Single-Pass List Partitioning

Leonor Frias [1] Johannes Singler [2] Peter Sanders [2]

[1]Dep. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya

[2]Institut für Theoretische Informatik, Universität Karlsruhe

MuCoCos'08

Outline

1. Introduction
2. Problem Definition
3. The SinglePass Algorithm
4. Experiments
5. Conclusions
6. References

Motivation

Effectiveness of many parallel algorithms relies on partitioning the input into pieces.

Motivation

Effectiveness of many parallel algorithms relies on partitioning the input into pieces.

BUT most descriptions disregard how this is actually done (or just assume index calculations) ...

Motivation

Effectiveness of many parallel algorithms relies on partitioning the input into pieces.

BUT most descriptions disregard how this is actually done (or just assume index calculations) ...

ALTHOUGH there are common settings where the input cannot be partitioned so easily.

Motivation

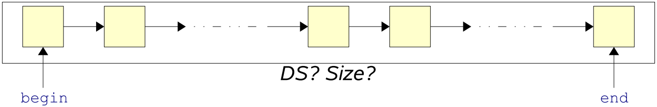
Effectiveness of many parallel algorithms relies on partitioning the input into pieces.

BUT most descriptions disregard how this is actually done (or just assume index calculations) ...

ALTHOUGH there are common settings where the input cannot be partitioned so easily.
Example: Sequences as input to algorithms in the Standard Template Library (STL), part of the C++ standard library.

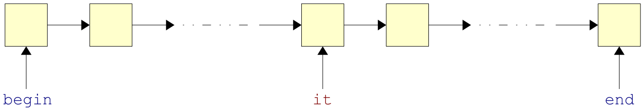
Algorithms in the STL

Input given using (*forward*) *iterators*, abstract from the underlying data structure.



Algorithms in the STL

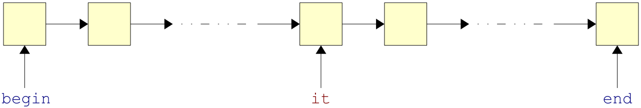
Input given using (*forward*) *iterator*s, abstract from the underlying data structure.



Operations on a *forward iterator* `it`:

Algorithms in the STL

Input given using (*forward*) *iterator*s, abstract from the underlying data structure.

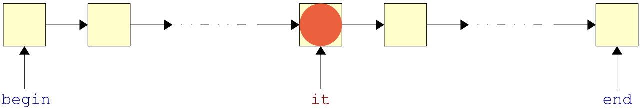


Operations on a *forward iterator* `it`:

- `*it`: Dereference.

Algorithms in the STL

Input given using (*forward*) *iterator*s, abstract from the underlying data structure.

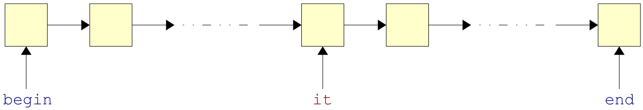


Operations on a *forward iterator* `it`:

- `*it`: Dereference.

Algorithms in the STL

Input given using (*forward*) *iterator*s, abstract from the underlying data structure.

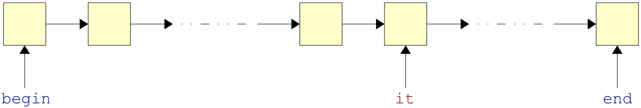


Operations on a *forward iterator* `it`:

- `*it`: Dereference.
- `++it`: Advance to next element.

Algorithms in the STL

Input given using (*forward*) *iterator*s, abstract from the underlying data structure.

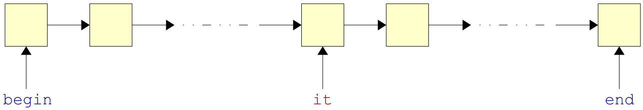


Operations on a *forward* *iterator* `it`:

- `*it`: Dereference.
- `++it`: Advance to next element.

Algorithms in the STL

Input given using (*forward*) *iterator*s, abstract from the underlying data structure.



