A Modified Parallel Heapsort Algorithm

Hiroaki Hirata, Kyoto Institute of Technology, Japan
Atsushi Nunome, Kyoto Institute of Technology, Japan

https://orcid.org/0000-0003-1911-3075

ABSTRACT

Sorting is a fundamental and essential problem required in the wide range of application fields, and so many sorting algorithms have been developed. Among those algorithms, heapsort is one of the most elegant and efficient sorting algorithms. But no parallel heapsort algorithm had been presented until the authors developed a restricted parallel algorithm a few years ago. This parallel algorithm had a restriction which makes it difficult to be used universally for general data sets. So, in this article, the authors present a modified parallel algorithm which is free from such restriction and can be used for any data set. This new algorithm can achieve almost the same performance as the restricted algorithm the authors developed before.

KEYWORDS

Heapsort, Parallel Algorithm, Parallelization, Sorting Algorithm, Thread-Level Parallel Processing, Thread-Level Speculation

INTRODUCTION

Sorting is a fundamental and essential problem required in the wide range of application fields, and so many sorting algorithms have been developed since the dawn of the computer era. Among those algorithms, heapsort (Williams, 1964) is one of the most elegant and efficient sorting algorithms. Theoretically its time complexity is \( O(n \log n) \) in the average case as same as quicksort or Merge sort. Since heapsort's worst-case complexity is also \( O(n \log n) \), it performs better than quicksort, whose worst-case complexity is \( O(n^2) \). On the other hand, heapsort has the advantage in less amount of auxiliary memory than Merge sort. However, there had not yet been a parallel heapsort algorithm found for many decades since the heapsort algorithm was found, in contrast with Merge sort, which is parallelizable.

Recently when the authors tried to shorten a time for constructing a k-dimensional tree (Bentley, 1975), they also challenged the development of a parallel heapsort algorithm as its sub-problem (Yamasaki, 2018). They succeeded to shorten the sorting time, but their parallel algorithm had a restriction which makes it difficult to be used universally for general data sets. The paper (Kitano, 2019) explored the removal of such restriction, and this article improves the idea in the paper to develop a new parallel heapsort algorithm and evaluates its performance on a real existing machine.

DOI: 10.4018/IJSI.2020070101

Copyright © 2020, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
HEAPSORT ALGORITHM

Figure 1 shows the C language-like pseudo-code for heapsort. In its first stage, the algorithm builds a heap tree bottom-up (at lines 2 and 3 of Figure 1). A heap tree is a half-ordered binary tree whose elements are stored in a linear array. Assume that \( n \) values are stored in elements of an array: \( a[1], a[2], \ldots, a[n] \). As shown in Figure 2, \( a[2k] \) and \( a[2k+1] \) are child nodes of \( a[k] \). In heap trees, the following condition must be satisfied: the value of a node is greater than or equal to the values of its child nodes. Which child node has the smaller value is not defined.

The procedure DownHeap\((k,m)\) of Figure 1 moves the value of the node \( a[k] \) downward in the tree that consists of \( a[1], a[2], \ldots, a[k], \ldots, a[m] \) so that the above condition of heap trees can be satisfied. In the example of Figure 3, the value 16 stored in \( a[k] \) is moved to nodes closer to leaves of the tree. The values 30, 24, and 17 (which are larger than 16) are shifted up toward the root of the tree. By repeating this “downheap” operation, from nodes whose children are the leaf nodes to the root node, a heap tree can be built.

In the second stage of heapsort (at lines 4-7 of Figure 1), the algorithm performs the following steps:

**Step 1:** It removes the last element of the array (\( a[n] \)) from the heap tree, and so the remaining heap tree consists of \( a[1], \ldots, a[n-1] \). It also moves the content of \( a[n] \) to temporary variable temp.

**Step 2:** In the heap tree, the largest value is stored in the root node (\( a[1] \)). So the algorithm moves it to the last element of the array (\( a[n] \)).

**Step 3:** Now the root node (\( a[1] \)) is empty. The algorithm restores the content of the variable temp to \( a[1] \).

Figure 2. Heap tree

```
int HeapSort() {
    for (i = n/2; i > 0; i--)
        DownHeap(i, n);
    while (n > 1) {
        tmp = a[n]; a[n] = a[1]; a[1] = tmp;
        DownHeap(1, --n);
    }
}
```
Related Content

Modelling Information Demand in an Enterprise Context: Method, Notation, and Lessons Learned
[www.igi-global.com/chapter/modelling-information-demand-enterprise-context/76619?camid=4v1a](www.igi-global.com/chapter/modelling-information-demand-enterprise-context/76619?camid=4v1a)

Katana: Towards Patching as a Runtime Part of the Compiler-Linker-Loader Toolchain
[www.igi-global.com/article/katana-towards-patching-runtime-part/46149?camid=4v1a](www.igi-global.com/article/katana-towards-patching-runtime-part/46149?camid=4v1a)
Strategies for Static Tables
www.igi-global.com/chapter/strategies-static-tables/21068?camid=4v1a

The Impact of Ideology on the Organizational Adoption of Open Source Software
www.igi-global.com/chapter/impact-ideology-organizational-adoptions-open/29482?camid=4v1a