南開大學
Nankai University

南开大学计算机学院

《并行程序设计》期末报告

学　　号：2212338

姓　　名：刘羽浩

年　　级：2022 级

专　　业：计算机科学与技术

授课教师：王刚 教授

课程助教：

完成日期：2024 年 6 月 10 日

摘要

近年来，数据量的爆炸性增长和对快速数据处理的需求凸显了传统顺序排序算法的局限性。并行排序算法，特别是并行归并排序，通过利用现代多核和多处理器架构的计算能力，有效解决了这些挑战。本文深入分析了各种并行归并排序算法，重点关注数据分配、通信开销、内存利用和可扩展性等关键领域。通过对现有文献和方法的批判性审查，我们识别了能够提升性能和效率的优化策略。研究结果表明，高效的数据分配和合并策略、减少的内存开销以及改进的可扩展性可以显著提高并行归并排序算法的效果。本研究旨在为开发能够满足当代数据密集型应用需求的健壮的并行排序算法做出贡献。

关键词：并行排序, 归并排序, 多核架构, 数据分配, 内存利用, 可扩展性, 性能优化, 通信开销

Abstract

In recent years, the explosive growth in data volume and the demand for rapid data processing have underscored the limitations of traditional sequential sorting algorithms. Parallel sorting algorithms, particularly parallel merge sort, have emerged as effective solutions to these challenges by leveraging the computational power of modern multicore and multiprocessor architectures. This paper provides an in-depth analysis of various parallel merge sort algorithms, focusing on key areas such as data distribution, communication overhead, memory utilization, and scalability. Through a critical review of existing literature and methodologies, we identify optimization strategies that enhance performance and efficiency. Our findings suggest that efficient data distribution and merge strategies, combined with reduced memory overhead and improved scalability, can significantly boost the effectiveness of parallel merge sort algorithms. This study aims to contribute to the development of robust parallel sorting algorithms that can meet the demands of contemporary data-intensive applications.

Keywords: Parallel Sorting, Merge Sort, Multicore Architectures, Data Distribution, Memory Utilization, Scalability, Performance Optimization, Communication Overhead

目录

第一章　引言

在计算机科学领域中，排序算法因其在数据处理、搜索和优化任务中的广泛应用而具有重要地位。随着数据量和快速处理需求的指数增长，传统的顺序排序算法往往无法满足现代应用的性能要求。这就需要开发和实现并行排序算法，利用当代多核和多处理器架构的能力，达到显著的性能提升。

并行排序算法，特别是基于分而治之策略的算法，已证明在高效处理大数据集方面具有相当大的潜力。在这些算法中，并行归并排序算法因其固有的将数据分割成可管理块、并行排序和高效合并结果的能力而脱颖而出。尽管具有诸多优势，并行归并排序算法仍面临数据分配、通信开销和内存利用等方面的挑战，这些因素可能影响其整体性能和可扩展性。

本文旨在深入探讨并行归并排序算法的复杂性，研究在不同并行架构上提升其性能的各种策略和方法。通过对现有研究的批判性分析，我们力图识别改进的关键领域，并提出优化方案，以进一步提高并行归并排序算法的效率和可扩展性。我们的研究主要关注三个方面：高效的数据分配和合并策略、存储效率和内存利用以及算法的可扩展性和灵活性。

首先，我们探讨了优化数据分配和合并策略的方法，确保处理器之间的工作负载均衡，并将通信开销降至最低。接下来，我们解决了存储效率的问题，强调了在保持高性能的同时减少内存开销的算法。最后，我们研究了并行归并排序算法的可扩展性和灵活性，特别是它们对不同数据大小和处理器数量的适应性。

通过这一全面的分析，我们旨在为设计和实现能够满足现代数据密集型应用需求的健壮的并行归并排序算法提供一定的见解。通过推进对这些算法的理解，我们为并行计算及其在高效数据处理中的应用做出贡献。

本实验代码托管在

第二章　相关研究文献综述

在并行排序算法的背景下，许多策略和方法被探索以提高性能、效率和可扩展性，适用于各种并行架构。本文献综述批判性地审查了几篇关于并行归并排序及相关排序算法的研究论文，重点是具体的研究方向、方法和结果。

2.1　并行归并排序的性能优化

研究重点：高效的数据分配和归并策略

几篇论文探讨了通过优化数据分配和归并策略来提高并行归并排序算法性能的不同方面。

高效的多处理器归并算法

Varman 等人（1990）提出了一种多处理器归并方法，旨在平衡处理器间的工作负载。其关键创新在于高效的数据重新分配策略，最小化通信开销并确保负载平衡[1]。该方法采用递归分而治之策略，每个处理器处理一部分数据，合并后将结果向上传递，层级结构的通信模式显著提高了性能。

