesh structure is invariant.

3.3 Parallelization of the incomplete LU factorization

Incomplete LU factorization of the A matrix produces the L and U factors of the preconditioner. The nonzero structures of the L and U factors are predetermined during the setup phase. Entries that are common to the structure of A and the structure of L or U are copied from A and the rest are zeroed out so that the factorization proceeds in place. The sequential algorithm used is a row-based algorithm. Row k (the source row) updates row i (the target row) if L_{ik} is nonzero. Figure 7 shows an outline of the sequential algorithm.

The parallel incomplete LU factorization algorithm's computational structure is driven by the distrib-

```
Algorithm Sequential_ILU
    for i ← 1 to num_rows do
        for k < i such that L_ik ≠ 0 do
            Update row i using row k
        end for
    end for
```

Figure 7: Sequential incomplete LU factorization

ution of the rows of the matrix among the processors. The rows of L and U are distributed among the processors in the same manner as the rows of the matrix A. Each processor is responsible for carrying out all updates onto rows owned by it, and sends completed rows to other processors needing them for updating their owned rows.

An important issue is the order in which a row's sources update it. In the sequential semantics, a target row's sources update it in increasing order of their index. For example, if row i is updated by rows k_1, k_2, and k_3 ($k_1 < k_2 < k_3$), then row i is updated by the rows in the order k_1, k_2, k_3. If the parallel algorithm is allowed to carry out these updates in an arbitrary order, results that are not consistent with the sequential semantics may be obtained. This is best understood by means of the example. Assume that a nonzero exists in position $U_{k_1k_2}$. L_{ik_2} will be nonzero because of the premise that row k_2 updates row i. During the update of row i using row k_1, the value of L_{ik_2} will get modified by subtraction of the product $L_{ik_1} * U_{k_1k_2}$. L_{ik_2} is subsequently used during the update of row i using row k_2. Therefore, the update of i using k_2 is affected by the update of i using k_1. If the order of updates were reversed, the update of i using k_2 will not reflect the change in the element L_{ik_2}, and the results obtained will, in general, be different.

The current parallel version uses an algorithm that maintains the sequential semantics by enforcing the restriction that a target's sources update it in increasing order of their index. This is done by maintaining, for each target, the index of the next source that may update it at any instant. Notice that this could be stricter than is really necessary because the dependences may be such that two source rows that update the same target may be independent of each other. A precise analysis of the dependence structure may be used to maintain, for each target, the subset of its sources, any member of which may update it next.

In spite of restricting the order of updates onto a target, there is considerable parallelism in the incomplete LU factorization, since several targets may be updated simultaneously on different processors. The parallel algorithm also uses dynamic scheduling of updates based on which source rows become available (either from within the processor or from other processors). This allows targets to be considered based on the availability of their sources. A key feature of the algorithm is the notion of "relevant" rows. A relevant row on a processor is a source row that may be used to update one or more owned targets. Each processor maintains a queue of indices of relevant rows that have become available. As each relevant available row is removed from the queue, it is used on whatever owned targets it is allowed to update. Some of these updates may cause targets to become available for use as sources. Some of the updates may cause some other source to become relevant. When there are no more relevant rows available in the queue, the processor waits for some relevant row to arrive from another processor. Figure 8 shows an outline of the parallel algorithm executed on each processor.

The algorithm above does not show (for ease of presentation) an important feature of the actual implementation. When a row is completely updated, it is not necessarily sent out immediately. Several rows may be sent together in one message. A run-time parameter controls the desired size of messages. Proper use of this feature may result in significant lowering of the communication overhead.

3.4 Parallelization of the conjugate gradient solve

The preconditioned conjugate gradient solve of $Ax = b$ involves four main types of operations:

(a) vector norm

(b) vector dot product

(c) sparse matrix vector product

(d) sparse triangular solve

The first two operations are implemented by using reduction primitives. The sparse matrix vector product involves the A matrix, whereas the sparse triangular solves involve the L and U factors of the preconditioner.

The parallel sparse matrix vector multiply Av involves communication of values of the vector v. The rows of the A matrix are distributed among the processors, and the corresponding v values are present on the same processor. Let w denote the vector result of the

Algorithm Parallel_ILU
Send out owned rows with no updates onto them
Initialize set of relevant sources
Initialize queue of relevant and available sources
while not all owned rows are done **do**
    while queue is not empty **do**
        Remove source k from queue
        for each owned target i of k **do**
            if k is the next source to update i **then**
                Update row i using row k
                if i is completely updated **then**
                    Send row i to needy processors
                    if i is relevant **then**
                        Add i to set of relevant sources
                        Add i to queue
                    end if
                else
                    Determine the next source k' of i
                    Add k' to set of relevant sources
                    Add k' to queue if it is available
                end if
            end if
        end for
        if k is still relevant **then**
            Add k back to the queue
        end if
    end while
    Wait for a message containing some row k
    if k is relevant **then**
        Add k to the queue
    end if
end while

Figure 8: Parallel incomplete LU factorization algorithm

matrix vector multiply Av. Each processor computes the w values corresponding to rows in A owned by it. The first step in this process is an exchange among the processors of the relevant values in the vector v. The values of w_i are then computed by performing the dot products between row i of A and the vector v. The nonzero structure of row i determines which values of v are needed for this dot product. During the setup phase, the set of values to be communicated between the processors is predetermined.

