RAL-92-080

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX RAL-92-080

Initial Experiences in Porting a Three-Dimensional Semiconductor Device Modelling Program to the Intel iPSC/860

R F Fowler B W Henderson and C Greenough

December 1992

Initial Experiences in Porting a Three-Dimensional Semiconductor Device Modelling Program to the Intel iPSC/860

R.F. Fowler[†], B.W. Henderson[‡] and C. Greenough[†]

[†] *Mathematical Software Group*
[‡] *Parallel Processing Group*

October 1992

Abstract

EVEREST is a three-dimensional device modelling package used to study the electrical behaviour of semiconductor devices. A set of partial differential equations describing the current flow within the device is solved using a mixed mesh of tetrahedral and hexahedral elements. The highly non-linear nature of the equations requires that a flexible solution strategy be used to make the software robust and efficient. Within the solution process large sparse non-symmetric linear systems are solved using iterative solvers such as CGS with preconditioning. Run times of many hours or even days are required for realistic devices on current workstations.

EVEREST consists of over 100000 lines of Fortran and was developed for scalar architectures. The main aim of this work was to investigate how such an application could be adapted to run on the iPSC/860. A review of some of the possible parallelisation techniques is made.

Most techniques for the parallel implementation of such mesh based calculations on MIMD machines involve a partitioning of the mesh. An extension of an algorithm due to Farhat has been implemented in a pre-processor to partition the device meshes used by EVEREST. Aspects of load balancing and the need to minimise the number of interface nodes in the decomposition are discussed. Some initial results using the partitioned mesh to perform the system matrix assembly are presented.

Methods that could be used to implement a parallel preconditioned CGS solver on the iPSC/860 are reviewed and some performance estimates made using the results of certain BBS tests on this machine.

Presented at the BECAUSE Workshop, INRIA, Sophia–Antipolis, France, 12–16 October 1992

Mathematical Software Group
Computational Modelling Division
Rutherford Appleton Laboratory
Chilton, Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

As part of the ESPRIT project BECAUSE we have investigated the implementation of a semiconductor device modelling package on the Intel iPSC/860. The device modelling software that we have used is the EVEREST suite which solves the drift diffusion equations to model general silicon semiconductor devices. The purpose of this work was to assess the amount of effort required to move existing field modelling codes onto the iPSC machine and to find out the potential benefits available in doing so.

In the following sections of this paper we first present an overview of the EVEREST software, the numerical problem it solves and give some results showing where the CPU time is spent in the serial implementation. A short review is then made of the main approaches that could be used for a parallel implementation of the Solver Module. Section 5 discusses the problem of mesh partitioning and the pre-processor that we have developed for use with EVEREST device meshes. The results of an initial version of the Solver module that performs the matrix assembly phase of the calculation in parallel are then given in Section 6. We then make some estimates of the expected performance of a parallel linear solver based on results of tests that have been run on the iPSC/860. Finally some conclusions are given.

2 Semiconductor Device Modelling

Semiconductor device modelling can be used to predict the static and dynamic response of devices such as diodes, MOSFETs, etc. In particular device designers need to be able to calculate properties such as the current flow in the device as a function of the applied voltages on the contacts. From such information it is possible to determine important parameters like the current gain of a transistor, or its maximum operating frequency. Device simulation also gives insight into exactly where current flow occurs and the cause of problems such as "latch-up" in fast switching of CMOS devices. The high cost of fabrication of test devices and the constant drive for smaller and faster devices means that simulation is a vital tool in the semiconductor industry.