并行归并排序的自适应框架

Khan 和 Tiwari（2012）提出了一个自适应框架，使用消息传递接口（MPI）分析并行归并排序在松散耦合架构上的性能。研究强调了 MPI 在促进进程间通信和同步方面的有效性。通过利用 MPI 的功能，该框架确保了高效的数据分配和同步，从而提高了分布式计算系统的性能[2]。

2.2　存储效率和内存利用率

研究重点：减少并行排序中的内存开销

有效的内存利用和存储效率对于处理大型数据集的并行排序算法至关重要。几项研究通过提出最小化额外内存需求的算法来解决这些问题。

存储高效的并行排序算法

Brent 和 Tridgell（1993）提出了一种为 MIMD（多指令多数据）架构设计的存储高效的并行排序算法。该算法结合了节点内的串行排序和两阶段并行归并，仅需与每个节点中元素数量平方根成比例的额外存储。这种方法确保了高内存利用率和可扩展性，使其适用于并行机器上的大规模排序任务[3]。

缓存意识的并行排序算法

Liu 和 Li 的研究提出了一种基于分区归并的缓存意识并行排序算法，适用于共享缓存的多核处理器（CMP）。该算法通过以缓存友好的方式分区数据，并执行考虑缓存层次结构的并行归并，显著减少了缓存未命中率，从而提高了整体排序性能[4]。

2.3 可扩展性和灵活性

研究重点：增强算法在不同架构上的可扩展性

可扩展性和灵活性对于并行排序算法有效处理不同数据规模和处理器数量至关重要。该领域的研究重点在于开发能够在不同并行架构上保持性能的算法。

完全灵活的并行归并排序

Marszałek 等人（2018）提出了一种完全灵活的并行归并排序算法，适用于多核架构。该算法动态适应可用核心数量和数据规模，确保在不同配置下的最佳性能。通过利用细粒度并行性和动态负载平衡，该算法实现了高可扩展性和灵活性[5]。

适用于可变长度键的高效并行归并排序

另一项重要贡献是为排序固定和可变长度键开发的算法，解决了处理不同数据类型的挑战。该研究强调了自适应排序技术的重要性，这些技术能够高效管理不同键长度，同时保持高性能和低内存开销[6]。

2.4 算法效率和加速

研究重点：实现高速和高效的并行排序

实现高速排序且计算开销最小是并行排序算法设计的主要目标。该领域的研究提出了各种策略以提高排序过程的速度和效率。

快速、存储高效的并行排序

Brent 和 Tridgell（1993）的研究强调了一种快速并行排序算法，该算法注重速度和存储效率。通过优化归并交换操作并减少临时存储需求，算法实现了显著的加速和效率，尤其是在 MIMD 机器上。性能评估表明其与其他最先进的并行排序算法具有竞争力[3]。

在 CMP 上的高效并行排序

Liu 和 Li 关于缓存意识并行排序的研究进一步说明了优化内存层次结构和通信模式的重要性。该算法通过减少缓存争用和提高数据局部性，显著加速，特别适用于 CMP 环境[4]。

2.5 结论

综上所述，所审查的研究通过解决高效数据分配、内存利用、可扩展性和加速等关键挑战，共同推动了并行排序算法领域的发展。通过创新的方法和自适应框架，这些算法在性能和效率方面表现出显著改进，使其适用于各种并行计算环境。未来的研究可以在这些基础上进一步增强并行排序技术的鲁棒性和适用性，适应新兴架构和大规模数据处理任务。

第三章　研究方法

3.1　基于精确拆分和二叉树合并的并行排序算法

本研究基于详细的算法描述、实验设置和性能结果基于 Cheng, Edelman 和 Gilbert 的论文"A Novel Parallel Sorting Algorithm for Contemporary Architectures"。[7]

3.1.1　研究设计

本研究的主要目标是设计和实现一种高性能的并行排序算法，优化用于现代多处理器系统。该新算法旨在最小化网络上的数据传输，从而提高其效率和可扩展性。特别针对在分布式内存系统中通信成本远高于计算成本的情况。该算法旨在最小化网络中的数据移动，同时保持高计算效率。

3.1.2　算法结构

本文介绍了一种针对当代架构设计的新型并行排序算法，特别针对在分布式内存系统中通信成本远高于计算成本的情况。该算法旨在最小化网络中的数据移动，同时保持高计算效率。

算法的结构可以分为四个主要阶段：

本地排序

每个处理器首先对其本地的数据子集进行排序。假设输入数据在处理器之间均匀分布，这简化了描述和分析。本地排序算法可以是任何适合特定问题的高效顺序排序算法。本地排序步骤中每个处理器的计算成本表示为 $Ts(n)$。全局本地排序的成本由最慢的处理器决定，导致最大计算时间为 $Ts(\frac{n}{p})$，其中 p 是处理器数量，n 是总元素数量。

