Parallel In-Place Algorithms: Theory and Practice

Yan Gu, Omar Obeya, and Julian Shun

(Based on slides by Yan Gu)
You can put more memory on a machine, but they are expensive

Purchase price of RAX XT24-42S1 with 72 CPU cores (Xeon Gold 5220)

Rental price of AWS EC2 x1e-series multicore instances
Space-efficiency is crucial for shared-memory parallel algorithms

• Allows you to run larger inputs on your machine

• Decreases monetary costs

• Reducing memory footprint can improve performance due to lower memory traffic and better cache utilization
Parallel in-place algorithms have been gaining attention recently, but they are still underexplored.

- Duplicate removing [HL89]
- Merge and mergesort [GL91, GL92]
- Samplesort [ZCZ99, AWFS17]
- Search problems (backtrack and branch-and-bound) [PPSV15]
- Generating search tree layout [BCH'18]
- Radix sort [OKFS19]
- Partition [KW20]

Yet, there are no standard definition on what “parallel in-place” means.

Yet, there are no general approaches to designing parallel in-place algorithms.
1. Models for parallel in-place (PIP) algorithms
2. New PIP algorithms and a general approach
In-place in the sequential setting

Output

$O(1)$

$O(\log n)$

$O(\text{polylog}(n))$
But it doesn’t quite work in the parallel setting...

Input
But it doesn’t quite work in the parallel setting...

## Input

Limiting total auxiliary space → Limiting overall parallelism

Space-parallelism tradeoff in the in-place PRAM model [Langston93]
Can we achieve both?

• Can we get high parallelism?
  • Low span

• Can we achieve small auxiliary space?
  • Each processor should use a small auxiliary space, similar to the sequential setting (e.g., $O(\log n)$ words)

• Can we have clean computational models that capture both needs, but are still simple to use?
  • Need to decouple the analysis of auxiliary space and the analysis of span
The binary fork-join (work-span) model

• An algorithm is measured by work (number of operations) and span (length of longest sequential dependence)

• A fork instruction creates two subtasks that can be run in parallel

• After they finish, they join and continue
The binary fork-join (work-span) model

• Benefits of this model:
  • High-level, and algorithm designers need not to deal with system-level details such as load-balancing, task scheduling, synchronization, which are error-prone and can significantly complicate an algorithm
  • Algorithm design and analysis are independent of P (#processors)

• Can we design parallel in-place (PIP) algorithm also based on this model?
New models in this paper

• Strong PIP model
  • Achieve small (polylogarithmic) span and auxiliary space simultaneously

• Relaxed PIP model
  • Achieve sub-linear span and auxiliary space simultaneously

• Our models decouple the analysis between span and auxiliary space
  • Low span is useful in practice, not just for high parallelism, but also for reducing cache misses and global synchronization
The strong PIP model

• We assume:
  • The sequential execution uses $O(\log n)$-word auxiliary space in a stack-allocated fashion for an input size of $n$
  
• Stack-allocated fashion: when we allocate memory after a fork (or function call) it must be reclaimed before the associated join (or function return)

• A strong PIP algorithm uses $O(P \log n)$ total auxiliary space on $P$ processors using a randomized work-stealing scheduler (e.g., Cilk)
  • The “busy-leaves” property [BL99]
An strong PIP algorithm example

```java
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A+n/2, n-n/2);
    return L+R;
}
```

Work: $O(n)$
Span: $O(\log n)$

Sequential auxiliary space: $O(\log n)$

Total on P processors: $O(P \log n)$
The strong PIP model

• We assume:
  • The sequential execution uses \( O(\log n) \)-word auxiliary space in a stack-allocated fashion for an input size of \( n \)

• The strong PIP model is very restrictive
  • Does not allow for heap space
  • We do not have many work-efficient PIP algorithms in this model
The relaxed PIP model

• We assume:
  • The sequential execution uses $O(\log n)$-word auxiliary space in a stack-allocated fashion, and $O(n^\epsilon)$ shared (heap-allocated) auxiliary space ($0 < \epsilon < 1$)
  • Allows us to design many more work-efficient PIP algorithms
Outline of this talk

1. Models for parallel in-place (PIP) algorithms
2. New PIP algorithms and a general approach
## PIP algorithms

<table>
<thead>
<tr>
<th>Model</th>
<th>Problems</th>
<th>Work-efficient</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Permuting tree layout</td>
<td>✓</td>
<td>[6]</td>
</tr>
<tr>
<td></td>
<td>Reduce, rotating</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Strong PIP Model</td>
<td>Scan (prefix sum)</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>Filter, partition, quicksort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging, mergesort</td>
<td></td>
<td>✓</td>
<td>[11]</td>
</tr>
<tr>
<td>Set operations</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Relaxed PIP Model</td>
<td>Random permutation</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>List and tree contraction</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Merging, mergesort</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Filter, partition, quicksort</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>(Bi)Connectivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum spanning forest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [6]: Berney et al., IPDPS 2019
- [11]: Blelloch, Ferizovic, Sun, SPAA 2016

*: main contribution
General approach for relaxed PIP algorithms

• Decomposable Property

Auxiliary space used is bounded by auxiliary space for sub-problem

Provide a tradeoff between space and parallelism
Relaxed PIP algorithm design using the Decomposable Property

