An efficient parallel set container for multicore architectures

Álvaro DE VEGA [a], Diego ANDRADE [a,1] and Basilio B. FRAGUELA [a]
[a] *University of A Coruña*

Abstract.
Multicores are now the norm and new generations of software must take advantage of the presence of several cores in a given architecture. Parallel programming requires specific skills beyond from those required for the development of traditional sequential programs. The usage of parallel libraries is one of the best ways to facilitate parallel programming, as they does not require new compilers and they allow to parallelize sequential codes without big efforts by the programmer. This paper presents an efficient and portable parallel set container. This container has been used to program parallel versions of several algorithms. The experimental results show that the container facilitates the programmability and achieves a good performance on multicore systems.

Keywords. Multicore architectures, parallel library, data containers, data parallelism

1. Motivation

Parallel libraries are a good method to facilitate the expression of parallelism to programmers. Libraries have several advantages with respect to the usage of parallel languages [4] or compiler directives [3], as they provide a more portable solution that shortens the development time and do not require new compilers or the learning of new languages.

Data container types are widely used in modern programming, as they are a normalized method to store the data managed by a program. Object-oriented languages are the natural implementation vehicle for these containers, since we are defining new data types. Besides they facilitate the construction of containers of complex data types (e.g. objects with runtime polymorphism). Features such as polymorphism and operator overloading are very useful in this context, as they provide a more convenient notation for the representation of operations.

Data parallelism is preferably exploited in the parallelization of data containers (instead of task parallelism), as the methods that manage these containers may be parallelized by dividing the container into different parts, on which these operations are applied concurrently. This paper presents a parallel data set container type for C++ which uses efficiently the resources available in multicore architectures. The data type supports the API of the Standard Template Library (STL) [12] and additional methods which can

[1] Corresponding Author: Office 0.03, Facultade de Informática, Campus de Elviña s/n, A Coruña, Spain E-mail: diego.andrade@udc.es.

be used to explicitly or implicitly express parallelism. The underlying parallelization of the data type and its operations is done using the Intel Threading Building Blocks (TBB) library [9].

The rest of this document is organized as follows. Section 2 outlines the main operations implemented by our library. Section 3 describes the test programs implemented using the library and the performance numbers obtained. Section 4 discusses the related work and Section 5 summarizes the conclusions.

2. The concurrent_set data type

The `concurrent_set` data type supports the API of the `std::set` type of the STL library. This facilitates code reuse as well as the gradual adaptation of existing codes, which can first replace the standard STL sets by `concurrent_set` with no effort and later use `concurrent_set` additional methods when and wherever desired. The programmer sees a single flow of execution, being the operations on concurrent_sets implicitly parallel. The container is polymorphic, as sets containing elements of any data type may be defined. In addition, it offers methods for all the operations that are usually applied on sets. The meaning of some of these operations is defined as described in the standard set theory: Is_a_member_of, Is_subset_of, Union, Intersection, Complement and Symmetric difference. Other operations provided that are not specified in the standard set theory are:

Selection When it is a applied on a set A of elements of type T, given a predicated $Pred : T \rightarrow \{true, false\}$ it generates a subset $B = \{a \in A / Pred(a) = true\}$

Application Given a set A of elements of type T, and a function $f : T \rightarrow T$, this operation applies this function on each member of A.

Reduction Given a set A of elements of type T and a reduction function $r : (T, T) \rightarrow T$, the reduction function is applied on all the members of A until it returns a single element.

Map Given a set A of elements of type T and a function $f : T \rightarrow S$, this operation applies this functions on each member of A and it returns a set $B = \{f(a)/a \in A\}$

Relationship Given two sets A and B, and defined the cartesian product of these data sets $A \times B$ as the set $\{(a, b)/a \in A, b \in B\}$, this operation builds a set that contains all the pairs of $A \times B$ that fulfill a predicate p that defines the relationship.

The library also provides more complex operators such as an efficient parallel MapReduce which are not detailed here due to space reasons.

The library is publicly available in https://forxa.mancomun.org/projects/ctl/

Figure 1 shows an easy example of usage of the `concurrent_set` data type which illustrates its simplicity. Line 1 shows the only header that must be included to enable the use of the library. The instruction in line 5 creates a `concurrent_set` of elements of type `char`. Let us notice that it uses one of the standard constructors of the `stl::set` data type and that `concurrent_set` is defined in the `ctl` namespace. This set is filled with the `char`'s of the string initialized in line 4. Line 6 executes a `map` operation on the set using the implicit parallelism provided by the library. This `map` operation applies the function declared on line 2 on all the elements of the set. This function substitutes each char by the next char in the alphabet.

```
1  #include "concurrent_set.h"
2  inline char sum1(const char c) {return c+1;}
3  void main() {
4          char str[]= "abcd";
5          ctl::concurrent_set<char> set(str, &str[sizeof(str)-1]);
6          set.map(sum1, set);
7  }
```

Figure 1. Example of map operation implemented using the `concurrent_set` data type

2.1. Implementation details

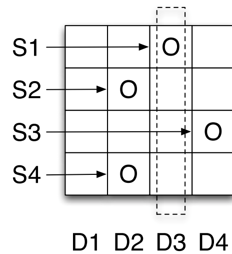

Figure 2.: Grid layout used in the map operations. Rows S_1 to S_4 represent the source subsets and columns D_1 to D_4 represent the destination subsets.

