

Multitasking TORT Under UNICOS: Parallel Performance Models and Measurements

D. A. Barnett
Lockheed Martin Corporation
PO Box 1072
Schenectady, NY 12301

Y. Y. Azmy
Oak Ridge National Laboratory
PO Box 2008, MS 6363
Oak Ridge, TN 37831
yya@ornl.gov

Abstract

The existing parallel algorithms in the TORT discrete ordinates were updated to function in a UNICOS environment. A performance model for the parallel overhead was derived for the existing algorithms. The largest contributors to the parallel overhead were identified and a new algorithm was developed. A parallel overhead model was also derived for the new algorithm. The results of the comparison of parallel performance models were compared to applications of the code to two TORT standard test problems and a large production problem. The parallel performance models agree well with the measured parallel overhead.

1 Introduction

TORT is a three-dimensional, neutral particle transport code which is widely used in the nuclear industry (Rhoades, 1997). A large TORT problem typically involves several million spatial cells, several hundred directions and tens of energy groups. While TORT is optimized to solve large problems, a large TORT problem can easily consume the majority of a machine's resources, including memory, CPU cycles and disk space. Memory is usually the scarcest resource, and while TORT employs sophisticated memory management techniques, it will always work best with as much memory as it can allocate. In such a situation, it is essential that TORT also make maximum use of multiple processors so that a large job can minimize the time it resides on the system.

Version 2 of TORT included two different multitasking implementations (Rhoades, 1989). These were based on multitasking libraries which were specific to the CRAY CTSS operating system that were later converted to UNICOS. However, both of the extant methods had shortcomings which it was necessary to overcome in order to create an efficient parallel version of the code for UNICOS. UNICOS is a time-sharing operating system in which fairness is the watchword. Every process with equal priority gets an equal share of CPU cycles: A process is allowed to run for a period of time called a quantum (1/60 s) then that process is suspended and a different process is given the CPU. The operating system also looks for every opportunity to shift CPU resources from processes which are idle to processes which are active. In general, this means that multitasking programs which synchronize in a period shorter than the quantum incur a substantial penalty.

To create an efficient multitasking program in such an environment requires attention to two particular attributes of the code: minimization of synchronization points and minimization of redundant operations. The former attribute tries to insure that a process will run at least until its quantum expires. The latter attribute insures that a process will accomplish the most useful work in that period. This paper describes how these two objectives were implemented in TORT and how a parallel performance model was used to measure the degree to which we have succeeded in achieving them.

2 CRAY Macrotasking

Version 2 of TORT used direct calls into the parallel processing library under CTSS to start and stop

tasks and to lock global arrays when multiple tasks accessed them directly; it also used a CRAY FORTRAN-specific concept of TASK COMMON to give each task its own private version of a COMMON block. Collectively, this approach to multiprocessing was referred to as macrotasking since parallelism was obtained at the subroutine level and under the direct control of the programmer. While these calls are no longer the recommended way to write multiprocessing code, they are still available under UNICOS.

The multiprocessing paradigm of macrotasking is a shared memory model. The shared memory model makes it simple to distribute information among tasks, but tasks must serialize writes into globally shared memory locations. Macrotasking provides both lock and barrier routines to assure this synchronization.

To aid the performance of parallel programs in a time-sharing system, the UNICOS macrotasking synchronization routines do not immediately block (and thereby yield the CPU) when a task enters a lock or a barrier which is not satisfied. Instead, the library routines may repeatedly test the synchronization hardware in a tight loop for a certain period of time; this is called *spin waiting*. The interesting aspect of the spin waiting under UNICOS is that the amount of CPU time devoted to it is determined automatically by the system itself. Under a heavy system load, a task will spin wait for a short period; under a light load, a task may spin wait longer. The system documentation (CRI, 1996) calls this *autotuning*. In our initial experience with macrotasking, this led to the counter-intuitive behavior that runs made on a heavily loaded system actually reported using less CPU time than the same run made on a lightly loaded system.