The sparse triangular solves ($Lq = r$ and $Us = t$) are also preprocessed during the setup phase. The schedule of operations on each processor is predetermined. The schedule specifies when each owned equation is to be solved, when solution values are to be received and sent, and what processors to send to or receive from. Each processor's schedule depends on the schedule of the processors sending solution values to it. Machine parameters such as the cost of sending a word between processors relative to the cost of a floating point multiply/add are taken into account so as to allow the schedule computation to be more precise at predicting the order in which solution values will reach a processor. This influences the order in which equations are solved.

Table 1: Characteristics of the examples used.

Number of vertices	Number of edges	Number of rows	Nonzeros in A	Nonzeros in ILU
5,407	10,723	16,225	242,896	340,469
11,481	22,835	34,447	516,094	723,137
21,451	42,751	64,357	965,186	1,352,735
42,930	87,022	128,794	1,955,984	2,724,020
10,207	22,027	306,25	489,596	689,074
31,568	96,512	94,708	2,031,526	7,033,088

4 Preliminary Performance Results

In this section, we report performance results from our initial implementation of parallel FIELDAY. We will see from these results that the overall execution time is dictated by two variables: (i) total number of PCG iterations performed, and (ii) execution time of each PCG iteration. Both of these are affected by the ordering and partitioning methods used as well as by the number of partitions made. In the following, we present some preliminary observations from our recent studies on several 2-D and 3-D device simulations carried out on multiple processors. Observations like these are an important step towards increasing the understanding of the performance behavior of the nonlinear problems encountered in semiconductor device simulations.

All our parallel experiments were carried out on a 16-processor IBM SP2 with thin nodes at the IBM T. J. Watson Research Center. Each node of this system has a POWER2 model 390, RS/6000 processor with 64KB data cache, 66.5 MHz clock speed, 64-bit wide memory bus, and 128 MB memory. On the SP2, the processors are interconnected via a high performance switch, the details of which can be found in [17]. The processor-switch interface is managed by a special adapter card. Explicit message passing is the parallel programming paradigm. In our experiments, message passing was handled by the Parallel Operating Envi-

Table 2: Effect of partitioning the finite element graph on the ILU structure

Problem	Number of nonzeros in the ILU preconditioner				
	P=1	P=2	P=4	P=8	P=16
Preconditioning with level-1 fill					
5,407	340,469	345,866	348,992	353,798	360,743
11,481	723,137	733,013	736,199	742,820	753,590
21,451	1,352,735	1,357,460	1,363,454	1,370,432	1,382,459
42,930	2,724,020	2,735,228	2,745,209	2,760,626	2,810,399
10,207	689,074	693,226	696,610	705,082	717,097
Preconditioning with level-2 fill					
5,407	437,843	448,193	456,071	468,647	484,028
11,481	929,897	947,039	956,939	972,899	997,724
21,451	1,740,881	1,752,620	1,769,756	1,787,708	1,816,061
42,930	3,523,304	3,549,218	3,574,814	3,609,029	3,689,900
10,207	934,195	940,102	942,577	958,192	981,586
31,568	7,033,088	7,248,098	7,671,365	8,407,802	8,457,380

ronment (POE) using the MPI protocol [16].

4.1 Examples

For this study, we considered several 2-D and 3-D examples. For illustration purposes, we selected a few of these and they are listed in Table 1 along with some key characteristics. All our examples represent N-FET type devices. In the rest of the discussion, we refer to these by the number of vertices (shown in column 1) in the corresponding finite element mesh. The number of edges in the finite element mesh and the resulting number of variables (i.e., the number of rows in the coefficient matrix A) are shown in columns 3 and 4, respectively. Note that there are three variables at each vertex of the mesh and each problem we considered has four contact variables. Also shown in Table 1 are the number of nonzeros in the resulting coefficient matrix A and in its ILU factors. The structure of the L and U factors is obtained based on the natural ordering (assigned by a mesh generator) to the vertices of the finite element mesh. Among the examples listed in Table 1, 31568 is from a 3-D problem, and the remaining are from 2-D problems. The 5407, 11481, 21451, and 42930 correspond to the same problem with higher levels of mesh refinement. We use these four examples in our scalability study.

4.2 Effect of reordering on fill in ILU

As mentioned earlier in Section 3.1, we reorder the vertices according to the color assigned to them by the recursive spectral bisection (RSB) partitioner. We refer to such a reordering as the RSB ordering. Each processor configuration results in a separate RSB ordering. This reordering affects the amount of fill produced in the ILU factorization and also the bandwidth of the preconditioner. Shown in Table 2 are the number of nonzero entries in the ILU preconditioner allowing level-1 and level-2 fill when the vertices are ordered for partitioning the mesh into 1, 2, 4, 8, and 16 partitions*. Note that the amount of fill-in increases with the number of partitions although the ILU is computed using the same level. Comparing the fill when one processor is used with the total fill when sixteen processors are used, for the 2-D problems considered and for level-1 ILU, the increase in the fill is between 2% and 6% and, for the level-2 ILU, the increase is between 4% and 11%. Note that the proportion of the fill is higher as the partition size assigned to a processor gets smaller or when a smaller mesh is divided among larger number of processors (e.g., there

