Scalable Computing: Practice and Experience
Volume 9, Number 3, pp. 179–184. http://www.scpe.org
ISSN 1895-1767

SINGLE-PASS LIST PARTITIONING

LEONOR FRIAS*, JOHANNES SINGLER†, AND PETER SANDERS†

Abstract. Parallel algorithms divide computation among several threads. In many cases, the input must also be divided. Consider an input consisting of a linear sequence of elements whose length is unknown a priori. We can evenly divide it naïvely by either traversing it twice (first determine length, then divide) or by using linear additional memory to hold an array of pointers to the elements. Instead, we propose an algorithm that divides a linear sequence into p parts of similar length traversing the sequence only once, and using sub-linear additional space. The experiments show that our list partitioning algorithm is effective and fast in practice.

Key words: parallel processing, sequences, algorithmic libraries

1. Introduction. An algorithm is a well-defined computational procedure that takes input values and produces output values. A parallel algorithm divides computation among several threads. For data-parallel algorithms, the input must be also divided into (independent) parts of similar size, so that parallel computation is effective. Most parallel algorithm descriptions disregard how the input is actually divided (or assume that the input can be divided by index computations). If we want to use such algorithms in practice, we have to deal with sequences that are non-trivial to partition.

In particular, we focus on the algorithms in the Standard Template Library (STL), which is a part of the C++ programming language [1]. In this setting, the input consists of a sequence of elements, given as a pair of iterators. The standard requires these to satisfy only the forward iterator concept. The only feasible operations for a forward iterator are dereferencing, and advancing to the next element. Thus, a forward iterator sequence is a linear sequence of unknown length (e. g. a sequence implemented as a singly-linked list). Traversing it is inherently sequential. Even if the sequence comes with an associated length variable, this information is lost when passing iterators, as required by most STL algorithms. Also, keeping the length up to date is inefficient for operations like splitting one list into two, at a known iterator. This is why `std::list`, for example, does not guarantee the calculation of `size()` to take only constant time.

However, the speedup of a parallelized program is limited by the sequential portion, according to Amdahl's law. Hence, making the sequential partition before the paralleling processing as fast as possible, is of utmost importance.

Given that the length of the sequence is unknown, one could think of first traversing the sequence to determine its length, and then traversing it a second time to actually divide the sequence. We call this algorithm TraverseTwice. However, traversing the sequence can be expensive, so we do not want to pay for it twice. The elements can be spread in memory cache-unfriendly, and/or calculating the next element may be costly. To avoid this, one could also think of using a dynamic array, storing the pointers to the elements there, effectively converting the sequence to a randomly accessible one. We call this algorithm PointerArray. However, this is very costly in terms of additional space. We subsume both algorithms as the *trivial* solutions.

In this paper, we present an algorithm that divides such a linearly traversable sequence into p parts of similar length using only a little additional space, and accessing each element exactly once. In the next section, we first formally define the problem, called *list partitioning*. Then, we present our single-pass list partitioning algorithm. Next, we present some experiments on list partitioning: we evaluate the algorithms by themselves as well as their impact in parallel performance. Finally, we sum up the results.

2. Problem Definition. A linearly traversable sequence of unknown length n, given by two forward iterators, is to be divided into p parts of almost equal length. Let the ratio r be the quotient of the length of the longest part and the length of the shortest part. It is a good quality measure for the partitioning, since it correlates to the efficiency of processing the parts in parallel, given that processing time is proportional to a

*Universitat Politècnica de Catalunya, Dep. de Llenguatges i Sistemes Informàtics. Jordi Girona Salgado, 1-3. 08034 Barcelona, Spain. lfrias@lsi.upc.edu Supported by ALINEX project (TIN2005-05446) and by grants number 2005FI 00856 and 2006BE-2 00016 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund. This work was partially done while visiting Universität Karlsruhe.

†Universität Karlsruhe, Institut für Theoretische Informatik, ITI Sanders. 76128 Karlsruhe, Germany. {singler, sanders}@ira.uka.de Partially supported by DFG grant SA 933/3-1.