It is possible to defeat this behavior by setting the spin wait hold time to a fixed value (called *fixed tuning*) via the library command `tsktune`. In order to measure consistent CPU times in the performance model developed here, it was necessary to set hold time to zero. However, even though turning off autotuning generally results in the least CPU time for a program, it does not result in the least wall-clock time. The purpose of spin waiting is to keep tasks synchronized without having to yield the CPU; since context switches are expensive operations, the best performance is obtained when they are minimized.

3 Original Multitasking Algorithms

In Cartesian geometry, the loss term of the transport equation does not couple the angular flux between any of the discrete directions. Therefore, the sweep of distinct directions along an X row can be performed in parallel. This forms the basis for the parallel algorithms used in TORT. Version 2 of TORT contained two different parallel algorithms: octant parallel (OP) and row parallel (RP). In the OP method, two tasks were started for each row visited by the sweeping algorithm: One swept the row solving all the discrete directions in the octant for $\mu_n > 0$ (where μ_n is the direction cosine of the ordinate in the X direction), the other task swept in the other direction solving all the $\mu_n < 0$ directions. The obvious limitation of this algorithm to a factor of two speed-up was reasonable at the time it was implemented since the target platform, a CRAY X-MP, had only two processors. A more subtle limitation was that the problem was limited to either vacuum or periodic boundary conditions in X; other boundary conditions would have required the use of boundary values calculated from the previous iteration which, in general, retards convergence. OP also provided the first glimpse as to how important it would be to maximize the amount of computation performed between synchronizations. Since the OP method started a new task for each row, it was essential that the amount of work each task performed was much larger than the task start-up time (Rhoades, 1989).

The RP method attempted a finer grained parallelism. Here, a new task was started for each direction in a single octant. This allowed for a higher degree of parallelism and it eliminated the constraint on boundary conditions since all tasks rendezvoused at the end of the row sweep and therefore the most

recently calculated exiting angular flux information was always available. The drawbacks to RP as implemented in version 2 of TORT were that it was also hand-coded explicitly for two processors and, because it started a new task for each direction, there was less work for each task to accomplish before task termination at the end of the row.

4 Parallel Performance Model of the Direction Parallel Method

For version 3 of TORT, the multitasking code was revived for use under the CRAY UNICOS operating system. In addition to adapting the code to UNICOS macrotasking, significant work was done to improve the RP method. To distinguish the improvements made to the old RP method, the new algorithm was dubbed Direction Parallel, which, while still in the spirit of RP, shares very little code with the original method (Azmy, 1996). The initial implementation of Direction Parallel (DP1) featured two significant alterations of the RP algorithm:

1. The call to start a new task was moved out of the loop over angles so that only as many tasks which are requested in the input are started at the beginning of each row sweep.
2. Rather than statically assigning a specific task to each angle, each task is allowed to grab an angle to solve from a queue. This is referred to as *dynamic scheduling*.

The idea behind dynamic scheduling is to reduce the amount of synchronization time required while increasing the amount of useful computations each task may perform. Thus, as each task reaches the end of a row sweep along one discrete ordinate, it checks if there is any remaining discrete ordinates in the queue and if there are, it takes the next angle and performs the row sweep for that angle. As long as retrieving an angle from the queue does not block, the task will continue to run until the end of the quantum.

When a multitasked program is executed in parallel on a time-sharing system, the maximum wall clock speed-up that can be achieved is largely controlled by the other work on the system. The best that can be hoped for is that, when there are free processors, all of the tasks started by TORT will be in a state to execute. This is called the READY state in UNICOS. When a process' quantum expires or it blocks in a system call, the next available process in the READY queue is assigned to the processor and its execution proceeds. Therefore, the goal in optimizing a multitasked program is to insure that tasks have as much to do as possible and that redundant work is minimized. Such non-useful work, i.e., operations not performed in the sequential code, or multiply performed in the parallel code, is referred to as *overhead*.