EVEREST is a software package for the simulation of general silicon semiconductor devices in three dimensions and was developed as part of ESPRIT Project 962 [1]. The software solves the following set of time-dependent non-linear partial differential equations over the device:

$$\epsilon\nabla^2\psi = -\rho \tag{1}$$

$$\frac{\partial p}{\partial t} = \nabla.\mathbf{J}_p - R \tag{2}$$

$$\frac{\partial n}{\partial t} = -\nabla.\mathbf{J}_n - R \tag{3}$$

The hole and electron currents themselves ($\mathbf{J}_p$ and $\mathbf{J}_n$) are given by the equations:

$$\mathbf{J}_p = -q\mu_p(V_T\nabla p + p\nabla(\psi - V_T\log(n_i))) \tag{4}$$

$$\mathbf{J}_n = q\mu_n(V_T\nabla n - n\nabla(\psi - V_T\log(n_i))) \tag{5}$$

There are three basic unknowns at each point in space ψ, n and p, though the code actual works in terms of a transformed set of variables ψ, ϕ_n and ϕ_p. ψ is the electrostatic potential and ϕ_p and ϕ_n are the quasi-Fermi potentials. The other terms in the equations are given from appropriate physical models, and may depend on the solution variables.

These equations are solved on a finite element type mesh composed of hexahedral and tetrahedral elements, the latter being necessary to allow for non-rectangular geometries and for adaptive refinement of the mesh [2]. The automatic adaption is an important feature since it is very difficult to specify a mesh in three dimensions that captures the solution while not using an excessive number of elements.

The partial differential equations are discretized on the mesh using a finite volume (control region) method. The resulting set of sparse equations is highly non-linear and is solved using a Newton-Raphson iteration. Since the computation times involved can be very long (hours or even days for fully three-dimensional problems on current workstations) it is important to solve these as efficiently as possible. A number of strategies are employed in EVEREST for this purpose, some of which we mention here:

- Continuation - each bias case is solved using the previous solution as the starting guess. If the Newton method fails to converge, the bias step is reduced and we try to solve an intermediate problem to get a better starting guess.
- Damping of the Newton updates are made if necessary.
- Gummel and coupled iterations - the default solution method is to start out using the Gummel method, where the three unknowns ψ, ϕ_p and ϕ_n are solved one a time holding the other two fixed. When sufficiently close to the solution a fully coupled method, solving for all three unknowns together, is used.
- The linear systems, $A\mathbf{x} = \mathbf{b}$, that arise in each Newton step are solved with pre-conditioned CG methods, ICCG for symmetric systems and CGS or Bi-CGSTAB for the non symmetric ones.

This combination of techniques gives a solution method that is both robust and efficient while not requiring excessive memory, e.g. 32 Mbytes is adequate for basic structures.

3 Analysis of Workload of Serial Program

As a starting point for understanding the existing code and selecting the parallel method to employ, a timing analysis was made of the EVEREST solver module on a selection of problems. While we already knew what parts of the code were likely to be most expensive it is useful to have actual figures when considering the costs and benefits of particular parallel methods.

The problems considered included the major device types (field effect and bipolar) with static and time dependent solutions. An example with automatic mesh adaption was also included. Table 1 summarises some of these examples.

Name	N_n	N_e	Mesh	Steps	Run time (Sec.)
MOS1	1416	1040	H	3	565
MOS4	9755	8320	H	3	10782
NPN	6582	10490	M	6	88850
COMBIC	1179	532	H	6	16091
ADAPT	6582	10490	M	1	954

Table 1: Test problem sizes and run times. N_n and N_e are the numbers of nodes and elements in the mesh, which is either made of hexahedra (H) or a mixed mesh (M) of hexahedra and tetrahedra.

The run times refer to an implementation on a SUN Sparc IPC workstation. Up to 1 day of run time can be used with a mesh of only 6500 nodes. We note that the run time of the mesh adaption example (ADAPT) is very much less than other large simulations. This is because the mesh adaption is performed only on the doping profile (which needs no calculation) and the variation in potential, which can be found cheaply by solving just Poissons equation. Once a refined mesh has been generated in this way it is usual to perform a number of more expensive calculations on it, as in the NPN test. It seems that it is not worthwhile to deal with the complexity of including adaptive refinement in a parallel version since this is relatively cheap and can be done as a pre-processing step on a serial machine. This would not be the case when adaption on current is included in the package.

The CPU time on this machine was analyzed in two ways, firstly using the builtin timing routines within the program that give information on logical blocks and secondly using compiler generated profile information.