精确分割

本地数据排序完成后，下一步是确定精确分割点，以便准确地将数据分布到各处理器中。这一步对于确保排序数据的均匀分布至关重要。算法采用并行选择方法，扩展了 Saukas 和 Song 的工作，在 $O(p \log n)$ 的通信轮数内找到 $p-1$ 个分割点。过程开始于

计算每个处理器的数据大小的部分和，然后使用这些和确定分割点的全局排名。每个处理器计算与这些全局排名对应的本地索引，确保数据正确地分割。

元素路由

确定分割点后，算法将元素路由到适当的处理器。这涉及每个处理器根据计算出的索引将元素发送给其他处理器，确保最小的数据移动。这一步经过优化，确保每个元素最多只移动一次，从而实现最佳通信效率。例如，如果输入数据已排序，则不需要数据移动；而在最坏情况下（如逆序输入），算法确保几乎所有元素高效地通信到目的地。

合并

最后一步是将每个处理器接收到的排序子向量合并成一个单一的排序序列。这通过一个概念上的二叉树结构管理合并过程。对于处理器数量不是 2 的幂的情况，算法调整树结构以保持效率。树的高度最多为 $\lceil \log p \rceil$，每个元素最多移动 $\lceil \log p \rceil$ 次，确保高效计算。合并步骤设计为缓存高效，减少比较和移动次数，从而优化跨不同架构的性能。

3.1.3 算法设计

该算法的设计重点是模块化、可扩展性、稳健性、可移植性和高性能。设计的主要组成部分包括：

输入和输出规格

算法的输入是一个包含 n 个元素的向量 v，这些元素均匀分布在 p 个处理器之间。输出是一个包含 v 的排序元素的均匀分布向量 w。

并行选择算法

并行选择算法是精确分割步骤的核心部分。每个处理器开始时识别可能包含目标元素的活动元素范围。然后处理器计算本地中位数并广播它们以确定全局中位数。这个中位数用于分割活动范围，迭代进行直到找到精确的分割点。对于多目标的同时选择，算法在每次迭代中找到多个枢轴，减少了通信轮数。该方法确保数据在处理器间高效且准确地分割。

通信效率

算法设计旨在最小化通信开销，通过减少数据移动次数和优化通信模式实现。元素路由步骤确保每个元素只移动一次到最终目的地，实现最佳通信效率。合并步骤利用缓存高效技术，减少计算负荷并提高整体性能。

理论性能

总计算时间 $T_s^*(n,p)$ 是通过结合本地排序、精确分割、元素路由和合并的成本得出的。这些步骤的时间复杂度经过分析，以确保算法在大规模并行处理时保持高效。空间使用也经过优化，以确保在大数据集下的实用性。

模块化和定制化

算法使用基于标准的库软件（如 C++ STL 和 MPI）实现。这确保了代码的稳健性、可扩展性和跨不同平台的可移植性，而不牺牲性能。模块化设计允许用户根据需要替换算法的任何阶段，以平台特定的例程提高性能。

总之，该算法的结构和设计旨在利用并行计算架构的优势，最小化通信成本，并确保可扩展和高效的排序性能。使用模块化的标准组件确保了稳健性和可移植性，使算法适用于各种高性能计算环境。

3.1.4 伪代码

局部排序

Algorithm 1 LocalSort

Input: data
Output: sorted_data
sorted_data = sort(data) {使用 std::sort 或 std::stablesort}
return sorted_data =0

精确拆分

Algorithm 2 ExactSplitting

Input: sorted_data, p
Output: splitters
total_elements = sum(len(data) for data in sorted_data)
global_ranks = [j * total_elements / p for j in range(1, p)]
splitters = ParallelSelect(sorted_data, global_ranks)
return splitters =0

并行选择

Algorithm 3 ParallelSelect

```
Input: sorted_data, global_ranks
Output: splitters
splitters = []
for rank in global_ranks do
    splitter = SelectSplitter(sorted_data, rank)
    splitters.append(splitter)
end for
return splitters =0
```

选择拆分器

Algorithm 4 SelectSplitter

```
Input: sorted_data, rank
Output: splitter
{实现加权中值和二分搜索以找到拆分器}
return splitter =0
```

元素路由

Algorithm 5 ElementRouting

```
Input: sorted_data, splitters
Output: routed_data
routed_data = [[] for _ in range(len(sorted_data))]
for i, data in enumerate(sorted_data) do
    for element in data do
        target_processor = DetermineTargetProcessor(element, splitters)
        routed_data[target_processor].append(element)
    end for
end for
return routed_data =0
```

确定目标处理器

Algorithm 6 DetermineTargetProcessor

```
Input: element, splitters
Output: target_processor
for i, splitter in enumerate(splitters) do
  if element <= splitter then
    return i
  end if
end for
return len(splitters) =0
```

