CONF-961245--3

Recent Improvements in the Performance of the Multitasked TORT on Time-Shared Cray Computers*

Y. Y. Azmy

Oak Ridge National Laboratory**
Oak Ridge, Tennessee 37831-6363

to be presented at the
OECD/NEA meeting on 3D Deterministic Radiation Transport Computer Programs,
Feature, Applications and Perspectives
Paris, France
December 2-3, 1996

* Research sponsored by the U.S. Department of Energy.
**Managed by Lockheed Martin Energy Research Corp. for the U. S. Department of Energy under Contract DE-AC05-96OR22464.

DISCLAIMER

RECENT IMPROVEMENTS IN THE PERFORMANCE OF THE MULTITASKED TORT ON TIME-SHARED CRAY COMPUTERS*

Y. Y. Azmy
Oak Ridge National Laboratory
P.O. Box 2008, MS 6363
Oak Ridge, TN 37831

Abstract

Coarse-grained angular domain decomposition of the mesh sweep algorithm has been implemented in ORNL's three dimensional transport code TORT for Cray's macrotasking environment on platforms running the UNICOS operating system. A performance model constructed earlier is reviewed and its main result, namely the identification of the sources of parallelization overhead, is used to motivate the present work. The sources of overhead treated here are: redundant operations in the angular loop across participating tasks; repetitive task creation; lock utilization to prevent overwriting the flux moment arrays accumulated by the participating tasks. Substantial reduction in the parallelization overhead is demonstrated via sample runs with fixed tunning, i.e. zero CPU hold time. Up to 50% improvement in the wall clock speedup over the previous implementation with autotunning is observed in some test problems.

Introduction

Since its inception TORT has been geared towards solving large problems that typically required mainframe platforms in the *supercomputer* class of the time.[1] The advent of powerful workstations in the late eighties that rival mainframes in speed and capacity led to porting TORT to these new platforms. Nevertheless, with judicious programming to take full advantage of their advanced capabilities, like vector processing, mainframes still lead workstations in performance albeit with an admittedly diminishing margin. Extremely large production problems, over 3 million cells with S_{16} angular quadrature, place a high demand on all components of a computer system that is typically met only on large machines such as Cray supercomputers. For this reason we continued to support TORT on Cray computers running the UNICOS operating system, in addition to several workstation platforms.

To enhance the performance of TORT even further it was equipped with multitasking capability at the macrotasking level.[2] Measured performance in this early implementation failed to produce large speedup factors in spite of the extremely coarse grain of the parallel runs. Recently the multitasking option[3] has been restructured to eliminate large chunks of parallel overhead thus providing

Managed by Lockheed Martin Energy Research, Inc. under contract DE-AC05-96OR22464 with the U.S. Department of Energy.

speedup factors exceeding 5 in some modestly large problem configurations.[3] The resulting multitasking option included two types of domain decompositions, one across the right and left halves of angular space termed the Octant Parallel method, the other across angular directions within an octant termed the Direction Parallel method. The Octant Parallel method suffers a few drawbacks,[3] and does not offer better performance over the Direction Parallel method with two participating tasks. Hence the Octant Parallel method has been eliminated from TORT; the multitasking option in TORT now refers exclusively to the Direction Parallel method even if not explicitly specified. In spite of the good parallel performance exhibited by the multitasked TORT it was recognized that the increase in parallel overhead with the number of participating tasks is large and attempts to reduce it were initiated.

The first step in this process was to construct and validate a parallel performance model on the Cray Y/MP that provided insight into the dependence of the CPU time on various problem parameters as well as the number of participating tasks.[4] The performance model enabled us to identify bottlenecks that thwart parallel performance. These include: redundant operations in the angular loop across participating tasks; excessive number of tasks created repeatedly; utilizing locks to prevent overwriting the angular and spatial moment arrays accumulated by the participating tasks; CPU hold time. Among these, the first three are under the programmer's control, while the last is almost entirely set by the operating system depending on machine loading during execution. [The user can set the CPU-hold time within the code but without runtime information on machine loading this would amount to a gamble destined to failure].