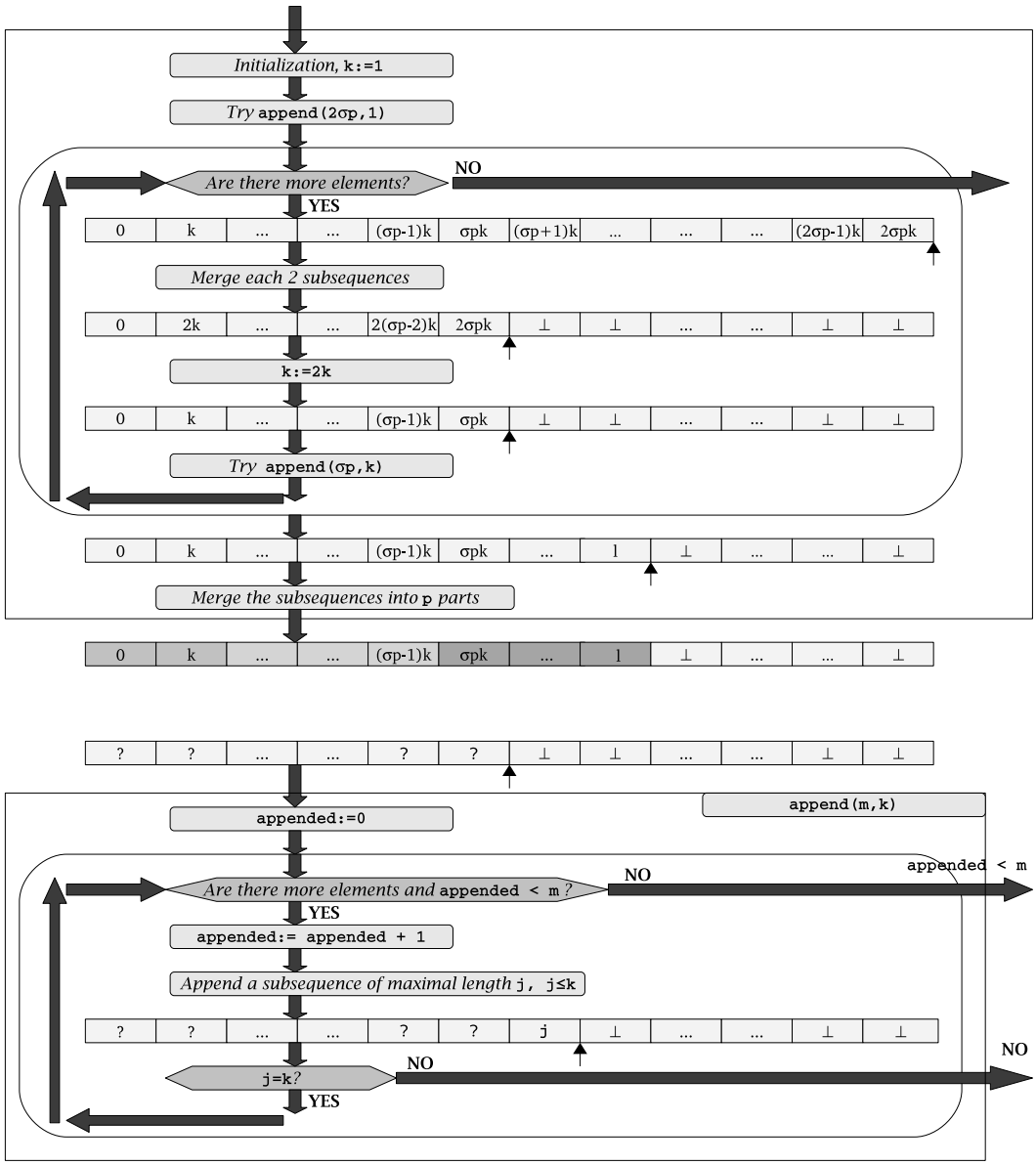

FIG. 2.1. *Basic* SINGLEPASS *list partitioning algorithm scheme.*

part's length. Thus, to guarantee good efficiency, r should be upper-bounded by a constant R at any time, only depending on a tuning parameter, namely the oversampling factor $\sigma \in \mathbb{N} \setminus \{0\}$.