• Suppose that there is an algorithm satisfying the decomposable property with work $W(n) = O(n \text{ polylog}(n))$ and $O(\text{polylog}(n))$ span. Then, there is a relaxed PIP algorithm for the same problem with $W(n)$ work, $O(n^\epsilon \text{polylog}(n))$ span, and $O(n^{1-\epsilon})$ auxiliary space for some $0 < \epsilon < 1$. 
Random Permutation as an example

\[ \text{\textbf{KNUTHSHUFFLE}}(A, H) \]
\[
\textbf{for} \ i \leftarrow n \ \textbf{to} \ 1 \ \textbf{do} \\
\text{swap}(A[i], A[H[i]])
\]

This algorithm can be parallelized \cite{SGB+15}, with \( O(n) \) work and \( O(\log n) \) span w.h.p. \cite{BFGS20}

However, the amount of auxiliary space is \( O(n) \), for data structures to resolve conflicts

\[ H[i] \] is randomly drawn between 1 and \( i \)

This serial algorithm is in-place
Decomposable property

**KNUTHSHUFFLE**(A, H)

for \( i \leftarrow n \) to 1 do

swap(A[i], A[H[i]])

<table>
<thead>
<tr>
<th>Iterate</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H = )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>( A = )</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>


Decomposable property

\textbf{KNUTH\textsc{Shuffle}(A, H)}

\textbf{for } i \leftarrow n \textbf{ to } 1 \textbf{ do}

\textbf{swap}(A[i], A[H[i]])

Iterate  1  2  3  4  5  6  7  8
\begin{tabular}{c}
\hline
$H =$ \\
\hline
1 & 1 & 2 & 4 & 2 & 3 & 4 & 2 \\
\hline
\end{tabular}

\begin{tabular}{c}
\hline
$A =$ \\
\hline
a & b & c & d & e & f & g & h \\
\hline
\end{tabular}

Work on the second half first
Decomposable property

\begin{algorithm}
\textbf{Knut\textit{h}Shuffle}(A, H)
\begin{algorithmic}
\For{$i \leftarrow n \text{ to } 1$}
\State $\text{swap}(A[i], A[H[i]])$
\EndFor
\end{algorithmic}
\end{algorithm}

Iterate 1 2 3 4 5 6 7 8
\begin{array}{cccccccc}
H = & 1 & 1 & 2 & 4 & 2 & 3 & 4 & 2 \\
A = & a & e & f & g & h & c & d & b
\end{array}

Work on the second half first
Decomposable property

**KnuthShuffle**($A$, $H$)

for $i \leftarrow n$ to 1 do
    swap($A[i]$, $A[H[i]]$)

Iterate 1 2 3 4 5 6 7 8

$H = \begin{array}{cccccccc}
1 & 1 & 2 & 4 & 2 & 3 & 4 & 2 \\
\end{array}$

$A = \begin{array}{cccccccc}
a & e & f & g & h & c & d & b \\
\end{array}$

Then work on the first half
**Decomposable property**

\[ \text{KNUTH\_SHUFFLE}(A, H) \]

\[ \text{for } i \leftarrow n \text{ to } 1 \text{ do} \]

\[ \text{swap}(A[i], A[H[i]]) \]

<table>
<thead>
<tr>
<th>Iterate</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H = )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

\( A = \)

- \( a \)
- \( b \)
- \( c \)
- \( d \)
- \( e \)
- \( f \)
- \( g \)
- \( h \)
Decomposable property

**KNUTHSHUFFLE**($A, H$)

```markdown
for $i \leftarrow n$ to 1 do
    swap($A[i], A[H[i]]$)
```

Work on $k$ elements per batch, for a total of $n/k$ rounds

Only needs $O(k)$ auxiliary space for resolving conflicts per round

This gives an $O(n)$ work relaxed PIP algorithm for random permutation, with sublinear span and space
Experiment setup

• 72-core Dell PowerEdge R930 (with two-way hyper-threading) and 1TB of main memory

• Implemented using Cilk Plus

• Comparing to Problem Based Benchmark Suite (PBBS), containing state-of-the-art multicore implementations
Our PIP algorithms are competitive with or faster than the best non-in-place versions, mainly due to a smaller memory footprint and fewer memory accesses.
Our PIP algorithms have good scalability with respect to input size and thread counts, similar to the best non-in-place parallel algorithms.
Space Usage

• The PBBS algorithms are not in-place, and require auxiliary space linear in the input size
• Memory overhead of our PIP algorithms:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input size (MB)</th>
<th>Memory usage (MB)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>7629.4</td>
<td>7636.2</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>Filter</td>
<td>7629.4</td>
<td>7636.9</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>Random permutation</td>
<td>762.9</td>
<td>791.2</td>
<td>3.7%</td>
</tr>
<tr>
<td>List contraction</td>
<td>762.9</td>
<td>773.5</td>
<td>1.4%</td>
</tr>
<tr>
<td>Tree contraction</td>
<td>1144.4</td>
<td>1154.9</td>
<td>0.9%</td>
</tr>
</tbody>
</table>
Models for parallel in-place (PIP) algorithms

- Strong and relaxed PIP models, based on the binary fork-join model
- Decouples the analysis between parallelism and auxiliary space, and leads to practical algorithms

New PIP algorithms and a general approach

- Decomposable property: convert a non-PIP algorithm to relaxed PIP
- New PIP algorithms for scan, filter, sort, merge, random permutation, list and tree contraction, (bi)connectivity, minimum spanning forest
- Competitive with or faster than state-of-the-art in practice