*Informally, in level-1 ILU, only the fills produced by eliminating the original non-zeros are retained but higher-level fill produced in the elimination of the level-1 fill is discarded. The definition can be extended to higher levels of fill.

is 6% increase in fill with the 5407 case whereas the increase in fill is only 3% with the 42930 case). For the 3-D case (31568 problem), we show the number of nonzeros for level-2 fill only since the problem does not converge with level-1 fill. We observe that the increase in fill for the 3-D problem is more than 20% when the number of partitions is increased from 1 to 16. Although the increase in the fill is moderate to high, the effect on the convergence rate is significant in all cases, as we will see in the following.

Table 3: Number of iterations (Newton & CG) after partitioning the finite element graph

P	Problem				
	5407	11481	21451	42930	10207
Preconditioning with level-1 fill					
1	(6,185)	(5,151)	(5,328)	(5,296)	(19,691)
2	(6,308)	(6,507)	(5,262)	(5,660)	(19,772)
4	(5,280)	(5,256)	(5,309)	(5,523)	(19,809)
8	(5,505)	(5,250)	(5,417)	(6,776)	(16,1451)
16	(5,334)	(5,532)	(6,552)	(5,916)	(15,1207)
Preconditioning with level-2 fill					
1	(7,151)	(6,212)	(5,246)	(5,194)	(19,723)
2	(6,302)	(6,298)	(5,225)	(5,806)	(19,706)
4	(5,229)	(6,376)	(5,309)	(5,676)	(18,633)
8	(6,417)	(5,336)	(6,307)	(5,503)	(20,1126)
16	(6,399)	(5,360)	(5,449)	(6,1168)	(18,1116)

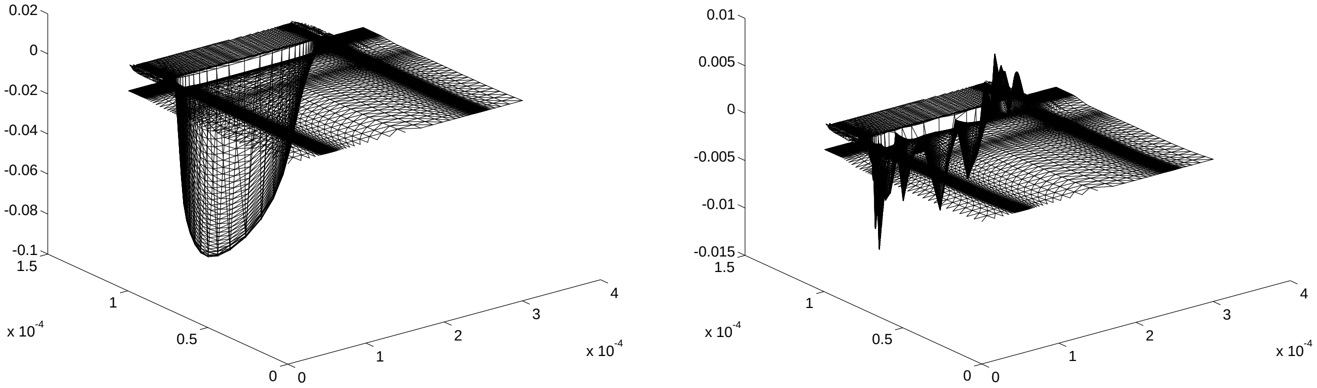
4.3 Effect of reordering on the convergence rate

In Table 3, we show the number of Newton iterations and the total number of PCG iterations required to solve a single bias point problem. These are presented as a pair (x, y) for each set of processors – x is the number of Newton iterations and y is the number of PCG iterations. Results for both level-1 ILU and level-2 ILU are shown. In all cases, the recursive spectral bisection is performed on the original mesh. We observe that the number of Newton iterations does not change significantly with the increase in the number of processors and in some cases there is a slight decrease, which we consider to be accidental, pending further investigation. However, as the number of processors is increased (equivalently, as the number of partitions is increased), the total number of PCG iterations increases and, in some cases, this increase is quite dramatic. When the number of partitions is increased from 1 to 16, with level-1 fill, the total number of PCG iterations increases anywhere from 68% (21451 case) to 250% (11481 case). With level-2 fill, the increase in the total number of PCG iterations is anywhere from 54% (10207 case) to 502% (42930 case). In other words, the convergence rate deteriorates and there is no obvious pattern. We note that the ordering effect on the convergence rate of PCG methods has been reported extensively in the literature (see, e.g. [4] and [18]). To our knowledge, no single satisfactory solution has been proposed for determining an ordering for finite element problems for unstructured meshes that is good both from the convergence rate and parallel efficiency points of view.

