In place shared-memory sorting algorithms

Chris Rinard
Quicksort History:

Invented in 1951 by Tony Hoare

Architecture of the time is in a museum now
TL;DR

- This paper presents IPS\(^4\)O: a parallel, in-place version of samplesort.
- At the time of writing, Quicksort and variants are the predominantly used sorting alg.

“You have to outperform quicksort in every respect in order to replace it.”
Improvements to quicksort

- Strictly in-place
- 2-3 pivots (20% better than single pivot)
- Parallel Quicksort (Tsigas, Zhang)
- Samplesort
Quicksort review

1: Choose Pivot
Quicksort review

1: Choose Pivot

2: Put pivot at its correct sorted position, all smaller elements before pivot, and all greater elements after pivot
Quicksort review

3: Quicksort the smaller and larger elements (left and right)
Samplesort

Basic idea: k-way Quicksort

3 Phases + recursion

1. Sampling
2. Classification
3. Distribution
4. Recurse
Sampling

1. Sample a * k - 1 randomly sampled inputs into array S.
2. Sort S
3. Pick splitters $s_0...s_{k-2}$ from S
Classification

1. For each element, find bucket index, and keep track of bucket size ($e$ in $b_i$ if $s_{i-1} < e \leq s_i$).
2. Classify each element of the input into correct bucket
3. Find memory locations of boundaries
Distribution

1. Copy elements from input array into buckets.
IPS$^4$O

4 Stages + recursion:

1. Sampling: bucket boundaries
2. Classification: Group input into blocks (in block, every elem in same bucket)
3. Permutation: Globally order blocks
4. Cleanup: Clean up partially filled or crossing blocks
<table>
<thead>
<tr>
<th>11</th>
<th>13</th>
<th>11</th>
<th>17</th>
<th>10</th>
<th>11</th>
<th>9</th>
<th>3</th>
<th>18</th>
<th>4</th>
<th>7</th>
<th>18</th>
<th>19</th>
<th>3</th>
</tr>
</thead>
</table>

**IPS₄O Sampling phase**
IPS$^4$O Sampling phase

K = 3, $\alpha$=2, $k\alpha$ - 1 elements

<table>
<thead>
<tr>
<th>11</th>
<th>13</th>
<th>11</th>
<th>17</th>
<th>10</th>
<th>11</th>
<th>9</th>
<th>3</th>
<th>18</th>
<th>4</th>
<th>7</th>
<th>18</th>
<th>19</th>
<th>3</th>
</tr>
</thead>
</table>

[Table showing the sampling process with highlighted elements]
### IPS$^4$O Sampling phase

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>13</td>
<td>11</td>
<td>17</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>18</td>
<td>4</td>
<td>7</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

$K = 3, \alpha=2, k\alpha - 1$ elements

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>13</td>
<td>11</td>
<td>17</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>18</td>
<td>4</td>
<td>7</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>10</td>
<td>17</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>4</td>
<td>11</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>
### IPS$^4$O Sampling phase

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>13</td>
<td>11</td>
<td>17</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>18</td>
<td>4</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$K = 3$, $\alpha = 2$, $k\alpha - 1$ elements

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>13</td>
<td>11</td>
<td>17</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>18</td>
<td>4</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>17</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>4</td>
<td>11</td>
<td>18</td>
<td>19</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### IPS$^4$O Sampling phase

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3 | 7 | 10 | 11 | 11 | 17 | 9 | 13 | 18 | 4 | 11 | 18 | 19 | 3 |

K = 3, α=2, kα - 1 elements

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3 | 7 | 10 | 11 | 11 | 17 | 9 | 13 | 18 | 4 | 11 | 18 | 19 | 3 |

K = 3, k - 1 splitters (picked equidistantly)
IPS$^4$O Sampling

$K = 3, k - 1$ splitters (picked equidistantly)

Create branchless decision tree, $k$ buckets
Performance Hack: IPS$^4$O bucket structure (branchless decision tree)

- Eliminates branch mispredictions: use of $a = (<>)?b:c$, easy to store

```plaintext
t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle
\quad //
f or i := 1 to n do  // locate each element
  j := 1  // current tree node := root
  repeat log k times  // will be unrolled
    j := 2j + (a_i > t_j)  // left or right?
  j := j - k + 1  // bucket index
  |b_j|++  // count bucket size
  o(i) := j  // remember oracle
```
Performance Hack: IPS$^4$O bucket structure (branchless decision tree)