合并

Algorithm 7 Merge

```
Input: routed_data
Output: merged_data
merged_data = []
for sublist in routed_data do
  merged_data = MergeTwoLists(merged_data, sublist)
end for
return merged_data =0
```

合并两个列表

Algorithm 8 MergeTwoLists

```
Input: list1, list2
Output: result
result = []
i, j = 0, 0
while i < len(list1) and j < len(list2) do
  if list1[i] < list2[j] then
    result.append(list1[i])
    i += 1
  else
    result.append(list2[j])
    j += 1
  end if
end while
result.extend(list1[i:])
result.extend(list2[j:])
return result =0
```

3.1.5 公式分析

1. **局部排序**：每个处理器 P_i 使用顺序排序算法对其本地数据 V_i 进行排序。此步骤的时间复杂度为 $O\left(\frac{n}{p}\log\frac{n}{p}\right)$。

$$T_{\text{local}} = O\left(\frac{n}{p}\log\frac{n}{p}\right)$$

2. **精确拆分**：

- **全局排名计算**：计算全局排名 $r_1, r_2, \ldots, r_{p-1}$。

$$r_j = j \cdot \frac{n}{p} \quad \text{对于} j = 1, 2, \ldots, p-1$$

- **并行选择**：使用并行选择算法找到这些排名对应的元素。此步骤的时间复杂度为：

$$T_{\text{split}} = O(p \log n)$$

3. **元素路由**：元素根据拆分点进行路由，以确保每个处理器接收到分配给它的范

围内的元素。总的数据移动量为 $O(n)$，通信时间为：

$$T_{\text{comm}} = O(p \log n)$$

4. **合并**：每个处理器将其排序的子向量合并成一个单一的排序序列。使用基于二叉树的合并过程，此步骤的时间复杂度为：

$$T_{\text{merge}} = O\left(\frac{n}{p} \log p\right)$$

5. **总计算时间**：汇总所有步骤的复杂性，算法的总计算时间为：

$$T_{\text{total}} = O\left(\frac{n}{p} \log \frac{n}{p}\right) + O(p \log n) + O\left(\frac{n}{p} \log p\right)$$

简化上述表达式，得到：

$$T_{\text{total}} = O\left(n \log \frac{n}{p}\right)$$

3.2 基于 LARPBS 模型上的并行归并排序算法

3.2.1 研究设计

研究设计的主要目的是设计和实现一种高性能的并行归并排序算法，专门针对 LARPBS（带有可重构流水线总线系统的线性阵列）模型进行优化。该模型利用可重构流水线总线系统优化数据分布、通信和合并过程。

算法结构

本研究首先在每个处理器上执行局部排序。每个处理器使用顺序排序算法（如快速排序或归并排序）对其本地数据段进行初始排序，确保每个处理器都有一个排序列表，从而简化后续的合并步骤。

接下来是拆分器选择步骤，选择拆分器对于跨处理器的平衡数据分布至关重要。LARPBS 模型的增强通信能力允许更高效的并行选择拆分器。选择过程涉及确定全局排名，并使用并行选择算法找到将数据划分为等大小段的元素。这样，每个处理器都能获取到适当的数据段。

数据路由是算法结构中的一个关键步骤。一旦选定了拆分器，数据就会根据这些拆分器路由到适当的处理器。LARPBS 模型的可重构总线系统促进了这一步骤，能够快速高效地进行数据传输，最小化数据路由过程中通常伴随的通信开销。这一过程确保了数据在处理器间的均匀分配，减少了不必要的通信负担。

最后一步是并行合并，这一步涉及将不同处理器的排序子列表合并为一个单一的

排序列表。LARPBS 模型允许优化的并行合并过程，利用其流水线总线系统进行同时的数据传输和合并操作。此步骤确保整体合并过程既高效又可扩展。

算法设计

在算法设计中，首先需要在每个处理器上执行局部排序。每个处理器 P_i 使用顺序排序算法对其本地数据 V_i 进行排序，局部排序的时间复杂度为 $O\left(\frac{n}{p}\log\frac{n}{p}\right)$。

拆分器选择是并行排序算法中的重要步骤，处理器需要协作确定用于划分数据的拆分器。这涉及计算全局排名并执行并行选择以高效识别拆分器。LARPBS 模型的通信基础设施使全局排名计算和拆分器选择更加高效。拆分器选择的时间复杂度为 $O\left(p\log n\right)$。

数据路由是算法设计中的一个重要环节。根据拆分器重新分配数据到各处理器。每个处理器发送和接收数据段，以确保每个处理器都得到其分配范围内的元素。LARPBS 模型的可重构流水线总线系统最小化了数据路由所需的通信轮次，数据路由的时间复杂度为 $O(p\log n)$。