W. l. o. g. we assume that the input sequence has length at least σp, i. e. $n \geq \sigma p$. Otherwise, if $p \leq n$, we can lower σ down so that $\sigma p \leq n$. If $p > n$, we reduce p to n to avoid that any part is empty (and therefore, $r = \infty$), which would not be sensible for our purposes.

Actually, we can consider this an online problem because the input is given one element at a time, without information about the whole problem. Thus, we can define a competitive ratio between the optimal offline algorithm and our online algorithm. For the optimal offline algorithm, the difference in part lengths is at most 1, which gives a ratio $r_{\mathrm{OPT}} = \lceil n/p \rceil / \lfloor n/p \rfloor \overset{n\to\infty}{\to} 1$.

3. The SinglePass Algorithm. Let L be a forward linearly traversable input sequence (e. g. a linked list). Our single-pass algorithm, denoted SINGLEPASS, keeps a sequence of boundaries $S[0 \ldots p]$, where $[S[i], S[i+1])$ defines the i^{th} subsequence of L. Inserting a subsequence into S means storing its boundaries in the appropriate places. A boundary is identified by its rank in L.

The basic SinglePass algorithm works as follows:

1. Let $k := 1$, $S := \{\}$.
2. Iteratively append to S at most $2\sigma p$ 1-element consecutive subsequences from L.
   $S = \{0, 1, 2, \ldots, 2\sigma p\}$
3. While L has more elements do:
   Invariant: $|S| := 2\sigma p$, $S[i+1] - S[i] = k$
   (a) Merge each two consecutive subsequences into one subsequence.
   $S[0, 1, 2, \ldots, \sigma p] := S[0, 2, 4, \ldots, 2\sigma p]$
   This results in σp subsequences of length $2k$.
   (b) Let $k := 2k$.
   (c) Iteratively append to S at most σp consecutive subsequences of length k from L.
   $S := \{0, k, \ldots \sigma pk, (\sigma p + 1)k, (\sigma p + 2)k, \ldots, l\}$,
   $\sigma pk < l \leq 2\sigma pk$
   If L runs empty prematurely, the last subsequence is shorter than k.
4. The $\sigma p \leq |S| \leq 2\sigma p$ subsequences are divided into p parts of similar lengths as follows. The $|S| \bmod p$ rightmost parts are formed by merging $\lceil |S|/p \rceil$ consecutive subsequences each, from the right end. The remaining $p - (|S| \bmod p)$ leftmost parts are formed by merging $\lfloor |S|/p \rfloor$ consecutive subsequences each, from the left end.

The algorithm (visualized in Figure 2.1) takes special care of the rightmost subsequence E, which might be shorter than the others, i. e. $|E| \leq k$. Let T be the part containing E, there is no part that consists of more subsequences than T. So, if exactly one part is longer than all the others (i. e. $|S| \bmod p = 1$), this is specifically T. In this case, T differs from the other parts in $|E|$ elements. As a whole, the algorithm guarantees that in the worst-case, two parts differ at most in one complete subsequence (i. e. in at most k elements).

The basic SinglePass algorithm needs $\Theta(\sigma p)$ additional space to store S. The time complexity is $\Theta(n + \sigma p \log n)$. This is proved as follows. We need to traverse the whole sequence, taking $\Theta(n)$ time. In addition, step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

