Design and Implementation of Parallelized Linked List Class Library Using Pthread Library

Hong-Soog Kim, Young-Ha Yoon, Dong-Soo Han
School of Engineering
Information and Communications University
P.O. Box 77, Yusong, Taejon 305-600, Korea

Abstract *In this paper, we introduce the PLLCL: Parallelized Linked List Class Library as the pre-treatment approach of parallelism in business computing areas, where the automatic parallelizing compiler is not yet successful. The primitive operations of linked list are parallelized using POSIX thread library for the compatibility across the various platforms. From the implementation, we identified the shortcomings of POSIX thread library and devised thread pool scheme in order to overcome the limitations. Experimental results showed the promising results for the pre-treatment approach of parallelism.*

Keywords: parallelized library, linked list, SMP, Pthread, parallelization

1 Introduction

Although parallel processor systems are getting popular, many hurdles such as the overhead of parallelization, maintaining sustained degree of parallelism and the difficulty of parallel programming are still in unsolved state [1]. Albeit the first two obstacles are inherent and unavoidable, the last one can be mitigated by proper techniques. In order to exploit the benefits of parallel processor system, it is prerequisite to parallelize application programs with appropriate parallel programming paradigm. In case of parallelizing the existing sequential program, automatic parallelizing compiler can be used [2, 3].

With the view of the parallelism, the automatic parallelizing compilers take the form of "post-treatment" of parallelism: programmer codes application programs with the sequential programming paradigm, then the automatic parallelizing compilers extract the potential parallelism from the source codes and generate parallelized programs using the specific parallel programming paradigm. Automatic parallelizing compiler techniques, however, have mainly focused on the scientific computations that have regular computation patterns [4].

In the business application domains, current automatic parallelizing compiler is not so much useful as in the scientific computation areas. The reason originates from the differences between the two application domains.

In scientific computations, FORTRAN has been used as major language and DO loop is used as main control construct and array as main data structure [5, 6]. In business application domains, however, the major languages support pointer operations that make complicated the analysis of program for parallelization of the existing sequential program [7]. Moreover, in these languages, WHILE loop is often used as control construct with dynamic data structure such as linked list, tree and graph.

To cope with the different situations and the limitations of automatic compiler techniques in business domains, we propose the "pre-treatment" of parallelism: the parallelized class libraries, which prepare the parallelized methods for common operations, if possible. The idea is as follows: First, the well designed class libraries are prepared by parallel programming experts, then the normal programmers use the methods which provide the parallelized operations but hide their complicated parallel mechanism. The normal programmers have only to use the methods provided by parallelized class library.

The paper is organized in five sections. Our parallelized linked list library is introduced in section 2. Here, we explain the related data structure, partition management and implementation of parallelized search. In section 3, performance evaluation results are presented with the experiment environment. In section 4, we summarize related work. Finally, we review our approach and discuss the future work in section 5.

2 Parallelized Linked List Class Library

For the evaluation of the validity of pre-treatment approach, we selected the linked list because it is common data structure for business applications such as table lookup, web search and search engine. We have designed and implemented the parallelized linked list class library (PLLCL) on the shared memory parallel processor systems. In implementing parallel operations, the POSIX thread is used.

To raise the reusability and extendibility of the library, the basic operations, such as insertion, deletion and search, of linked list are implemented as methods of class. Hence, other more complicated operations can be built by combining the functionality of these operations.

The parallelized operations that require the thread manipulations are hidden as private methods while the public methods, which use parallelized private methods internally, are exposed. Therefore, normal programmers have only to use the public methods to exploit the benefits of the hidden parallelized operations.

2.1 Primitive Operations of Linked List

There are three primitive operations in linked list: the insertion, deletion and search of the node with certain conditions. In case of implementing the linked list using array, insertion, deletion and conditional search operations on linked list with n nodes require $O(n)$ time complexity. When implementing the linked list at the cost of extra memory for pointer, the pure insertion and deletion operations with n nodes can be performed in $O(1)$ time complexity. However, the search operation with conditions still requires $O(n)$ time complexity.

Since the insertion and deletion operation also require search operation to locate the insertion point or the node to be deleted, search operations are critical. In order to speed up the search operation, we divide a linked list into p partitions, allocate one thread to each partition and get each thread search the their own partition. Partitions are also repartitioned as insertion and deletion operations make the partitions unbalanced.