Operations on a *forward iterator* `it`:

- `*it`: Dereference.
- `++it`: Advance to next element.

Forward sequence

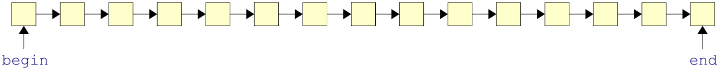
How to partition forward sequences or alike?

In compile-time:

1. The sequence is actually a random access sequence (e.g. an array)
    - More operations: `it + k`, `it - k`, `it2 - it1`, ...
    - Sequence length can be known in constant time

2. The sequence is not random access
    - Sequence length is unknown in constant time

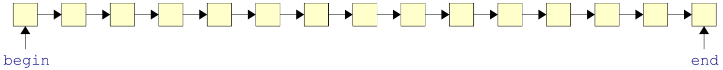
How to partition forward sequences or alike? (2)



Naïvely:

- TRAVERSETWICE
- POINTERARRAY

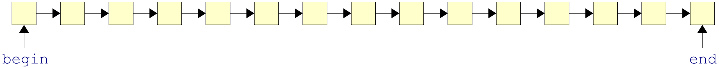
How to partition forward sequences or alike? (2)



Naïvely:

- TRAVERSETWICE
  1. Determine length (1st traversal)
  2. Partition (2nd traversal)
- POINTERARRAY

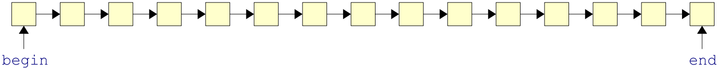
How to partition forward sequences or alike? (2)



Naïvely:

- TraverseTwice
- PointerArray
  1. Store pointers in a dynamic array (linear auxiliary memory)
  2. Trivial index calculation

How to partition forward sequences or alike? (2)

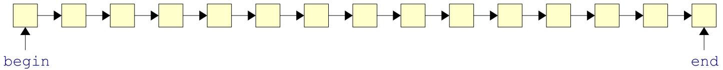


Naïvely:

- TRAVERSETWICE
- POINTERARRAY

Cannot this be done more efficiently?

How to partition forward sequences or alike? (2)



Naïvely:

- TRAVERSETWICE
- POINTERARRAY

Cannot this be done more efficiently?
Amdahl's law: speedup limited by the sequential portion.

Our contribution

An efficient sequential algorithm to divide *forward sequences*.

- Only one traversal
- Sub-linear additional space

List Partitioning problem

Given a *forward sequence*, divide it into p parts of almost equal length.

List Partitioning problem

Given a *forward sequence*, divide it into p parts of almost equal length.

Quality ratio r: $1 \leq \frac{|\text{longest part}|}{|\text{shortest part}|}$

r correlates to the efficiency of processing the parts in parallel (given that processing time is proportional to parts length)

List Partitioning problem

Given a *forward sequence*, divide it into p parts of almost equal length.

Quality ratio r: $1 \leq \frac{|\text{longest part}|}{|\text{shortest part}|} \leq R$

r correlates to the efficiency of processing the parts in parallel (given that processing time is proportional to parts length)

R: constant, depends only on a tuning parameter, namely the oversampling factor σ.

- $\sigma \in \mathbb{N} \setminus \{0\}$.

List Partitioning as an online problem

Only one element is given at a time, no global information.

List Partitioning as an online problem

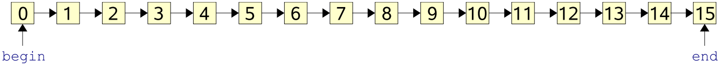
Only one element is given at a time, no global information.

Optimal offline algorithm: the difference in length between the parts is at most 1.
Quality ratio: $r_{\mathrm{OPT}} = \lceil n/p \rceil / \lfloor n/p \rfloor \overset{n \to \infty}{\longrightarrow} 1$.