In order to quantify the overhead behavior in DP1, a parallel performance model was constructed. Typically, a parallel performance model will describe how much elapsed time a process will require as the number of participating processors is varied. In an ideal system, this function would follow Amdahl's law. In the context of UNICOS time-sharing operation, however, it is not useful to make estimates of elapsed time since the code's performance will primarily depend on the load on the system, something which is not under the control of the programmer. The only aspect of the code's performance which is under the control of the programmer is the overhead. The goal, then, of this parallel performance model is to identify which areas of overhead contribute the most to preventing a code from realizing its maximum potential parallel speed-up.

In DP1, the sources of overhead are characterized as follows:

1. Creation of slave tasks: Slave tasks are created once for each left-right and right-left row sweep, thus the total extra time TORT spends creating tasks is:

$$T_{task}(ncpu) = \tau_{task} \times itnfl \times km \times jm \times Q \times (ncpu - 1), \quad (1)$$

where τ_{task} is the CPU time required to start a new task, $itnfl$ is the number of iterations performed (i.e., complete sweeps through all space and angles), km is the number of planes, jm is number of rows in a plane, Q is number of quadrants (which is always equal to four) and $ncpu$ is number of tasks requested in the TORT input. TORT's parent task participates in all calculations so only $ncpu - 1$ additional tasks are started.

2. Lock/unlock flux moments array: At the end of each direction's row sweep, each task accumulates its angular flux's contribution into the cell flux angular moments. Since the flux moments array is a global array, this accumulation must be performed under a lock (i.e., sequential mode). The overhead computed for this step is the cost of calling the lock and unlock functions. The total time TORT spends in the lock routines is:

$$T_{lock} = \tau_{lock} \times itnfl \times km \times jm \times N_{lock} \times mm_{w \neq 0}, \quad (2)$$

where τ_{lock} is the total CPU time to perform a single lock and unlock, N_{lock} is the number of times the lock/unlock is performed per row sweep and $mm_{w \neq 0}$ is the number of directions in the quadrature set for which the quadrature weight is not equal to zero.

3. Memory management: Even though each slave task can see all of the process address space, a small amount of information related to memory pointers and common block information private to each task is manipulated as each task starts. The total time TORT spends performing these memory management operations is:

$$T_{memory}(ncpu) = \tau_{memory} \times itnfl \times km \times jm \times Q \times ncpu, \quad (3)$$

where τ_{memory} is the total CPU time per memory operation.

4. Redundancy in the loop over angles: in the adaptation of RP to DP1, the dynamic scheduling of angles was implemented by retaining the old sequential loop over angles which each task executes in full. However, if the current angle index is not equal to that selected by the task from the dynamic scheduling queue, the task simply skips the row sweep for that angle. There are a few operations which must be performed even if the angle is not processed; these operations are related to updating certain pointers which reference per angle information. Because some of this time is distributed over the parallel tasks, the timing parameters were measured in two separate runs, one where the time of all tasks was measured by serializing the tasks and another run where only one task did all the work. The difference between the two measurements is the overhead. The total time TORT spends in this code section is:

$$T_{redunant}(ncpu) = ((ncpu - 1)\tau_{not-own} + \tau_{own} - \tau_{vector}) \times itnfl \times km \times jm \times mm \quad (4)$$

where τ_{own} is the CPU time spent in a loop when the task owns the angle and $\tau_{not-own}$ is the CPU time spent in the loop when the task does not own the angle, τ_{vector} is the CPU time for one task to do all the work in vector mode, and mm is the total number of directions in the quadrature set.

5. Hold time: As mentioned before, the UNICOS libraries which implement the macrotasking directives can cause a process which has entered a synchronization point (such as a lock) to spin wait. Because the exact amount of time a task will spin wait is determined dynamically by the system, there is no way to quantify this CPU time penalty. For the purposes of the parallel performance model, this capability was disabled in TORT by calling the function `tsktune` with hold time set to zero (CRI, 1996).