2.2 Data Structure for Parallelized Search

For efficient parallel search operations, The PLLCL tries to keep the even partition size. PLLCL has a private variable *concurrency_level* that keeps the number of threads used for parallelized searches. The number of partition p is determined by concurrency level. The position array and partition pointer array are added to facilitate the search operation. Figure 1 depicts the eight partitions

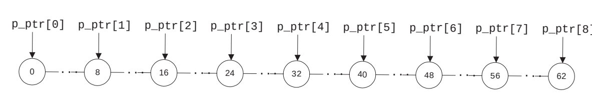

Figure 1: Data structures for partition balancing

of 63 nodes with position array and pointer array when the *concurrency_level* is eight.

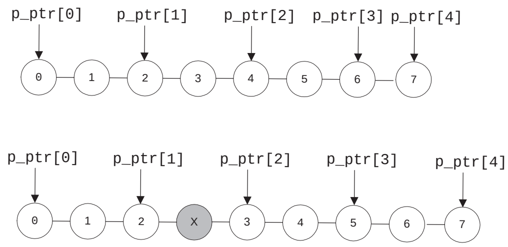

Figure 2: Node insertion

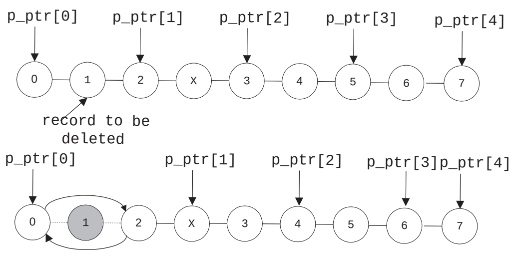

Figure 3: Node deletion

When the number of partitions is p, the position array $position[]$ is an array of $p+1$ integers, which hold the position number of beginning node of each partition. The partition pointer array $p\_ptr[]$ is an array with $p+1$ pointers, which point the beginning node and ending node of each partition. In this scheme, partition P_i is designated by $p\_ptr[i]$ and $p\_ptr[i+1]$. The position number of the beginning node in the partition P_i is stored in $position[i]$. Position array is used to determine the size of partition as well as choosing an appropriate partition to locate the node for insertion and deletion without searching the whole nodes.

2.3 Partition Management

As the insertion and deletion operations are performed, the balanced partitions would become jagged. In order to keep the balanced partitions, the linked list is repartitioned.

For every insertion operation, the partition pointers $p\_ptr[i]$ moves one backward when the node is inserted in partition P_j ,where $i > j$. This makes the all partitions even except for the last partition. Figure 2 depicts the insertion operation and related activities for maintaining balanced partition size.

When the last partition becomes large compared to other even partition, the speed of parallel search may depend on the search operation on the last partition. Therefore, repartition is needed. Current implementation of PLLCL adopts seven repartitioning polices. They are categorized into micro-repartitioning and macro-repartitioning policies.

The micro-repartition policies determine the time of repartitioning based on the $concurrency\_level$ (i.e. the number of partitions). When the number of nodes in the last partition exceeds the $concurrency\_level$, repartitioning is performed. These policies have an advantage of keeping the partitions balanced always but incur frequent repartitionings. The micro-repartitioning polices are subdivided into three polices: micro-immediate, micro-moderate and micro-delayed repartitioning policy. Each of micro-repartitioning polices performs the repartitioning when the exceeding number of nodes in the last partition becomes one, half of the $concurrency\_level$ and $concurrency\_level$ respectively.

The macro-repartitioning policies repartition the linked list based on the current partition size. Current partition size is the number of nodes in each partition except for last one. There are four macro-repartitioning policies: macro-quarter, macro-half, macro-3quarter and macro-delayed repartitioning policy. Each of these macro-repartitioning policies repartitions the linked list when the exceeding number of nodes in the last partition becomes quarter, half, three quarter and whole of current partition size respectively. These policies minimize the number of repartitioning but the last partition is more likely to unbalanced.

In case of node deletion, PLLCL moves the partition pointers $p\_ptr[i]$ one forward if the deleted node is in partition P_j ,where $i > j$. Activities related to deletion operations are depicted in Figure 3. The repartition policies for deletion operation are similar to those of insertion operations.

2.4 Parallelized Search and Partition Management

Parallel search is implemented by threaded search operations. Each thread T_i ($1 \leq i \leq concurrency\_level$) searches the nodes in partition P_i with the given key. A thread, which firstly finds the node with the given key, halts the executions of other search threads and returns the pointer of the node. For the compatibility of PLLCL across the various platforms, Pthread library is used for the implementation. As the Pthread standard supports C language interface only [8, 9], the creation of search threads of PLLCL is implemented as follows: First, we define the dummy C linkage interface function that is declared as friend in C++ class and then call the threaded search method in dummy C linkage interface function. Finally, the dummy C linkage interface function is passed to the *pthread_create*() as third argument [10, 11, 12].