In the following, we further explore the reasons for the deterioration in the convergence rate with the increase in the number of processors. For this, we consider the 10207 case and compare the behavior of the solution of the linear system on 1 and 8 processors. The number of Newton iterations required on 1 and 8 processors are 19 and 16, respectively. We consider the 4th Newton iteration on 8 processors (127 PCG iterations) and the 5th Newton iteration on 1 processor (50 PCG iterations). Other than the fact that at these Newton iterations both 1 and 8 processors take the most number of PCG iterations, we have picked these iterations arbitrarily. These two cases do not necessarily correspond to the solution of the same linear system. We have made observations similar to those we make in the following at other Newton iterations as well.

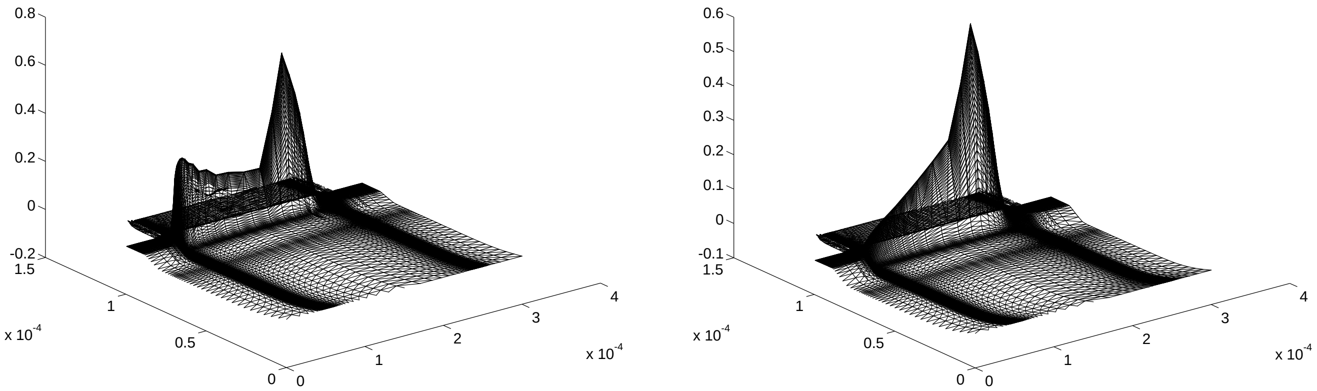
In both cases, we plot the difference in the final (converged solution) and the intermediate solution at various PCG iteration points. Note that the solution represents an update to ψ, n, or p. We consider only the updates to the ψ variable. These are shown in Figure 9 and Figure 10 for the 1 and 8 processors cases, respectively. We note that for the 1 processor case, the error terms have many frequency components (see the solution error at the end of iteration #21, Figure 9(b)) and these are eliminated rapidly. However, in the 8 processor case, there are a few dominant frequencies in the error and these persist over tens of PCG iterations.

We now consider the original mesh structure and the ordering that produced the 8 partitions. This is shown in Figure 11(a). The partition numbers are defined by the color coding scheme indicated at the top (left to right in increasing order). Within each partition, the vertices are ordered consecutively. A careful examination of the partitioning reveals that the RSB partitioner, which takes into account only the graph structure and not the numerical dependencies among vertices, has divided among all 8 processors the narrow band of finely discretized mesh in the gate re-



(a) (b)

Figure 9: Error in the updates to ψ solution at the end of (a) 11 iterations and at the end of (b) 21 iterations (Newton iteration 5, P=1).



(a) (b)

Figure 10: Error in the update to ψ solution at the end of (a) 11 iterations and at the end of (b) 61 iterations (Newton iteration 4, P=8).

gion between the source and the drain. (This includes the inversion layer.) This ordering and the associated partitioning produces a good computation and communication load balance, but prevents fast flow of information among the variables at adjacent vertices in the critical region. Reordering of the vertices and discarding of the resulting fill both contribute to the poor convergence rate.

To overcome this drawback (which is really a trade-off between convergence rate and computation/communication load balance), we are currently considering several other ordering and partitioning schemes. Following is a slight modification to our original strategy that improves the situation, but it is not our final solution.

The partitions carved out by the RSB partitioner affect communication costs incurred in the assembly phase, the ILU phase, and the PCG iterations (primarily the forward and back solves involving sparse triangular systems). Communication patterns in the ILU computation phase and the sparse triangular solves are determined by the structure of the L and U factors of the preconditioner and not by the structure of the original matrix A. Taking this into account, we modified our partitioning strategy. Instead of partitioning the vertices of the original mesh by considering only the original edges, we partitioned the vertices taking into account the original edges as well as those

Table 4: Effect of partitioning the preconditioner graph on the ILU structure

Problem	Number of nonzeros in the ILU preconditioner				
	P=1	P=2	P=4	P=8	P=16
Preconditioning with level-1 fill					
5,407	340,469	344,600	350,159	353,711	359,840
11,481	723,137	730,334	738,431	743,666	754,214
21,451	1,352,735	1,361,930	1,368,956	1,375,577	1,395,509
42,930	2,724,020	2,737,552	2,752,967	2,765,624	2,783,594
10,207	689,074	691,900	696,223	705,433	719,134
Preconditioning with level-2 fill					
5,407	437,843	447,254	456,731	467,912	482,831
11,481	929,897	941,531	962,351	984,398	1,012,145
21,451	1,740,881	1,757,324	1,779,482	1,798,484	1,840,868
42,930	3,523,304	3,547,028	3,579,338	3,606,572	3,662,747
10,207	934,195	942,097	954,142	982,264	1,010,578
31,568	7,033,088	7,306,064	7,540,358	7,595,282	7,799,660