In the second stage of the process of reducing the parallelization overhead we implemented modifications aimed at circumventing the major contributors. Thus to reduce the redundancy in the angular loop we analyzed its contents to determine dependencies, then using dynamic scheduling techniques suitable for shared memory architectures allowed tasks to skip the loop for angles they do not own. Then we eliminated the repetitive creation of tasks by constructing an infinite loop in which the slave tasks constantly undertake additional work when available. Obviously this led to additional synchronization points that contribute to parallelization overhead and potential CPU idleness; however measured overhead suggests that these are more than compensated for by the reduction in total task creation CPU time overhead. Finally we avoided the use of locks by allowing each participating task to accumulate its contribution to the angular and spatial moments of the flux in private arrays which are subsequently summed across tasks after the mesh has been swept in all angles of an angular quadrant. This requires additional storage space for the private arrays and implies the *serialization* of the accumulation process of the flux moments. The former is typically a small fraction of the total storage requirement in most applications while the latter is not too severe a penalty since the resulting loops performing the summation across tasks are highly vectorizable with relatively long vectors for most applications.

This paper is organized as follows. We start with a brief review of the parallel performance model constructed and validated on Los Alamos National Laboratory's Y/MP rho machine. This is followed by a description of the modifications implemented to reduce the parallelization overhead from each of the three overhead sources described above. A subsequent section includes measured performance results aimed at quantifying the effect of the reduction in parallelization overhead. The last section summarizes this work and our main conclusions.

The Parallel Performance Model

One of the major difficulties in characterizing parallel performance on time-shared computers such as the Cray is its dependence on machine loading during execution. Competition for resources essentially reduces the possibility of overlapping execution of more than one participating task thus

suppressing speedup. Measured parallel performance for TORT on Los Alamos National Laboratory's rho machine, an 8-CPU Cray Y/MP running UNICOS 8.0, accounted for this dependence by repeating tests in a variety of machine loading regimes.[3] These measurements also revealed the strong dependence of the total CPU time consumed by all participating tasks, which can serve as a gauge of parallelization overhead, on machine loading. Since it is practically impossible to quantify machine loading to establish its influence on parallel performance in general, and on parallelization overhead in particular, we aimed only at reducing its influence on the total CPU time. This is accomplished by first observing that the sensitivity of the CPU time to machine loading is largely due to the CPU hold time which by default is controlled by the operating system. When machine load is light the operating system allows a process to hold a CPU until a task is ready to execute on it thereby eliminating wasted time in re-engaging a new CPU; the time the CPU remains on hold is counted towards the CPU cost of the process. In contrast, if the machine load is heavy the operating system assigns any unoccupied CPUs to other processes thus reducing CPU hold time but incurring additional delays in wall clock time. The characteristic behavior resulting from this feature of Cray's multitasking is the opposite trend of the CPU and wall clock times for a given problem using the same number of tasks, i.e. one increases as the other decreases. In order to minimize the dependence of the CPU time on machine loading we set the parameter **holdtime** to zero via a call to the system routine *tsktune* in a special version of TORT. As shown shortly this results in measured CPU time that increases linearly with the number of participating tasks that can be modeled as detailed below.

Since the main objective for the performance model is to quantify parallelization overhead we construct it for the *increase* in CPU time over the sequential run of the same problem. As such the parallelization overhead is comprised of all operations in the multitasked algorithm that are not performed in the original sequential algorithm. We have identified six differences (not counting the CPU hold time set to zero here) and we proceed to develop models for each if possible.