In management of search threads, our initial implementation was to create threads whenever a search request occurs and get them away when search request is completed. The check on search completion was implemented with *pthread_join*() function, which does not support anonymous join. Consequently, parallelized search operation had to wait until all threaded searches were completed although only the result of firstly successful threaded search was meaningful in most cases. Anonymous join is required in this case, when the result of firstly completed thread is meaningful, such as parallel search in PLLCL. Hence, the initial thread management scheme is revealed to provoke the considerable overheads that diminish the speed-up gains from the parallelized search.

To avoid this overhead, we reimplemented the thread management as thread pool in which threads were prepared in the PLLCL initialization, activated and deactivated in accordance to the search request initiation and search request completion. This implementation reduced the overhead related to thread creation and destruction. The Pthread standard, however, does not support anonymous join, which is indispensable for situations such as parallel search method of PLLCL. This shortcoming was resolved by using primitive semaphores and some flag variables. By using semaphores and some flag variables, threads in thread pool are activated by the search request and deactivated by the search completion notification of other thread or failure of the search with non-existing key.

3 Performance Evaluations

To evaluate the current implementation of PLLCL, we measured the performance overhead of partition management and execution time of parallelized search in PLLCL over sequential search. The experiment environment is as follows.

> *H/W*: 4-way Pentium III Xeon 500MHz with 512K L2 cache, 1GB main memory
> *OS*: Solaris 8 Beta Refresh Intel Platform
> *Compiler*: GNU C++ compiler 2.8.1

Figure 4 presents the relationship between number of repartitions and number of appended node. This result confirms the expectation that the macro-repartitioning policy is inferior to micro-repartitioning policy with respect to number of repartitioning.

Figure 5 shows the relationship between cumulative time of append operations and number of appended nodes. This result reveals that the append time is independent on the partition management policies. Moreover, the time of append with the partition management is approximately equal to or slightly more than that of serial append without the partition management. From the Figure 4 and Figure 5, our partition management does not invoke overheads in append (or insert) operations. Therefore, we can conclude that the micro-repartitioning policies are superior to macro-repartitioning policies in maintaining balanced partitions for parallelized search.

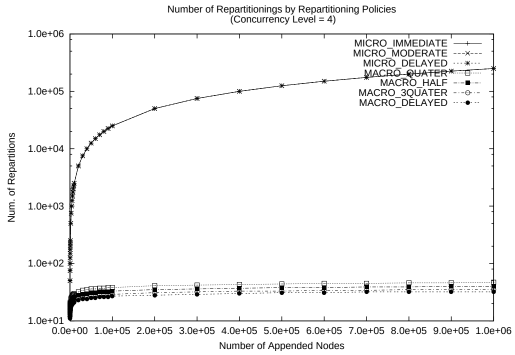

Figure 4: Number of repartitionings

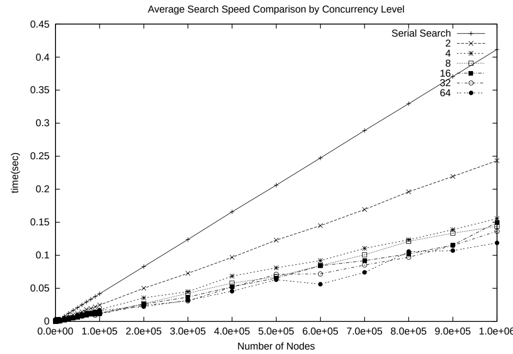

Figure 6: Average search time

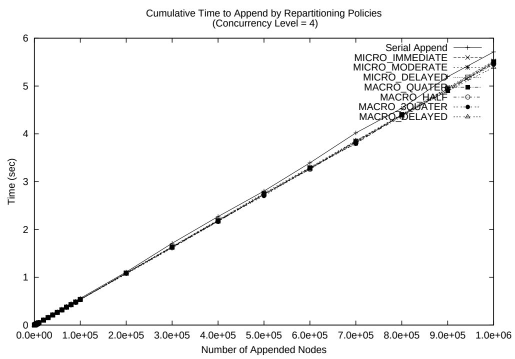

Figure 5: Cumulative time to append

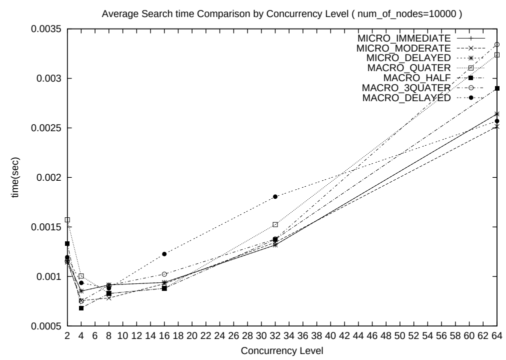

Figure 7: Average search time vs. Repartitioning policy

In Figure 6, the average search time of parallelized version is compared to that of serial version. The average time is calculated over one hundred experiments. Here, micro-moderate repartitioning policy is used and the *concurrency_level* (i.e. number of threads in thread pool) is varied. Across all concurrency levels, parallelized search has less execution time than serial search. Figure 7 presents the average search time varying the repartitioning policies. This shows every repartitioning policy has approximately minimum search time when *concurrency_level* is four.

