Dynamic Data Structures in Shared Memory: OpenMP Implementations

Coromoto León, Gara Miranda, and José M. Rodríguez-Rosa *

University of La Laguna. Dpto. Estadística, I.O. y Computación.
Avda. Astrofísico Fco. Sánchez, s/n, 38271 La Laguna, Spain
{cleon,gmiranda}@ull.es,boriel@gmail.com
http://nereida.deioc.ull.es

Abstract. The objective of this paper is to accomplish a computational study of the dynamic data structures implemented under the parallel shared memory paradigm using the OpenMP tool. The study problem consists in maintaining a set of generic items allocated in memory, each one with an associated numeric value. A value can enter or leave the set at any time and the number of items in memory is neither constant nor predictable. Thus, a dynamic data structure must be used to represent the set. The first approach uses insertion sort algorithms with doubly-linked lists. It is enough to put into manifesto the problems arisen when parallelizations with shared memory models are proposed. Three implementations of this algorithm have been developed: one using semaphores, another using critical regions and a last one which maps dynamic memory allocations onto static ones. A more detailed study involves the analysis of more complex algorithms based on merge, hash tables and binary trees. The obtained computational results show the main lacks of OpenMP related to the dynamic memory management.

Key words: Shared Memory, Dynamic Data Structures, OpenMP

1 Introduction

The objective of this work is to accomplish a computational study of the dynamic data structures implemented under the parallel shared memory paradigm. The aim is to measure the efficiency of the proposed algorithms for handling such data structures. Besides, the suitability for using OpenMP [1] with such *fine-grained* parallelism structures is also evaluated. For such study, the Intel C++ compiler - wich provides support for OpenMP - has been chosen.

In order to carry out the study, the following *problem* was analyzed: during program run-time a set of items is allocated in memory. At any time, a new value will be added to or extracted from the set. Extraction is done according to some predefined criteria and the total amount of items in memory are neither constant

* This work has been supported by the EC (FEDER) and the Spanish Ministry of Education and Science inside the 'Plan Nacional de I+D+i' (TIN2005-08818-C04-04). The work of Gara Miranda has been developed under grant FPU-AP2004-2290.

nor predictable. Thus, there are two kinds of events: insertion and extraction. An *insertion event* happens when an element is inserted into the set, meanwhile an *extraction event* occurs when an element is removed from the set according to some predefined conditions.

For this study, the *smallest value* has been defined as the extraction criterium. This modelizes some computer science problems such as *task queues* in which shorter jobs are executed first. Due to the dynamic nature of the set, the usage of arrays was discarded in favour of linked lists. In order to increase efficiency, the set is maintained *ordered* using a doubly-linked list, so that, removing *the smallest value* is just a single time-constant operation. The insertion of a new value into the set requires to *traverse* the list looking for the right place for the new element to be inserted and so, keeping the structure ordered. The simulations carried out consist in generating a fixed number of *insertion events* of a random numeric sequence. The number of generated events is defined as the *problem size* and is denoted by N. Contrary to the sequential case, in which insertion and manipulation of lists is rather trivial, parallel insertion of elements into a dynamic list might somewhat tricky [2], since execution threads must maintain the integrity and consistency of the dynamic data structures in use. Section 2 presents a brief description of the implemented list traversal parallel algorithms. Some of the computational results obtained are depicted in section 3. Finally, the conclusions and some lines of future work are presented.

2 List Traversal Parallel Algorithms

The first approach to the previously stated experiment was to use a parallel implementation of the *insertion sort* algorithm: in order to insert an element into the list, every thread has to traverse it from the beginning until it finds the right location for the item. Both, traversing and inserting operations, must be thread-safe to ensure data structure integrity. Implementations using this method have a quadratic order of computational complexity. The sequential implementation for the insertion sort method - denoted by SEQ - was used as reference to compare the parallel implementation results. CRITICAL denotes a first approach to parallel list insertion using critical regions which proves to be a rather inefficient method. There has also been implemented a parallel version of SEQ using OpenMP threads named OMP. Every thread uses OpenMP locks to preserve list integrity. STATIC represents the same implementation as OMP, but using its own memory manager to map dynamic memory into static memory structures, e.g., using large static and contiguous arrays as a *memory pool*.

