A Multi-Threaded Time Domain TLM Algorithm for Symmetric Multi-Processing Computers

Poman P.M. So and Wolfgang J.R. Hoefer

Computational Electromagnetics Research Laboratory
Department of Electrical and Computer Engineering
University of Victoria
PO Box 3055, Victoria, BC, V8W 3P6, Canada

E-MAIL: POMAN.SO@ECE.UVIC.CA

***Abstract* — A multi-threaded TLM algorithm will be presented in this paper. The algorithm enables time domain field simulators to exploit the full computing power of multi-processor computers, resulting in improved performance of field solver based CAD tools running on multi-processor computers.**

I. Introduction

The processing power of Windows-based computers has been improving steadily in recent years. The improvement is due to the increase in CPU frequency, decrease in memory access time, and the advances in symmetric multi processing (SMP) technology. Intel Pentium based computers with dual processors are available from major computer makers such as COMPAQ, DELL, HP, IBM, and SGI. Super SMP computers with multiple RISC processors are available from COMPAQ, CRAY, HP, IBM, SGI and SUN. These multi-processor computers have multiple high performance CPUs that can work either independently or cooperatively with each other. If field solvers such as TLM are to exploit the processing power of the extra CPUs in the multi-processor computers, their underlying algorithms must be re-structured to contain multiple threads. A thread is a path of execution in a program unit; a multi-thread program defines multiple execution paths that can be executed in parallel should the computer have the capability to do so. On single processor computers, the threads are executed in a time-sharing fashion; on *N*-processor SMP computers, *N* threads can be executed in parallel.

A traditional single-threaded TLM program performs computation in a sequential manner via a system of do-loops. Just recompiling the program with a new compiler for the multi-processor computers could not transform a single-threaded algorithm to a multi-threaded version because the compiler cannot change the program semantics. In the following sections, we will present a general approach for transforming a single-threaded TLM algorithm into a multi-threaded implementation.

II. Overview Of The TLM Algorithm

The fundamental procedures of the TLM algorithm are the scattering and transfer of voltage impulses on a TLM mesh. Impulses incident on a node are scattered in all directions and then propagate to the neighboring nodes in a manner similar to a water wave propagating away from the center of its initial disturbance. In matrix form, the impulse scattering phenomenon at any node can be expressed as:

$$[V]^r = [S] \cdot [V]^i \tag{1}$$

Where $[V]^i$ and $[V]^r$ are the vectors of the incident and reflected voltage impulses, and $[S]$ is the impulse scattering matrix, [1]. Initially, the TLM mesh contains only incident impulses. After applying equation (1), the TLM mesh contains reflecting impulses. The propagation effect is modeled by swapping impulses on the transmission lines that connect two neighboring nodes. This sequence of operations forms the TLM algorithm. When boundaries and sources are present in the mesh, additional operations must be included. Tong and Fujino [2] developed an efficient scattering algorithm by eliminating unnecessary floating-point operations. Since then, researchers have been revisiting their TLM codes to reduce floating-point operations in the scattering procedure. A recent and better-known scattering algorithm that achieves a minimum number of floating-point operations per scattering step is given in Trenkic's GSCN formulation [3]. However, as pointed out by

0-7803-6540-2/01/$10.00 (C) 2001 IEEE

Mangold et. al. [4], floating-point operation is just one of the issues that must be addressed when using TLM code to solve real engineering problems requiring a large TLM mesh. In such a situation, memory access time becomes a bottleneck. Mangold *et. al.* have outlined a unified scattering-transfer procedure to minimize the impact of memory access time overhead.

Besides reducing floating-point operations and memory access time overhead, the performance of TLM code can be improved further by using massively parallel computers. So and Hoefer, [5] and [6], have ported SCN algorithms to a number of massively parallel computers. These computers all have a large number (8 to 32K) of CPUs; each CPU is used to model a TLM node. In order to take advantage of these massively parallel computers (MPPs), the scattering and transfer operations must be implemented using parallel languages designed for those computers. Unfortunately, these computers are expensive, hence not widely available for general use.

Since the current trend in high-performance computing architecture is towards multi-processor computers, a multi-thread TLM algorithm that exploits the full power of these computers must be developed. Because scattering and transfer of impulses are the two CPU intensive operations of the TLM algorithm, we will concentrate our discussions on these two procedures. Nevertheless, the multi-threaded paradigm is very general and can be applied to other critical paths of a field solver as well as to other time and frequency domain based field solvers.

III. Division of A TLM mesh into Sub-Regions

The scattering and transfer procedure of the TLM algorithm are always nested inside a system of iteration loops that control the order of execution per time step; Listing 1 shows the semantics of such a system of loops. The scattering function in Listing 1, `Scatter_impulses`, loops over the entire three-dimensional TLM mesh. In order to support the multi-thread programming paradigm, the scattering function must be modified to loop over a sub-region of the mesh. The `Scatter_impulses` function in Listing 2 contains a parameter list that allows the calling function to control the size of the sub-region to be used in the scattering operation. Similar modifications can be made for the impulse transfer and signal processing functions as well as for the field animation procedures.

