KAPL-P-000199
(K97141)
CONF-981003--

PARALLEL PERFORMANCE OF TORT ON THE CRAY J90: MODEL AND MEASUREMENT

A. Barnett, et. al.

October1997

NOTICE

KAPL ATOMIC POWER LABORATORY SCHENECTADY, NEW YORK 10701

Operated for the U. S. Department of Energy
by KAPL, Inc. a Lockheed Martin company

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Parallel Performance of TORT on the CRAY J90: Model and Measurement

Allen Barnett
Lockheed Martin Corporation
PO Box 1072
Schenectady, NY 12301
Ph: (518)395-6734
Fax: (518)395-6664

Yousry Y. Azmy
Oak Ridge National Laboratory
PO Box 2008, MS 6363
Oak Ridge, TN 37831
Ph: (423)574-8069
Fax: (423)574-9619
E-mail: yya@ornl.gov

1. Introduction

A limitation on the parallel performance of TORT (Ref. 1) on the CRAY J90 is the amount of extra work introduced by the multitasking algorithm itself. The extra work beyond that of the serial version of the code, called overhead, arises from the synchronization of the parallel tasks and the accumulation of results by the master task. The goal of recent updates to TORT was to reduce the time consumed by these activities (Ref. 2). To help understand which components of the multitasking algorithm contribute significantly to the overhead, a parallel performance model was constructed and compared to measurements of actual timings of the code.

2. Description of the Parallel Performance Model

The overhead in the TORT CRAY Macrotasking-based (Ref. 3) algorithm arises from seven sources. The individual contribution of each of these to the total overhead is described below.

1. Creation of the Slave Tasks: The slave tasks are created only once per run and the time is in the millisecond range (~0.2 msec). Hence, this source of overhead is excluded from the model.

2. CPU Hold Time: In order to improve parallel performance in a time-sharing environment, the CRAY multitasking libraries will cause a task to spin-wait for a certain amount of time (the hold time) rather than releasing the CPU immediately upon reaching a synchronization point. This source of overhead is directly dependent on runtime conditions, such as machine loading, which the user has no way of predicting or even obtaining from the operating system during execution. Hence, this source of overhead is excluded from the model. Furthermore, in order to get the measured performance to conform with the model under this restriction, we set the CPU hold time to zero via a call to the system routine **tsktune**. This causes all tasks to release the CPU upon entering synchronization points.

3. Serial Accumulation of Task Contributions to Shared Arrays: In the present implementation each task accumulates its contribution to shared arrays in its own set of private arrays during the sweep of a single row, then, upon completion of the sweep, the master task sums these private arrays into their shared counterparts. This source of overhead is incurred at the rate of once per participating task ($\nu \equiv ncpu$), per flux iteration ($I_{flux} \equiv itnfl$), per row along the x-dimension ($J \times K \equiv jm \times km$), per quadrant in angle space, e.g. $\eta, \xi > 0$ ($Q \equiv 4$). Hence the contribution to the parallel overhead from this component is given by:

$$T_{accum}(\nu) = \tau_{accum} \times \nu \times I_{flux} \times J \times K \times Q. \quad (1)$$

The contribution of this component of overhead was measured by instrumenting a special version of TORT with a timer around the code section in subroutine **row** which performs this summation and then running TORT with 1 through 32 tasks. The measured overhead time is divided by the number of participating tasks, then the resulting points are fitted to a constant function to produce $\tau_{accum} = 4.011 \times 10^{-5}\text{sec}$.

4. Memory Management: This is the CPU time consumed moving data in memory to locations where participating tasks are programmed to find them. The memory management overhead is incurred at two locations. The first is in subroutine **row** to place current row data at the proper memory locations and to identify the scratch space for each participating task. The second is in subroutine **rowdp** where every task locates the above mentioned information in memory. This penalty to CPU time is incurred at a rate similar to the serial accumulation activity described above:

$$T_{memory}(\nu) = \tau_{memory}(\nu) \times I_{flux} \times J \times K \times Q. \tag{2}$$

The two parts of the memory management component were measured per instance. The measured values were then fitted to models that describe the dependence of each part on ν,

$\tau_{memory}(\nu) = 5.535 \times 10^{-5}\nu$.

5. Barrier Assignment: In the present version of TORT 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. The overhead for assigning one barrier is of the order $10\mu\text{sec}$; hence we exclude the contribution to the performance model from the barrier assignment activity.