- Eliminates branch mispredictions: use of $a = (<>)? b : c$, easy to store
- Better than this, you can unroll the loop

```plaintext
t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle \quad //
for i := 1 to n do  // locate each element
    j := 1 \quad // current tree node := root
    repeat \log k \times \quad // will be unrolled
        j := 2j + (a_i > t_j) \quad // left or right?
    j := j - k + 1 \quad // bucket index
    |b_j|++ \quad // count bucket size
    o(i) := j \quad // remember oracle
```
Performance Hack: IPS\textsuperscript{4}O bucket structure (branchless decision tree)

- Eliminates branch mispredictions: use of \( a = (<>)? b : c \), easy to store
- Better than this, you can unroll this loop
- In practice, authors note “up to 2x faster than std::sort”

```
t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle
\text{for } i := 1 \text{ to } n \text{ do } \text{ // locate each element}
  j := 1 \text{ // current tree node := root}
  \text{repeat log } k \text{ times } \text{ // will be unrolled}
    j := 2j + (a_i > t_j) \text{ // left or right?}
  j := j - k + 1 \text{ // bucket index}
  |b_j|++ \text{ // count bucket size}
  o(i) := j \text{ // remember oracle}
```
IPS$^4$O

4 Stages:

1. **Sampling**: bucket boundaries
2. **Classification**: Group input into blocks (in block, every elem in same bucket)
3. **Permutation**: Globally order blocks
4. **Cleanup**: Clean up partially filled or crossing blocks
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>17</td>
<td>4</td>
<td>13</td>
<td>18</td>
<td>4</td>
<td>11</td>
<td>18</td>
<td>19</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

| 3 | 7 | 10 | 11 | 11 | 17 | 4 |
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k=3$, each thread has $k$ “buffer blocks”
IPS\textsuperscript{4}O Classification

\( t = 2 \), split into \( t \) “stripes”

\( k = 3 \), each thread has \( k \) “buffer blocks”

Block size limited -- for this case = 2
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k=3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case = 2
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k = 3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case = 2
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k = 3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case $= 2$
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k=3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case $= 2$
IPS\textsuperscript{4}O Classification

\( t = 2 \), split into \( t \) “stripes”

\[ \begin{array}{ccc}
2 & 3 & 0 \\
3,7 & 11 & \text{ } \\
7 & 11 & \\
10 & 11 & 17 & 4
\end{array} \]

\( k = 3 \), each thread has \( k \) “buffer blocks”
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k = 3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case $= 2$
IPS$^4$O Classification

$t = 2$, split into $t$ “stripes”

$k=3$, each thread has $k$ “buffer blocks”

Block size limited -- for this case = 2
IPS$^4$O

4 Stages:

1. Sampling: bucket boundaries
2. Classification: Group input into blocks (in block, every element in same bucket)
3. Permutation: Globally order blocks
4. Cleanup: Clean up partially filled or crossing blocks
IPS$^4$O Block Permutation

Buffer Blocks

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>17</td>
</tr>
</tbody>
</table>

Memory blocks

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
IPS\textsuperscript{4}O Block Permutation

Buffer Blocks

Memory blocks

Goal:

How do I find these?
IPS$^4$O Block Permutation

Buffer Blocks

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>17</td>
</tr>
</tbody>
</table>

Memory blocks

| 10 | 11 |  3 |  7 | 13 | 18 |

Goal:

|  3 |  3 |  4 |  7 |  9 | 10 | 11 | 11 | 11 | 13 | 18 | 11 | 11 |

How do I find these? Prefix-sum!
IPS\textsuperscript{4}O Block Permutation

Memory blocks

\begin{itemize}
  \item \text{write}_0
  \item \text{read}_0 \text{ write}_1
  \item \text{write}_2
  \item \text{read}_1 \text{ read}_2
\end{itemize}

10 11 3 7 13 18

Thread 0 (primary bucket 0) Thread 1 (primary bucket 1)

4 9 17 4,3 11 18,19
IPS\textsuperscript{4}O Block Permutation

Memory blocks

write\textsubscript{0}  read\textsubscript{0}  write\textsubscript{1}  read\textsubscript{1}  write\textsubscript{2}  read\textsubscript{2}