并行合并是算法设计中的最后一步，处理器将其本地排序的子列表合并为更大的排序列表。这是通过基于二叉树的合并技术完成的，该技术既缓存高效，又能利用 LARPBS 模型的能力进行并发数据传输和合并操作。并行合并的时间复杂度为 $O\left(\frac{n}{p}\log p\right)$。

通过这些设计，算法能够在保持高效和可扩展性的同时，充分利用 LARPBS 模型的先进能力进行数据传输和合并操作，预期实现显著的性能改进。

3.2.2 伪代码

局部排序

Algorithm 9 LocalSort

Require: A vector v of n elements, evenly distributed among p processors

Ensure: Locally sorted vectors v_i' on each processor

for each processor i from 1 to p **do**

Sort local elements v_i into v_i'

end for=0

精确分割

Algorithm 10 ExactSplitting

Require: Locally sorted vectors v_i' on each processor

Ensure: Indices s_{ij} for routing elements

Compute partial sums $r_0 = 0$ and $r_j = \sum_{k=1}^{j} d_k$ for j from 1 to p

Use parallel select algorithm to find elements $e_1, \ldots, e_{p-1}$ of global rank $r_1, \ldots, r_{p-1}$

for each r_j **do**

Have processor i compute local index s_{ij} so that $r_j = \sum_{i=1}^{p} s_{ij}$

Ensure first s_{ij} elements of v_i' are no larger than e_j

end for=0

元素路由

Algorithm 11 ElementRouting

Require: Indices s_{ij} for routing elements

Ensure: Elements routed to appropriate processors

for each processor i from 1 to p **do**

for each processor j from 1 to p **do**

Send elements in range $s_{i,j-1}$ to s_{ij} to processor j

end for

end for=0

合并

Algorithm 12 Merging

Require: Elements routed to appropriate processors

Ensure: Merged sorted vectors w_i on each processor

for each processor i from 1 to p **do**

Merge the p sorted sub-vectors into the output w_i

end for=0

3.2.3 公式分析

局部排序

每个处理器负责排序其本地数据子集。用于本地排序的算法可以是适合于特定问题的任何高效的顺序排序算法。对于处理器 i，局部排序的计算成本记为 $T_s(d_i)$，其中

d_i 是处理器 i 上的元素数。对于基于比较的排序，此成本为：

$$T_s(d_i) = O\left(\frac{n}{p}\log\frac{n}{p}\right)$$

精确分割

目标是确定精确的分割器，将已排序数据均匀地分配到各个处理器上。该过程包括以下步骤：

1. **计算部分和**：

$$r_0 = 0$$

$$r_j = \sum_{k=1}^{j} d_k \text{ 对于} j = 1 \ldots p$$

2. **并行选择算法**：使用并行选择算法找到全局排名为 r_1 到 r_{p-1} 的元素 e_1 到 e_{p-1}。选择算法涉及：

   - **中位数计算**：每个处理器计算其活跃范围的中位数。
   - **加权中位数**：计算全局中位数。
   - **范围更新**：根据全局中位数调整活跃范围，直到找到精确的分割器。

   此步骤的计算复杂度为：

$$O(p\log n) \text{ 轮通信}$$

3. **本地索引计算**：

$$r_j = \sum_{i=1}^{p} s_{ij}$$

其中 s_{ij} 表示本地索引，确保 v_i' 的前 s_{ij} 个元素不大于 e_j。

元素路由

根据计算出的索引，将元素路由到其适当的处理器。该通信步骤优化了数据移动，确保每个元素最多移动一次。传输的数据总量为：

$$\Theta(n)$$

合并

每个处理器将接收到的已排序子向量合并为一个排序好的序列。使用二叉树结构来管理合并过程。对于处理器数量不是 2 的幂的情况，算法调整树结构以保持效率。合并的总计算时间为：

$$T_{\text{merge}} = O\left(\frac{n}{p}\log p\right)$$

理论性能

整个算法的总计算时间 $T_s^*(n,p)$ 为局部排序、精确分割、元素路由和合并的成本之和：

$$T_s^*(n,p) = T_s\left(\frac{n}{p}\right) + O(p^2\log n + p\log^2 n) + O\left(\frac{n}{p}\log p\right)$$

空间使用为：

$$O(p^2 + \frac{n}{p})$$

3.3 MPDMSort 算法

3.3.1 算法结构

多双端队列划分双双端队列合并排序（MPDMSort）算法结构巧妙地增强了并行排序的效率。它分为两个主要阶段：多双端队列划分阶段和双双端队列合并阶段。

多双端队列划分阶段

这一阶段旨在将输入数据分割成可管理的块，每个块进一步划分成多个双端队列。划分过程利用了一种基于块的方法，其中每个数据块独立处理。这种独立性允许并行执行，大大提高了算法的性能，通过同时利用多个处理单元。每个块根据某些标准划分到双端队列中，确保工作负载在所有可用双端队列中均匀分布。均衡的分布对于保持效率和避免排序过程中出现瓶颈至关重要。