6. Barrier Synchronization Overhead: Synchronizing a barrier consumes CPU time while tasks spin at the barrier waiting for the arrival of the remaining participants in a calculation. It was measured in a simple test code as a function of the number of participating tasks and the loop index within which the barriers are synchronized. The four main barriers used in TORT are synchronized at the same rate as for the serial accumulation component, except it is not incurred if $\nu \leq 1$. Hence the contribution to the parallel performance model from the Barrier Synchronization component is given by:

$$T_{barrier}(\nu) \equiv \tau_{barrier}(\nu) \times I_{flux} \times J \times K \times Q \times 4 \tag{3}$$

if $\nu > 1$ and zero otherwise and $\tau_{barrier}(\nu) = 6.751 \times 10^{-5}\nu + 1.121 \times 10^{-6}\nu^2$.

7. Angle Loop Redundancy: This includes a call to the function **iselfsch**, which increments a global counter, and a test of ownership of an angle by the calling task to determine whether to process it or skip to the next discrete ordinate. While the CPU time consumed by the global counter is small, the number of times it is called is proportional to the number of ordinates in the quadrature. Measuring the global counter overhead for the purpose of including it in the parallel performance model presented two complications:

a. It is very small: Measuring the global counter overhead inside of TORT on a per instance basis yielded inaccurate results. A special code was written to measure the overhead of multiple instances of **iselfsch** in a loop; even then, because **iselfsch**'s overhead is smaller than the clock overhead, the latter had to be subtracted from the former to improve its accuracy.

b. The resulting value for the global counter overhead increased linearly with $\nu \leq 10$ or 12, then behaved randomly for larger ν. Furthermore, the value obtained would have greatly overpredicted the total overhead. We hypothesize that this unexpected behavior is due to contention for the lock. Indeed by performing the measurements described above within a lock itself, the measured **iselfsch** overhead was largely constant, $\tau_{selfsc} = 1.4 \times 10^{-6}$ sec. Clearly, contention is unavoidable in a production environment so that the above estimate for the model parameter τ_{selfsc} must be viewed as a lower bound. This source of overhead is incurred at the rate:

$$T_{selfsc}(\nu, \rho) = 2\tau_{selfsc} \times \nu \times \rho \times I_{flux} \times J \times K \times M, \quad (4)$$

where $M = mm$, the number of angles in the quadrature, and $\rho = 1$ implies the lower bound on this time penalty in the absence of contention. Contention will result in a faster increase in this component with increasing ν in a generally unpredictable way. Hence in subsequent figures we plot a range for this time component between its lower bound, $\rho = 1$, to twice that value, i.e. $\rho = 2$.

8. Total Overhead Model: Adding each of the modeled overhead components together yields:

$$T(\nu, \rho) \equiv T_{memory}(\nu) + T_{accum}(\nu) + T_{barrier}(\nu) + T_{selfsc}(\nu, \rho). \quad (5)$$

3. Construction of the Parallel Performance Model

The parallel performance model (Eq. 5) was constructed for a modified version of TORT's Test Problem 6, a "large concrete building". The quadrature was changed from S_6 to S_{16} and both group 1 and group 2 were solved. The basic parameters of Test Problem 6 are:

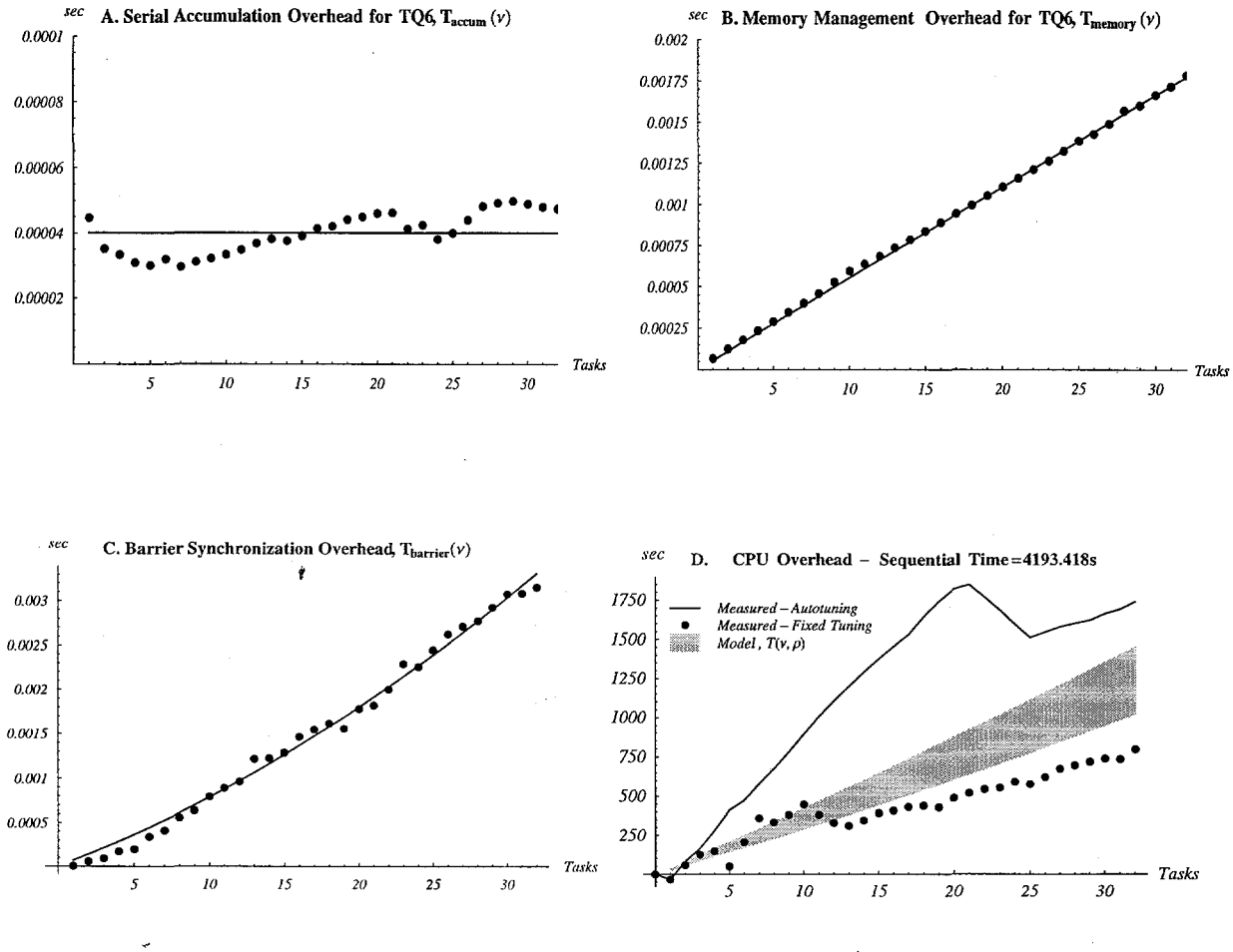
$$I = 117, J = 33, K = 27, I_{flux} = 17, M = 320. \quad (6)$$

Figures 1.A-1.C show the fits to the various performance model parameters, while Figure 1.D shows the total overhead model (with ρ varying from 1 to 2) compared to the actual mesured overhead of TORT for both autotuning and fixed tuning runs.

4. Conclusions

The performance model we developed for the CRAY J90 version of TORT predicts the overhead to within about 20% for this problem. This is well within expectations given the sources of unquantifiable behavior in a time sharing system, including both memory and lock contention and machine load. By setting the CPU hold time to zero, we were able to measure relatively consistent values for each of the sections of code which are specific to the multitasking implementation.

Figure 1. Performance Model Parameter Fits and Comparison to Measured Results



Of the sources of overhead which are within the control of the user, the dynamic scheduling feature in the inner-most loop over angles contributes the most to the overhead. Whether or not switching to a static scheduling algorithm would improve parallel performance remains to be investigated. The scheduling quantum under UNICOS is approximately 1/60 second. If a row sweep requires less time than this, then dynamic scheduling would be desirable, since generally a processor will always be ready for more work. If a row sweep requires more than 1/60 second, then each sweep will be interrupted anyway and the round-robin scheduling of equal priority processes will effectively keep the tasks synchronized.

5. References

1. ORNL/TM-13221, The TORT Three-Dimensional Discrete Ordinates Neutron/Photon Transport Code (TORT Version 3), WA Rhodes, DB Simpson, October 1997

2. "Multitasking the Three-Dimensional Transport Code TORT on CRAY Platforms", YY Azmy, DA Barnett, CA Burre, Proceedings of the 1996 Topical Meeting on Radiation Protection and Shielding, April 21-25, 1996

3. SR-2080, UNICOS System Libraries Reference Manual