Int J Parallel Prog (2007) 35:493–505
DOI 10.1007/s10766-007-0053-2

OpenMP Implementation of SPICE3 Circuit Simulator

Tien-Hsiung Weng · Ruey-Kuen Perng · Barbara Chapman

Received: 6 November 2006 / Accepted: 26 January 2007 / Published online: 20 July 2007

Abstract In this paper, we describe our experience of creating an OpenMP implementation of the SPICE3 circuit simulator program. Given the irregular patterns of access to dynamic data structures in the SPICE code, a parallelization using current standard OpenMP directives is impossible without major rewriting of the original program. The aim of this work is to present a case study showing the development of a shared memory parallel code with minimum effort. We present two implementations, one with minimal code modification and one without modification to the original SPICE3 program using Intel's *taskq* construct. We also discuss the results of the case study in terms of what future compiler tools may be needed to help OpenMP application developers with similar porting goals. Our experiments using SPICE3, based on SRAM model simulation, were compiled by the SUN compiler running on a SunFire V880 UltraSPARC-III 750 MHz and by the Intel icc compiler running on both an IBM Itanium with four CPUs and Intel Xeon of two processors machines. The results are promising.

Keywords OpenMP SPICE circuit simulator · Shared-memory programming model

T.-H. Weng (✉) · R.-K. Perng
Department of Computer Science and Information Engineering,
Providence University, Taichung, Taiwan, R.O.C
e-mail: thweng@pu.edu.tw

R.-K. Perng
e-mail: rkperng@pu.edu.tw

B. Chapman
Department of Computer Science, University of Houston, Houston, TX, USA
e-mail: chapman@cs.uh.edu

1 Introduction

In modern Very Large-Scale Integrated circuit (VLSI) design, Electronic Design Automation (EDA) tools are used to accelerate the design cycle and to help engineers to improve their designs. SPICE3 is the most important EDA tool used in circuit design. It is a general purpose circuit simulation program for DC, transient, linear AC, pole-zero, sensitivity, and noise analyses developed by UC Berkeley [1,2] and is written in C. Several commercial codes are based on SPICE. It is used to simulate circuits for various applications from switching power supplies to SRAM cells and sense amplifiers. Doing so requires the simultaneous solution of a number of equations that capture the behavior of electrical/electronic circuits. The number of equations can be quite large for a modern electronic circuit with transistor counts from several hundred thousands to some millions, and thus the simulation of circuits has become complex and quite time-consuming. Thus, a shared memory parallel program version is needed to achieve cost-effective performance.

Circuit simulator programs have been parallelized using Pthreads [3]. Although particularly useful for task parallelism where it can provide good performance, Pthread offers a low-level and cumbersome programming model. It requires major code rewriting and thus a major porting effort. Moreover, the resulting code is difficult to maintain in view of the many calls to Pthreads library routines and explicit coding of parallelism.

OpenMP [4] is an industry standard for shared memory parallel programming agreed on by a consortium of software and hardware vendors. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared-memory architectures. It is considerably easier for a non-expert programmer to develop a parallel application under OpenMP than under either Pthreads or the de facto message passing standard MPI. OpenMP also permits the incremental development of parallel code. Thus it is not surprising that OpenMP has quickly become widely accepted for shared-memory parallel programming.

For many scientific applications, especially numerical codes written in Fortran, parallelization is chiefly a matter of distributing the computation in loops that modify large arrays. Thus parallelization via OpenMP is simply a matter of inserting directives to indicate parallel regions and loops, and specifying which variables are shared or private with few modifications of the original source code needed. Unfortunately, this is not the case for other applications, particularly if they are written in C/C++. Challenges arise in the OpenMP implementation of C codes with dynamic linked-list data structures such as the SPICE3 circuit simulator, but are also encountered in agent-based model simulations [5], such as simulation of the immune response to a pathogen, financial markets applications, and more. In [5] Massaioli et al. discuss three techniques for realizing pointer-chasing loops in OpenMP: (1) By explicit decomposition of the lists into approximately equal-sized chunks, storing pointers to these chunks in an array, and then adding *omp for* worksharing directives. With this approach, the list decomposition is difficult to parallelize. (2) By adding the *omp taskq* directive, which is available only in the Intel KSR KAP/Pro compiler, but is not (yet) part of the official standard and thus introduces a portability problem. (3) By adding an *omp* *single nowait* clause to independently parallelizable linked-list loops. If the size of the system being simulated is sufficiently large, this may scale well.