Algorithm

Let $\sigma = 2$, $p = 3$

L:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

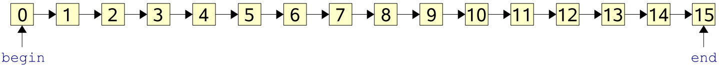
begin

end

Algorithm

Let $\sigma = 2,\ p = 3$

L:



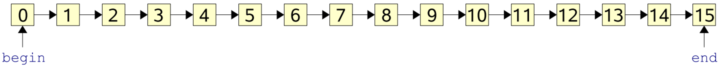
$k = 1,\ \ S = \{\}$

1. Initialization.

Algorithm

Let $\sigma = 2$, $p = 3$

L:



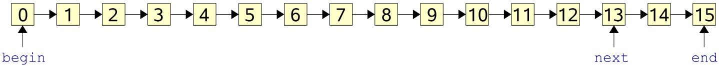
$k = 1, \;\; S = \{\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.

Algorithm

Let $\sigma = 2,\ p = 3$

L:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

begin next end

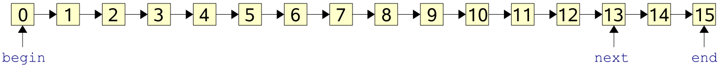
$k = 1, \quad S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:

Algorithm

Let $\sigma = 2,\ p = 3$

L:



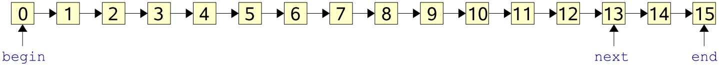
$k = 1, \quad S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:
    1. Merge each two consecutive subsequences into one. $S[0, 1, 2, 3, 4, 5, 6] := S[0, 2, 4, 6, 8, 10, 12]$

Algorithm

Let $\sigma = 2$, $p = 3$

L:



0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15

begin

next

end

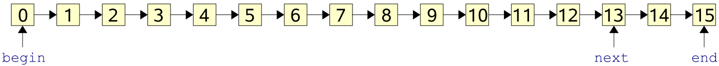
$k = 1$, $\quad S = \{0, 2, 4, 6, 8, 10, 12\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:
   1. Merge each two consecutive subsequences into one. $S[0, 1, 2, 3, 4, 5, 6] := S[0, 2, 4, 6, 8, 10, 12]$
   2. Let $k := 2k$.

Algorithm

Let $\sigma = 2,\ p = 3$

L:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

begin

next

end

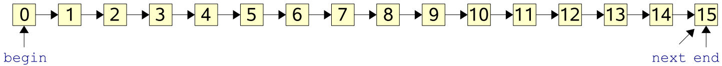
$k = 2, \quad S = \{0, 2, 4, 6, 8, 10, 12\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:
    1. Merge each two consecutive subsequences into one. $S[0, 1, 2, 3, 4, 5, 6] := S[0, 2, 4, 6, 8, 10, 12]$
    2. Let $k := 2k$.
    3. Iteratively append to S at most σp consecutive subsequences of length k from L.

Algorithm

Let $\sigma = 2$, $p = 3$

L:



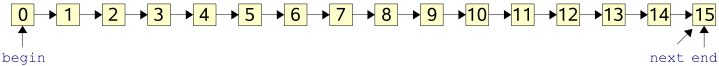
$k = 2$, $\quad S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:
    1. Merge each two consecutive subsequences into one. $S[0, 1, 2, 3, 4, 5, 6] := S[0, 2, 4, 6, 8, 10, 12]$
    2. Let $k := 2k$.
    3. Iteratively append to S at most σp consecutive subsequences of length k from L.

Algorithm

Let $\sigma = 2$, $p = 3$

L:

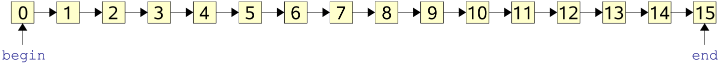


$k = 2$, $\quad S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

1. Initialization.
2. Iteratively append to S at most $2\sigma p$ 1-elem subsequences from L.
3. While L has more elements do:
    1. Merge each two consecutive subsequences into one.
       $S[0, 1, 2, 3, 4, 5, 6] := S[0, 2, 4, 6, 8, 10, 12]$
    2. Let $k := 2k$.
    3. Iteratively append to S at most σp consecutive subsequences of length k from L.
4. Merge the subsequences in S to obtain p subsequences.