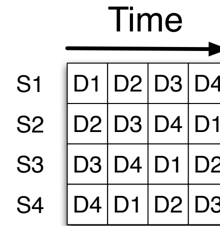

Figure 3.: Order in which each task, which processes one subset of A, starts the processing of the subsets of B. Rows S_1 to S_4 represent the subsets of A and D_1 to D_4 represent the subsets of B.

The `concurrent_set` data type is defined as a C++ template which allows to define sets containing elements of any type. The parallelization of all the operations is achieved by dividing internally the set in a number of subsets. The structure is organized as a `std::vector` of `std:set`'s. The concurrency is implemented using the Intel Threading Building Blocks library which allows to define dynamically the tasks that process concurrently each subset and provides automatic load balancing among the cores available. We developed an alternate implementation of the library using OpenMP but it was up to twice slower for some operations. The number of subsets defaults to the amount of hardware threads available in the system, but it can be selected by the user for each `concurrent_set`. The mapping of objets to subsets is done according to a hash function which tries to balance the number of elements among the subsets. Again, while a default one is provided, a user-defined one can be supplied.

The implementation of the operations with reduced parallelism, like for example a insertion of one isolate element in the set, is done simply by applying the sequential version of the operation on the corresponding subset. This is the case of several operations already present in the API of the `std::set` type of the STL library. The remaining operations are implemented using efficient strategies which take advantage of the presence of several cores. The parallelism is limited in these operations mainly by two factors. (1) The impossibility of different tasks to operate concurrently on the same inner set or subset. (2) The mapping of the `concurrent_set` elements to subsets is determined by a hash function of the contents of each element. Therefore, if the operation modifies one element, it may change its location. Thus, the operations performed by each task

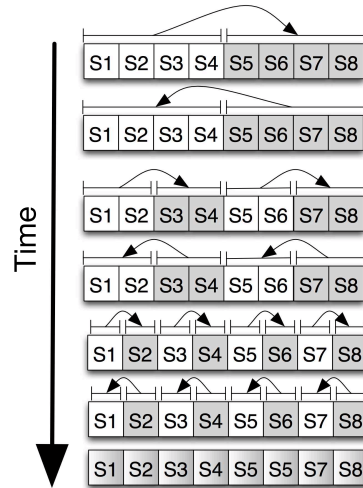

Figure 4. Scheme to process a all-to-all operation when the two operand set are the same set A. S_1 to S_8 represent the subsets of A.

may potentially affect any subset, which limits largely the performance as factor (1) established. These operations are classified in four categories for which a separate strategy has been designed to perform the operation efficiently. They are now explained in turn.

- *Map operations.* This category includes the operations where the same function is applied on all the elements of the set. `Application` and `Map` are examples of this category. If the function applied on each element of the set does not modify its contents, there is not risk that the element has to be relocated in a different subset after the function is applied. In that case, it is safe to apply the operation concurrently on the elements of different subsets. If the function may modify the contents of the element and conversely the subset where it is located, it is not safe to apply the function concurrently, as the same subset could be affected by different tasks. The solution is to create an auxiliary grid layout, like that one represented in Figure 2, which has one row per source subset (those where the elements are before they are modified) and one column per destination subset (where the elements are placed once modified). Each task applies the function on the elements of its associated source subset and stores them in the appropriate columns of its associated row. As the elements already exist and they are simply modified, the grid stores only the pointers to the original elements (once modified). Finally, the elements (the pointers) are moved from this grid to the corresponding subsets of a new set. In this movement, the processing of each column of this grid can be performed concurrently by a different task, as it is guaranteed that the elements of different columns are inserted in different subsets of the destination structure.
- *Comparison operations.* Given two sets A and B, this kind of operations searches for occurrences of each element in A in set B and viceversa. Symmetric difference and intersection are examples of this kind of operations. The search of the elements of set A in set B has a complexity of $O(N \cdot log(M))$, N and M being the cardinalities of A and B respectively. This complexity can be reduced because the elements in a set are always ordered according to their hash function, thus, the search of the first element of A starts

at the first element of B and continues until the same element, or an element with a hash function greater than the searched element, is found. The search of the second element of A starts where the search of the first element finished. The process continues until the end of set B is reached, or all the elements of A have been searched. This strategy allows to reduce the complexity of this search to $O(log(N + M))$. If both sets have the same number of subsets and they use the same policy to distribute their elements among them (same hash function and comparator), the search of the elements on each subset of A is limited to the corresponding subset of B. When this is not the case, a copy of set B is created with the same number of subsets and using the same policy to distribute its elements as in set A.