1. Slave task overhead: This is incurred in the creation of slave tasks, via calls to *tskstart*, and when the master task waits for all slave tasks to complete execution via calls to *tskwait*. These occur at the frequency of once per flux iteration, per row of computational cells, per slave task. The total number of tasks, master plus slaves, n is selected by the user at run time via the input variable **ncpu**. Denoting the slave task overhead T_{slave}, its dependence on the number of tasks is represented by,

$$T_{slave}(n) = 4\,\tau_{slave}\,\lambda\,K\,J\,(n-1)\,, \quad n \geq 1, \tag{1.a}$$

where λ is the total number of flux iterations for all energy groups, K, J, are the number of computational cells in the z-, and y-dimensions, respectively, assuming a uniform mesh for simplicity, and τ_{slave} is the CPU time consumed in creating a single task, and waiting for a slave task to finish executing. The factor of 4 in Eq. (1) represents the four quadrants in angular space because the mesh sweep in TORT is performed in the positive and negative μ within the same task. The model parameter τ_{slave} was measured via a simple test code to be of the order,

$$\tau_{slave} = 10^{-4}\ sec. \tag{1.b}$$

2. Lock assign and release overhead: Locks are assigned only once per run and are not explicitly released; they simply vanish upon termination of execution. The CPU time overhead for assigning a lock was measured to be of the order 2×10^{-6} *sec*. Since there is only a total of four locks this contribution is negligible and is ignored in the model.
3. Lock arm and disarm overhead: Locks are deployed to protect shared arrays from overwriting during the process of accumulating angular flux contribution to the angular and spatial moments flux. Locks are armed, and disarmed via calls to routines *lockon*, and *lockoff*,

respectively, at a frequency of once per flux iteration, per row of cells, per angular direction with nonzero weight. Hence this source of parallelization overhead contributes,

$$T_{lock}(n) = N_{lock}\ \tau_{lock}\ \lambda\ K\ J\ M_{\omega\neq 0}\ , \tag{2.a}$$

where $M_{\omega\neq 0}$ is the number of angular directions with non zero weights, and $N_{lock} \leq 4$ is the number of locks utilized which is problem dependent. The CPU time penalty per lock was measured to be,

$$\tau_{lock} = 2.8\times 10^{-6}\ sec. \tag{2.b}$$

4. Lock collisions: Tasks encountering one another at an armed lock effectively execute in sequence and can potentially waste CPU time if held in waiting until the lock is disarmed. This source of overhead is run-time dependent and difficult to predict and model; hence it is not included in the present model.

5. Memory management: This activity includes moving data from the master task memory locations to private memory locations to be used by the slave tasks, and creating pointers to all necessary private arrays for each participating task. This is done once per flux iteration, per row of cells, per quadrant in angular space, per task (including the master). Hence the contribution to parallelization overhead from memory management is modeled by,

$$T_{memory}(n) = 4\ \tau_{memory}\ \lambda\ K\ J\ n\ , \tag{3.a}$$

where the measured value for the model parameter is,

$$\tau_{memory} = 2\times 10^{-5}\ sec. \tag{3.b}$$

6. Redundancy in the angular loop: The loop over angular directions in TORT involves several initialization activities and index incrementing that must be performed correctly. When a task executes these sections of the loop for directions it does not own this amounts to redundant operations that waste CPU time. Modeling this contribution to the CPU penalty is complicated by the fact that the CPU cost of the loop depends on whether or not the task owns the angular direction, which is also different from the loop cost in the sequential run. Denoting the CPU time required to span the angular loop once in each of the above three cases by τ_{own}, τ_{not}, τ_{seq}, respectively, and noting that the angular loop is executed once per discrete ordinate, per flux iteration, per row of cells, per task, the angular loop redundancy's contribution to the parallelization overhead is,

$$T_{redundancy}(n) = \lambda\ K\ J\ M\ [\tau_{own} + (n-1)\tau_{not} - \tau_{seq}]\ ,\ n \geq 1, \tag{4.a}$$

where M is the total number of discrete ordinates. The measured values for the model parameters are,

$$\tau_{own} = 7.4\times 10^{-6}\ sec.\ ,\quad \tau_{not} = 10^{-5}\ sec.\ ,\quad \tau_{seq} = 8.1\times 10^{-6}\ sec.\ . \tag{4.b}$$

Note that in modeling this component we ignored the effect of the number of computational cells in a row, because the loops involved in the angular loop overhead are highly vectorizable.

