Deque-free work-optimal parallel STL algorithms

Daouda Traoré[1], Jean-Louis Roch[1], Nicolas Maillard[2], Thierry Gautier[1], and Julien Bernard[1]

[1] INRIA Moais research team, CNRS LIG lab., Grenoble University, France
[2] Instituto de Informática, Univ. Federal Rio Grande do Sul, Porto Alegre, Brazil
[1] `Fristname.Lastname@imag.fr`, [2] `nicolas@inf.ufrgs.br`

Abstract. This paper presents provable work-optimal parallelizations of STL (Standard Template Library) algorithms based on the work-stealing technique. Unlike previous approaches where a deque for each processor is typically used to locally store ready tasks and where a processor that runs out of work steals a ready task from the deque of a randomly selected processor, the current paper instead presents an original implementation of work-stealing without using any deque but a distributed list in order to bound overhead for task creations. The paper contains both theoretical and experimental results bounding the work/running time.

1 Introduction

The expansion of multicore computers, including from two to dozens of processors, has recently led to a new surge of research on the parallelization of commonly used algorithms with special attention to the C++ *Standard Template Library* STL [1,2,3,4,5]. Most STL algorithms admit fine grain recursive parallelism (see Sect. 2); also implementations [6,3,4,5] commonly rely on *work stealing*, a decentralized thread scheduler: whenever a processor runs out of work, it steals work from a randomly chosen processor.

Yet, work stealing achieves provably good performances [7,8,9,10]. In the sequel, let W be the parallel work, *i.e.* the number of unit operations; let D be the parallel depth, often denoted T_∞ too since it corresponds to the theoretical time on an unbound number of processors. Then, the following classical bounds hold: with high probability (denoted w.h.p.), the number of steals is $\mathcal{O}(p.D)$ and the execution time of a recursive computation on p processors is $T_p \leq \frac{W}{p} + \mathcal{O}(D)$, a similar bound being achieved in multiprogrammed environments [8].

However, work W and depth D are antagonist criteria that often cannot simultaneously be minimized. Let W_{seq} be the minimal sequential work. For some problems, any parallel algorithm with $D = o(W)$ has a work $W = W_{seq} + \Omega(W_{seq})$. For instance, any prefix computation $-$ *i.e.* STL's `partial_sum` $-$ with depth $D = o(W_{seq})$ requires asymptotically a work $W \geq 2W_{seq}$ [11]. Thus, divide&conquer parallel prefix with depth $D = \log^{\mathcal{O}(1)} W_{seq}$, $-$e.g. Ladner-Fisher algorithm$-$ takes a time greater than $\frac{2W_{seq}}{p}$. This compares unfavorably to the tight lower bound $\frac{2W_{seq}}{p+1}$ when p is small, up to 4. Similar lower bounds occur for STL `unique_copy` and `remove_copy_if` [12].

To minimize the work while preserving the depth, we propose in Sect. 3 an implementation of work stealing that performs the recursive on-line coupling of two distinct algorithms: one is sequential, that enforces a minimal work overhead; the other one is parallel, and minimizes the depth on an unbound number of processors. Differently from classical implementations of work stealing [7,9,10], our scheme does not rely on a deque of tasks but on a list of successful steals, so it is called *deque-free*. It relies on two operations on the work, called `extract_par` and `extract_seq`; extending previous works on parallelism extraction [13,4].

In Sect. 4, we state that, based on a processor on online tuning of the chunk sizes in the loops, the list is managed with work overhead $\mathcal{O}\left(pD\log^{\mathcal{O}(1)} W\right)$. This result is used in Sect. 5 to provide parallel STL algorithms for `partial_sum`, `unique_copy`, `remove_copy_if`, `find_if`, `partition`, and `sort` that all asymptotically achieve the tight lower bound, *e.g.* $\frac{2W_{seq}}{p+1}$ for `partial_sum`. The deque free algorithm has been implemented on top of Kaapi [9]. Experimentations on a 16-core computer, presented in Sect. 6, confirm this theoretical result, outperforming Intel TBB [3] and MCSTL [5].