Table 5: Number of iterations (Newton & CG) after partitioning the preconditioner graph

P	Problem				
	5407	11481	21451	42930	10207
Preconditioning with level-1 fill					
1	(6,185)	(5,151)	(5,328)	(5,296)	(19,691)
2	(6,308)	(5,283)	(5,341)	(5,443)	(19,716)
4	(5,298)	(5,296)	(5,313)	(5,434)	(18,1436)
8	(6,325)	(5,339)	(5,417)	(8,1271)	(16,1325)
16	(6,346)	(5,299)	(5,338)	*	(16,1353)
Preconditioning with level-2 fill					
1	(7,151)	(6,212)	(5,246)	(5,194)	(19,723)
2	(6,232)	(6,310)	(6,392)	(5,509)	(18,601)
4	(5,178)	(5,245)	(5,300)	(5,657)	(19,665)
8	(6,245)	(5,278)	(5,297)	(5,650)	(20,872)
16	(5,235)	(5,323)	(5,286)	(6,570)	(19,945)

$*$ – did not converge after 2000 CG iterations in Newton step 5

produced by the fill allowed by the fill-level. Again the RSB partitioner is used, and the vertices reordered according to their color. We refer to this as the RSB ordering of the preconditioner graph. The partitioning for 8 processors produced by this ordering is shown in Figure 11(b). The effect of partitioning the preconditioner graph on the number of nonzeros in the ILU (with level-1 and level-2 fill), for the examples we consider, is tabulated in Table 4. Comparing Tables 2 and Table 4, we note that for the 2-D cases, the differences in the nonzeros in the ILU are small (both level-1 and level-2 fills). However, for the 3-D case, there is a significant reduction in the fill added (from 20% to 11%) when the preconditioner graph is used for partitioning the mesh. We are further investigating the reasons behind this observation. The effect of this partitioning strategy on the number of Newton and PCG iterations is tabulated in Table 5. Comparing these iteration counts with those tabulated in Table 3, we note that, for the examples we considered, the new ordering reduces the number of PCG iterations. However, the total number of PCG iterations, in general, increases with the number of processors used. Comparing the PCG iteration count for P=1 and P=16 cases, we observe that the increase in the total PCG iteration count is anywhere from 3% (21451 case) to 98% (11481 case) with level-1 fill and from 16% (21451 case) to 193% (42930 case) with level-2 fill.

4.4 Parallel scalability with problem size

In most parallel applications, unless they are embarrassingly parallel, as the number of processors is increased, the overhead in solving the same problem gets larger and the performance deteriorates. Instead of looking at the performance of the same problem on larger number of processors, an accepted measure of performance for distributed systems is *scalability* of an implementation. Loosely stated, scalability is the ability to maintain the same level of performance on a larger set of processors as that on a smaller set, by increasing the problem size. Good scalable implementations (or algorithms) are those where the problem size needs to be increased linearly with the number of processors, the rationale being that most resources (e.g., memory) increase at most linearly with the number of processors. If the overheads because of insufficient parallelism and communication costs are such that the problem size must grow in a superlinear manner with the number of processors, then the implementation is not considered scalable. In this section, we present some results to estimate the goodness of our implementation of parallel FIELDAY.

For the scalability analysis we selected four problems: 5407, 11481, 21451, and 42930. All four cases represent the same N-FET type device with successively higher levels of mesh refinement (the number of vertices in the finite element mesh roughly doubles from 5407 through 42930). All involved single bias point computations and assumed the same mobility model. In our scalability experiments, we computed the 5407 problem on 2 processors, 11481 problem on 4 processors, 21451 problem on 8 processors, and 42930 problem on 16 processors. In all cases, a level-2 ILU preconditioner was applied. In these experiments, we measure the time spent in the three major components: computing the Jacobian and the RHS (JCB-&-RHS), computation of the ILU preconditioner, and the linear system solves using the PCG method including convergence tests for PCG and for the Newton iterations. The sum of the time spent in these three components is the total execution time. We ignore the time spent in the initial setup phase, which is relatively insignificant especially when multiple bias point simulations are carried out. Performance measurements from our experiments are presented in Table 6. All timings reported are in seconds. Shown in the third column of that table is the total execution time on the corresponding number of processors.

Although we are approximately doubling the problem size with the number of processors, we cannot directly compare the total execution time for each problem to establish the quality of scalability achieved by our current implementation. This is because the convergence rate for the Newton iterations as well as that for the PCG iterations differ as the problem size is changed and as the number of partitions is changed. For that reason, we normalize the time spent in the three major components by the number of Newton iterations and the number of PCG iterations. Listed in columns 8 and 9 of Table 6 are the costs per Newton iteration for computing JCB-&-RHS and the ILU, respectively. The cost per PCG iteration is shown in the final column. For each problem, we have also shown the memory requirements per processor in columns 4 and 5. Column 4 gives the peak memory requirement, whereas column 5 gives the memory requirement during the DC solution phase.