Table 2 gives some typical results from the builtin timing routine. The majority of time is found

Block	MOS1	MOS4	COMBIC	NPN	ADAPT
Level 1					
Mesh input	1.87	0.69	0.07	0.09	0.03
Doping input	1.49	0.47	0.02	0.06	0.06
Initial soln.	11.17	5.64	0.08	2.57	4.10
Gummel iter.	46.37	38.12	0.05	22.71	22.76
Coupled iter.	37.19	54.53	0.06	74.48	nc
Output	1.65	0.47	0.01	0.07	4.02
Transient	nc	nc	99.68	nc	nc
Refinement	nc	nc	nc	nc	28.01
Level 2					
Linear soln.	40.62	69.22	88.10	82.11	8.02
Form Jacobian/RHS	41.69	22.32	11.11	14.87	33.73
Level 3					
PCGS solver	31.49	61.09	86.66	78.50	nc
ICCG solver	7.43	6.87	0.02	2.98	7.56

Table 2: Percentage of time spent is basic blocks of the Solver. The different levels refer to depth in the calling tree, so that Level 1 results include time shown in lower levels, etc.

to be taken up in the formation of the Jacobian matrix and right hand side vector and solution of the resultant linear system. This is the case in both static and transient examples, even though the higher level calling blocks are different. These two parts are most dominant in the longer runs when they account for up to 99.2% of the total time. Hence a parallel implementation must deal with these areas efficiently. However, we cannot just ignore the rest of the program since it will have implications on data movement and distribution, and if large numbers of processors are used it will require parallelisation of all of the code.

Some typical results from the compiler generated subroutine level profiling of one such simulation are shown in Table 3. The table only shows those subroutines taking above 1% of the total time, with a few words outlining the function. It should be possible to perform the matrix assembly and right-hand side evaluation in parallel the since operations are independent, apart from update of common variables. The linear algebra is more complex and is dominated by the forward and backwards

Linear algebra				
Subroutine function	MOS1	MOS4	COMBIC	NPN
forward subs.	11.7	26.6	27.1	28.1
backward subs.	11.7	25.3	27.8	28.4
diag. matrix-vector prod.	3.0	3.0	8.6	6.5
vector addition	4.2	4.3	5.0	5.7
diagonal scaling	2.4	2.4	7.2	5.3
vector scaling	2.7	2.5	2.8	3.3
vector subtraction	2.5	2.7	2.8	3.3
dot product	1.5	1.4	1.2	1.5
maximum value of vector	1.0	0.9	1.1	1.2
zero vector	1.8	0.6	0.1	0.4
Matrix/RHS assembly				
Subroutine function	MOS1	MOS4	COMBIC	NPN
sparse matrix update	13.2	6.5	3.2	3.5
evaluate element contr.	7.1	2.4	0.7	1.5
discrete terms	4.5	1.4	0.2	0.6
divergence terms	4.0	3.4	1.2	1.7
mobility model	2.1	nc	0.5	0.8

Table 3: Compiler generated subroutine level profiling of some test problems. The subroutines are divided into those associated with the linear algebra and those to do with the formation of the Jacobian matrix and right-hand side. Figures are the percentage of the total time due to this routine.

substitution operations on large sparse matrices. These operations arise due to the pre-conditioning of the iterative methods and are further emphasised by the fact that the Solver Module uses the implementation due to Eisenstat [3] which avoids a direct sparse matrix-vector product at the cost of additional substitution operations. An effective pre-conditioning is required because the iterative methods can often fail to converge without it.

4 Parallel Methods for Device Modelling

There has been a great deal of research done on solving partial differential equation problems on parallel computers. The methods that might be applied to the device equations can be classified into two main types, true domain decomposition methods and ones based on a geometric decomposition for the matrix assembly followed by a parallel solution of the global linear problem. We briefly consider these options here.