2 Related work: Parallel STL and work stealing

Among the works related to the parallelization of STL (see [5] for a survey), we focus on parallelizations based on recursive range partitioning with random access iterators and work stealing. In [2], a one-dimensional structure (a range) is converted into a two-dimensional structure (a collection of subranges) each subrange being processed sequentially by a given thread. Additional merge operations, possibly parallel, may be involved to complete the work. Threading Building Blocks (TBB) [3] and the Multi-Core Standard Template Library (MCSTL) [5] are both dedicated to multicore computers and based on Cilk's [7] work stealing implementation. TBB and MCSTL use recursive range partitioning: each subrange is recursively split in order to balance the load by work stealing up to a given sequential threshold. In TBB, this threshold may be adapted by the runtime – default *ideal* strategy – or tuned by the user. In MCSTL, a fixed threshold is used; additionally a partitioning in p subranges is implemented with OpenMP. Finally, although its main concern was cryptography, the paper [4] has introduced basic techniques also developed here but was restricted to only few STL algorithms.

Then, using work stealing, those libraries provide asymptotic optimal provable performances when recursive range splitting is work optimal: this is the case for instance for STL `for_each` and `accumulate`. However, recursive parallelism may introduce an arithmetic overhead, as seen in the introduction for `partial_sum`, implemented in TBB by the Ladner-Fisher algorithm with a non optimal work $2W_{seq}$. To reach the optimal work $\frac{2p}{p+1}W_{seq}$, MCSTL implements `partial_sum` by static partitioning in $p+1$ subranges at the price of poor performances on a multiprogrammed environment. A similar implementation is provided for `unique_copy`. None of these libraries provide a parallel remove_copy_if. The next section presents an implementation of work stealing which deals with work optimality.

3 The deque-free work stealing algorithm

Implementations of work stealing enforce the optimization of the sequential execution of the parallel algorithm according to the *work-first* principle [7]. Potentially parallel tasks are pushed on a local deque according to some sequential order; when a processor completes a task, it just pops the next one from the head of its deque if ready, thus following the sequential order. When the deque is empty, the processor becomes a thief, stealing a ready task from a randomly chosen victim. Although it is based on the work first principle, the deque-free work stealing presented in Algorithm 1 replaces the sequential execution of the parallel algorithm by the execution of a work optimal sequential algorithm from which, at any time, a fraction of the work may be extracted.

As is common, the computation is modeled by a DAG that unfolds dynamically as the computation proceeds. However, the unfolding is performed by the scheduling operations, which occur at each stealing attempt. Each node in the DAG corresponds to a work stream $[I_1, \ldots, I_\perp[$ of sequential instructions. For some STL algorithms – *e.g.* transform, for_each – it may correspond to a range of indexes, but not in the general case.

At any time and for any node, it is assumed possible to extract some work from the computation in progress through an operation, named extract_par in the sequel, without blocking the victim. Indeed, on a steal by P_s, let $w = [I_f, I_\perp[$ the instructions that the victim P_m has yet to perform. Then extract_par extracts from w a subrange $[I_k, I_\perp]$ with $k > f$ and creates a new node which range $w' = [I'_{k'}, I'_{\perp'}[$. The thief P_s processes w' while the victim P_m keeps on the sequential execution of w, now restricted to $[I_f, I_k[$. P_s starts the execution of w' behaving as P_m with its own thieves. Note that w' is different from the range $[I_k, I_\perp]$, since it encompasses the parallel work necessary to process what would have been a sequential work in the initial range.

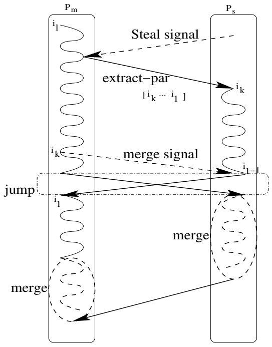
The execution of the deque-free algorithm performed by P_m is described in Algorithm 1. It is structured in two loops: the inner nano-loop corresponds to the computation of the work. The outer micro-loop corresponds to the course of P_m's thieves according to the sequential order in w. Braces indicate critical sections. The figure on the right shows the synchronization scheme between P_m and P_s, similar to the distributed list homomorphism skeleton DH [13] but in an on-line context. Synchronizations are nested and the DAG of nodes corresponding to a given execution matches a Cilk strict multithreaded computation [7].