We observe from Table 6 that the computation time of the matrix assembly and right hand sides (JCB-&-RHS) remains approximately the same when the problem size and the number of processors are doubled. This indicates that the parallel computations in this step scale well. This is to be expected since the computations are data parallel and only the nearest neighbor communication is involved. Similarly, although not entirely data parallel, the ILU computations also seem to scale relatively well. However, when we compare the time spent in each PCG iteration, we observe that scalability is poor. Each PCG iteration involves two vector dot products, two matrix vector multiplies, four triangular solves, and two infinity norm computations. Of these, the first two are data parallel. The infinity norm computation, involving global reduction, is less data parallel, but is a relatively minor component. The sparse triangular solves, on the other hand, involve recurrence relations and are not data parallel. Moreover, the computation to communication ratio is also small giving rise to high overhead of parallelism.

We note here that in the above we have used the number of vertices in the mesh as a measure of computation cost associated with a problem. This is not entirely correct since the number of edges in the mesh or the sparsity structure of the matrices involved (both before and after preconditioning) plays an important role in the total computation time. For a complete understanding of scalability, one should take this into account. This shortcoming in our study is further underscored when we consider the memory requirement per processor for the four problems studied. The memory requirement per processor is shown in columns 4 and 5 of Table 6. Memory requirement grows approximately with the amount of work and the also with the overhead of parallelism. For the 42930 problem, the memory required per processor during the solution phase is higher by a factor of about 1.5 when compared to the problems solved on 4 and 8 processors. This indicates that the 42930 case incurs more work. Currently, we are analyzing this case further to understand whether this is useful work or is an over-

Table 6: Scalability results for 2-D meshes with triangular elements.

Mesh Size	Number of Processors	Total Exe. Time	Peak Mem. per Processor (MB)	DC Memory per Processor (MB)	# Newton Iterations	# PCG Iterations	Time per JCB & RHS	Time per ILU	Time per PCG Iteration
5,407	2	47.4	16.0	15.0	6	232	1.25	1.60	0.131
11,481	4	66.0	21.0	20.0	5	245	1.36	1.92	0.202
21,451	8	84.4	29.0	22.0	5	297	1.30	1.98	0.229
42,930	16	195.7	59.0	31.0	6	570	1.37	2.08	0.307

head resulting from parallelizing the computations.

4.5 Solution of large 3-D problems

We now consider the solution of two relatively large problems representative of real-life simulations. For this we used the test case 31568 which has right prismatic mesh elements. We consider two simulations, one involving computation of 34 bias points and the other involving 68 bias points. The first case involves determining the I_d–V_d output characteristics when the gate voltage is held constant at 1.1 V and and the drain voltage is varied from 0.0 V to 3.3 V in steps of 0.1 V. In the second case, I_d–V_g (linear) characteristics are determined by first holding the drain voltage constant at 0.1 V (for the first 34 bias points) while the gate voltage is varied from 0.0 V to 3.3 V in steps on 0.1 V; after that the drain voltage is raised to 3.3 V and the gate voltage is reduced from 3.3 V to 0.0 V in steps of 0.1V. We ran these experiments on the 16-processor IBM SP2 (with thin nodes) as well as on a RS/6000 model 590 workstation with 1GB memory.

We note here that the model 590 workstation has four times larger data cache and four times wider memory bus as compared to that on the thin nodes of the SP2. There are also differences in the load/store operations between the registers and the data cache. On various benchmarks involving sparse and irregular computations, we observed that the model 590 workstation is approximately 2 to 2.5 times faster than the single processor of the SP2 that we used in our experiments.

We present results from our experiments in Table 7.

Table 7: Execution time (in seconds) for the 31568 problem on RS/6000 590 workstation and on a 16-processor SP2

Test	System	Total Exe. Time	Total Time for JCB & RHS	Total Time for ILU	Total Time PCG Iters
I_d vs. V_d	RS6K/590	35,907	4,287	30,385	
	SP2/16	25,522	381	6,173	18,943
I_d vs. V_g	RS6K/590	45,987	5,776	38,722	
	SP2/16	33,216	512	8,161	23,903

In that table, we have shown the total execution time as well as the cumulative time spent in computing the JCB-&-RHS, in computing the ILU, and in the PCG iterations. For the runs on the workstation, the last two components are reported by their sum. We note that in computing JCB-&-RHS the parallel version does well. As noted earlier, the computations in this component are data parallel and parallelism is easy to exploit.

When we compare the execution time for computing the ILU and the PCG iterations, the parallel version performs very poorly in both cases (a speedup of about 1.4). Recall that the processors on SP2 are considerably slower than the RS/6000 590 workstation. To understand the performance difference in the two cases on 16 processors, it is instructive to compare the total number of Newton iterations and PCG iterations performed in each case. In the I_d–V_d case (34 bias points), a total of 230 Newton iteration and 12732 PCG iterations were performed whereas, in the I_d–V_g case (68 bias points), a total of 304 Newton iterations and 15930 PCG iterations were performed. Note that in both cases, the ordering and partitioning of the vertices of the mesh are identical. The difference lies in the numerical values in the coefficient matrices. The same partitioning in one case results in better convergence rate than the other.