The worst case ratio r is bounded by $\frac{\sigma+1}{\sigma}$. The worst case occurs when just one complete subsequence was appended after reducing the list. W. l. o. g., to analyze the average ratio, we consider only complete subsequences, therefore $\sigma p \leq |S| < 2\sigma p$. The average ratio is upper-bounded by

$$\begin{aligned}
\mathbb{E}r &< \frac{1}{\sigma p} \sum_{\ell=\sigma p}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} = \frac{1}{\sigma p} \left(\sigma + \sum_{\ell=\sigma p, p \nmid \ell}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \right) \\
&= \frac{1}{\sigma p} \left(\sigma + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{\sigma + \ell + 1}{\sigma + \ell} \right) = \frac{1}{\sigma p} \left(\sigma p + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma + \ell} \right) \\
&= 1 + \frac{1}{\sigma p} \left((p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma + \ell} \right) = 1 + \frac{1}{\sigma p} \left((p-1)(\Psi(2o) - \Psi(\sigma)) \right) \\
&\approx 1 + \frac{1}{\sigma p} \left((p-1)(\ln(2o) - \ln(\sigma)) \right) = 1 + \frac{1}{\sigma p} \left((p-1) \ln(2) \right).
\end{aligned}$$

E. g., for $\sigma = 10$ and $p = 32$, the longest subsequence is at most 10% longer than the shortest one, expectedly 7% longer.

A generalization of this algorithm performs step 3a and 3b only every m^{th} loop iteration. In the remaining iterations of the main loop, S is doubled in size, so that space for additional subsequences is needed. This is equivalent to increasing the oversampling factor to σn^γ with $\gamma = 1 - 1/m$.

The generalized SinglePass algorithm needs as many iterations of Step 3 as the basic algorithm, i. e. $\Theta(\log n)$ iterations. The additional space increases, but sub-linearly, growing with $O(\sigma p n^\gamma)$. The time complexity of this algorithm is $\Theta(n + \sigma p(n^\gamma + \log n))$.

In the worst case, the longest sequence and the shortest sequence have length $(n^\gamma + 1)k$ and $(n^\gamma)k$, respectively. It holds $\sigma p n^\gamma k \approx n$, so $k \approx \frac{n^{1/m}}{\sigma p}$. Subsuming this, the lengths of the subsequences do at most differ by $\frac{n^{1/m}}{\sigma p}$ elements, i. e. the difference decreases relatively to n, as n grows. Therefore, the bound for r also converges to 1.

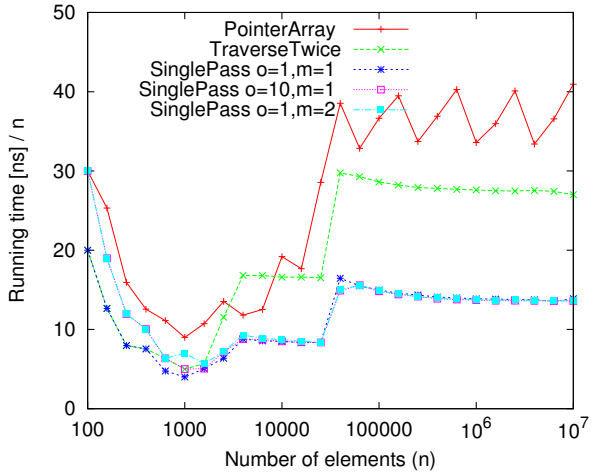

FIG. 4.1. *Running times of the list partitioning algorithms for* $p = 4$.

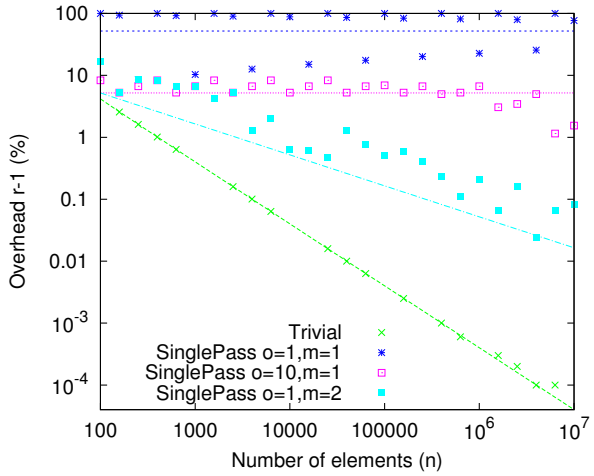

FIG. 4.2. *Quality of the list partitioning algorithms for* $p = 4$. *We show the worst-case overhead ratio* $h = r - 1$, *as well as its expectancy. The results are in %. Note that the missing points are actually 0.*

Generally speaking, the choice of m trades of time and space versus solution quality. The larger m, the more memory and time is used, but r becomes better. This is the same effect as would be caused by a dynamically growing σ. Choosing $m = 2$ appears to be a good compromise.