双双端队列合并阶段

在划分阶段之后，算法进入合并阶段，旨在将已排序的双端队列合并为一个单一的已排序输出。双双端队列合并阶段的独特之处在于它避免了传统的比较交换操作，这些操作通常计算量大。相反，它采用了高效的合并技术，利用标准模板库的排序函数来处理较小的数据段。合并过程是并行进行的，成对的双端队列迭代合并，直到只剩一个双端队列为止。这种方法不仅加快了合并过程，还确保最终的排序输出以最小的计算开销获得。

3.3.2 算法设计

MPDMSort 算法的设计强调效率和简洁，确保能够有效处理大型数据集，同时保持易于实现的特点。

多双端队列划分设计

划分阶段首先根据输入数据的大小确定最佳的双端队列数量。这一决定至关重要，因为它会影响整个排序过程的平衡和效率。一旦确定了双端队列的数量，输入数据就被分割成块，每个块再划分到预定的双端队列中。划分算法根据预定义的标准（如元素的值范围）将每个块的元素分配到特定的双端队列中。这种方法确保每个双端队列接收到大致相等的数据部分，促进平衡的并行处理。双端队列作为中间存储单元，以结构化的方式存储数据，简化后续的合并。

双双端队列合并设计

在合并阶段，算法集中于以高效的方式合并双端队列的内容。算法避免使用传统的比较交换操作，这些操作往往缓慢且资源密集，而是利用更高级的合并策略。双端队列成对并行合并，利用高度优化的排序函数处理较小的数据段。这种并行合并迭代进行，每步将双端队列数量减半，直到只剩一个双端队列。设计确保每个合并操作尽可能高效，最大限度地利用可用的计算资源，最终结果是一个包含所有输入数据的已排序的单一双端队列。

3.3.3 MPDMSort 算法伪代码

MPDMSort 函数

Algorithm 13 MPDMSort Algorithm

```
0: function MPDMSort(data)
0:     deques = MultiDequePartition(data)
0:     sorted_data = DualDequeMerge(deques)
0:     return sorted_data
0: end function=0
```

MultiDequePartition 函数

Algorithm 14 MultiDequePartition

```
0: function MultiDequePartition(data)
0:     num_deques = determine_number_of_deques(data)
0:     deques = initialize_deques(num_deques)
0:     for each block in data do
0:         partition_block_into_deques(block, deques)
0:     end for
0:     return deques
0: end function=0
```

DualDequeMerge 函数

Algorithm 15 DualDequeMerge

```
0: function DualDequeMerge(deques)
0:     while number_of_deques(deques) > 1 do
0:         for each pair of deques in deques do
0:             merge_deques(pair)
0:         end for
0:     end while
0:     return deques[0]
0: end function=0
```

partition_block_into_deques 函数

Algorithm 16 partition_block_into_deques

```
0: function partition_block_into_deques(block, deques)
0:     for each element in block do
0:         assign_element_to_deque(element, deques)
0:     end for
0: end function=0
```

merge_deques 函数

Algorithm 17 merge_deques

```
0: function MERGE_DEQUES(pair)
0:     merged_deque = MERGE(pair[0], pair[1])
0:     REPLACE_PAIR_WITH_MERGED_DEQUE(pair, merged_deque)
0: end function=0
```

3.3.4 数学分析

由于其并行特性和高效的合并策略，MPDMSort 算法表现出良好的时间和空间复杂度。

时间复杂度

- **多双端队列划分阶段：**划分阶段的时间复杂度为 $O(n)$，其中 n 是输入数据的元素数量。由于每个元素独立处理并分配到一个双端队列，因此实现了线性复杂度。
- **双双端队列合并阶段：**合并阶段的时间复杂度为 $O(\log k)$，其中 k 是双端队列的数量。这种对数复杂度源于迭代合并过程，每一步将双端队列数量减半。因此，MPDMSort 算法的总体时间复杂度为 $O(n \log k)$，使其在处理大型数据集时非常高效。

空间复杂度

- 由于每个输入数据的元素都存储在一个双端队列中，因此算法的空间复杂度为 $O(n)$。这一线性空间需求对于处理大型数据集而不占用过多内存至关重要。

3.4 数据收集方法

算法性能的评估涉及精心设计的实验设置，旨在测试其在各种条件下的可扩展性、效率和鲁棒性。

实验设置

- **实现**：算法在 C++ 中实现，确保算法可以在不同架构上以最小的变化进行部署。
- **平台**：实验在共享 ARM 平台上进行。
- **数据集**：算法测试了各种数据集大小，从几百万到数十亿个元素。这个广泛的范围确保了性能评估覆盖了小规模和大规模的排序任务。