Getting p subsequences of similar length

L:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

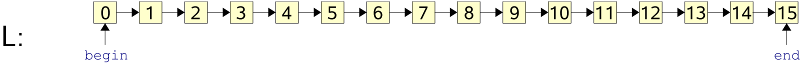
begin

end

$S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences *(s = 8)*

Getting p subsequences of similar length

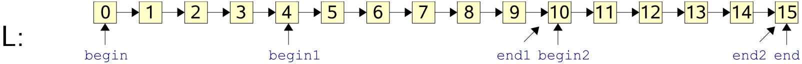


$S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences $(s = 8)$

$s \bmod p$ rightmost subsequences: *merge* $\lceil s/p \rceil$ subsequences

Getting *p* subsequences of similar length



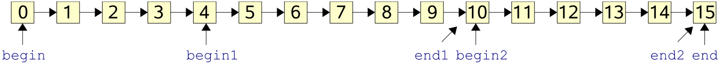
$S = \{0, 2, [4, 6, 8, [10), 12, 14, 15)\}$

At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences *($s = 8$)*

$s \bmod p$ rightmost subsequences: *merge* $\lceil s/p \rceil$ subsequences

Getting p subsequences of similar length

L:



$S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

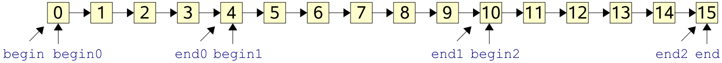
At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences $(s = 8)$

$s \bmod p$ rightmost subsequences: *merge* $\lceil s/p \rceil$ subsequences

$p - (s \bmod p)$ leftmost subsequences: *merge* $\lfloor s/p \rfloor$ subsequences

Getting p subsequences of similar length

L:



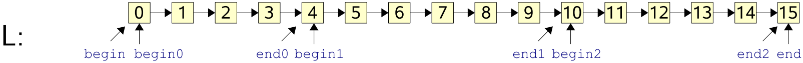
$$S = \{[0, 2, 4), 6, 8, 10, 12, 14, 15\}$$

At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences *($s = 8$)*

$s \bmod p$ rightmost subsequences: *merge* $\lceil s/p \rceil$ subsequences

$p - (s \bmod p)$ leftmost subsequences: *merge* $\lfloor s/p \rfloor$ subsequences

Getting p subsequences of similar length



$S = \{0, 2, 4, 6, 8, 10, 12, 14, 15\}$

At the beginning of step 4:
$\sigma p \leq s = |S| - 1 \leq 2\sigma p$ subsequences *($s = 8$)*

$s \bmod p$ rightmost subsequences: *merge* $\lceil s/p \rceil$ subsequences

$p - (s \bmod p)$ leftmost subsequences: *merge* $\lfloor s/p \rfloor$ subsequences

Special care with the last subsequence in S, which may be *not* full. The algorithm guarantees that two parts differ in length in at most in k elements.

Analysis

Auxiliary space (i.e. $|S|$): $\Theta(\sigma p)$

Time: $\Theta(n + \sigma p \log n)$.

- L traversal: $\Theta(n)$
- Step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

Analysis

Auxiliary space (i.e. $|S|$): $\Theta(\sigma p)$

Time: $\Theta(n + \sigma p \log n)$.

- L traversal: $\Theta(n)$
- Step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

Ratio:

- worst-case: r bounded by $\frac{\sigma+1}{\sigma}$.
- average: $\mathbb{E} r < \frac{1}{\sigma p} \sum_{\ell=\sigma p}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \approx 1 + \frac{1}{\sigma p} \left((p-1) \ln(2) \right)$

Analysis

Auxiliary space (i.e. $|S|$): $\Theta(\sigma p)$

Time: $\Theta(n + \sigma p \log n)$.