4. Experiments. We have implemented our SINGLEPASS algorithm in the general form, so it subsumes the two variants. We have implemented it in C++ and we have included it into the MCSTL [2]. MCSTL stands for Multi-Core Standard Template Library and is a parallel implementation of the standard C++ library. Besides, we have implemented the two naïve algorithms, namely TRAVERSETWICE and POINTERARRAY algorithms. Dynamic arrays have been implemented using the STL `vector` class.

We have performed two kinds of experiments. First, we present the evaluation of the list partitioning algorithms isolatedly. Then, we present the impact of the list partitioning algorithm as part of two parallelized STL algorithms.

4.1. Comparison of List Partitioning Algorithms. We have compared all the algorithms both measuring the running time as well as the quality of the results. Concerning quality, we have computed both the worst-case ratio r and its expectancy. For a better plot reading, we have rescaled these results using the *overhead* ratio h. h is defined from the ratio r as $h = r - 1$. It must be noted that the actual quality of the results is deterministic with respect to a problem size. That is, the quality of our solution does not depend on the specific input data but only on its size.

Setup.. We have tested our program on an AMD Opteron 270 (2.0 GHz, 1 MB L2 cache). We have used GCC 4.2.0 as well as its libstdc++ implementation, compiling with optimization (`-O3`).

Parameters for Testing. We have repeated our experiments at least 10 times. The focus is on results for $p = 4$. Recall that as p grows, r becomes smaller.

For SINGLEPASS, there are the following parameter combinations: *1)* ($o = 1$, $m = 1$), *2)* ($o = 10$, $m = 1$) and *3)* ($o = 1$, $m = 2$). Therefore, it uses $\Theta(p)$, $\Theta(10p)$, and $\Theta(\sqrt{n}p)$ additional space, respectively.

Results. Figure 4.1 summarizes the performance results, and Figure 4.2 the quality results. We see that the performance of the SINGLEPASS algorithm is very good. In particular, it takes only half the time compared to the TRAVERSETWICE algorithm, and even less compared to the POINTERARRAY algorithm. That is, we can divide a sequence into parts using virtually the same time as for only traversing it once. Further, the varying running times for POINTERARRAY must be due to the amortization of the vector doubling cost (i. e. depending on how much of the allocated memory has been actually used by the vector).

The quality of the solution for our algorithm improves with the amount of additional space allowed for the auxiliary sequence S (i. e. increased o or m). The simplest variant ($o = 1$, $m = 1$) has a worst-case ratio of 2 and an average ratio of 1.5. In addition, the overhead $r - 1$ is divided by o (in our case, $o = 10$). When setting