To verify the model, τ_{task} and τ_{lock} were measured with a separate program which performed the operations in a tight loop; the values of τ_{memory} , τ_{own} , $\tau_{not-own}$, and τ_{vector} came from a special version of TORT which was instrumented to measure the various time values. The total parallel overhead

performance model is the sum of the first four of these contributions:

$$T_{DP1}(ncpu) = T_{task}(ncpu) + T_{lock} + T_{memory}(ncpu) + T_{redundant}(ncpu). \quad (5)$$

Two of the TORT test problems were solved in both parallel and serial modes on a CRAY Y/MP with eight processors. The measured values of the timing parameters are given in Table 1.

Table 1. Timing Parameters for DP1 (seconds)

τ_{task}	τ_{lock}	τ_{memory}	$\tau_{not-own}$	τ_{own}	τ_{vector}
1×10^{-4}	2.8×10^{-6}	2×10^{-5}	1×10^{-5}	7.4×10^{-6}	8.1×10^{-6}

The first TORT test problem solved was the fifth sample job from the TORT verification suite (TP5). Its parameters are given in Table 2. Figure 1 shows a comparison of (5), $T_{DP1}(ncpu)$, and the actual

Table 2. Test Problem 5 Model Parameters

$itnfl$	km	jm	mm	N_{lock}	$mm_{w \neq 0}$
20	18	12	66	3	52

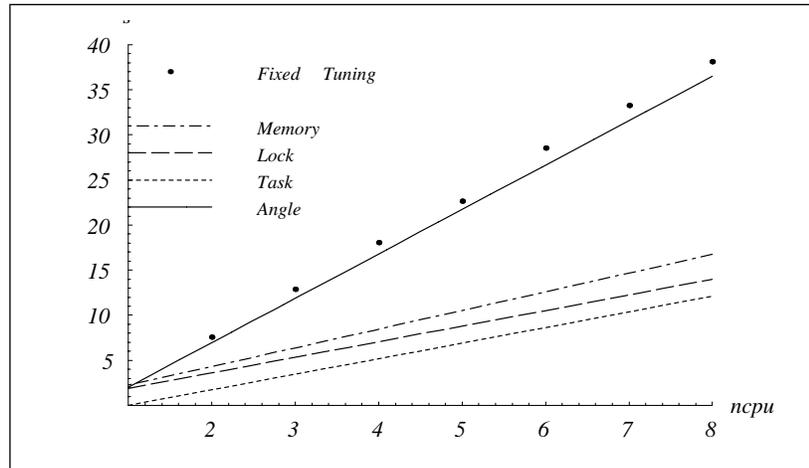


Figure 1. DP1: TP5 CPU Overhead (Sequential Time = 36.257 s)

measured overhead using an modified version of TORT 3.1 (with hold time set to zero). Figures 1 and 2 show the cumulative contribution of each component of the overhead, so the distance between each curve represents that component's contribution to the overhead.

The parallel overhead performance model was also verified for the sixth TORT test problem (TP6). The TP6 model parameters are given in Table 3. The comparison of the model to the measured over-

Table 3. Test Problem 6 Model Parameters

$itnfl$	km	jm	mm	N_{lock}	$mm_{w \neq 0}$
17	33	27	60	3	48

head is shown in Figure 2. It can be seen that the model agrees very well with the measured overhead in both test cases. It is also evident that the task starting and the angle loop redundancy are the chief contributors to the overhead.

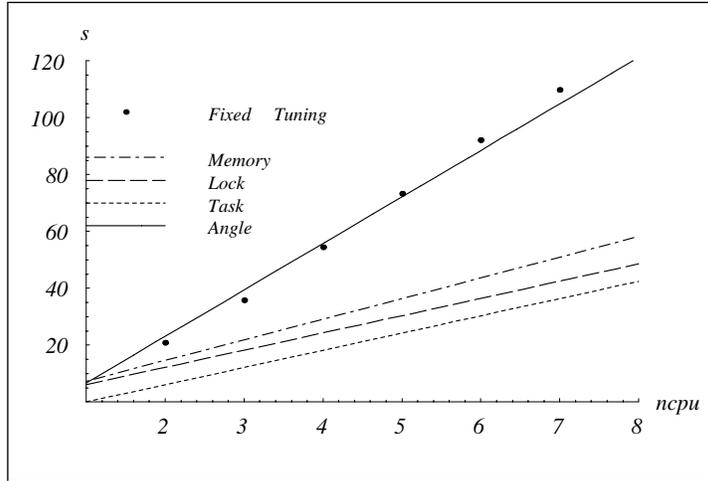


Figure 2. DP1: TP6 CPU Overhead (Sequential Time = 423.143 s)