- L traversal: $\Theta(n)$
- Step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

Ratio:

- worst-case: r bounded by $\frac{\sigma+1}{\sigma}$.
- average: $\mathbb{E}r < \frac{1}{\sigma p} \sum_{\ell=\sigma p}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \approx 1 + \frac{1}{\sigma p}\left((p-1)\ln(2)\right)$

E. g. if $\sigma = 10$ and $p = 32$, then $r <= 1.1$ and $\mathbb{E}r < 1.07$

Generalization of the SINGLEPASS Algorithm

Performs *merge* steps only every m^{th} loop iteration.

In the remaining iterations, S is doubled in size, so that more subsequences can be added.

Generalization of the SINGLEPASS Algorithm

Performs *merge* steps only every m^{th} loop iteration.

In the remaining iterations, S is doubled in size, so that more subsequences can be added.

Thus, the total number of iterations is kept the same: $\Theta(\log n)$.

Generalization of the SINGLEPASS Algorithm

Performs *merge* steps only every m^{th} loop iteration.

In the remaining iterations, S is doubled in size, so that more subsequences can be added.

Thus, the total number of iterations is kept the same: $\Theta(\log n)$.

Equivalent to increasing the oversampling factor to σn^{γ} with $\gamma = 1 - 1/m$.

Analysis

$n^{\gamma} = \frac{n}{\sqrt[m]{n}} = \sqrt[m]{n^{m-1}}$
Auxiliary space (i.e. $|S|$): $O(\sigma p n^{\gamma})$.

Time: $\Theta(n + \sigma p(n^{\gamma} + \log n))$.

Analysis

$n^\gamma = \frac{n}{\sqrt[m]{n}} = \sqrt[m]{n^{m-1}}$

Auxiliary space (i.e. $|S|$): $O(\sigma p n^\gamma)$.

Time: $\Theta(n + \sigma p(n^\gamma + \log n))$.

Ratio:

$|\texttt{longest}| = (\sigma n^\gamma + 1)k$ $|\texttt{shortest}| = \sigma n^\gamma k$

$\frac{|\texttt{longest}|}{|\texttt{shortest}|} = 1 + \frac{1}{\sigma n^\gamma} = 1 + \frac{\sqrt[m]{n}}{\sigma n} \overset{n\to\infty}{\longrightarrow} 1$

Choosing m

The choice of m trades off time and space versus solution quality (better r as m larger).

Choosing m

The choice of m trades off time and space versus solution quality (better r as m larger).

Some interesting cases:

- $m = 1$: *merge* is performed each iteration → simple SINGLEPASS Algorithm
- $m = 2$: *merge* is performed once each two iterations
  - $n^{\gamma} = \sqrt{n}$
  - Auxiliary space: $O(\sigma p \sqrt{n})$
  - Time: $\Theta(n + \sigma p(\sqrt{n} + \log n))$.
  - Ratio: $1 + \frac{1}{\sqrt{n}}$

Choosing m

The choice of m trades off time and space versus solution quality (better r as m larger).

Some interesting cases:

- $m = 1$: *merge* is performed each iteration → simple SINGLEPASS Algorithm
- $m = 2$: *merge* is performed once each two iterations
  - $n^{\gamma} = \sqrt{n}$
  - Auxiliary space: $O(\sigma p \sqrt{n})$
  - Time: $\Theta(n + \sigma p(\sqrt{n} + \log n))$.
  - Ratio: $1 + \frac{1}{\sqrt{n}}$

$m = 2$ appears to be a good compromise.

Implementation

C++ implementation

Algorithms

- generalized SINGLEPASS
  - included in the MCSTL [SSP]
    MCSTL = Multicore STL, parallel implementation of the STL
- TRAVERSETWICE
- POINTERARRAY

Testing

Performance and quality results for $p = 4$.
Quality evaluated according the overhead $h = r - 1$.