Our aim in this work was to realize an OpenMP implementation of the SPICE3 circuit simulator with as few modifications to the sequential program as possible. Our approach relies on performing loop transformations and then adding OpenMP directives to the resulting loops. We discuss both possibilities for improving the OpenMP SPICE3 parallel program and developing it with minimum effort using a task queue without modification of the original program. Then, we present the result of our evaluation of this parallel version of SPICE3 on SunFire SPARC-III, Itanium, and Xeon platforms and give our conclusions and future plans.

2 OpenMP Implementation

In this section, we give an overview of the original sequential SPICE3 program simulator. Next, in Sect. 2.2 we present our OpenMP implementation of SPICE3. We describe the steps taken to create the OpenMP program. In Sect. 2.3, we discuss the possibilities for and challenges of further improvement of the parallel program. We also discuss our development of a parallel SPICE3 implementation with minimum effort and without modification to the original program using Intel's *omp taskq* in Sect. 2.4.

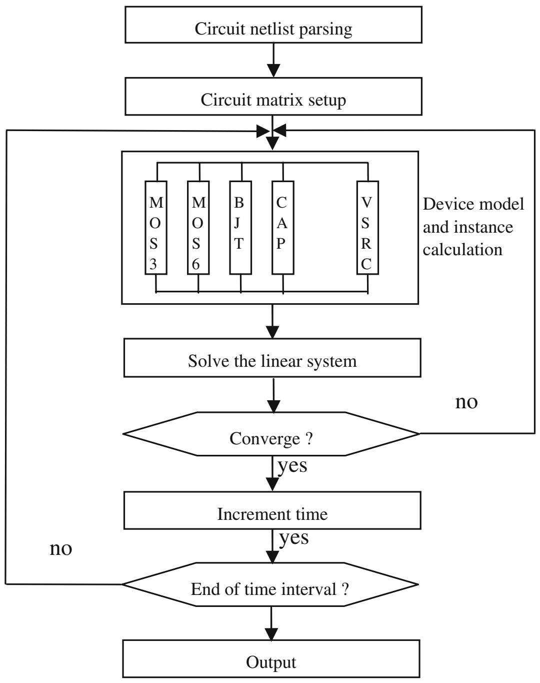
2.1 Brief Overview of Sequential SPICE3

SPICE3 provides several types of circuit simulations (or analyses) for modern VLSI design. Among these, the transient simulation is the most frequently used one. Figure 1 shows the basic configuration of the SPICE3 transient simulation algorithm. The circuit netlist describing the connection of the electronic devices in the circuit simulated is first parsed by SPICE3 and the appropriate data structures are generated. Then, the matrix representing the circuit is created and the data structures related to the matrix are set up. Actual transient analysis occurs next. For each time point in the transient analysis, the model calculations for each device, such as MOSFET, resistor, or capacitor device, are performed. The electrical parameters such as conductance and current for each instance instantiated from the corresponding device model are computed and put into the matrix elements. After the device model and instance calculations, all elements in the matrix for the linear system in transient analysis are ready for the sparse matrix solver in SPICE3. Then the matrix calculations for the linear system, such as the LU decomposition and forward/backward elimination in each iteration, are carried out until convergence is obtained. This process will continue until the final transient time is reached. Finally, the simulation results for all the time points simulated are displayed on the screen or stored in an output file.