In addition to verifying the parallel performance model, the performance measurements for DP1 also showed that there was a significant amount of parallelization overhead. For an eight task run with TP6, a 641 second TORT run consisted of over 130 seconds of overhead, an additional 31% over the sequential CPU time. The performance model indicated that the largest sources of overhead came from the task starting and the angle loop redundancy; the task starting because the time to start a task is quite large and the angle redundancy because of the high frequency (namely, once per angle) at which it is incurred.

5 The Improved Direction Parallel Method

For version 3.1 of TORT, the parallel algorithm was modified to address the issues uncovered by the performance model testing of DP1. The second version of the Direction Parallel algorithm (DP2) differs in the following ways from DP1:

1. The starting of the slave tasks is hoisted out of the row loop up to the initialization code; that is, the slave tasks are created only once at the beginning of the run. Rather than starting and stopping them repeatedly, task synchronization is handled through the use of barriers.
2. Rather than locking the summation operation of the flux moments array, each task now has its own private moments array where it accumulates its partial contributions. The master task is responsible for summing the individual contributions into the global array in sequential mode.
3. The loop over angles was reorganized to reduce the amount of redundancy, though some redundancy remains due to TORT's use of zero-weight directions to separate groups of directions with common polar angles, i.e., η -levels.

A new parallel overhead performance model was constructed to describe DP2. It includes the following attributes (not all of which contribute significantly to the overhead):

1. Creation of the slave tasks: The slave tasks are now created only once per run. The CPU time for creating a task is in the millisecond range so this source of overhead is excluded from the model.
2. Hold time: As described above, this source of overhead is not quantifiable so it is excluded from the model. When the measurements are made on TORT timing, hold time is set to zero.
3. Serial accumulation of the task contribution to the flux moments: In the serial version of the code, the row sweeping routine itself adds the angular flux contributions to the global flux moments array. Because each slave task computes its own partial moments array, the master task must take an extra step to sum the partial moments together. This source of overhead is:

$$T_{accum}(ncpu) = \tau_{accum} \times itnfl \times km \times jm \times Q \times ncpu, \quad (6)$$

where τ_{accum} is the time required to perform a single row's accumulation. Unlike the other sources of overhead, the amount of time spent in this operation depends on the length of the row and the size of the scattering source expansion, L . When this parameter is measured, a problem specific value of τ_{accum} is obtained (see Table 4).

4. Memory management: This is unchanged from the DP1 model, see (3).
5. Barrier Assignment: In DP2, synchronization among the participating tasks is accomplished via barriers. There is a total of five barriers, two of which are used only if the left boundary condition is reflective. The barriers are *assigned* only once at the beginning of a run (assignment merely means that memory is allocated to store the data structure which represents the barrier). The overhead for assigning one barrier is on the order of 10 μ s, hence it is excluded from the performance model.
6. Barrier synchronization overhead: The overhead from barrier synchronization occurs when tasks spend time waiting at the barrier for the remaining participants to arrive. As discussed before, this is minimized by setting hold time to zero, but there are still CPU cycles expended in detecting when the barrier wait count is satisfied. A simple test code was written to measure the barrier overhead as a function of the number of participating tasks, $ncpu$. The test code demonstrated a weakly quadratic behavior in $ncpu$, possibly as a result of contention for the barrier memory itself.

Without a left reflective boundary condition, there are four barriers used in TORT and they are synchronized at the same rate as serial accumulation and memory management:

$$T_{barrier}(ncpu) = 4 \times \tau_{barrier}(ncpu) \times itnfl \times km \times jm \times Q \times ncpu \quad (7)$$

where $\tau_{barrier}(ncpu)$ is the time required to synchronize at a single barrier. As mentioned above, this is not a constant value, but depends on the number of participating CPUs.