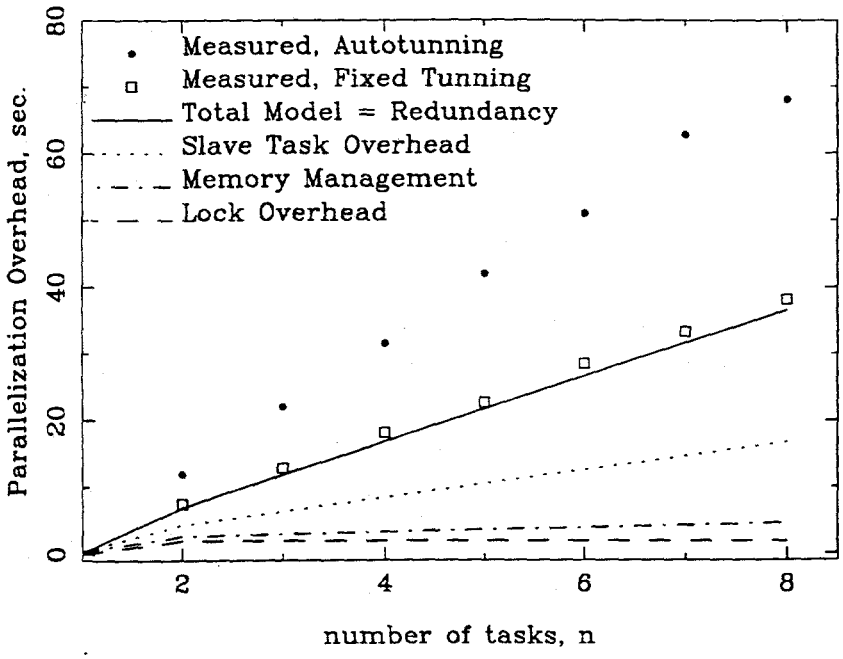
Combining the modeled sources of parallelization overhead, Eqs. (1-4), we obtain,

$$T(n) = T_{slave}(n) + T_{lock}(n) + T_{memory}(n) + T_{redundancy}(n)\ ,\ n \geq 1. \tag{5}$$

In order to validate the performance model, Eq. (5), we solved TORT's test problems 5, and 6, denoted TP5, and TP6, respectively, with autotuning and with fixed tunning using $n = 1, \cdots, 8$. Measuring the components of the parallelization overhead individually would have perturbed the total CPU substantially so we measured only the total CPU time in the numerical experiments

described here. The measured parallelization overhead, i.e. increase in total CPU time from the sequential run case, with autotuning and zero CPU hold time for TP5, and TP6 are depicted *vs* n in Figs. 1, and 2, respectively. The difference between these two sets of points represents the CPU time wasted in waiting at the machine loading level during the runs that produced these measurements. In addition to the measured parallelization overhead, also plotted in Figs. 1 and 2 are the model components, Eqs. (1-4), and the model total, Eq. (5). Comparison of the measured and model overhead total curves demonstrates excellent agreement thereby validating the model; we extrapolate this result to imply validation of the model components also.

Fig. 1. Parallelization Overhead for TP5: Measured Values with Autotuning (bullets), Fixed Tunning (squares), and Model (solid line) Total, as Well as Model Components.