5 Conclusions

Parallelization of three-dimensional semiconductor device simulations pose special problems especially when irregular grids are used. In this paper, we have described our experience in parallelizing the FIELDAY code, which is a general purpose finite element based code for solving the drift-diffusion (DD) equations. We have focused on the steady-state solution (using DC analysis) of the DD equations. The problem is linearized using Newton iterations and the linear systems are solved using preconditioned conjugate gradient type iterative methods. We have presented some *very preliminary* performance results on a 16-processor IBM SP2. Our preliminary experiments indicate that irregular computations can be parallelized for efficiency and we have seen acceptable performance on a per iteration basis. However, overall performance is quite poor. The reason for this behavior is that we have used a partitioning method that takes into account only the graph representation of the irregular grid. Although a good load balance of both the computation and communication is achieved, the resulting reordering adversely affects the structure of the preconditioner used and the convergence rate deteriorates rapidly. The effects become worse as the number of partitions is made larger to solve a problem on a larger set of processors. Our latest experiments indicate that by enabling the partitioner to distinguish between the edges in the direction of strong coupling and those in the direction of weak coupling, the convergence rate can be improved significantly without sacrificing parallelism. This understanding can be brought in by incorporating additional information about the underlying physical process in the mesh representation. We are continuing further investigation taking into account the underlying physical processes as well as the numerical properties when the mesh is being partitioned.

Acknowledgement: The authors would like to thank Giovanni Fiorenza and Bob Walkup for many fruitful discussions on Fielday device simulator. Giovanni also conducted the experiments on RS/6000 model 590 workstation, the results of which are reported in this paper for comparison purposes.

References

[1] Barnard, S. T. and Simon, H. D., "Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems", *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101-117, 1994.

[2] Buturla, E. M., Cottrell, P. E., Grossman, B. M., and Salsburg, K. A., "Finite-Element Analysis of Semiconductor Devices: The FIELDAY Program", *IBM J. Res. Develop*, vol. 25, pp. 218–231, 1981.

[3] Cole, D. E., *et al.*, "The Use of Simulation in Semiconductor Technology Development", *Solid-State Electronics,* vol. 33, pp. 591–623, 1992.

[4] D'azevedo, E. F., Forsyth, P. A., and Tang, W.-P., "Ordering Methods for Preconditioned Conjugate Gradient Methods Applied to Unstructured Grid Problems", *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 944–961, 1992.

[5] Dutton, R. W. and Pinto, M. R., "The Use of Computer Aids in IC Technology Evolution", *Proc. IEEE* , vol. 74, no. 12, pp. 1730–1740, 1986.

[6] Fischetti, M. V., "Monte Carlo Simulation of Transport in Technologically Significant Semiconductors of the Diamond and Zinc-Blende Structures. Part I: Homogeneous Transport", *IEEE Trans. Elec. Dev.*, vol. ED-38, p. 634, 1991.

[7] Forghieri, A., Guerrieri, R., Ciampolini, P., Gnudi, A., Rudan, M., and Baccarani, G., "A New Discretization Strategy of the Semiconductor Equations Comprising Momentum and Energy Balance", *IEEE Trans. on CAD/ICAS* , vol. CAD-7, no. 2, pp. 231–242, 1988.

[8] Ieong, M. K., and Tang, T.-w., "A General Hydrodynamic Solver for Deep Submicron Silicon Devices", *Proceeding of the IWCE*, pp. 111-i-114, 1994.

[9] Knepper, R. W. *et al.*, "Advanced Bipolar Transistor Modeling: Process and Device Simulation Tools for Today's Technology", *IBM J. Res. Develop*, vol. 29, no. 3, pp. 218–228, 1985.

[10] Knepper, R. W. *et al.*, "Technology CAD at IBM", *Technology CAD Systems*, edited by Fasching F., Halama S., and Selberherr S., pp. 25–62, 1993.

[11] Laux, S. E., Fischetti ,M. V., and Frank, D. J., "Monte Carlo Analysis of Semiconductor Devices: The DAMOCLES Program", *IBM J. Res. Develop*, vol. 34, p. 466, 1990.

[12] Pinto, M. R., Rafferty, C. S., and Dutton, R. W., "PISCES II: Poisson and Continuity equation solver," Stanford University, Stanford, CA 94305, Sep. 1984.

[13] Rudan M., and Odeh, F., "Multi-dimensional Discretization Scheme for the Hydrodynamic Model of Semiconductor Devices", *COMPEL*, vol. 5, pp. 149–183, 1986.