7. Angle loop redundancy: In this version of TORT, the only angle loop redundancy is a call to the system routine `iselfsch` to get the next angle from the queue and the test to determine if this task should perform a row sweep along this angle. While the actual time to call `iselfsch` is quite small, it occurs at a rate of $ncpu$ times mm . Thus, with many tasks and a large direction set, this may be a significant contributor to the overhead. Measuring the global counter overhead presented two problems:
 - a. It is quite small, to the extent that it was necessary to take into the account the cost of calling the timing routine itself in order to get an accurate result.
 - b. The resulting value for the global counter overhead increased linearly for $ncpu \leq 10$, then behaved randomly for larger $ncpu$. Furthermore, the value obtained would have greatly overpredicted the total overhead. We hypothesize that this unexpected behavior is due to contention for the lock associated with the global counter. By performing the measurement of `iselfsch` within a lock, the measured overhead was largely constant, about 1.4 μ s. Contention is unavoidable in a production environment, so this value must be viewed as a lower bound.

The total self-scheduling overhead is:

$$T_{selfsch}(ncpu, \rho) = 2 \times \tau_{selfsch} \times \rho \times itnfl \times km \times jm \times mm \times ncpu \quad (8)$$

where $\rho = 1$ implies the lower bound on this time penalty in the absence of contention. Contention will generally result in faster increase in this component with increasing $ncpu$ in a generally

unpredictable way. Hence, in subsequent figures, a range for this component is plotted, from $\rho = 1$ to $\rho = 2$.

The total performance model for DP2 is then the sum of these components:

$$T_{DP2}(ncpu, \rho) = T_{memory}(ncpu) + T_{accum}(ncpu) + T_{barrier}(ncpu) + T_{selfsch}(ncpu, \rho). \quad (9)$$

6 Application of the DP2 Parallel Performance Model

The parallel overhead performance model (9) for DP2 was evaluated for several versions of TORT's test problems on a 32 processor CRAY J90. Again, the code was instrumented to measure the timing parameters; the values are given in Table 4.

Table 4. Timing Parameters for DP2 (times in seconds)

τ_{accum}	τ_{memory}	$\tau_{barrier}(ncpu)$	$\tau_{selfsch}$
TP5-320: 1.155×10^{-5}	5.535×10^{-5}	$6.751 \times 10^{-5} ncpu$	1.4×10^{-6}
TP6-720: 4.011×10^{-5}		$+ 1.121 \times 10^{-6} ncpu^2$	
LM1: 7.662×10^{-5}			

The first problem is TP5 modified to have a 320 angle quadrature, $mm = 320$. The comparison of the model to the measured sequential code is shown in Figure 3. In this Figure, the contribution of

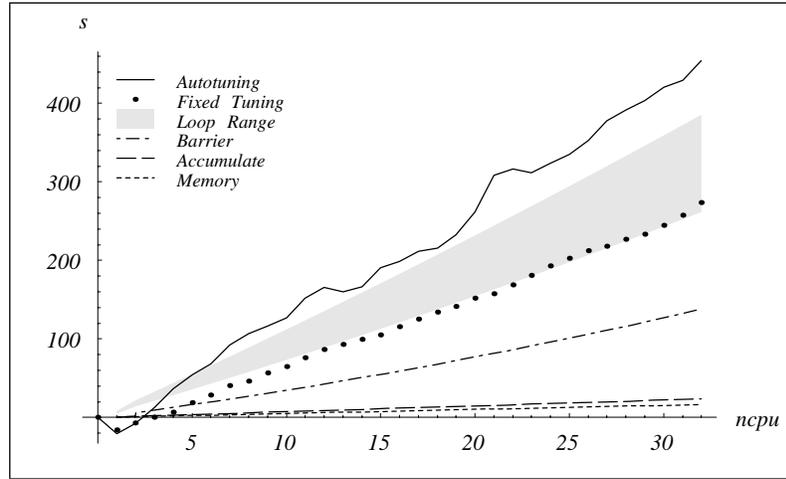


Figure 3. DP2: TP5-320 CPU Overhead (Sequential Time=349.443 s)