4.1 Domain decomposition methods

The classic domain decomposition method is that due to Schwarz, whereby the domain is divided into a number of overlapping subdomains. Within any one subdomain a self contained problem is defined by assuming that the new boundary edges that arise from the decomposition are Dirichlet boundaries. The values on the new boundaries are first set by some initial guess and are then updated from the neighbouring subdomains when new solutions are obtained. Many texts describe this method in more detail, see for example [4].

From the point of view of adapting an existing device modelling program to run in parallel this technique would appear to be straightforward to implement. If the existing device mesh were decomposed into a set of suitable overlapping regions then the new boundaries could be defined as a new type of contact (which implies Dirichlet boundary conditions). This would allow the existing Solver Module to run on each node of the parallel machine with only the minimum of modifications to exchange boundary updates at suitable points and to check for global convergence. However the Schwarz method suffers from a number of draw backs which could make it very inefficient:

- The overlap region represents at least one layer of additional elements to be assembled for each processor. This will increase the total workload and limit efficiency especially for three-dimensional meshes.
- The convergence of the non-linear process will be slower in the parallel version than the serial one [6]. This is likely to represent a significant problem as the device equations are highly non-linear. Convergence improves as the amount of overlap is increased, but this is at the expense of increasing the total amount of work.

Other methods exist based on decomposition into non-overlapping subdomains, which avoids the additional work and should give better convergence properties. One such method is based on solving the Schur complement problem for the interface nodes. If the domain is split into just two subdomains for example, and the nodes are renumbered so that those on the interface have the highest numbers, then the linear system to be solved takes the form:

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

where each A_{ij} represents s block matrix and u_i and f_i are the components of the update and right-hand side vectors respectively. If we can solve for the interface values u_3 first, then the remaining two parts of the solution for u_1 and u_2 are independent and can be performed in parallel.

The Schur complement problem for the interface nodes is to solve

$$Su_3 = \hat{f}_3$$

where,

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

and

$$\hat{f}_3 = f_3 - A_{31}A_{11}^{-1}f_1 - A_{32}A_{22}^{-1}f_2$$

A lot of research has been done on such methods, much of it for problems in computational fluid dynamics (CFD), see for example [5]. The size of the Schur complement problem is proportional to the number of interface nodes and this can be large in three dimensions. An efficient solution method is required and [5] suggests that a parallel iterative solver is needed. However, it is also necessary to use a good preconditioner and it seems that there are many possibilities to select from. We are not aware of any work which explores the appropriate preconditioning to use for the device equations or indeed whether such a preconditioner exists.

In the absence of clear evidence that such an approach would work we looked at the alternative method to domain decomposition, that of using a global parallel solver.

4.2 Parallel iterative solvers

Implementation of a parallel version of the the conjugate solver routines on a distributed memory MIMD machine is possible and offers the advantage that we will be solving the same numerical problem as in the serial case, and hence should expect similar convergence properties and behaviour. The fact that such an approach is viable on at least one MIMD architecture is shown by Pommerell *et al* [7]. They solve matrices arising from the semiconductor device problem on a 64 processor system using parallel PCGS and ICCG solvers. Good performance is reported in terms of speed up (40-50 on 64 processors). However they present no results for the actual performance of the system (elapsed time or MFLOP/s) and have used a simulation of the system for timing. One characteristic of the system that they do state is that the time to send one word of data (8 bytes) from one processor to another is close to that to compute it, i.e. the time for 1 FLOP. Other aspects such as message latency are not mentioned. We note that the iPSC/860 can perform 1 FLOP in about $0.1\mu s$ while taking almost $3\mu s$ to send an 8 byte word (with very long messages). Thus the iPSC/860 has a much worse communication - computation balance.

Taking some typical timing results for just the linear algebra part of the Solver module we find that the CPU time of the serial code can be classified into three major types shown in Table 4.

Operation	Example	Time (%) CGS
Parallel vector	$\alpha\mathbf{x} + \mathbf{y}$	31.8
Limited parallel vector	$\mathbf{x}.\mathbf{y}$	2.7
Forward/back subs.	$L^{-1}\mathbf{x}$	63.4

Table 4: Distribution of CPU time for the linear solvers between 3 types of operations.