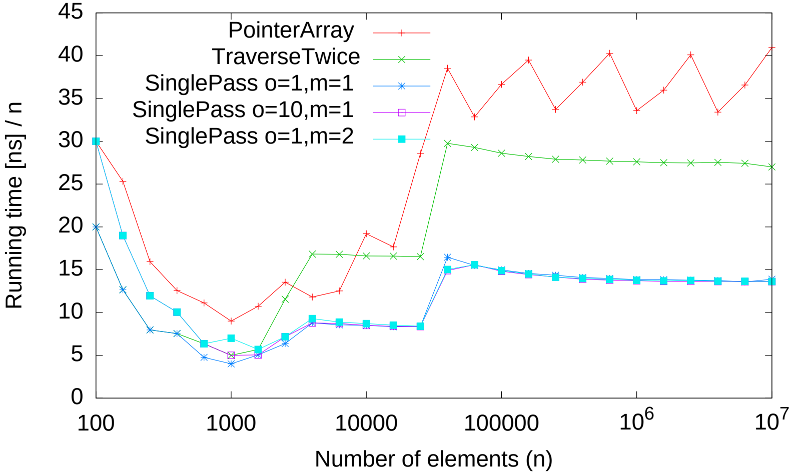
Setup

- AMD Opteron 270 (2.0 GHz, 1 MB L2 cache).
- GCC 4.2.0 + libstdc++, optimization (-O3).

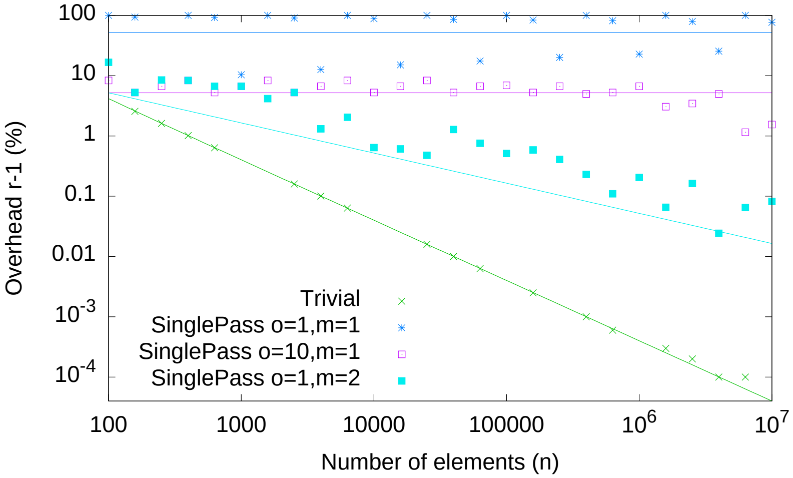
Parameters for SINGLEPASS

- $(o = 1,\ m = 1)$, $\Theta(p)$ space
- $(o = 10,\ m = 1)$, $\Theta(10p)$ space
- $(o = 1,\ m = 2)$, $\Theta(\sqrt{n}p)$

Time results



Quality results



Conclusions

We have solved the list partitioning problem using only one traversal and sub-linear additional space.

Conclusions

We have solved the list partitioning problem using only one traversal and sub-linear additional space.

Our experiments have shown that our algorithm is very efficient in practice.

Conclusions

We have solved the list partitioning problem using only one traversal and sub-linear additional space.

Our experiments have shown that our algorithm is very efficient in practice.

The larger m, the better the quality, trading off memory.
In the worst-case:

- $m = 1$: $h = 1/\sigma$
- $m > 1$: h decreases exponentially with n.

For large input instances and in most practical situations, no difference with optimally partitioned sequences.

Further reading

[SK08] describes some of the problems and challenges in parallelizing algorithms in the context of the C++ standard library.

References

Johannes Singler and Benjamin Kosnik.
The libstdc++ parallel mode: Software engineering considerations.
In *International Workshop on Multicore Software Engineering (IWMSE)*, 2008.

Johannes Singler, Peter Sanders, and Felix Putze.
The Multi-Core Standard Template Library.
In *Euro-Par 2007: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694. Springer Verlag.