Generally, the extracted work w' of P_s differs from the stolen range $[I_k, I_\perp]$. Moreover, a subsequent merge operation may be required to complete the result r of $[I_1, I_\perp[$ from the ones of both $[I_1, I_k[$ and $[I'_{k'}, I'_{\perp'}[$. This merge operation may be empty, *e.g.* for_each. However, in general, it consists into non-blocking completion of $[I_1, I_k[$ – resp. $[I'_{k'}, I'_{\perp'}[$ – based on a merge_m– resp. merge_s – operation by P_m – resp. P_s –. Each of both operations corresponds to a sequential stream of instructions, that can be executed in sequential in a non-blocking way if no steal occurs. Both streams are parallel; the scheme is similar to the distributed list homomorphism skeleton DH [13].

The `merge_m` and `merge_s` instructions to perform depend on the state of the last thief P_s of P_m when P_m completes I_{k-1}. Two cases arise. Either P_s has previously completed w_s; then the results from both P_s and P_m can be merged by P_m to complete r; in this case `merge_s`$= \emptyset$. Or the thief has completed I'_{l-1} with $k' < l < \perp'$ and is currently processing I'_l, head of $w' = [I'_l, I'_{\perp'}[$. Then, P_m preempts P_s after I'_l; after synchronization, both perform their part of the distributed merge, respectively `merge_m` and `merge_s`. Also, the deque of tasks is replaced by a list of stolen computations managed locally on each victim P_m: at `extract_par`, P_m inserts at the head of P_m's list a pointer to the stolen work. After completion of `merge_m` P_m accesses the head of its thief list that corresponds to its next instruction, previously stolen by a thief P'_s; and proceeds to a new distributed merge instruction now between P_m and P'_s. P_m is then only interrupted when it performs a preemption on a stealer process, then waiting at most the completion of the current `run-seq` on P_s.

```
Init   v, w ← { extract-par }
// v is the victim, w the stolen work
// v initialized w.SignalMerge to false
// v initialized w.SignalJump to false
thiefList ← ∅
// Micro-loop
while ( { w ≠ ∅ }) or ( { thiefList ≠ ∅ }) do
    // Nano-loop:
    while($I_s$ ← { w.extract-seq })≠ $I_\perp$)
    $I_s$.run_seq
    if { w.SignalMerge = true } then
        { v.thiefList.insertHead( thiefList ) }
        thiefList ← ∅ ; { w.SignalJump ← true }
        merge_s
    end
    if (thiefList = ∅) break else
        $w_s$ ← thiefList.head
        { $w_s$.SignalMerge ← true }
        while( { $w_s$.SignalJump ≠ true })
        yield
        jump ; merge_m
    end
end
{ w.SignalJump ← true }
goto 1
```

Algorithm 1: The Deque-Free Algorithm.



The main process P_m runs the sequential algorithm on its container.

P_s calls `extract_par` and runs $[I'_k, I'_\perp]$ in parallel.

The arithmetic work is processed in the nano-loop. To amortize overhead, at each step, P_m extracts from w its next sequential chunk I_s of instructions by `extract_seq` and runs the sequential algorithm on it. It stops only when preempting a thief, thus few times if the depth is small. The next section states some choices for the `extract_seq` chunks in order to achieve optimal work for STL algorithms.

4 Theoretical bounds for online granularity

Let $w = [I_k, I_\perp[$ be the stream of instructions that an arbitrary process P_m is processing at a given top. Let $W_{seq}(w)$ – resp. $D(w)$ – be the work – resp. *potential depth*– in number of unit time instructions to complete w in sequential – resp. in parallel with an unbounded number of steal operations–. The next theorem states that extracting blocks of $D(w)$ operations with `extract_seq` achieves an optimal asymptotic sequential work while not increasing the potential depth on an unbounded number of processors,