The three types of operation are: parallel-vector, vector operations which can be performed without communication; limited parallel-vector operations, like the dot product where some communication is required; and the forward and backward substitution operations.

The key problem is seen to be the sparse matrix substitutions, which are normally solved in a sequential manner with little scope for parallel execution. The method suggested for parallelisation of these operations in [7] is based on a renumbering of the nodes such that sub-blocks of the matrix can be treated in parallel. This renumbering is illustrated in Figure 1.

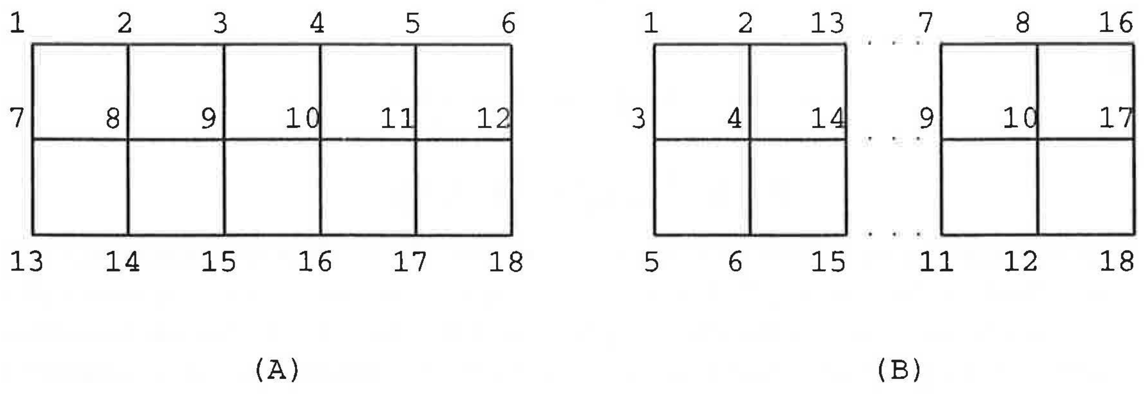

Figure 1: The original grid (A) has a natural numbering of the nodes. Half the nodes are mapped to each processor and a balanced red-black renumbering performed (B).

The original mesh (A) with natural numbering has first been partitioned so that the nodes on the left are mapped to processor 1 and the rest to processor 2, as in (B). In the substitution operations, we have one unknown corresponding to each node and nodes that are mapped to the same processor will be solved sequentially. Hence dependencies between processors are restricted to nodes along the interface (3,4,9,10,15,16 in the original numbering). In the renumbered system (for the forward substitution case) nodes 1-6 on processor 1 and 7-12 on processor 2 can be solved in parallel. Nodes 13-15 can only be solved when the results for nodes 7-9 are provided from processor 2. Nodes 16-18 do not have any dependencies on processors 1, but have been effectively "coloured black" so that there is an equal number of each colour on every processor (the balanced colouring of [7]). This can be extended to unstructured meshes and more complex decompositions, though more than two colours may be required in some cases.

Hence with suitable modification of the data distribution it should be possible to parallelise all the basic operations involved in the preconditioned CGS and ICCG solvers.

One additional cost that needs to be considered is that of the convergence rate of the iterative methods due to the renumbering of the nodes. It is known that "natural" numberings generally lead to better preconditioners than some others such as red-black orderings or band-width minimisation ones. Heiser *et al* [8] have reported that the reordering for parallelisation costs about 15-40% in terms of extra iterations. While not insignificant, such an overhead is acceptable if good speed up can be obtained.

5 Mesh Partitioning

All the methods mentioned in the previous section require that the device mesh is partitioned in some way between the processors. There are many ways in which this can be done and the optimal partitioning will depend on the solution method to be used. For the matrix and right-hand side vector assembly a partitioning that ensures reasonable load balancing is required. For a hexahedral mesh over a device of only one material this just requires the same number of elements are assigned to each processor. In addition it is important to minimise the number of nodes on the interface between two domains since the variables associated with these will have updates from both processors and hence require communication.