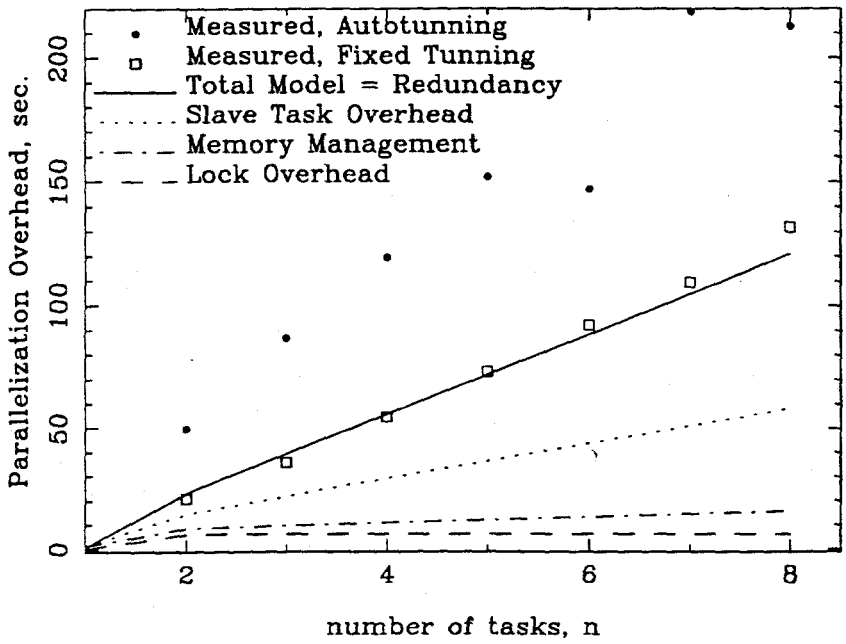


It is evident from Figs. 1 and 2 that the three major contributors to parallelization overhead are the CPU hold time, redundancy in the angular loop, and the slave task overhead. Furthermore, parametric studies with the performance model indicated the significant contribution of locks overhead to the parallelization penalty in case of very large problems with high order angular quadrature. As discussed earlier setting the CPU hold time to zero drastically reduces wall clock speedup, hence no attempt is made to reduce its impact on parallel performance. In the following three sections we describe our approach to improve the other three major sources of parallel inefficiency.

Eliminating Redundancy in Angular Loop

Analysis of the angular loop in subroutine *rowdp* that implements the multitasking algorithm revealed total independence of the individual instances in this loop. Since scheduling of the angular directions to the participating tasks is assigned dynamically, we modified the loop to start with a test on whether the task owns this instance of the loop index. [A task owns a direction index by grabbing it from a global counter and incrementing it by one]. If the task owns the direction index

Fig. 2. Parallelization Overhead for TP6: Measured Values with Autotuning (bullets), Fixed Tunning (squares), and Model (solid line) Total, as Well as Model Components.



it sweeps the row along it, otherwise it skips to the end of the loop. As such the only redundancy left in the loop is essential for its correct execution, and is indeed minimal.

Eliminating Repetitive Task Creation

The large contribution of the slave task overhead to the parallelization penalty is due, at least in part, to the large number of times the slave tasks are created. In particular, slave tasks are created every time the master task sweeps a row of cells in all discrete ordinates in a quadrant in angular space, and are released upon its conclusion. This is a direct consequence of the way multitasking programming is designed, and the simplistic coding practiced so far for the purpose of facilitating the debugging and verification processes prior to optimizing performance. The purpose of this effort is to reduce the contribution to overhead from the slave task creation by moving this activity up the subroutine hierarchy and stalling the slave tasks until more work is made available by subroutine *row*. In this scheme the slave tasks are created only once per run.

Our strategy in accomplishing this is to move the task creation as far up the subroutine hierarchy as necessary to encompass all mesh sweeps within a run, yet not too far up to miss variables-setting as the input data are read. Hence we moved this, as well as several related activities such as lock and barrier assignment, to the bottom of subroutine *input*. At such a point during execution all pointers within the container array **d** have been computed; these are necessary for the slave tasks to find needed data in their correct location, and to use private scratch room for their individual computations. The slave tasks are created to run a new subroutine, *slave*, with two arguments: the container array, **d**, and the pointer to the first position in each task's private copy of the **comrowv** common block, **lcomp**. For the most part *slave* is an infinite loop that waits until the master task

calls the interface subroutine *rowdp* before doing the same. [Recall that the master task loads each task's **comrowv** with correct data for the present row to be swept before it calls *rowdp*. This, together with shared data that the slave tasks are able to locate through pointers to positions in **d**, provides them all data necessary to proceed with a row sweep as before].