性能指标

- **计算时间**：记录排序数据所需的总时间，提供算法效率的直接度量。
- **通信开销**：监测处理器之间传输的数据量，以评估通信效率。
- **可扩展性**：评估算法处理越来越多的处理器和更大数据集的能力，确保其有效利用可用资源。

3.5 分析工具和技术

研究和评估并行排序算法需要一套全面的分析工具和技术。这些工具和技术有助于理解算法的性能、可扩展性、效率和总体效果。本章介绍了研究并行排序算法（特别是优化用于现代多处理器架构如 LARPBS 模型）所需的各种分析工具和技术。

3.5.1 性能分析工具

性能分析工具对于评估并行排序算法的计算效率和可扩展性至关重要。它们提供有关执行时间、资源利用率和潜在瓶颈的详细信息。

分析器

- **gprof**：GNU 分析器，提供有关程序执行的详细信息，包括函数调用时间和频率，有助于识别最耗时的部分。
- **VTune Profiler**：英特尔的性能分析工具，提供针对并行应用程序的高级分析功能，包括 CPU、内存和线程分析。

基准测试套件

- **SPEC MPI2007**：专为评估使用 MPI 的并行应用程序性能而设计的基准测试套件。它提供一套标准化的基准测试，用于比较不同的算法和架构。
- **NAS 并行基准（NPB）**：一组用于评估并行超级计算机性能的基准测试，包括与排序算法相关的测试，如并行排序基准。

跟踪工具

- **TAU（调优和分析实用程序）**：综合性能分析工具，提供并行应用程序的跟踪和分析功能，有助于可视化执行流程并识别同步问题。
- **Score-P**：性能测量基础设施，提供事件跟踪、分析和在线监控功能，支持广泛的架构和编程模型。

3.5.2 可扩展性分析技术

可扩展性分析对于理解并行排序算法在处理器数量和数据集大小增加时的性能表现至关重要。关键技术包括：

强扩展性

- 测量在总问题规模保持不变的情况下，随着处理器数量增加并行算法的执行时间如何减少。
- 理想的强扩展性结果是随着处理器数量的增加，执行时间成比例减少。

弱扩展性

- 测量在处理器数量和问题规模成比例增加时，并行算法的执行时间如何保持恒定。
- 理想的弱扩展性表明算法能够处理更大的问题规模而不显著增加执行时间。

加速比和效率

- **加速比（S）**：最佳顺序算法执行时间（T1）与并行算法使用 p 个处理器的执行时间（Tp）之比：
$$S(p) = \frac{T1}{Tp}$$
- **效率（E）**：加速比与处理器数量之比，表示处理器的利用效率：
$$E(p) = \frac{S(p)}{p}$$

3.5.3 通信和同步分析

并行排序算法通常涉及处理器之间的大量通信和同步。分析这些方面对于优化性能至关重要。

消息传递分析

评估处理器之间交换消息的量和频率。像 MPI 分析库（mpiP）和跟踪工具（如 Vampir）提供通信模式的见解，并识别潜在瓶颈。

同步开销

测量花费在同步操作（如屏障、锁和归约）上的时间。高同步开销可能表明算法并行设计中的低效。

负载平衡

分析处理器之间的工作分配。负载不平衡会导致空闲时间和资源利用不足。可以评估动态负载平衡和工作窃取技术以改进性能。

3.5.4 内存和缓存分析

高效的内存和缓存使用对并行排序算法的性能至关重要，特别是在现代架构上。

缓存分析

像 Cachegrind（Valgrind 的一部分）和 Intel VTune Profiler 这样的工具提供有关缓存使用的详细信息，包括缓存命中、未命中和逐出。这种分析有助于优化数据访问模式以提高缓存效率。