Since direct insertion has quadratic order of complexity, a more time-efficient sorting algorithms were developed. These algorithms use an *extended* doubly-linked list in which some nodes are data whilst others are metadata that mark positions within the list. Such nodes are called *mark-nodes*. Each mark-node within the list marks the position at which following values are equal or higher than the current node. Thus, having information about a mark-node will save the thread to traverse all the list from the beginning when looking for the inser-

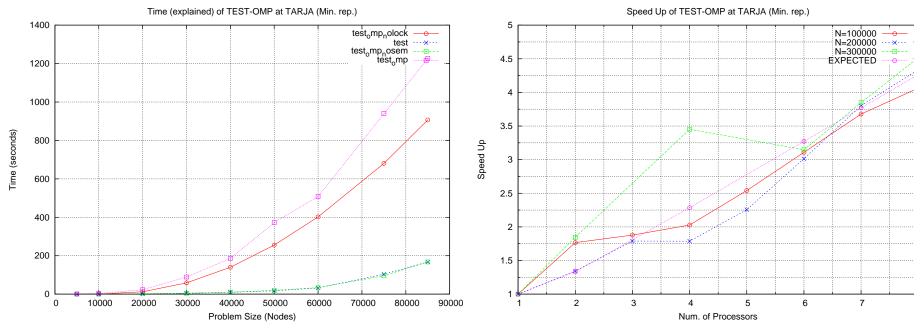

Fig. 1. OMP algorithm. Execution time and speed-up

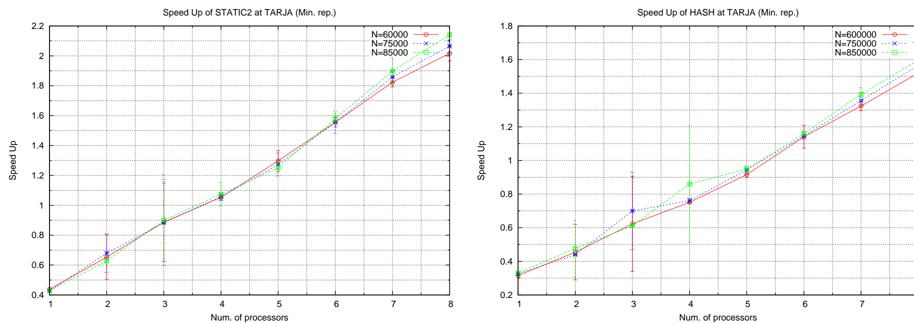

Fig. 2. Speed-up for STATIC and HASH implementations

tion position. In general, these methods resulted some orders of magnitude faster than previous ones. Several experiments using such methods were designed according to the dynamic structure used to locate the mark-nodes. The sequential (SEQHASH) and parallel (HASH) implementation use a hash table of fixed size to map the mark-nodes. Alternative versions which use an unbalanced binary search tree to look for mark-nodes instead of hash tables (SEQTREE and TREE) were also implemented.

3 Computational Results

Experiments have been run over a 64 bits processors Intel Itanium 2 machine. Time measure was done using Posix system calls which returned *wall clock* times. Every experiment was executed five times and average values are considered. Results with an error greater than its standard deviation were discarded.

Figure 1 shows the execution time in seconds for the OMP algorithm with a single thread. This profile decomposes time execution showing that most

time is spent in locks creation and locks usage (`test_omp_nolock` and `test_omp respectively`). Figure 2 shows speed-up with different number of nodes for HASH and STATIC algorithms. OpenMP locks creation and implicit cache-flushes are highly expensive, but indispensable to maintain dynamic data integrity [3]. For higher number of nodes, insertion sort methods tend to overpass their corresponding sequential implementations. For hybrid methods, only HASH shows slight speed-up whilst TREE results seem to behave more erratically.

4 Conclusions

The first conclusion is that fine-grained parallel algorithms with dynamic data structures are not suitable for usage with OpenMP. Locks creations are considerably expensive and locks usage also carries out some performance detriment due to implicit cache flushes [4]. Thus, thread competition seems not to be the bottle neck of these implementations - the largest the list, the less competition among threads.

Dynamic data structures and coarse-grained parallel implementations have a better performance as we can check using another strategy for parallel list sorting based on merge sort. In this case, each thread maintains its own list continuously sorted, thus not having to interact with other threads. At the end of the random sequence generation, all lists are merged in parallel into a main global list. Although this method proved to be very efficient, it did not meet the previous criteria for random extractions: they can happen *at any time* so, extractions - if implemented - would be a little more expensive because we must check each thread list to get the smallest value. However, we found these results interesting enough to be taken into account since they seem to point out some cache issues related to OpenMP locks.

References

1. OpenMP Architecture Review Board, OpenMP Application Program Interface. Version 2.5, `http://www.openmp.org` (2005)
2. Akl, S. G.: Parallel Computation. Models and Methods. Prentice-Hall, (1997)
3. Hoeflinger J.P., Supinski B.R.: The OpenMP Memory Model. Proc. of the First International Workshop on OpenMP - IWOMP'05. (2005)
4. Süß, M., Leopold, C.: A User's Experience with Parallel Sorting and OpenMP, In: Proc. of the Sixth European Workshop on OpenMP - EWOMP'04, pp. 23–38. (2004)