2.2 Transforming Sequential Application to OpenMP Version

In order to reduce the effort in developing our parallel code, we first try to compile the original code with the auto-parallelization option of the Sun compiler switched on to find loops that may be a good candidate for potential parallelization. Unfortunately,

Fig. 1 Basic configuration of SPICE3 simulator



few loops are parallelizable, and their computational workload is very light. The most time consuming loops are the matrix calculation and model and instance calculation and these are not recognized as being parallelizable by the compiler.

In this work, we focus exclusively on parallelizing the model and instance calculation part, shown in Fig. 1. We refer to it as the *device loading routine*, because here all the model parameters related to the device, and the parameters for the instantiations of the device, are computed and loaded into the corresponding matrix elements. Many devices, such as MOSFET, resistor, capacitor, diode, and bipolar transistor, are supported by SPICE3. For each device, SPICE3 provides at least one model for the instances corresponding to this device used in the circuit simulated. For example MOS3 is one of the models for instances of the MOSFET device. Parameters such as the conductance and current are calculated according to the model equations built into the device loading routines. The conductance calculated will contribute to the elements of the matrix used in the linear system for simulation, while the calculated current will be entered into the right-hand-side of the linear system.

In this paper, we use an SRAM circuit as an example to demonstrate the SPICE3 simulation in its OpenMP implementation. A typical SRAM architecture is shown in Fig. 2. The SRAM circuit has a data input bus (*data_in*), a data output bus (*data_out*), an address bus (*addr*), and a write enable (*wr_ena*) pin. The data presented at *data_in* will be stored in a word line specified by *addr* when *wr_ena* is asserted. The data, stored in a word line and specified by *addr*, will be read and output to *data_out* if *wr_ena* is disabled.

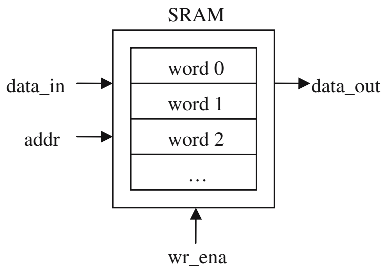


Fig. 2 A typical SRAM architecture

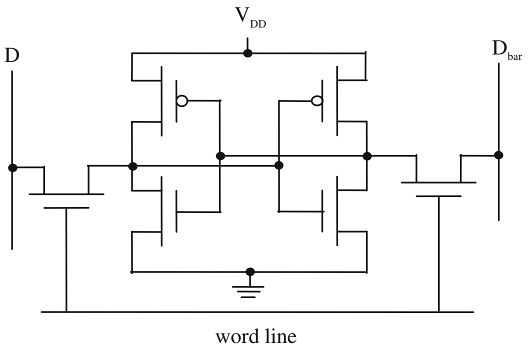


Fig. 3 The six-transistor CMOS SRAM

For the 1 K SRAM used in this study, there are 1 K storage cells built in the circuit. About 32 word lines are formed in the circuit and each word line has 32 storage cells with each storage cell representing one bit data. The storage cell is constructed by a basic six-transistor CMOS SRAM cell, which is shown in Fig. 3. There are six MOSFETs in the cell. The SRAM cell is designed by two cross-coupling CMOS inverters, which consists of four MOSFETs, to form the main cell for data storage, and two transmission-gate NMOS transistors to connect the main cell with the complementary bit lines (D and D_{bar}).