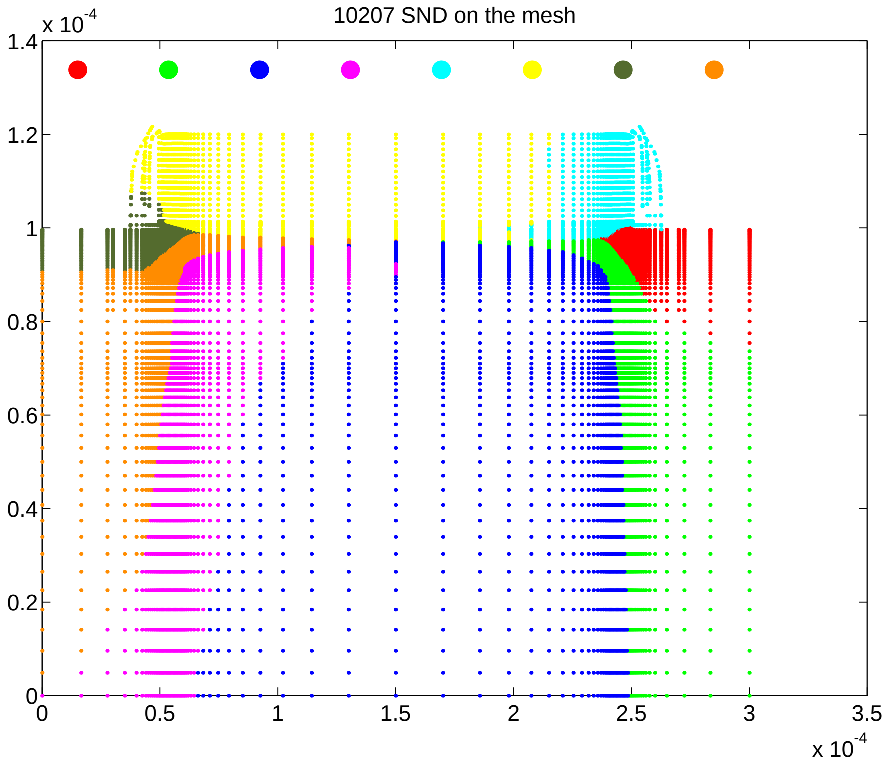
[14] Scharfetter, D. L., and Gummel, H. K., "Large-signal Analysis of a Silicon Read Diode Oscillator", *IEEE Trans. Elec. Dev.*, vol. ED-16, no. 1, pp. 64–77, 1969.

[15] Selberrher, S., Schutz, S., and Potzl, H.W., "MINIMOS - a Two-dimensional MOS Transistor Analyzer,", *IEEE Trans. Elec. Dev.*, vol. ED-27, no. 8, pp. 1540–1550, 1980.

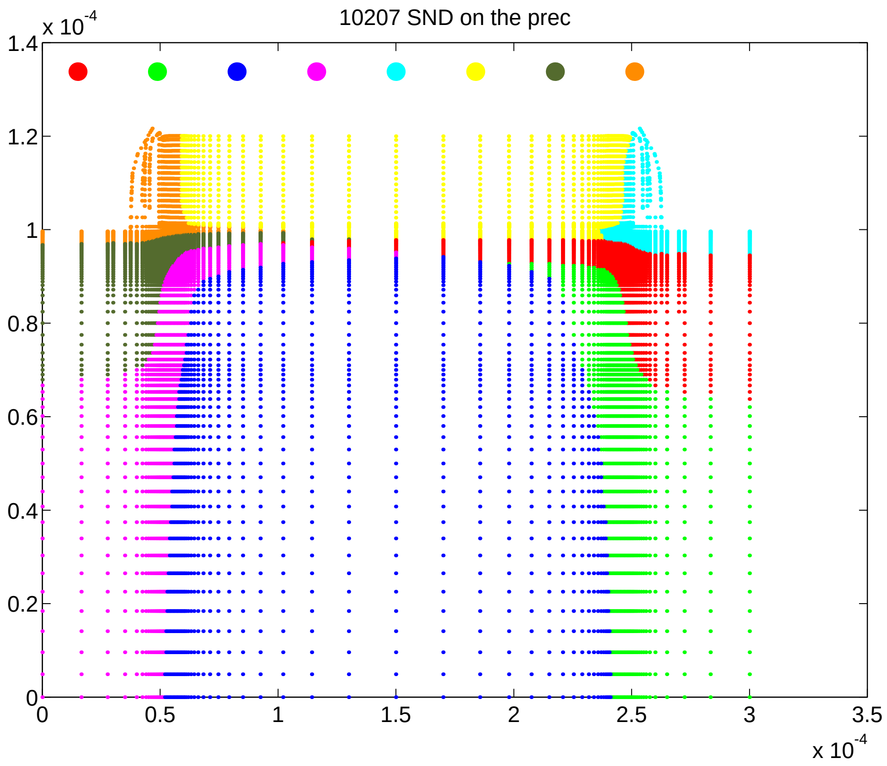
[16] Snir, M., Hochschild, P., Frye, D., and Gildea, K. The communication software and parallel environment of SP2. *IBM Systems Journal*, 34(2), 1995.

[17] Stunkel, C., Shea, D., and Abali, B. The SP2 communication subsystem. *IBM Systems Jounal*, 34(2), 1995.

[18] Tomacruz, E. R., *Applications of Parallel Processors to Technology Computer-Aided Design Problems*, Ph.D. Thesis, Memorandum No. UCB/ERL M94/101, Electronics Research Laboratory, University of California, Berkeley, California, 1994.



(a)



(b)

Figure 11: Partitioning of 10207 mesh with SND partitioning: (a) based on the connectivity of the original mesh, (b) based on the connectivity defined by preconditioner. (This is a color plate)