For the solution of the linear algebra problem using a global solver the partitioning is based on nodes rather than elements. Apart from the need to have the same number of nodes on each processor we also need to consider the communication costs associated with the most time consuming operation, the sparse matrix substitution. Following the colouring and reordering techniques discussed in Section 4.2 we will need to exchange data between processors for nodes that are on the boundary with another subdomain. This is similar, but not identical to the requirement for the assembly partitioning.

The importance of the mesh partitioning will depend on the characteristics of the parallel computer. The ratio of communication time to the calculation time is particularly important. The iPSC/860 is poor in this respect since it takes of the order of 30 times longer to send a value as to calculate it. The latency, or start up time to send a message, can also be important since decompositions that minimise the number of interface nodes also tend to create more neighbouring subdomains, so more messages of shorter length are sent. The iPSC/860 has a message half performance length of $N_{1/2} \approx 486$ ([9]) which has to be taken into account when selecting the best decomposition. In this paper we are most interested in the behaviour with large meshes and limited numbers of processors.

While it is possible to repartition data between the matrix assembly phase and the linear solution, we have sought to avoid this by looking for a single partition that balances work in both phases and limits the total communication cost.

Many algorithms have been developed for mesh decomposition. Simulated annealing may give the best results though it can be very expensive. Orthogonal recursive bisection and eigenvalue recursive bisection [10] are cheaper techniques that give reasonable results. We have developed a preprocessor that uses the domain decomposition method due to Farhat [11], it would be easy to modify this to use one of the more recent algorithms.

The Farhat algorithm attempts to decompose a given mesh into p subdomains with N_e/p elements each (or as near as possible), where N_e is the total number of elements. The basic steps are:

1. Count the number of elements about each node (called the weight).
2. Find the node with the lowest weight and use the elements connected to this as the start of the element list for the new subdomain.
3. Add the elements surrounding those already selected to the list until the desired length is reached.
4. Remove the selected elements from the list and update the weights.
5. Repeat from 2 until p subdomains are found.

This works well for small p but for $p > 8$ the method sometimes fails to find a set of connected regions. Because of this we modified the algorithm so that it would try a range of possible decompositions. This can be done since it is usually the case that there is more than one node of minimum weight to use as a seed for the next subdomain. By varying this choice we can usually find a number of successful decompositions without disconnected subdomains. As several valid decompositions can be generated in this way we can select the one that is "best", such as having the least number of interface nodes. The preprocessor will automatically try to generate a number of possible decompositions and write out the best one. For small problems there can be a difference of about 20% between the best and the worst decomposition in terms of the number of interface nodes.

A further complication is in the case of mixed meshes, as are often used in EVEREST, where the work per element is not constant. Timing measurements indicate that the assembly of a hexahedral element costs about 1.8 times that of a tetrahedral one in the present software. To allow for this the algorithm uses a weighted target for the number of elements in each subdomain. An example of the mesh decomposition for a two-dimensional MOSFET is shown in Figure 2.

Having partitioned the elements it is necessary to find a balanced partitioning for the nodes. This is done as follows:

1. Nodes that lie totally within any subdomain are allocated to that processor.
2. Interface nodes are assigned to processors trying to avoid giving any one processor more than N_n/p, where N_n is the total number of nodes.
3. To improve the load balance, a sweep is made through all interface nodes changing the subdomain they are assigned to if this improves the load balance. Nodes can still only be assigned to subdomains that they are next to.

This simple method gives a good load balance even when using a mixed mesh in the examples we have tested.