In our study of the performance of SPICE3 circuit simulation, we used the MOS3 model to model all MOSFET devices in SRAM circuits. Therefore the time-consuming part of the original sequential routine was the *MOS3load* function, which is the device-loading routine in SPICE3. It contains a nested pointer loop traversing an orthogonal linked-list. The actual size of the source code of the loop is approximately 1.3 K LOC (Lines of Code) and Fig. 4 reproduces the compact example code. The size of iterations of the loop depends on the size of the circuit; and the number of devices such as transistor, capacitor, etc. simulated may vary widely. Currently, it is not possible to parallelize a pointer-chasing loop by just directly adding an OpenMP directive in a portable manner: the *omp taskq* directive available in the Intel compiler is not a standard feature. Since there is a *return* statement within this loop, it has multiple exits, and cannot be parallelized without modification in any case. Likewise, we cannot employ an *omp parallel for reduction* to obtain the sum of the values for each element of the linked list.

```
int MOS3load(inModel,ckt)
   GENmodel *inModel;
   register CKTcircuit *ckt;
{  register MOS3model *model = (MOS3model *) inModel;
   register MOS3instance *here;
   ........
  for( ; model != NULL; model = model->MOS3nextModel ) {
      /* loop through all the instances of the model */
      for (here = model->MOS3instances; here != NULL; here=here->MOS3nextInstance) {
         .........
         if ( ckt->CKTmode & MODETRAN ) {
            error = NIintegrate(ckt,&geq,&ceq,here->MOS3capbd, here->MOS3qbd);
            if(error) return(error);
         }
        ........
      // Right hand side of Ax = b
       *(ckt->CKTrhs + here->MOS3gNode) -=   (model->MOS3type * (ceqgs + ceqgb + ceqgd));
        ....
      //   Sum of contributions for the element of matrix A
       *(here->MOS3DdPtr) += (here->MOS3drainConductance);
        ........
   }}  /* end of for loop */
   return(OK);
}
```

Fig. 4 Compact sequential code of MOS3load in SPICE3

There is a straightforward way to parallelize the sequential nested loop shown in Fig. 4. First, at the level of the circuit matrix setup of Fig. 1, we introduce a data structure to store the address of each linked-list element of an instance in an array of pointers, *MOS3instanceArray[i]*, as well as to keep track of the total number of elements in the lists in a variable *model->MOS3instanceCount*. Second, we perform loop coalescing to reduce the number and nesting level of loops, as well as to generate loops with larger loop iteration counts. The result is shown in Fig. 5.

This loop now involves an array of pointers and integer index instead of pointers. Finally, we may now directly add a *parallel omp for* directive to the loop since the loop iterations are independent except for the following: first, shared pointers point to the variables that are used to update the right hand side of the linear system, $Ax = b$, and second, shared pointers point to the elements of matrix A, which is used to sum the contributions for those elements. The *omp critical* synchronization directive is used to resolve this conflict, as shown in Fig. 5. We have chosen to present the large number of private variables required, given the lack of a *default(private)* clause; we believe that such a clause would be beneficial for such codes. In our case, there are 56 private variables to be declared manually and given the presence of pointer variables and aliasing, automatic scoping of variables in parallel regions (as proposed in [6]) would be highly desirable although we are aware that it may be difficult for the compiler to perform.

Other model device-loading functions such as *CAPload* (capacitor load), *DIOload* (diode load), *VSRCload* (voltage-source load), and many more as shown in Fig. 6, have a program structure very similar to *MOS3load*, so they can be parallelized the same way.

```
int MOS3load(inModel,ckt)
    GENmodel *inModel;
    register CKTcircuit *ckt;
{   ......
   register MOS3model *model = (MOS3model *) inModel;
   register MOS3instance *here;
   ......
   MOS3instance **MOS3instanceArray;
   MOS3instanceCount = model->MOS3instanceCount;
   MOS3instanceArray = model->MOS3instanceArray;
#pragma omp parallel default(none) shared(ckt,
CONSTKoverQ,MOS3instanceCount,MOS3instanceArray)
  #pragma omp for private(vt,Check, SenCond,EffectiveLength,DrainSatCur, SourceSatCur, \
     GateSourceOverlapCap,GateDrainOverlapCap,GateBulkOverlapCap,Beta,OxideCap,vgs,vds,vbs, \
     vbd,vgb,vgd,xfact,vgdo,delvbs,delvbd,delvgs,delvds,delvgd,cbhat,cdhat,tempv, \
     cdrain,capgs,capgd,capgb,von,evbs,evbd,vdsat,cdreq,xrev,xnrm, ceqbd,ceqbs,ceqgb, \
     ceq,geq,vgs1,vgd1,vgb1,arg,sarg,sargsw,error,gcgs,ceqgs,gcgd,ceqgd,gcgb,model,here)
    for( i = 0; i < MOS3instanceCount; i++) {
         here = MOS3instanceArray[i];
         model = here->MOS3modPtr;
         ......
       #pragma omp critical(lockA)
        {  // Right hand side of Ax = b
           *(ckt->CKTrhs + here->MOS3gNode) -=   (model->MOS3type * (ceqgs + ceqgb + ceqgd));
             ......
          //   Sum of contributions for the element of matrix A
           *(here->MOS3DdPtr) += (here->MOS3drainConductance);
           *(here->MOS3GgPtr) += ((gcgd+gcgs+gcgb));
           ......
        }  /* end critical  */
}  /* end of for loop */
   return(OK);
} /* end of MOS3load() */
```