- *All-to-all operations.* In this kind of operations, every element of set A is combined with each one of the elements of set B. `Relationship` is an example. The processing of the elements of each subset of A can be performed concurrently if the elements of B are not modified. If they are modified, then it is necessary to lock the access to each subset of B. If each task accesses the subsets of B in the same order, there is a big contention and the locks and waits are systematic. The interference among tasks is minimized if each task starts the processing of B at a different subset, as it is represented in Figure 3. When sets A and B are the same, establishing locks on the access to this set is more problematic. We solve this problem by splitting the set into two halves. Figure 4 represents the process. In a first stage, the elements in the subsets of the first half are combined with those in the second half. The processing of each subset in the first half is conducted concurrently by different tasks and the the subsets in the second half must be locked to avoid concurrent accesses by different tasks. Reciprocally, the elements in the second half are combined with those in the first one. The process is repeated recursively on each half until each one contains only one element, which is finally combined with itself.
- *Reduce operations.* The reduction on each subset is performed concurrently. Then, these partial results are combined. Some reductions finish when a given element is found. In our parallel implementation, the tasks are synchronized using a flag which is activated when one of the tasks finds the value. All the tasks are terminated when they find this flag activated.

3. Evaluation

The library has been validated with a set of codes parallelized using intensively the `concurrent_set` data type and its associated operations. These programs are:

- **Air control** This program implements a simple air control simulator. The air space is represented by a square space, where planes are represented by points which move according to a velocity in a given direction. In each simulation time, the program checks if the distance between two of the planes is too short. In that case, it reports a danger of collision.
- **Shortest path** This program implements the search of the shortest path between two points in a graph.
- **Barnes-Hut algorithm** This program performs the simulation of the evolution of a dynamic system where a number of particles interact according to a force whose effects diminish with the distance following the Barnes-Hut algorithm [1].

Table 1. Comparison of the programmability provided by Intel TBBs and `concurrent_set` using three quantitative metrics

Code	`concurrent_set`			Intel TBBs		
	SLOC	PE	V	SLOC	PE	V
Air control	85	338037	11	120	524608	15
Shortest path	120	1062305	25	172	1967884	30
Barnes-Hut	369	2126524	44	390	2350400	46
Delaunay	166	374757	23	206	721169	25

Table 2. Times (in milliseconds) of the sequential and parallel versions of the test programs and speedup (in parenthesis) calculated against sequential versions

Code	Seq.	1 thread	2 threads	4 thread	8 threads	16 threads	24 threads
Air control	28966	33418(0.87)	16316(1.78)	7863(3.68)	3987(7.27)	2812(10.3)	1393(20.79)
Shortest path	5566	5685 (0.98)	2852(1.95)	1487(3.74)	752(7.40)	494(11.27)	378(14.72)
Barnes-Hut	21098	21324(0.99)	11686(1.81)	6333(3.33)	3425(6.16)	2226(9.48)	1467(14.38)
Delaunay	1952	1899(1.02)	1323(1.47)	953(2.04)	789(2.47)	732(2.6)	765(2.55)

The system is simulated through a series of discrete simulation times where the interaction of each particle in the systems with all the other particles is calculated.

- **Delaunay refinement** This program refines an unstructured mesh of triangles so that it fulfills the Delaunay property [10], i.e., no angle in the mesh is less than 30 degrees.

The library has been tested from two points of view: expressivity and performance. We have evaluated the expressivity of the library following the methodology proposed in [5], which relies on three quantitative metrics: the number of source lines of code [14], the programming effort (PE) [6], and the cyclomatic number (V) [8]. The SLOC metric is influenced by the user programming style, while the two other metrics attenuate the influence of this factor. The programming effort (PE) is a function of the number of unique and total, operands and operators found in a program. The operands stand for the constants and identifiers, while the operators are the symbols or combinations of symbols that affect the value or ordering of operands. This programming effort metric is approximately proportional to the programming effort required to implement an algorithm. Finally, the cyclomatic number V is equal to P +1, P being the number of decision points or predicates in a program. The smaller V, the less complex the program is. These metrics are used in Table 1 to compare two implementations of the algorithms, one using Intel TBB library and the other one using our library. The PE and V metrics were collected using the C3MS tool used in [5]. The usage of our library implies, on average, a 20% reduction in the total size of the code with respect to the TBB version, despite the fact our library is used only in small pieces of the code. Our library also improves clearly the programmability of these benchmarks with respect to the Intel TBBs in terms of programming effort and cyclomatic number. Additionally, let us recall that our library implements the API of the `std::set` and this API is widely used by the programmers. Thus, these programmers can use easily our library, and the large amount of existing codes which already use this interface have an almost immediate translation to our library. Regarding the highly parallel operations not present in the STL API, the proposed API offers a quite natural method to express this parallelism (see Fig. 1).