Clearly, correct execution of the multitasked code demands provision of correct shared data as described above, and proper synchronization among the participating tasks. The four locks previously installed in *rownvp* are sufficient to prevent the overwriting of shared data by participating tasks if left in place; however they are eliminated as described in the following section. Additional synchronization points are defined via a total of five barriers, three of which are newly installed in the latest version of the code. These introduce additional parallelization overhead and the potential for wasted CPU time by tasks held in waiting at a barrier, but the working assumption here is that this increase will be dominated by the reduction due to the fewer slave tasks created.

Eliminating Locks

This effort was initiated to enhance performance by eliminating the use of locks, and their contribution to parallelization penalty which is substantial in problems employing high order angular quadratures. Clearly correct accumulation of the flux moments requires an alternative mechanism to avoid overwriting and it is understood that the reduction in CPU time will reflect the balance between that mechanism's cost and lock overhead.

There are at least two strategies for accomplishing this goal that were considered. The first strategy, which has been completed and is hereby being reported, is comprised of creating private arrays for each participating task to accumulate its contribution to the flux angular and spatial moments and PCR coefficients. These private arrays are created in subroutine *input*, and since they do not overlap with one another the locks become obsolete and therefore have been completely eliminated. The cost of implementing this modification is embodied in the storage of two new private arrays. The first is of length $2\times$**ima**, **ima** $\equiv$ maximum number of cells in a row, and contains the PCR coefficients for each cell's right and left edges. The second is of length $(4+$**lms**$)\times$**ima**, **lms** $\equiv$ number of angular moments computed, and contains the cell-averaged, x-, y-, and z-moments of the scalar flux, and the angular moments of the flux for each cell. These arrays are reset to zero upon each entry into subroutine *rowdp* and contain the contribution of each task to the respective arrays upon return from it. In subroutine *row* a loop is executed to accumulate these private arrays in their proper locations in memory.

It is evident from the description of the present strategy that the elimination of the locks alters the iteration history and the converged solution only by as much as numerical imprecision, a fact that was verified in all the test runs presented shortly. The main drawback of this strategy is that the accumulation of the private arrays performed in subroutine *row* is sequential. This leads to the second strategy, namely to multitask the accumulation process, which might be considered for future implementation if deemed necessary to improve the parallel efficiency of large applications, namely multitasking this process.