each component of the overhead is plotted cumulatively, so the area between the curves denotes each source's contribution to the total. The broad gray band represents (9); it is represented as a range of values due to $T_{selfsch}(ncpu, \rho)$, with ρ varying from 1 to 2. The dots show the measured values of the overhead for hold time equal to zero (fixed tuning). The Figure also shows what happens when the system is allowed to choose the spin wait time dynamically; this is the top line in the Figure - Autotuning. In this case, the overhead penalty increases more rapidly, implying faster deterioration in parallel efficiency, but potentially higher wall-clock speed-up.

A modification of Test Problem 6 was also used to verify the parallel overhead model. In this case, the quadrature was increased to 720 ordinates, $mm = 720$. The comparison of measured to modeled overhead is shown in Figure 4.

A larger problem (LM1, with approximately 2.5 million cells, 320 angles, and 1 energy group) was

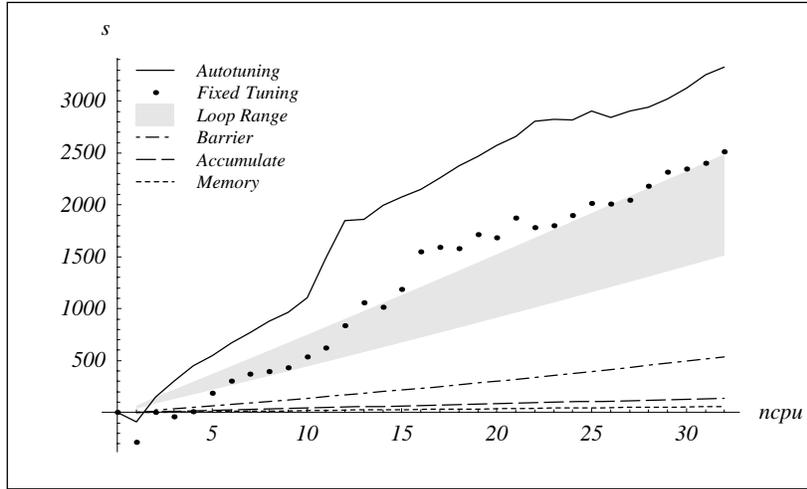


Figure 4. DP2: TP6-720 CPU Overhead (Sequential Time = 8453.5 s)

also used to compare measured parallel performance against the model. The parameters for this problem are given in Table 5 and its measured performance and model components are shown in Figure 5.

Table 5. LM1 Model Parameters

<i>itnfl</i>	<i>km</i>	<i>jm</i>	<i>mm</i>
11	135	144	320

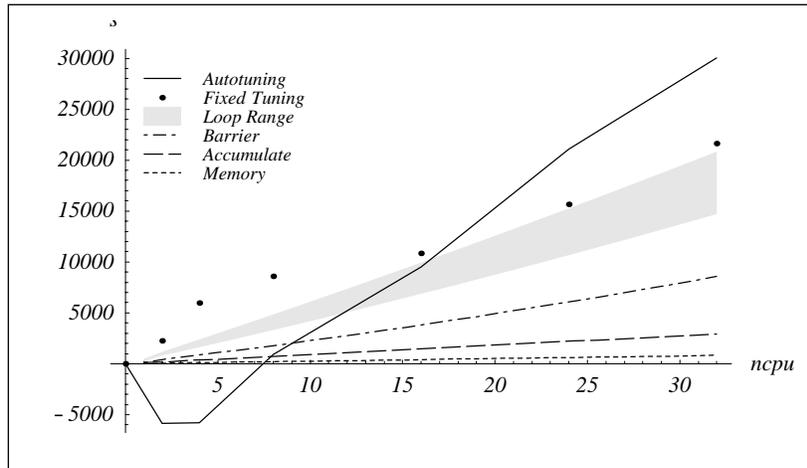


Figure 5. DP2: LM1 CPU Overhead (Sequential Time=70233 s)