The preprocessor for mesh decomposition is a self contained module that just reads the selected mesh file, in the standard ASCII (neutral file) format used through out the EVEREST suite. It takes

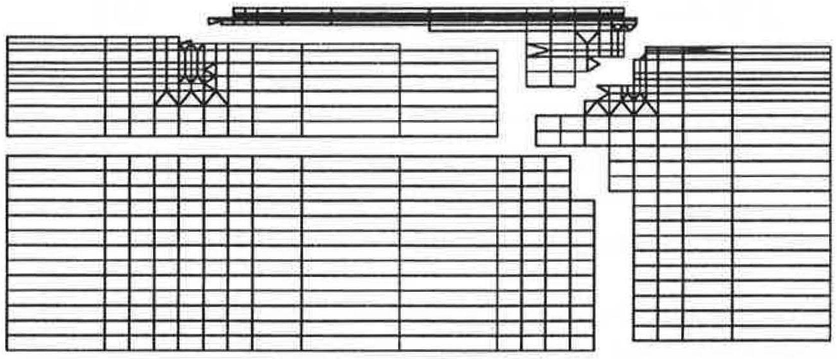

Figure 2: Mesh decomposition of a 2D MOSFET into 4 subdomains.

typically a few minutes (SUN 4 / 5000 nodes), to calculate a reasonably good decomposition and then writes the results to a new set of neutral files, one for each subdomain. These can then be picked up the Solver module running on the iPSC/860.

6 Parallelisation of the Matrix Assembly

To test the parallelisation of just the matrix assembly phase of the computation we have developed a new version of the EVEREST Solver module. This performs the assembly and right-hand side in parallel using the subdomains, but then sends the result to one processor so that the serial version of the linear solver can be used. This will be of limited use since the linear algebra step is represents about half of the total time, but it does allow us to test decomposition.

Due to the complexity of the data structures that are used within the code (some of which are to save memory) this required a significant amount of effort. Synchronous communication was used for sending data from all processors to the controlling node and returning the updates to them.

For fast access all the relevant data files are stored on the Concurrent File System (CFS) of the iPSC/860. Processor 0 is given control of the simulation and is responsible for reading commands and relaying them to other processors and writing results files. The parallel assembly is used for both coupled and Gummel iterations and also for damping, when only the right-hand side vector is evaluated. The range of different techniques used in the Solver add to the work required to produce a parallel version.

Table 5 gives some timings for the case of one and two processors. The test problem is a bipolar device with 372 nodes and 828 elements, all hexahedra.

The computation time for assembly is very close to half in each case with two processors, showing that the decomposition has worked. The results obtained are identical to just using one processor. The communication costs are about $1/8^{th}$ of the computation time with two processors, but this gets worse as the number of processors is increased. In this test the assembly phase was about 42% of the total time so the speed up is limited to ≈ 1.2 in this case. Increasing the number of processors will not improve this significantly and a parallel version of the linear algebra is required to get worthwhile performance.

Operation	1 processor	2 processors	
	T_{calc} (ms)	T_{calc} (ms)	T_{comm} (ms)
Poisson	123	62	10
RHS Poisson	63	32	2
Electron	555	276	14
Coupled	1092	548	74
Current eval.	5.3	4/1.3	1/2.9

Table 5: Results for parallel assembly on iPSC/860. The times are milliseconds per call.

7 Performance Estimates for Parallel PCGS on the iPSC/860

We can estimate the performance of a parallel preconditioned conjugate gradient solver on the iPSC/860 using the data in Table 4 along with results from the BECAUSE Benchmark Set (BBS) [9]. The tests in the BBS include the basic operations that are performed by the PCGS solver.

BBS 2.4.2 involves forward and backwards substitution on a sparse matrix of the type that occurs in the EVEREST solver. The parallel implementation is based on the mapping and renumbering discussed in Section 4.2 with a slicewise decomposition of the problem domain and the results are given in [9]. We may expect slightly better performance with the decomposition used in this paper, since there will be fewer interface nodes and hence less communication. Nevertheless the results should be a fair approximation to the potential speed up. Other BBS tests give the speed up we can expect for the other important operations, such as dot product and maximum value of a vector. All these results can be combined since there is no inconsistent assumption about the data distribution between them.

Operation	Time (%)	S_{1300}	S_{9000}	S_{36000}
Parallel vector	31.8	8	8	8
Limited parallel vector	2.7	0.7	2.8	4.9
Forward/back subs.	63.4	0.7	1.8	4.0
Weighted Speed up		1.0	2.4	4.8

Table 6: Estimated speed up of the EVEREST linear solver based on BBS test results for 8 processors on the iPSC/860. S_n is the speed up with n nodes.