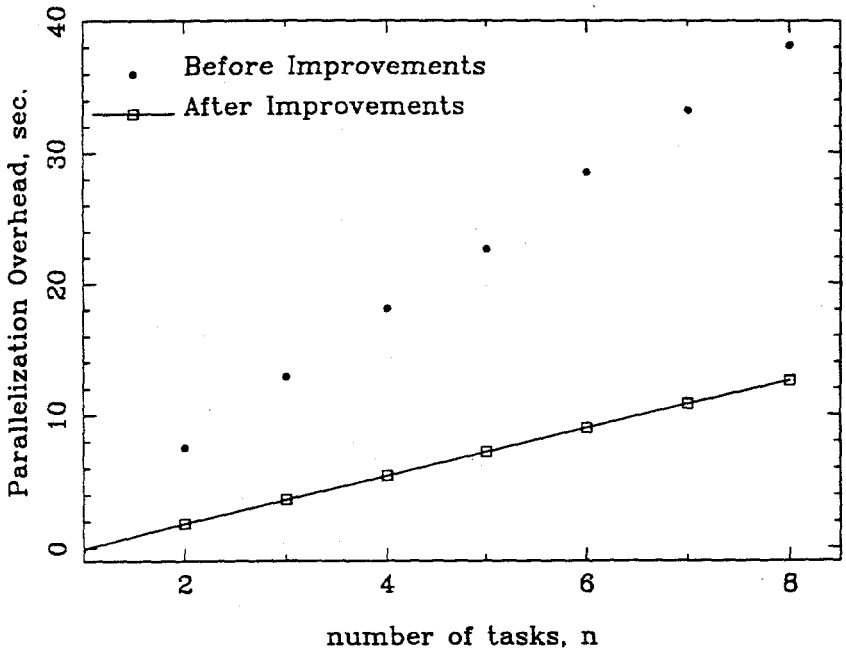
Testing and Measured Performance

Upon implementation of the improvements detailed above in TORT we repeated the numerical experiments with TP5 and TP6 with fixed tunning to determine the effect on parallelization overhead. In all runs we checked the converged solution and iteration history for the multitasked execution against their sequential counterparts and observed perfect agreement to all printed figures.

[This is in contrast to earlier results presented in Ref. 3 due to a bug that allowed the shared variable **pclr** which contains the partial currents on the *x=const* faces to be overwritten asynchronously].

The measured total parallelization overhead for TP5, and TP6 before and after the improvements presented here are shown in Figs. 3, and 4, respectively. These exhibit a reduction in parallelization penalty to 25-35% of its value before these improvements, implying successful achievement of our goal. However, comparing the measured penalty to the only remaining component in the performance model, namely the memory management, reveals large disagreement. We conjecture that this is mainly due to new contributors to overhead, mainly the barriers overhead and the serial accumulation of the flux angular and spatial moments. Plans are underway to update the performance model to account for these changes and identify additional candidates for improved performance.

Fig. 3. Measured Total Parallelization Overhead for TP5 with Fixed Tunning Before (bullets), and After (squares) the Improvements Presented in This Paper.

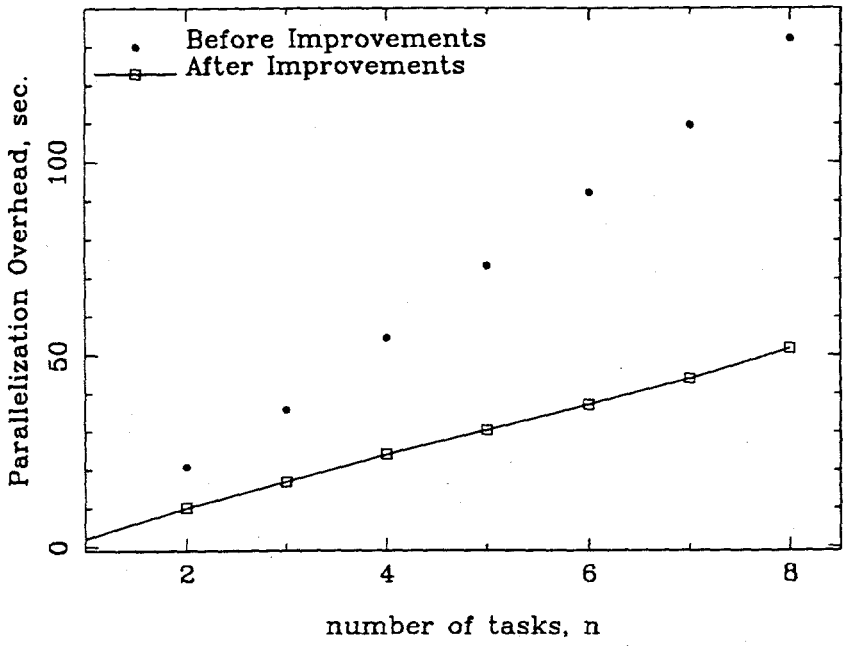


Finally, in order to put the effect of these improvements into perspective we executed TP5 and TP6 with the original, and with S_{16} quadrature sets on LANL's Y/MP with **ncpu** = 1, $\cdots$,8, and we plot the speedup in the measured Wall Clock times in Figs. 5, and 6, respectively. Comparing these with the measured speedup factors reported in Ref. 3 on a typically loaded machine we observe an improvement that is of the order of 50% for the eight-tasks case.