Theorem 1. *If the following two hypothesis hold:*
(1) $D(w) \geq \log_2 W_{seq}(w)$, *(2)* $\forall \epsilon > 0$, $\lim_{W_{seq}(w) \to \infty} \frac{D(w)}{W_{seq}^{\epsilon}(w)} = 0$,
and if each `extract_seq` *operation returns* $D(w)$ *unit time instructions, then for all* $\delta > 0$*, the number of calls to* `extract_seq` *is bounded by* $(1+\delta)\frac{W_{seq}(w)}{\log_2 W_{seq}(w)}$ *when* $|w|$ *tends to infinity.*

Proof. From hypothesis 2, $\forall \delta, \exists w_0, \forall W_{seq}(w) \geq w_0, D(w) < \frac{\delta}{2} W_{seq}^{\frac{\delta}{2+\delta}}$; and then, $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2}\frac{W_{seq}}{D(w)}$. Now, let the interval of iterations be split in two parts: (1) For the iterations such that there remain more than $W_{seq}^{\frac{2}{2+\delta}}$ operations: there are at least $D(W_{seq}^{\frac{2}{2+\delta}})$ operations to be extracted at each call to `extract_seq`, which, by hypothesis 1, is larger than $\log\left(W_{seq}^{\frac{2}{2+\delta}}\right) = \frac{2}{2+\delta} \log W_{seq}(w)$. Therefore, there are at most $W_{seq}(w)$ divided by this number calls to `extract_seq`, *i.e.* $(1+\frac{\delta}{2})\frac{W_{seq}(w)}{\log W_{seq}(w)}$. (2) For the next (fine-grained) iterations, there are at most $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2}\frac{W_{seq}(w)}{D(w)}$ operations to be performed (for any $W_{seq}(w) \geq w_0$), and therefore at most this number of `extract_seq` calls (if each one includes only one operation). Therefore, and with hypothesis 1, the number of calls to `extract_seq` is, in this phase, less than $\frac{\delta}{2}\frac{W_{seq}(w)}{\log W_{seq}(w)}$. Adding the two bounds, one gets that the number of `extract_seq` calls is at most $\left(1+\frac{\delta}{2}+\frac{\delta}{2}\right)\frac{W_{seq}(w)}{\log W_{seq}(w)}$, hence the expected result.

Thus, the overhead induced by `extract_seq` operations in the nano-loop is asymptotically upper bounded by $\frac{W_{seq}}{\log W_{seq}}$ while the micro/nano loops does not increase the initial potential depth D. This bound is similar to direct recursive partitioning by work stealing when recursivity is halved at a grain $\Omega(D)$, but then at the price of an increase of the depth by a factor $\rho > 1$. The first hypothesis seems reasonable, since $\log_2 W_{seq}(w)$ is a lower bound on the potential depth if $W_{seq}(w)$ is optimal. The second means that the work to be performed is much larger than the critical path, which is verified for STL algorithms with polylog depth. Yet, the order of $W_{seq}(w)$ may be unknown, *e.g.* for STL `find_if`. In order to extract either $\log_2(W_{seq}(w))$ (by `extract_seq`) or a fraction (by `extract_par`) of w, a third level of control of the instruction flow is used, called the *macro-loop* [4]. It consists in spanning an *a-priori* unknown set of instructions in steps (or chunks) of size $s_1, s_2, \ldots s_m$, analogously to Floyd's algorithm. The macro-step i of size s_i may start only after completion of step $i-1$. Eventually, the i-th chunk will contain extra instructions, besides $I_\perp$. But, to obtain $\sum_{i=1,m} s_i \simeq W_{seq}(w)$, in the sequel $s_i = \frac{\sum_{j=1,i-1} s_j}{\log \sum_{j=1,i-1} s_j}$ (with s_1 being some constant value). Then the macro-loop preserves asympt. the work $W_{seq}(w)$ while increasing the potential depth by a factor at most $\log W_{seq}(w)$. Embedding the micro-nano loop in the macro-loop is straight-forward and leads to the final deque-free work stealing algorithm, whose performance is stated in the next theorem:

Theorem 2. *If the parallel execution of $w = [I_1, \ldots, I_\perp[$ on an unbounded number of processors performs $(1+\alpha)W_{seq}(w)$ operations with $\alpha \geq 0$, then, w.h.p., the [macro-micro-nano] deque-free algorithm completes w on p processors in time:*

$$T_p(w) = \frac{\alpha+1}{\alpha+p} W_{seq}(w) + \mathcal{O}\left(D(w)\log^2 W_{seq}(w)\right), \textit{when } W_{seq}(w) \to \infty.$$

Proof. Using macro-steps only increases the number of instructions from W_{seq} to $W_{seq}(w) + \mathcal{O}\left(\log W_{seq}(w)\right)$. From theorem 1, this number is increased by the nano-loop up to $W_{seq}(w)\left(1 + \mathcal{O}\left(\frac{\log W_{seq}(w)}{W_{seq}(w)}\right) + \mathcal{O}\left(\frac{1}{\log W_{seq}(w)}\right)\right) = W_{seq}$ asymptotically. If only one processor P_m runs the program, it will never be preempted and will simply run the sequential algorithm, *i.e.* W_{seq} operations. On $p \geq 2$ processors, $p-1$ perform `extract_par` operations, completing at most $W_{par}(m) = (1+\alpha)W_{seq}(m)$ operations where m is the instructions stream corresponding to those $p-1$ thefts; while P_m completes $W_{seq} - W_{seq}(m)$ operations. Since each call to `extract_par` extracts a fraction of the work, the victim whole number of `extract_par` operations in the macro-loop is $\mathcal{O}\left(\log^2 W_{seq}(w)\right)$ w.h.p; then the sequential process waits $\mathcal{O}\left(D(w)\log^2 W_{seq}(w)\right)$ for all preemption operations. Since P_m is never idle, except when preempting a theft or eventually during the last macro-step of size at most $\frac{W_{seq}(w)}{\log W_{seq}(w)}$: $T_p(w) = W_{seq}(w) - W_{seq}(m) + \mathcal{O}\left(D(w)\log^2 W_{seq}(w)\right)$ (1). Besides, due to work stealing, w.h.p. $T_p(w) \leq \frac{\alpha+1}{p-1} W_{seq}(m) + \mathcal{O}\left(D(w)\log^2 W_{seq}(w)\right)$ (2). Elimination of $W_{seq}(m)$ in (2) from (1) leads to $T_p = \frac{\alpha+1}{\alpha+p} W_{seq}(w) + \mathcal{O}\left(D(w)\log^2 W_{seq}(w)\right)$.

This parallel coupling of two algorithms is totally adaptive, without any overhead when run on a single processor. The next section illustrates its optimality on a few STL algorithms.

5 Application to the STL

The generic three loops deque-free workstealing is specialized here to the STL.

for_each, transform, accumulate, inner_product: the deque-free implementation is direct since $\alpha = 0$: the work is defined by a subrange [f,l[of indexes; `extract_par` extracts the last half of the victim range; `run-seq` performs the call to the native sequential STL. In `for_each` and `transform`, there is no merge (nop). In `accumulate` and `inner_product`, `merge_m` sums the local result with the result of the thief, while `merge_s` is nop. An optimal parallel time $T_p(w) = \frac{W_{seq}(w)}{p}$ is asymp. reached. However, by halving recursive calls at a depth $\log W_{seq}(w)$, the sequential execution of the recursive partitioning algorithm is asympt. optimal. Then, here, our scheme does not improve recursive partitioning by work stealing. The parallel time is $T_p = \frac{W_{seq}}{p} + \mathcal{O}\left(\log^2 W_{seq}\right)$ which asympt. reaches the lower bound.

partial_sum. The macroloop splits the range in $\mathcal{O}\left(\log W_{seq}\right)$ subranges of size $s_i = \frac{\sum_{j=1,i-1} s_j}{\log \sum_{j=1,i-1} s_j}$ computed consecutively. Each macrostep is parallelized as follows: the work and `extract_par` are identical to `transform`. Only one process P_M follows the macrostep and preempts thieves, while other processes never preempt similarly to `transform`; `run-seq` performs a sequential `partial_sum`. `merge_m` gets from the thief the last computed prefix in its subrange and jumps to the first index after this subrange. `merge_s` gets from P_M the last computed prefix and apply a parallel `transform` to compute the final prefix of its subrange. Due to merge, the depth of a macrostep is $\mathcal{O}\left(\log^2 W_{seq}\right)$. Since $\alpha = 1$, the parallel time is $T_p = \frac{2W_{seq}}{p+1} + \mathcal{O}\left(\log^3 W_{seq}\right)$ which asympt. reaches the lower bound.