Fig. 5 OpenMP implementation of MOS3load in SPICE3

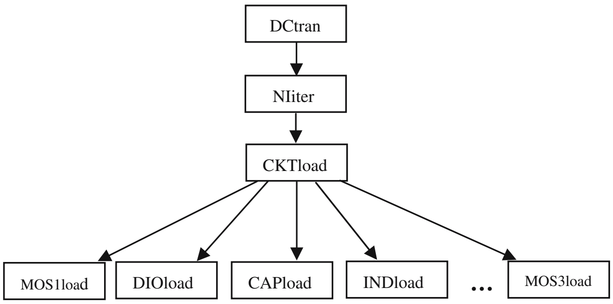


Fig. 6 Structure of a partial callgraph for a transient simulation of SPICE3 simulator program

The shortcoming of parallelizing this code by creating the parallel region within the *MOS3load* routine is that the routine is called many times, thereby involving considerable fork-join overheads in addition to the cost of the synchronization. Barrier and critical section overheads are, however, unavoidable.

2.3 Possible Improvements

The most important issue is to reduce the significant fork-join overhead incurred, since *MOS3load* is invoked several hundreds to thousands of times by other functions. The

```
……..
for (i=0;i<DEVmaxnum;i++) {
      if ( ((*DEVices[i]).DEVload != NULL) && (ckt->CKThead[i] != NULL) ){
         error = (*((*DEVices[i]).DEVload))(ckt->CKThead[i],ckt);
         if (ckt->CKTnoncon)
}}
…..
```

Fig. 7 Dynamically bound call from *CKTload* routine

total number of calls to *MOS3load* depends on the number of time points and the non-linear iteration counts at each time in the simulation. As the number of threads increases, fork-join overheads increase significantly. To improve this situation, we need to move the parallel region to include the calling functions. Unfortunately, it is non-trivial to do so. To explain the difficulties, we first manually created an incomplete or partial callgraph for a transient simulation of the SPICE3 program that shows invocations of our *MOS3load* function in Fig. 6. This callgraph is built from dynamically bound calls resulting from the variable pointer assignment mechanism. It is a quite time consuming task and it would be preferable to have a tool to assist in doing this, in particular to help a novice developer gain an understanding of the code.

Suppose we are able to move the *omp parallel* from the *MOS3load* function to *CKTload*. There will be no improvement because there is only one dynamically bound call from the *CKTload* routine; as shown in Fig. 7, this is realized by the value of the pointer *(*((*DEVices[i]).DEVload))(ckt->CKThead[i],ckt)*, which can point to *CAPload()*, *MOS2load()*, *MOS3load()*, *INDload()*, and/or many other device loading functions corresponding to calling relationships shown in Fig. 6. It can be improved by moving it to the function *NIiter*. From inside the *for(;;)* loop in the *NIiter* function, there is a call to the *CKTload* function; this loop iterates until the convergence criterion is met. Further, there is a call to *NIiter* from inside the *while(1)* loop of the *DCtran* function of Fig. 6; these calls continue until the final transient time is reached. In other words, the call from *DCtran* to *NIiter* represents the outer loop and the call from *NIiter* to *CTKload* represents the inner loop of Fig. 1. The parallelization of the code becomes more tedious, however.