Listing 1. A typical TLM control loop with its associated impulse scattering function, `Scatter_impulses`. The local variables `x_size`, `y_size` and `z_size` in the scattering function specify the dimensions of the TLM mesh. The *C++* syntax detail has been omitted from the functions.

```
Iteration(){
  for (n=0; n<num_of_iterations; n++){
    Inject_source_voltages();
    Scatter_impulses();
    Transfer_impulses();
    Enforce_boundary_conditions();
    Sample_output_signal_and_signal_processing();
    Compute_field_animation_data();
    Display_result();
} }
Scatter_impulses(){
  for (x=0; x<x_size; x++){
    for (y=0; y<y_size; y++){
      for (z=0; z<z_size; z++){
        Mesh(x,y,z).Scatter_impulses();
} } } }
```

Listing 2. A modified TLM scattering function.

```
Scatter_impulses(x1, y1, z1, x2, y2, z2){
  for (x=x1; x<x2; x++){
    for (y=y1; y<y2; y++){
      for (z=z1; z<z2; z++){
        Mesh(x,y,z).Scatter_impulses();
} } } }
```

IV. Creation of Helper Threads

The critical modification that transforms the iteration procedure in Listing 1 to a multi-threaded version is given in Listing 3. `Thread_Data` is an object type that provides storage for thread specific data, such as the indices for the mesh sub-region as well as the TLM operation to be performed on the sub-region. The use of these data is demonstrated in the `Thread` function.

In order for the `Iteration_Thread` function to work properly, all `Thread_Data` variables must have their `op` data member initialized to `Wait`. The `Iteration` function of the main program is responsible for setting the `op` data members to `Scattering`, `Transfer` and `Stop` at the appropriate time. This, in turn, causes the helper threads to execute the corresponding TLM functions over

a pre-defined sub-region of the TLM mesh. The helper threads will go into a wait state upon completion of the assigned operations.

The sub-region indices for the `Scatter_impulses` and `Transfer_impulses` functions can be computed by simply dividing the mesh equally along the longest axis of the TLM mesh. This is usually sufficient for computers with 2 to 4 processors. For SMP with more than 16 processors, a recursive procedure that divides the mesh up along all three axes may be implemented

The `Create_Iteration_Threads` function in Listing 3 is implementation dependent. A version for Windows MFC application is shown in Listing 4, for IBM AIX, `pthread_create` should be used to replace the Windows `AfxBeginThread` function.

Listing 3. A multi-thread iteration algorithm.

```
Iteration(){
  Thread_Data data[n];
  Initialize_Thread_Data(data,n);
  Create_Iteration_Threads(data,n);
  for (n=0; n<num_of_iterations; n++){
    Inject_source_voltages();
    Signal_iteration_thread(data,n,Scattering);
    Wait_until_all_threads_are_done(data,n);
    Signal_iteration_thread(data,n,Propagation);
    Wait_until_all_threads_are_done(data,n);
    Enforce_boundary_conditions();
    Sample_output_signal_and_signal_processing();
    Compute_field_animation_data();
    Display_result();
  } Signal_iteration_thread(data,n,Stop);
}
Thread(Thread_Data &d){
  while (d.op!=Stop){
    switch (d.op){
    case Scattering:
      Scatter_impulses(d.x1,d.y1,d.z1,
                       d.x2,d.y2,d.z2);
      d.op=Wait;    break;
    case Transfer:
      Propagate_impulses(d.x1,d.y1,d.z1,
                         d.x2,d.y2,d.z2);
      d.op=Wait;    break;
    case Wait:
    default: sleep(0);    break;
} }
```

Listing 4. A Windows MFC threads creation function.

```
Create_Iteration_Threads(data, n){
  for (m=0; m<n; m++){
    data.op     = Wait;
    data.thread = AfxBeginThread(Thread,data+n);
} }
```

V. Verification of the Algorithm

We have implemented the multi-threaded TLM algorithm for the Windows 95/98/ME/2000/NT and IBM AIX operating systems. A coax-to-waveguide transition, Figure 1, was used to evaluate the performance of the algorithm on an 8-processor IBM RS/6000 SMP computer. The structure was discretized with a mesh of $100\times50\times50\ \Delta l^3$, 250000 nodes. The observed processing times for 100 iterations using different numbers of threads are tabulated in Table 1. In principle, the improvement in performance should increase linearly with the number of threads. The maximum performance would be reached when the number of threads is equal to the number of processors. However, the results in Table 1 indicate that the expected maximum improvement did not materialize. This is because the system was heavily loaded with other jobs, and not all processors were allocated to execute the helper threads in parallel. By populating the system with a large number of threads, the TLM job was able to obtain more CPU time slices and hence, better performance.

Similar tests were done on some single-processor PCs. On these systems, the threads were executed sequentially in a time-sharing fashion. Furthermore, when the TLM engine and user interface were executed in separated threads, the software became more responsive to user inputs. Therefore, the multi-threaded algorithm can be used routinely on traditional single processor computers.