unique_copy, remove_copy_if. For both, our implementation is similar to `partial_sum` and achieves the similar lower bounds as for stream computations [12].

find, find_if: Our implementation relies on the macroloop: the macrostep i is here of size $s_i = \frac{\sum_{j=1,i-1} s_j}{\log \sum_{j=1,i-1} s_j}$. Each macrostep is processed as follows; `run-seq` is a sequential `find`; `extract_par` is identical to transform; `merge_s` is nop; `merge_m` compares the local index found (if any) with the one of the thief, possibly preempted, to take the minimum index. Parallel `find` performs early termination within a macrostep: if a process finds the first iterator N in `run-seq`, then it empties its work. Due to the microloop, all the other processes, when they try to merge with it, or to steal part of its work, terminate. This is also true as soon as one process −in particular the main process− reaches the upper bound r. The algorithm runs in time $T_p = \frac{W_{seq}}{p} + \mathcal{O}\left(\log^3 W_{seq}\right)$

partition and ***sort***: Our implementation of `partition` (on which the `sort` is based) [14] performs a parallel, in-place partition. The work corresponds to two subranges S_L and S_R and maintains pointers on the left-most (resp. right-most) block b_L in S_L (resp b_R in S_R); `run-seq` performs a specific partition on the distributed container $[b_L, b_R]$ to partition it, performing in average the same number of swap operations than a sequential partition of a contiguous container. Thus, it follows the STL partition scheme. `extract_par` splits S_L and S_R into two halves, the thief partitioning the right-most half of the left block, and the left-most half of the right block. Special care has to be taken when one whole half of the elements has already been partitioned, since the thieves always try to steal two blocks of elements. In this case, the remaining blocks of elements to be partitioned must be re-ordered, so that the algorithm may proceed. The algorithm runs in time $T_p = \frac{W_{seq}}{p} + \mathcal{O}\left(\log^2 W_{seq}\right)$.
`sort` is based on introspective sorting [14] which is based on quicksort. When the size n of input array is lower than a fixed tuned threshold g, the sequential

partition algorithm is applied. Else, the previous deque-free `partition` is performed. When completed, the two parts of the array are sorted in parallel with workstealing. It runs in expected time $T_p = \frac{W_{seq}}{p} + \mathcal{O}\left(\log^3 n\right)$.

6 Experimentations

We have implemented the deque-free work stealing algorithm on top of Kaapi [9] and specialized it to implement the previous algorithms: this implementation is referred as DFW below. Experimentations have been performed on an AMD Opteron NUMA machine (8 dual-core, 2.2 GHz, Lunix 2.6.23-1-amd64, architecture x86_64). All tests were run at least 10 times: the average, the fastest and the slowest execution time of the 10 executions are presented. We have used MCSTL_0.8.0-beta and the TBB 20_014 stable version. All programs (MCSTL, TBB, DFW) were compiled with the same gcc 4.2.3 and the same option -O2. All input data follow uniform distributions on an array of n doubles.

partial_sum. Figure 1 compares, for $n = 10^8$, DFW `partial_sum` to MCSTL [5] `partial_sum` which is based on a static splitting into $p + 1$ parts. The STL partial_sum average time of this experiment is 1.24s: the sequential execution with DFW exhibits no overhead. DFW outperforms MCSTL, and the runtimes are more stable. We argue that this is due to possible perturbations by system processes since, while work stealing is stable, static splitting is sensitive to any perturbation due to intermediate barriers. For this fine grain and memory intensive operation, performances do no scale above 4 processors, but when we have augmented the unit operation, the speedup has augmented. Figure 2 compares with the TBB implementation, which is recursive and depends on the chosen granularity. DFW yet reaches the theoretical bound $\frac{2W_{seq}}{p+1}$ with high stability; while TBB seems to scale and reaches the $T_p = \frac{2W_{seq}}{p}$ for some p.