The performance of the implementations that use `concurrent_set` has been compared with the performance of a sequential version of the programs. Table 2 shows the times in milliseconds of the sequential version of the program, and the parallel version implemented with our library for different numbers of threads. The speedup of the parallel version with respect to the sequential one is included between parentheses. The times where taken in a Intel Xeon hexa-core E7450 to 2.40 Ghz with 4 processor totaling 24 cores using the compiler gcc 4.1.2. The results show that the performance of the parallel version scales quite well with the number of cores. Delaunay presents low scalability because it has important non-parallel sections, and its parallelization introduces additional processes which are not present in the sequential version due to its high irregularity. For example, it performs speculative computations which are discarded if conflicts among threads are found.

4. Related work

Several works have tackled the improvement of the programmability of multicore systems using libraries. This section is focused in those which implement generic parallel data structures. The STAPL framework [13] defines `pContainers`, which are generic data structures that can be used in shared and distributed memory environments, and which can be composed hierarchically to achieve arbitrary degrees of nested parallelism. These two characteristics are shared with the Hierarchically Tiled Arrays (HTAs) [2], data structures which facilitate locality and parallelism of array intensive computations on both shared and distributed memory environments. The Intel Threading Building Blocks library [9] also provides several containers which can be used in shared memory systems, but they do not make use of the semantics of the containers to exploit data parallelism. The main advantage of our approach with respect to those ones is its conformance to the STL API. This facilitates its usage by programmers already familiar with that API and increases the migration and reuse of existing code.

Other works have also used the STL as a reference. For example HPC++ [7] is a library and language extension framework for portable parallel C++ programming. This library includes the Parallel Standard Template Library (PSTL) framework, which implements several containers (included a `set`), based in the STL API, and selected parallel versions of several algorithms which can be used in a distributed memory environment. Unlike our work, this one is focused on distributed memory environments and the number of parallel algorithms implemented is more limited, although it provides parallel versions of several containers (seven).

The Multi-Core Standard Template Library (MCSTL) [11] provides efficient parallel implementations of the algorithms in the STL API in shared memory environments. The main difference with our work is that it does not provide implementations of algorithms out of those in the STL API. The reason is that its main target is to provide parallelism simply by recompiling existing codes already written using the containers in the STL API.

5. Conclusions

This work presents a simple, portable and efficient parallel set container. The experimental results show that this parallel data type improves much the programmability in comparison to other alternatives for parallel programming such as Intel Threading Building Blocks. The usage of a STL-like interface softens the learning curve and facilitates the gradual adaption of existing codes. The codes implemented with this parallel container achieve a good performance taking advantage of the presence of an increasing number of cores and providing automatic load balancing.

Acknowledgements

This work has been supported by the Xunta de Galicia under project INCITE08PXIB105161PR and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project project TIN2010- 16735).

References

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, 1986.

[2] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. Fraguela, M. Garzarán, D. Padua, and L. Rauchwerger. Design and use of htalib–a library for hierarchically tiled arrays. *Procs. 19th Intl. Workshop on Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science*, 4382:17–32, 2007.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Procs. of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[5] C.H. González and B.B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *Procs. of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, pages 79–88, 2010.

[6] M.H. Halstead. *Elements of software science*. Elsevier New York, 1977.

[7] E. Johnson and D. Gannon. Hpc++: experiments with the parallel standard template library. In *Procs. of the 11th international conference on Supercomputing*, pages 124–131, 1997.

[8] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

[9] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 1 edition, July 2007.

[10] J.R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1-3):21 – 74, 2002.

[11] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. *Euro-Par 2007 Parallel Processing*, pages 682–694, 2007.

[12] A. Stepanov and M. Lee. The standard template library. Technical report, HP Laboratories Technical Report 95-11(R.1), 1995.

[13] G. Tanase, A. Buss, A. Fidel, H. Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, et al. The stapl parallel container framework. In *Procs. of the 16th ACM symposium on Principles and practice of parallel programming*, pages 235–246, 2011.

[14] D.A. Wheeler. Sloccount, a set of tools for counting physical source lines of code (sloc). *URL http://www. dwheeler. com/sloccount*.