The observations to make from Figures 3-5 are:

1. The total overhead which results when autotuning is enabled is usually larger, and less consistent, than the total overhead in the same TORT calculation with fixed tuning. In the former, the wait time, and hence the total overhead, is dynamically determined by the operating system; the programmer has no direct control over it. However, in terms of elapsed time, a problem with autotuning will always run faster than one with fixed tuning, since the system is in the best position to determine the load and adjust the spin wait time appropriately.
2. The next largest source of overhead is the call to `iselfsch` in the loop over angles. Even though the cost of a call to `iselfsch`, $\tau_{selfsch}$, is small, it occurs at a frequency proportional to the number of angles in the problem. Therefore, as the quadrature order is increased, it takes on a more signifi-

cant role.

3. The next largest source of overhead is the use of barriers to synchronize the slave tasks. Compared to the `iselfsch` call, barriers are invoked less frequently, but are individually more expensive. However, barriers are less expensive than the task start and wait combination used in DP1.
4. Overhead from the accumulation operation and the memory management are relatively minor contributors. This is an important component of overhead to characterize since a potential improvement to DP would be to distribute the partial sums of the scattering integral to the slave tasks. However, since the serial accumulation overhead is so small, there would be little advantage compared to the added complexity of implementing this idea.

The value of the improvements to TORT's Direction Parallel method can be seen by comparing the magnitude of the overhead between the two versions. For a version of TP6 which used the 320 angle quadrature, the sequential code on the J90 required 4000 s; with DP1 and eight tasks, the CPU time was 5424 s; with DP2, 4528 s. Thus, the percentage of additional CPU time for the parallel cases dropped from 36% to 13%. For the LM1 problem, the sequential time was 70233 s; for DP1 and eight tasks, CPU time was 98271 s; with DP2, 78841 s. Again, this represents a drop from an additional 40% to only 12% of the sequential CPU time.

7 Conclusions

After the first new parallel algorithm was implemented, a parallel overhead performance model was constructed and validated against the measured performance of the code. The parallel performance model identified the components of the parallel algorithm that are the most significant contributors to the overhead penalty, namely starting new slave tasks in the row sweep and the redundant operations in the loop over angles. With this knowledge, the slave task start up was moved out of the row sweep and replaced with barriers, thus keeping the same tasks available to perform work throughout a given run. Redundancies in the loop over angles were reduced and the locks on the accumulation of the flux moments were replaced with private accumulation arrays in each task. Together, these improvements, as implemented in the second new algorithm, produced a more efficient parallel code, thereby illustrating the main function of parallel performance models.

Two features of the work presented in this paper set it apart from the majority of endeavors in the area of multiprocessing applications reported in the literature. These are the production level of the TORT code and the time-sharing computing environment of the target platforms, which combine to enhance the relevance of our results to the TORT user community worldwide.

8 References

[Azmy, 1996] Azmy, Y. Y., Barnett, D. A., Burre, C. A., "Multitasking the Three-Dimensional Transport Code TORT on CRAY Platforms", Proceedings of the 1996 Topical Meeting on Radiation Protection and Shielding, No. Falmouth, Massachusetts, April 21-25, 1996, Vol. 2, 613, American Nuclear Society, LaGrange Park, IL (1996).

[CRI, 1996] UNICOS System Libraries Reference Manual, SR-2080 9.0, Cray Research, Inc., Mendota Heights, MN (1996).

[Rhoades, 1989] Rhoades, W. A., Flanery, R. E., "3-D Discrete Ordinates Calculations with Parallel-Vector Processors", Proceedings of the 1989 Topical Meeting on Advances in Nuclear Engineering Computation and Radiation Shielding, Sante Fe, New Mexico, April 9-13, 1989, Vol. 2, 69, American Nuclear Society, LaGrange Park, IL (1989).

[Rhoades, 1997] Rhoades, W. A., Simpson, D. B., The TORT Three-Dimensional Discrete Ordinates Neutron/Photon Transport Code (TORT Version 3), ORNL/TM-13221 (October 1997).