unique_copy. Figure 3 compares the runtime of DFW `unique_copy` to MCSTL on $n = 10^8$ double. MCSTL implements a static partitioning in $p + 1$ parts. The sequential STL execution time for this experiment is 0.61s. DFW performs much better than MCSTL and appears more stable. For this fine grain and memory intensive operation, performances do no scale above 4 processors.

remove_copy_if. Figure 4 shows the execution times obtained with DFW `remove_copy_if` with a predicate of $16\mu s$ and $n = 10^6$. At this larger grain, performances scales up to 16 core, reaching the $\frac{W_{seq}}{p}$ lower bound [12].

find_if. The experimentations consist in three measures with $n = 10^6$ where the position k of the first element to find is $10^2, 10^4, 10^6$. The time to compute `pred` is $\tau_{pred} = 35\mu s$. In table 2 DFW find_if provides speed up from $k = 10^6$, scaling up to 12 processors. In any case no speed down is observed with respect to the STL sequential time. This illustrates the DFW macroloop scheme in which one among the p processors always follows the sequential order.

sort. Table 1 shows the speed-up for sort w.r.t. sequential STL for $n = 6.410^5$ and $n = 3.2710^8$ and with 2, 8 and 16 threads. Threshold is $g = 10000$. DFW performs better than the two other ones for large n; this is not the case for small values even if close.

Speed-Up	**Sort**: input size n					
w.r.t. STL	$n = 6.410^5$			$n = 3.2710^8$		
sequential	p=2	p=8	p=16	p=2	p=8	p=16
DFW	1.6	2.3	1.3	1.9	6.9	8.1
TBB	1.9	3.4	1.8	1.9	4.9	5.1
MCSTL	1.7	2.1	2.0	1.8	5.1	7.1

Table 1. `sort`: speed-up w.r.t. STL.

DFW **find_if**: $n = 10^6$ elements;				
first matching is at position k				
k	p=2	p=8	p=12	p=16
k=10^2	0.99	0.99	0.99	0.99
k=10^4	1.92	5.51	6.02	6.35
k=10^6	1.99	7.89	11.4	13.1

Table 2. find_if: speed-up w.r.t. STL.

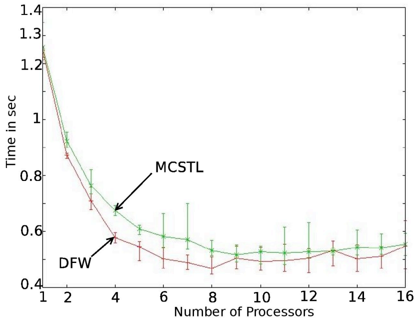


Fig. 1. partial_sum: runtime of DFW and MCSTL *vs.* number of processors, on $n = 10^8$ double.

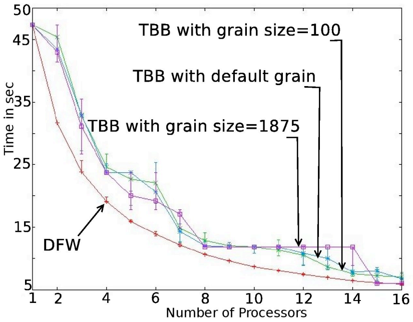


Fig. 2. partial_sum: runtime of DFW and TBB with $n = 3.10^4$ objects (unitary operation time $\tau_{op} = 1.5ms$).

7 Conclusions

We have introduced a generic deque-free work-stealing implementation that we have used to parallelize 80% of the STL algorithms for containers with random iterators. The proposed STL algorithms all achieve polylog potential depth on an unbounded number of processors, while the size of the scheme adapts automatically to the available resources and their speeds.