Table 1 CPU performance versus number of threads on a 8-processor IBM RS/6000 SMP computer. The last column is a measure of performance based on the elapsed time.

Number of Threads	Elapsed Time in seconds	CPU Time in seconds	kilo-nodes per second
1	60.32	29.67	414
2	32.64	30.03	766
4	30.78	29.70	812
8	29.95	30.04	835
16	19.91	29.94	1,256
32	12.43	28.66	2,011
64	11.80	27.10	2,119

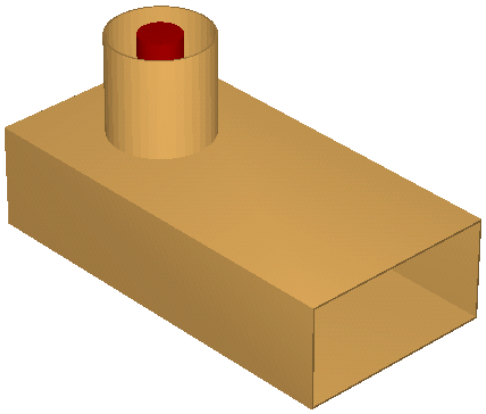

Figure 1 Coax-to-waveguide transition used to evaluate the performance of the new multi-threaded TLM algorithm; the structure was discretized with a mesh of $100\times50\times50$ Δl^3, 250000 nodes.

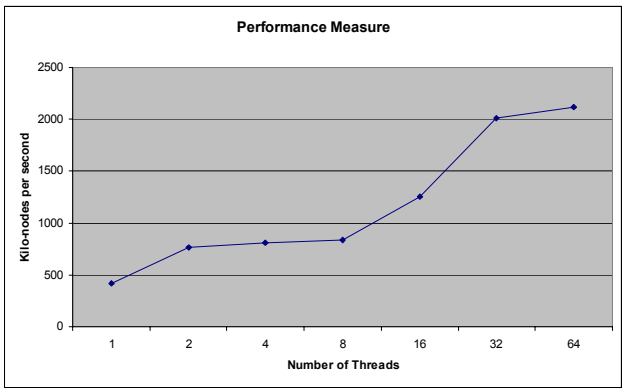

Figure 2 Performance enhancement as a function of the number of threads. The data for this graph are given in Table 1.

VI. Conclusion

A multi-threaded TLM impulse scattering and propagation algorithm has been presented. The technique is very general and can be applied to all critical paths of a TLM field solver to accelerate its field solving speed on multi-processor computers. In principle, drastic improvement in speed can be achieved by using a thread of execution on each available processor. However, our simulation results obtained with an 8-processor IBM RS/6000 SMP computer did not achieve the theoretical maximum performance. Moreover, by populating the processors with a large number of threads allows the field solver process to obtain more time slices from the operating system; this improves the field solver performance on a heavily loaded system.

Signal processing components and graphical user interface modules are also needed for computation of engineering parameters and display of results. However, these added features generally do not incur a significant overhead in CPU time. Two exceptions are the discrete Fourier transform (DFT) and field animation subroutines; they tend to consume a large portion of CPU time when a large number of frequency points are used, and high quality graphic output is demanded.

References

[1] P.B. Johns, *A Symmetrical Condensed Node for the TLM Method*, IEEE Trans. Microwave Theory and Tech. vol. MTT-35, no. 4, pp. 370-377, April 1987.

[2] C.E. Tong and Y. Fujino, *An Efficient Algorithm for Transmission Line Matrix Analysis of Electromagnetic Problems Using the Symmetrical Condensed Node*, IEEE Trans. Microwave Theory and Tech. vol. MTT-39, No.8, pp.1420-1424, August 1991.

[3] V. Trenkic, C. Christopoulos, and T. M. Benson, *Efficient Computation Algorithms for TLM*, in Proc. 1st International Workshop TLM Modeling — Theory and Application, pp. 77-80, Aug. 1-3, Victoria, B.C., Canada, 1995.

[4] T. Mangold, J. Rebel, W.J.R. Hoefer, P.P.M. So and P. Russer, *What Determines The Speed of Time-Discrete Algorithms?,* 16th Annual Review of Progress in Applied Computational Electromagnetics, pp.594-601, March 20-24, 2000, Monterey, California.

[5] P.P.M. So, C. Eswarappa and W.J.R. Hoefer, *Distributed Parallel TLM Computation and Digital Signal Processing for Electromagnetic Field Modelling*, an invited paper, International Journal of Numerical Modelling - Electronic Networks, Devices and Fields, vol. 8, no. 3/4, May-August 1995, pp 169-185, John Wiley & Sons Inc..

[6] P.P.M. So, C. Eswarappa and W.J.R. Hoefer, *Transmission Line Matrix Method on Massively Parallel Processor Computer*s, 9th Annual Review of Progress in Applied Computational Electromagnetics Digest, pp.467-474, March 1993, Monterey, California.