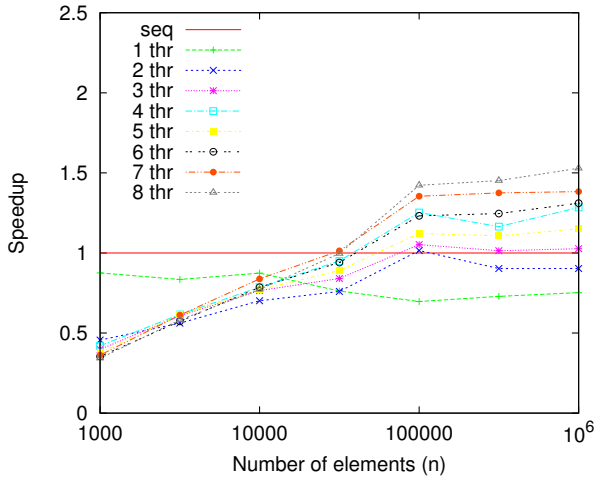

FIG. 4.3. *Speedup for STL list* `sort` *using* TRAVERSETWICE *partitioning.*

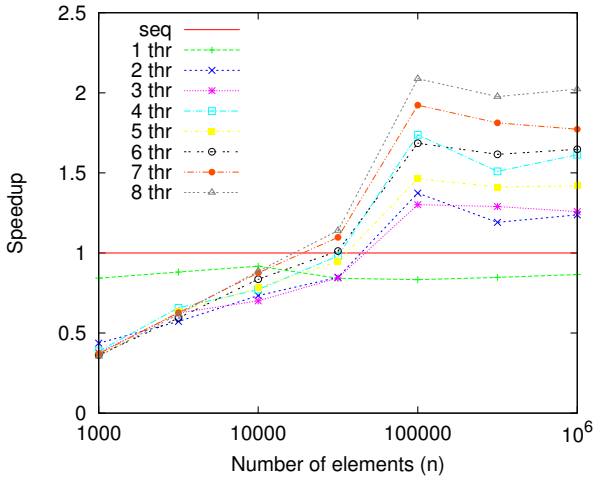

FIG. 4.4. *Speedup for STL list* `sort` *using* SINGLEPASS *partitioning.*

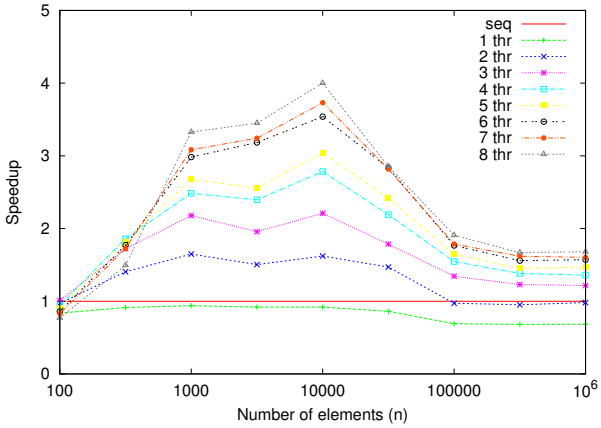

FIG. 4.5. *Speedup for* `accumulate` *using* TRAVERSETWICE *partitioning.*

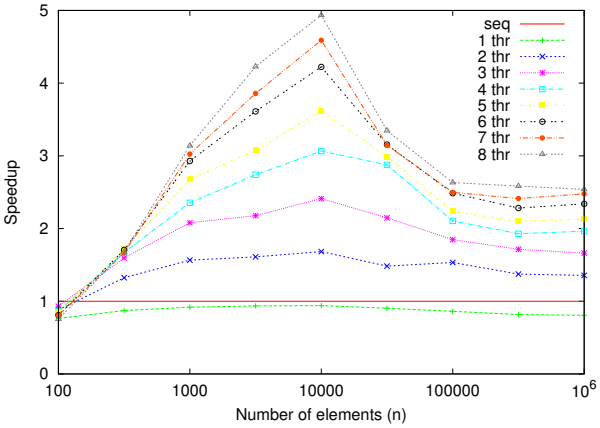

FIG. 4.6. *Speedup for* `accumulate` *using* SINGLEPASS *partitioning.*

m to 2, our algorithm behaves very well, converging to zero overhead. The average overhead ratio decreases exponentially with the input size n. Note that the (optimal) worst-case ratio achieved by the naïve algorithms also decreases exponentially, even faster.

4.2. Parallelization Using List Partitioning. After having evaluated the isolated performance for the list partitioning algorithms, we will investigate their impact in their intended domain, data-parallel algorithms.

Two interesting examples are `accumulate` and `list::sort` from the STL.
`accumulate` computes the sum (or some other reduction based on an associative binary operation) of a sequence of elements, starting with some initial value. Here, we consider the case of a sequence with forward iterators as input. `list::sort` stably sorts a doubly-linked list, whose size is not stored (in favor of fast splice operations). Its implementation is typically based on mergesort[1]. In the following, we first present parallel implementations using list partitioning for both algorithms. Then, we evaluate the impact in performance of different list partitioning algorithms.