A theoretical analysis is provided which proves work optimality for the STL presented algorithms with asymptotic ratio 1 w.r.t. to the lower bound on p identical processors, while this lower bound increases with p. In particular, the partial sum (prefix) computation is performed in asymptotic optimal work $\frac{2p}{p+1}W_{seq}$ without reference to the number of processors in the code, except inside the implementation of work stealing. To our knowledge, it provides the first provably work optimal parallel implementations of `partial_sum`, `unique_copy` and `remove_copy_if`. When compared to TBB and MCSTL, the experimentations exhibit good performances and stability w.r.t. static partitioning and direct recursive work stealing.

Since it achieves provable optimal performance, we think that this scheme is a basis to design of algorithms that oblivously adapt to the number of processor and their respective speeds. The considered algorithms (except sort) are related to a sequential linear traversal of the container; they are both processor and cache-oblivious on the CO-model. A perspective is then to extend this distributed scheme to algorithms with non-linear complexity in the input/output on distributed memory architectures.

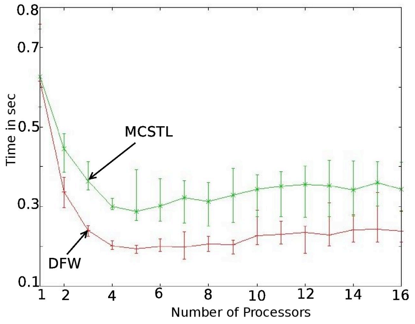


Fig. 3. unique_copy: runtime of DFW *vs.* MCSTL on of $n = 10^8$ double.

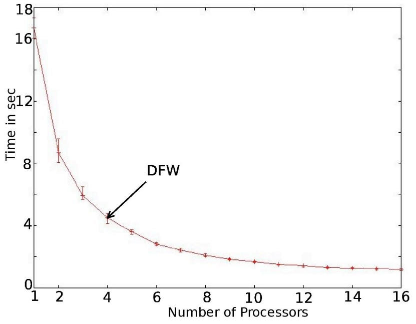


Fig. 4. `remove_copy_if`: runtime of DFW (predicate time $\tau_{pred} = 16\mu s$).

References

1. Musser, D.R., Derge, G.J., Saini, A.: STL tutorial and reference guide, second edition. Addison-Wesley, Boston, MA, USA (2001)
2. Austern, M.H., Towle, R.A., Stepanov, A.A.: Range partition adaptors: a mechanism for parallelizing stl. SIGAPP Appl. Comput. Rev. **4**(1) (1996) 5–6
3. Reinders, J.: Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)
4. Danjean, V., Gillard, R., Guelton, S., Roch, J.L., Roche, T.: Adaptive loops with kaapi on multicore and grid: Applications in symmetric cryptography. In: ACM PASCO'07, London, Canada (2007)
5. Singler, J., Sanders, P., Putze, F.: The multi-core standard template library. In LNCS 4641, S.V., ed.: Euro-Par 2007. (2007)
6. Yu, H., Rauchwerger, L.: An adaptive algorithm selection framework for reduction parallelization. IEEE Trans. Par. Dist. Syst. **17**(10) (2006) 1084–1096
7. Frigo, M., Leiserson, C., Randall, K.: The implementation of the cilk-5 multithreaded language. In: SIGPLAN Conf. PLDI. (1998) 212–223
8. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. **34**(2) (2001) 115–144
9. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: ACM PASCO, London, Canada (2007) 15–23
10. Chowdhury, R.A., Ramachandran, V., Blelloch, G.E., Gibbons, P., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: SIAM/ACM Symposium on Discrite Algorithms (SODA). (2008)
11. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. Journal of the ACM **27**(4) (1980) 831–838
12. Bernard, J., Roch, J.L., Traore, D.: Processor-oblivious parallel stream computations. In: 16th Euromicro Conf. PDP, Toulouse, France (2007)
13. Bischof, H., Gorlatch, S., Leshchinskiy, R.: Generic parallel programming using c++ templates and skeletons. In 3016, S.V.L., ed.: Domain-Specific Program Generation. (2004) 107–126
14. Traoré, D., Roch, J.L., Cérin, C.: Algorithmes adaptatifs de tri parallèle. In: RenPar'18 / SympA'2008 / CFSE'6, Fribourg, Switzerland (2008)