Conclusion

We used a parallel performance model constructed elsewhere to assess the sources of parallelization overhead and develop a strategy to reduce their impact on wall clock speedup. Specifically, we targeted the angular loop redundancy, slave task creation, and locks overhead, as these are the

Fig. 4. Measured Total Parallelization Overhead for TP6 with Fixed Tunning Before (bullets), and After (squares) the Improvements Presented in This Paper.



major contributors to parallel inefficiency under the programmer's control. By practically eliminating these, but introducing other necessary sources of overhead, we obtained a net reduction in overhead penalty to the range 25-35% of its earlier value. Furthermore, speedup factors that are 50% higher than before have been measured for modestly large test problems.

References

1. W. A. Rhoades and D. B. Simpson, "The TORT Three-Dimensional Discrete Ordinates Neutron/Photon Transport Code," *ORNL/TM-13221*, to be published.

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

3. Y. Y. Azmy, D. A. Barnett, and C. A. Burre, "Multitasking the Three-Dimensional Transport Code TORT on Cray Platforms," in *Advances and Applications in Radiation Protection and Shielding*, N. Falmouth, MA, April 21-25, 1996, Vol. 2, p. 613, American Nuclear Society, La Grange Park, Illinois (1996).

4. Y. Y. Azmy, D. A. Barnett, and C. A. Burre, Research to be published.

Fig. 5. Speedup in Measured Wall Clock Time on LANL Cray Y/MP for TP5 and TP6 with Original Angular Quadratures After Implementation of Improvements Described Herein.

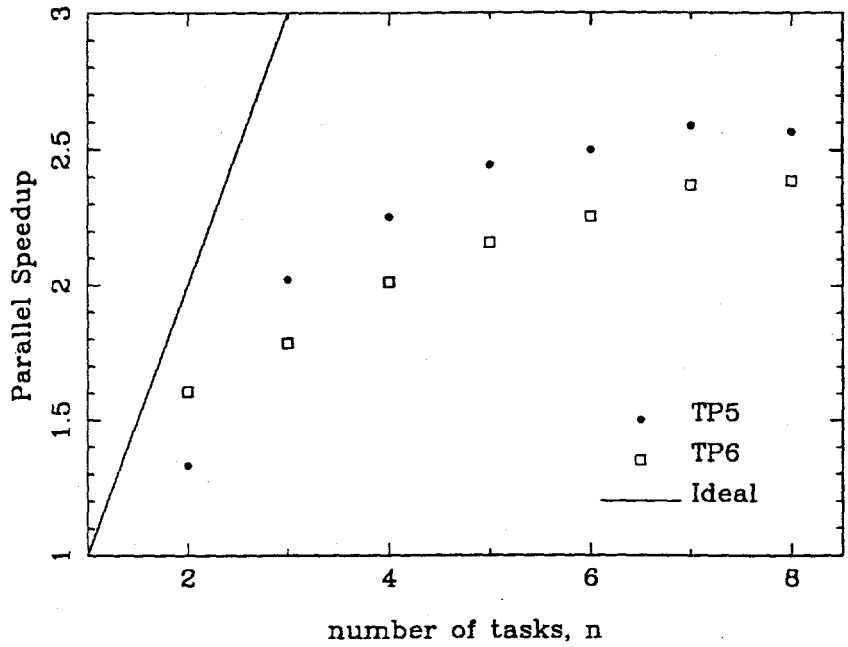


Fig. 6. Speedup in Measured Wall Clock Time on LANL Cray Y/MP for TP5 and TP6 with S_{16} Angular Quadrature After Implementation of Improvements Described Herein.