2.4 Implementing Parallel SPICE3 without Code Modification

In this section, we discuss the OpenMP implementation of SPICE3 using Intel's *omp taskq* and without any other modifications to the original program. The detailed description of Intel's *taskq* pragma, implemented in icc as a workqueueing model by the Intel compiler group can be found in [7,8]. The workqueuing model of Intel's *taskq* provides a flexible mechanism for specifying units of work that are not known or pre-computed at the beginning of the worksharing construct. In other words, the *taskq* and *task* pragmas are used to create dynamic tasks. It is extremely useful for handling dynamic data structures such as linked lists and trees, as well as for programs with control structures such as while loops and recursion. Conceptually, a *taskq* pragma causes the chosen thread to create an empty queue of tasks. The code inside

```
int MOS3load(inModel,ckt)
{  .................
  #pragma omp parallel shared(ckt, CONSTKoverQ,inModel)
   for( ; model != NULL; model = model->MOS3nextModel ) {
  #pragma intel omp taskq private(.....)
     for (here = model->MOS3instances; here != NULL; here=here->MOS3nextInstance) {
       #pragma intel omp task
        {  .....
           if (SendCond) continue;
           .....
           if (here->MOS3senPertFlag == OFF) continue;
            .....
        } /* end of omp task */
   }}  /* end of for loop */
  return(OK);
}
```

Fig. 8 Device instance level of granularity

a *taskq* should be a structured block; it is executed in single-threaded mode. Any *task* pragmas encountered while executing a *taskq* block specify that the enclosed work is associated with the queue and can be executed by any thread, and the unit of work is conceptually enqueued for subsequent execution.

Our implementation with few modifications discussed in Sect. 2.2 has been parallelized to perform device instance calculations per thread. To parallelize using *Intel omp taskq* at this level is not possible without more code modifications, because there are several branch statements such as *return* and especially *continue* in the inner loop of the *MOS3load* function as shown in Fig. 8, which is not a structured block and is not allowed by the compiler.

In order to parallelize it without any modification, we have to parallelize at a coarser granularity, namely at the level of device instances of a model (each model consists of many device instances) per task. This is shown in Fig. 9. With this level of granularity (model per thread), the load imbalance problem can occur, since each model may consist of a different number of device instances, but then the OpenMP directives can be inserted directly into original program.

The source code of Fig. 9 has been successfully compiled by Intel's OpenMP compiler on Linux Itanium of 4-CPUs machine. We include this experimental result in the next section.

3 Experiments

Our experimental results are based on an 8 K SRAM model simulation compiled with SUN compiler and run on a SunFire V880 Ultra SPARC-III 750 MHz with four CPUs and 4 G memory. In this simulation, *MOS3load* is invoked 780 times; this number depends on the number of time points simulated and the non-linear iteration counts of each time. An increase in this number does not affect the scalability of the performance. On the other hand, there are 61,584 total instances, which represent the size (number of iterations) of the coalesced loop iterations in *MOS3load*. This in turn depends on

```
int MOS3load(inModel,ckt)
{
#pragma omp parallel default(none) shared(ckt, CONSTKoverQ,inModel)
  #pragma intel omp taskq private(vt,Check, SenCond,EffectiveLength, \
                                  DrainSatCur,SourceSatCur, ........... ,model,here)
   for(model=inModel; model!=NULL;model=model->MOS3nextModel) {
   #pragma intel omp task
      for(here=model->MOS3instances; here!=NULL; here=here->MOS3nextInstance){
        ...
      #pragma omp critical(lockA)
       {    // Right hand side of Ax = b
            //  Sum of contributions for the element of matrix A
        } /* end critical */
   }}
   return(OK);
}
```