Table 6 shows estimates of speed up for the iPSC/860 based on the BBS results and the weights from Table 4. We have selected three typical sizes for the number of nodes of 1300, 9000 and 36000. While we can achieve about 60% efficiency using 8 processors on the largest problem, the smaller problems give very poor results. The half performance problem size is about 30000 nodes, and this will increase rapidly as the cube dimension is increased. The results are significantly worse than those reported in [7] for 64 processors. This is due to the poor communication performance of the iPSC/860. However, we cannot say that the machine discussed in [7] is "better" than the iPSC/860 since no absolute performance figures are quoted, only speed ups. If we made the same calculations as above for the iPSC/2 instead of the iPSC/860 we would get much higher speed up and efficiency. However the problem will always run in a shorter time on the iPSC/860 with the same number of processors.

The above efficiency does not include the cost of the extra iterations that are expected to be required due to the renumbering of the nodes for the forward and backward substitutions [8]. This could reduce the efficiency of the largest case to less than 50%.

8 Conclusions

We have analyzed an existing application software package to identify the computational kernels that are most expensive. This is an important first step in deciding whether it is possible to adapt a serial program to run in parallel or if it is necessary to redesign it using different algorithms.

For the EVEREST three-dimensional device modelling code most computational work is associated with the Solver Module in terms of the matrix assembly and the sparse linear algebra. The assembly can be performed in parallel by a suitable mesh decomposition and the method of Farhat has been shown to be suitable for the three-dimensional mixed element type mesh used in this software.

The linear algebra is more difficult to treat. The incomplete LU type preconditioning is intrinsically difficult to parallelise. While it is feasible on machines with good communication/computation balance, the iPSC/860 is rather limited in this respect and the performance will only be acceptable for large problems on limited numbers of processors.

The next generation of machines, such as the Intel Paragon, should address this communication/computation imbalance and be easier to use in an efficient manner.

References

[1] C.Greenough, C.J.Fitzsimons and R.F.Fowler, "Software for Modelling Semiconductor Devices in Three Dimensions" Report RAL-91-042 , Rutherford Appleton Laboratory (1991).

[2] J.V.Ashby, C.J.Fitzsimons, R.F.Fowler and C.Greenough, "The adaptive solution of three dimensional semiconductor device problems" Report RAL-91-020 , Rutherford Appleton Laboratory (1991).

[3] Eisenstat, S.C., 1981: Efficient implementation of a class of preconditioned conjugate gradient methods, SIAM J. Sci. Statis. Comp., vol. 2, p. 1.

[4] Carey, G.F., "Parallel Supercomputing: Methods, Algorithms and Applications", Wiley (Chichester) 1989.

[5] Chan, T.F, "Domain decomposition algorithms and computational fluid dynamics", Intl. J. Supercomputer Appl., Vol. 2, No. 4, p. 72, Winter 1988.

[6] G.Meurant, "Domain decomposition methods for PDEs on parallel computers", Intl. J. Supercomputer Appl., Vol. 2, No. 4, p. 5, Winter 1988.

[7] Pommerell, C. *et al*, "A set of new mapping and colouring heuristics for distributed memory parallel processors", Technical Report No 90/1, ETH Zurich, 1990.

[8] Heiser, G., Pommerell, C., Weis, J. and Fichtner, W. , "3D Numerical semiconductor device simulation", IEEE Trans. CAD, Vol. 10, No. 10, p. 1218 (1991).

[9] R.F.Fowler, B.W.Henderson and C.Greenough, "BBS results for the iPSC/2 and iPSC/860", Proceedings of the BECAUSE European Workshop, INRIA Sophia-Antipolis, Oct. 1992, to be published.

[10] R.D.Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations", Concurrency Pact. & Exper., Vol. 3, No. 5, p. 1 (1991).

[11] C.Farhat and E.Wilson, "A new finite element concurrent computer program", Intl. J. Numer. Methods in Eng., Vol. 24, 1771 (1987).