Parallel Implementations. Parallelizing `accumulate` is straightforward. Let the desired number of threads be p. The list is partitioned into p parts, using one of the described algorithms. In parallel, each thread accumulates its part. After that, the p results are combined sequentially.

[1]Note that mergesort can be implemented without explicitly splitting the sequence in the middle by using a bottom-up approach.

For the list sort, we partition the list as above and split it into the respective sublists. These are sorted by one thread each, in parallel. Recursive tree-shaped merging combines the results in $\log_2 p$ steps, each merge per se being done sequentially.

Parameters for Testing. We have repeated our experiments at least 10 times and report the average running time.

For SinglePass we have used the parameters $o = 1$ and $m = 2$. Deduced from the results in Section 4.1, this choice produces a high quality partitioning at a low overhead in running time and additional space. The data type for `list::sort` is a 16-byte element containing an 8-byte integer key. The values are randomly generated. As use case for accumulate, we approximately multiply double-precision floating-point numbers by summing up their logarithms.

Setup. We have run these experiments on an dual-socket AMD Opteron 2350 (2×4 cores, 2.0 GHz, 2×2 MB L3 cache).

Results. Figures 4.3 and 4.4 show the speedup for parallel `list::sort` using the TraverseTwice and SinglePass list partitioning algorithms, respectively. The PointerArray variant is not compared here since it has a linear space overhead.

In both cases shown, the achieved speedups in absolute terms are not particularly good. This is probably mostly due to the bad collective memory bandwidth caused by the random accesses to traverse the more and more scattered sublists. Nonetheless, for a large number of elements, the SinglePass list partitioning algorithm is significantly better than TraverseTwice because the sequential fraction is sped up by a factor of 2, and the parts are of very similar size.

Figures 4.5 and 4.6 show the speedup for parallel `accumulate` using the TraverseTwice and SinglePass list partitioning algorithms, respectively.

In this case, the achieved speedups in absolute terms are better because summing the logarithm of floating-point numbers is quite compute-intensive. Again, the performance using SinglePass is much better than with TraverseTwice, in particular for a large number of elements.

Overall, for large inputs, SinglePass obtains much better speedups than TraverseTwice both for `list::sort` and `accumulate`.

5. Conclusions. We have presented a simple though non-trivial algorithm to solve the list partitioning problem using only one traversal and sub-linear additional space. Our algorithm divides a linearly traversable sequence of unknown length n in time $\Theta(n + \sigma p(n^{1-1/m} + \log n))$ using $O(\sigma p n^{1-1/m})$ additional space. σ and m are tuning parameters of the algorithm.

Our experiments have shown that our algorithm is very efficient in practice. It takes almost the same time as if the list was just traversed, without any processing. Besides, very high quality solutions can be obtained. The larger m, the better the quality, trading off memory. If $m = 1$, the worst-case overhead ratio 1 is divided by the oversampling factor σ. If $m > 1$, the worst-case overhead ratio decreases exponentially with n.

Therefore, for large input instances and in most practical situations, processing perfectly equal parts and almost equal parts should take about the same time, because the time to process each of the parts fluctuates in the same order of magnitude. In addition to this, our approach computes the partitioning twice as fast as the naïve approach.

As a result, using our list partitioning algorithm as a substep of parallel algorithms, the overall performance is significantly improved compared to a naïve implementation.

REFERENCES

[1] *The C++ Standard (ISO 14882:2003)*, John Wiley & Sons, Ltd, 2003.
[2] J. Singler, P. Sanders, and F. Putze, *The Multi-Core Standard Template Library*, in Euro-Par 2007: Parallel Processing, vol. 4641 of LNCS, Springer Verlag, pp. 682–694.

Edited by: Sabri Pllana, Siegfried Benkner
Received: June 16, 2008
Accepted: July 28, 2008