4 Related Works

The approaches to utilize the parallelism can be categorized into post-treatment and pre-treatment by the extraction time of parallelism. The latter includes the parallelized library, and language extension to support the parallelism while the former indicates the automatic parallelizing compilers. With this classification, the related works can be summarized as follows.

Related to automatic parallelizing compilers, there have been many research projects that mainly deal with the FORTRAN languages. Examples of these projects are PIPS [13], ParaScope [14], Polaris [15] and SUIF [16].

Related to the expressing parallelism and multi-threading construct at user level, there have been two approaches. The first one is defining user-level multi-thread packages, like POSIX threads [17], Solaris thread [9] and DECthread [18]. The other one is extending sequential languages, such as C, with multi-threading functionality or defining new language with multi-threading functionality such as Java [19].

Related to the parallelized library approach, Elmasari et al. implemented a threaded communication library (TCL) [20]. The TCL is a user-level thread package that provides primitives to support programs under a multi-threaded program execution model. While the POSIX threads is solely for multi-threading, The TCL tightly coupled multi-threading and communication/synchronization.

Compared to TCL, PLLCL is distinguished from TCL in that the PLLCL is based on the POSIX thread for compatibility across the different platforms and mainly focused on the balanced load distribution among the threads for speed-up. In the PLLCL, prepared thread pool and partition management scheme are used to minimize the thread creation overhead and to speed-up the critical operation.

5 Concluding Remarks and Future Work

In this paper, we suggested one approach to deal with parallelism for multiprocessor systems. Our approach is pre-treatment of parallelism compared to the automatic parallelizing compiler techniques, which extract parallelism from existing programs.

Experimental evaluation ascertained the validity of our approach. We think our approach is complement to automatic parallelizing technique rather than orthogonal. Through the implementation of PLLCL, we found that the thread management facilities supported by current Pthread standard are insufficient. They are lack of anonymous join, thread enabling/disabling and thread pool for efficient implementation.

Currently we are trying to extend our experiences in PLLCL to tree data structure which is another important data structure used in business applications.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo CA, 1995.

[2] Michael Wolfe. Parallelizing compilers. *ACM Computing Surveys*, 28(1), March 1996.

[3] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company Inc., 1996.

[4] Christine Eisenbeis and J. C. Sogno. A general algorithm for data dependence analysis. Technical Report RR-1699, Recherche de l'INRIA, May 1992.

[5] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computer.* ACM Press, New York, 1996.

[6] Michael Wolfe. *Optimizing Supercompilers for Super-computers.* The MIT Press, Cambridge, MA, 1989.

[7] Dong-Soo Han and Takao Tsuda. Program analysis of optimizing compilers for record handling programs. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1248–1259, May 1996.

[8] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstation and Parallel Computers.* Prentice Hall, 1999.

[9] Sun Microsystems. *Multithreaded Programming Guide.* Sun Microsystems Inc., 1997.

[10] Scott Meyers. *Effective C++.* Addison Wesley, 1992.

[11] Magaret A. Ellis and Bjarne Strousrup. *The annotated C++ Reference Manual.* Addison Wesley, 1990.

[12] David R. Butenhof. *Programming with POSIX thread.* Addison-Welsley Longman Inc., 1997.

[13] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 212–219, July 1991.

[14] Mary W. Hall et al. Experiences using the parascope editor: an interactive parallel programming tool. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 33–43, 1993.

[15] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 1995.

[16] Byoungro So, Sungdo Moon, and Mary W. Hall. Measuring the effectiveness of automatic parallelization in suif. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 212–219, 1998.

[17] IEEE. *Portable Operating System Interface (POSIX) - Part 1 : System application: Programming Interface (API).* IEEE Standard Press, 1996.

[18] Digital Equipment Corporation. *Guide to DECthreads.* Digital Equipment Corporation, part number AA-2QDPC-TK, 1996.

[19] Stephen J. Hartley. *Concurrent Programming : The Java Programming Language.* Oxford University Press, February 1998.

[20] Nasser Elmasri, Herbert H. J. Hum, and Guang R. Gao. The threaded communication library: preliminary experiences on a multiprocessor with dual-processor nodes. In *Proceedings of the 9th ACM international conference on Supercomputing*, pages 195–199, 1995.