Fig. 9 All instances of a model level of granularity

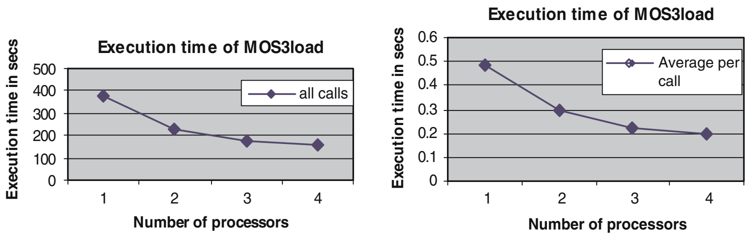
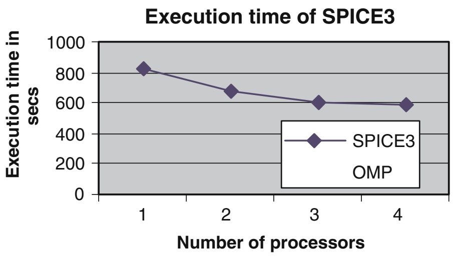


Fig. 10 The performance of OpenMP MOS3load function

Fig. 11 The performance of OpenMP SPICE3



the number of device instances involved in the circuit simulation. Figures 10 and 11 show that it performs well up to three processors for the entire SPICE3 program (note that the rest of SPICE3 is not parallelized here).

Other than the device model and instance calculation, the sparse matrix computation in SPICE3 is fairly time consuming. We are currently studying the parallelization of a public domain linear algebra sparse matrix package implemented in SPICE3 by Kundert [9]. It is not part of this paper. The code uses an orthogonal linked list data structure to store the sparse matrix. The matrix computation technique is based on the direct method with LU decomposition. With four processors, the fork-join overhead costs more than the execution time of each call to *MOS3load*, since its average execution time is only about 0.2 s. To scale well, we would need to simulate a larger number of device instances.

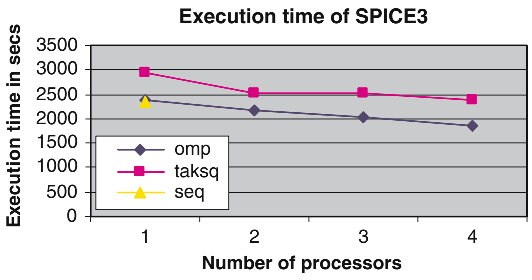


Fig. 12 The performance of OpenMP SPICE3 using Intel taskq runing on Itanium

We further performed the experiment with Intel's *taskq* as shown in Fig. 9 and compare it with the approach given in Fig. 5. Both were compiled with Intel icc compiler version 9.0 under the option –O2 –openmp and run on a 4-processor IBM eServer xSeries 380 Itanium 733 MHz with 16 GB memory. Again, we only parallelize the MOS3load part and not the rest of SPICE3, but this time we simulate based on a 16 K SRAM model simulation, which has a larger number of device instances. The results in Fig. 12 show that both perform well for the entire SPICE3 program if the matrix calculation part is also parallelized. But, with the *taskq* construct, significant runtime overheads are incurred compared to the original sequential code without –openmp compilation option labeled by '*seq*' in the figure.

The programs labeled "*CAPload*", "*MOS3load*", and "*CAPload+MOS3load*" correspond to the measurement of the whole SPICE with parallelized *CAPload()* function alone, *MOS3load()* function alone, both *CAPload()* and *MOS3load()* functions respectively; these versions have been parallelized using *omp parallel for* directive. Another three versions labels begin with "*Taskq*" correspond to the measurement of the whole SPICE with parallelized *CAPload()* function alone, *MOS3load()* function, and both *CAPload()* and *MOS3load()* function respectively; these three versions were parallelized using *omp taskq*. Each version takes 16 K SRAM based on MOS3 model simulation running on a DELL PowerEdge SC1420 Intel Xeon Processor 3.0 Ghz/2 M, Em64T 800 Mhz FSB *2 and 1G DDR400 memory. They were compiled with the Intel icc compiler version 9.1.038 under the option –O2 –openmp. Figure 13 shows the speedup of different versions of parallel SPICE. The result of parallelizing SPICE using *taskq* for both *MOS3load* and *CAPload* functions, labeled "*Taskq_MOS3load+CAPload*", gives comparable speedup to the other versions but at lower cost. In other words, it offers relatively good performance but requires less programming effort than the other versions.