10  11  

write
read
write
read
write
read

Dest bucket = 0
3  7

Dest bucket = 2
13  18

Thread 0 (primary bucket 0)  Thread 1 (primary bucket 1)

4  9  17  4,3  11  18,19
IPS$^4$O Block Permutation

Memory blocks

Thread 0 (primary bucket 0)  Thread 1 (primary bucket 1)

3 7  
4 9 17

13 18

4,3 11 18,19
IPS$^4$O Block Permutation

Memory blocks

write$_0$

read$_0$

write$_1$

read$_1$

write$_2$

read$_2$

Thread 0 (primary bucket 0)  Thread 1 (primary bucket 1)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,3</td>
<td>11</td>
<td>18,19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IPS$^4$O Block Permutation

Memory blocks

Thread 0 (primary bucket 0)  Thread 1 (primary bucket 1)

4 9 17  4,3 11 18,19

write$_0$  read$_0$  write$_1$  read$_1$  write$_2$  read$_2$
IPS$^4$O Block Permutation

Memory blocks

write$_0$
read$_0$
write$_1$
read$_1$
write$_2$
read$_2$

3 7
thread 0 (primary bucket 0)

10 11
thread 1 (primary bucket 1)

13 18

B_dest = 1

4 9 17

4,3 11 18,19
IPS\textsuperscript{4}O Block Permutation

Memory blocks

Thread 0 (primary bucket 0)

\[ B_{\text{dest}} = 1 \]

\begin{align*}
4 & \quad 9 & \quad 17
\end{align*}

Thread 1 (primary bucket 2)

\begin{align*}
4,3 & \quad 11 & \quad 18,19
\end{align*}
IPS$^4$O Block Permutation

Memory blocks

Thread 1 (done)

Thread 2 (done)

B_dest = 1

write$^0_0$

read$^0_0$

write$^1_1$

read$^1_1$

read$^2_2$

write$^2_2$
IPS\(^4\)O Block Permutation

Buffer Blocks

| 3 | 9 | 17 |

Memory blocks

3 7 10 11 13 18
IPS$^4$O

4 Stages:

1. **Sampling**: bucket boundaries
2. **Classification**: Group input into blocks (in block, every elem in same bucket)
3. **Permutation**: Globally order blocks
4. **Cleanup**: Clean up partially filled or crossing blocks
IPS$^4$O Block Permutation

Buffer Blocks

1 2 3
3 9 17

Memory blocks

7 10 11 11 13 18

What's wrong with this array? (Yes this is a question)
IPS\textsuperscript{4}O Block Permutation

Buffer Blocks

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>17</td>
</tr>
</tbody>
</table>

Memory blocks

| 7 | 10 |   |   | 11 | 11 |   |   | 13 | 18 |   |

1. Bucket overlap
2. Partially filled buffers
3. Last bucket can be in swap buffer
IPS^4O Cleanup

1. Bucket overlap
2. Partially filled buffers
3. Last bucket can be in swap buffer
Recursion structure
Performance Hack: Implementation of pointer arithmetic

128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS?
Performance Hack: Implementation of pointer arithmetic

128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS -- Read and write stored in 64-bit pointers, must be updated together

“The measurements reported in this paper were performed using somewhat non-portable implementations that use a 128-bit compare-and-swap instruction specific to x86 architectures (see also Section 6). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.”
Performance Hack: Implementation of pointer arithmetic

128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS -- Read and write stored in 64-bit pointers, must be updated together

“The measurements reported in this paper were performed using somewhat non-portable implementations that use a 128-bit compare-and-swap instruction specific to x86 architectures (see also Section 6). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.”
Performance Hack?: Implementation of pointer arithmetic -- does this matter?
Performance Hack?: Implementation of pointer arithmetic -- does this matter: It depends
Performance and Portability Bugs

Try it yourself: https://github.com/ips4o/ips4o-benchmark-suite

“For the run.sh command, you need an installation of the Intel® Integrated Performance Primitives (IPP) as well as Cilk Plus. For Cilk Plus, you require a compiler supporting the Cilk Plus C++ language extension or you need provide your own Cilk Plus library which you add to the CMakeLists.txt file.”
Summary: What it takes to publish a paper on sorting these days

1. Incremental improvement on algorithm
2. Portable
3. 30 pages of analysis
4. Involved runtime analysis
5. Write your own scheduler
6. I/O analysis
7. Branch mispredict analysis
8. Base case optimization