4 Conclusions and Future Work

We have developed an OpenMP SPICE3 circuit simulator program. The matrix and model device calculations are the two most time-consuming parts of the computation. We present our implementation of the device model and instance calculation part of SPICE3. Our goals were to minimize the effort required and the amount of modification of the original program. Our experimental results are promising in this respect, despite the data structures used. We discussed possible improvements; how-

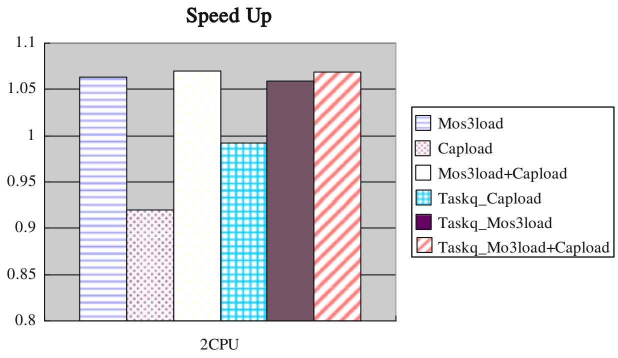


Fig. 13 The performance of OpenMP SPICE3 using Intel taskq runing on Xeon

ever, they do require more programming work. In the implementation of SPICE3, there are many while-loops, pointer-loops, and non-block structures, all require major rewriting if they are to be parallelized by current standard OpenMP features. As an alternative approach, we can use the *omp taskq* pragma since it is relatively easy to employ it without modifying the original code, and thus it enables us to develop a parallel application similar to our SPICE program using OpenMP with less effort. But for the workqueueing model implemented in the compiler, further improvement is essential.

The data handling of a bigger loop involves dubious variable declarations. This may affect the correctness of the program. In our case, there are explicit declarations of many private variables, and automatic data scoping may be needed. A compiler tool for a novice user to create a precise callgraph of the corresponding input program in the presence of dynamically bound calls (in Sect. 2.3) may aid programmer understanding of the code. We are continuing our work by creating an OpenMP version of the sparse matrix calculation (Sparse matrix package in SPICE3) that is also a time-consuming part of SPICE3.

Acknowledgements We are grateful to our colleagues in the Sun Center of Excellence in Geosciences at the University of Houston for their provision of a platform that enabled us to perform the experiments reported on in this paper.

References

1. Nagel, L.W.: SPICE2 - a computer program to simulate semiconductor circuits. University of California, Berkeley, ERL. Memo ERL-M520 (May 1975)
2. Quarles, T.L.: Analysis of performance and convergence issues for circuit simulation. University of California, Berkeley, ERL. Memo ERL-M89 (April 1989)
3. Lee, P.M., Ito, S., Hashimoto, T., Sato, J., Touma, T., Yokomizo, G.: A parallel and accelerated circuit simulator with precise accuracy. Procedings of the 15th International Conference on VLSI Design (2002)
4. OpenMP Architecture Review Board, Fortran 2.0 and C/C++ 1.0 Specifications. At www.openmp.org
5. Massaioli, F., Castiglione, F., Bernaschi, M.: OpenMP parallelization of agent-based models. Parallel Comput. **31**(10–12), 1066–1081 (2005)
6. Lin, Y., Terboven, C., Mey, D., Copty, N.: Automatic scoping of variables in parallel regions of an OpenMP program. WOMPAT 83–97 (2004)

7. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for intel SMP architectures. EWOMP. Rome, Italy (2002)
8. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallelism in OpenMP. EWOMP. Lund, Sweden (1999)
9. Kundert, K.: Sparse Matrix techniques. In: Ruehli, A. (ed.) Circuit Analysis, Simulation and Design. North-Holland (1986)