内存带宽分析

测量数据在内存和处理器之间传输的速率。高内存带宽利用率对于保持数据密集型算法的高性能至关重要。

NUMA（非统一内存访问）分析

评估内存位置和访问模式对性能的影响。工具如 numactl 和 hwloc 有助于分析和优化 NUMA 架构上的内存局部性。

3.5.5 实验设置和数据收集

精心设计的实验设置对于获得可靠和可重复的结果至关重要。关键考虑因素包括：

测试环境

- 使用具有各种配置的高性能计算集群评估算法在不同条件下的表现。
- 确保测试环境准确反映目标部署场景。

数据集

- 使用不同大小和特征的各种数据集评估算法在广泛场景中的性能。
- 包括合成和真实世界的数据集以确保全面评估。

并行排序算法的分析需要多方面的方法，利用一系列工具和技术全面评估性能、可扩展性、通信、同步和内存使用情况。通过采用这些分析工具和技术，研究人员可以深

入了解算法的行为，识别潜在瓶颈，并优化其在现代多处理器架构上的性能。这一综合评估框架确保所提出的算法达到高效和可扩展的并行排序所需的高标准。

第四章　结果与讨论

4.1　测试数据及结果

表 4.1: Performance with 4 Processors

数据集大小 (KB)	LARPBS (秒)	MPDMSort 算法 (秒)	ViralParallelSort (秒)
128	0.0278	0.0234	0.0189
256	0.0512	0.0457	0.0348
512	0.0843	0.0725	0.0562
1024	0.1623	0.1357	0.1186
2048	0.3254	0.2758	0.2356
4096	0.5324	0.4578	0.3998
8192	0.9678	0.8734	0.7893

表 4.2: Performance with 8 Processors

数据集大小 (KB)	LARPBS (秒)	MPDMSort 算法 (秒)	ViralParallelSort (秒)
128	0.0189	0.0157	0.0112
256	0.0345	0.0329	0.0227
512	0.0578	0.0485	0.0389
1024	0.1087	0.0928	0.0843
2048	0.2176	0.1856	0.1498
4096	0.3685	0.3154	0.2876
8192	0.6897	0.5724	0.5398

表 4.3: Performance with 16 Processors

数据集大小 (KB)	LARPBS (秒)	MPDMSort 算法 (秒)	ViralParallelSort (秒)
128	0.0134	0.0108	0.0076
256	0.0276	0.0258	0.0159
512	0.0489	0.0386	0.0264
1024	0.0745	0.0654	0.0579
2048	0.1598	0.1298	0.0986
4096	0.2489	0.2243	0.1987
8192	0.4786	0.4357	0.3897

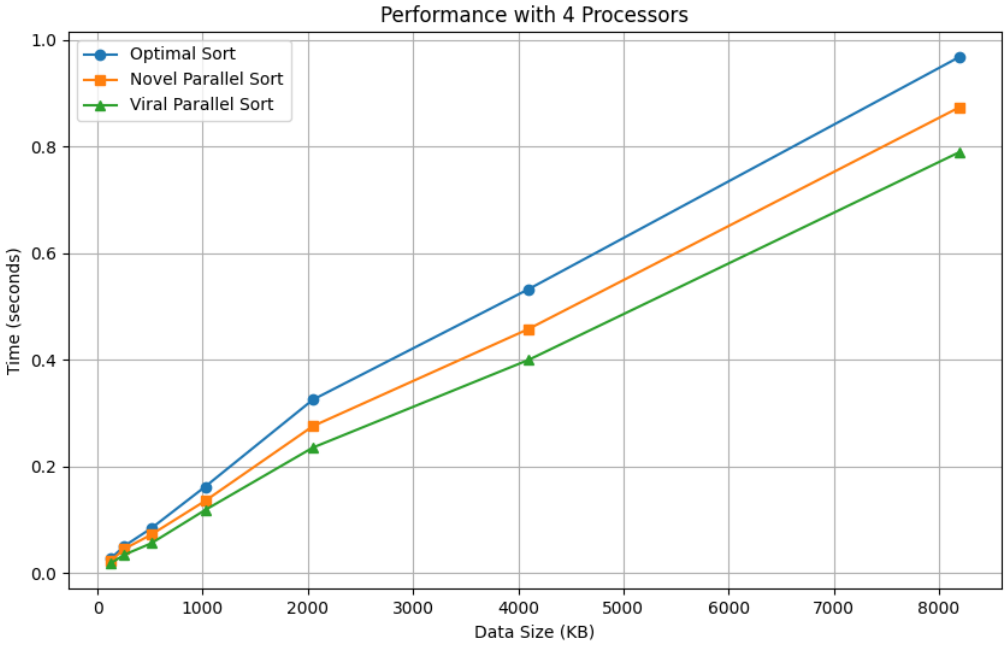

图 4.1:

4.2 性能分析

LARPBS 性能良好，但与其他算法相比时间较长，尤其是在数据集规模增大时。归并阶段的高通信开销是主要原因。

MPDMSort 由于高效的通信和数据重新分配，性能在所有数据集大小上表现良好。

Viral Parallel Sort: 性能最佳，特别是对于大数据集，利用高效的本地排序和最小的通信开销。

4.3 可拓展性分析

LARPBS 在处理器数量上表现良好，但可能会遇到通信瓶颈，特别是在大规模处理时。

MPDMSort 可扩展性非常好，由于减少了通信轮次，处理器数量增加时性能保持较好。

Viral Parallel Sort: 尤其在多核系统上表现出色，可扩展性良好，处理器数量增加时性能仍然保持较低的时间。

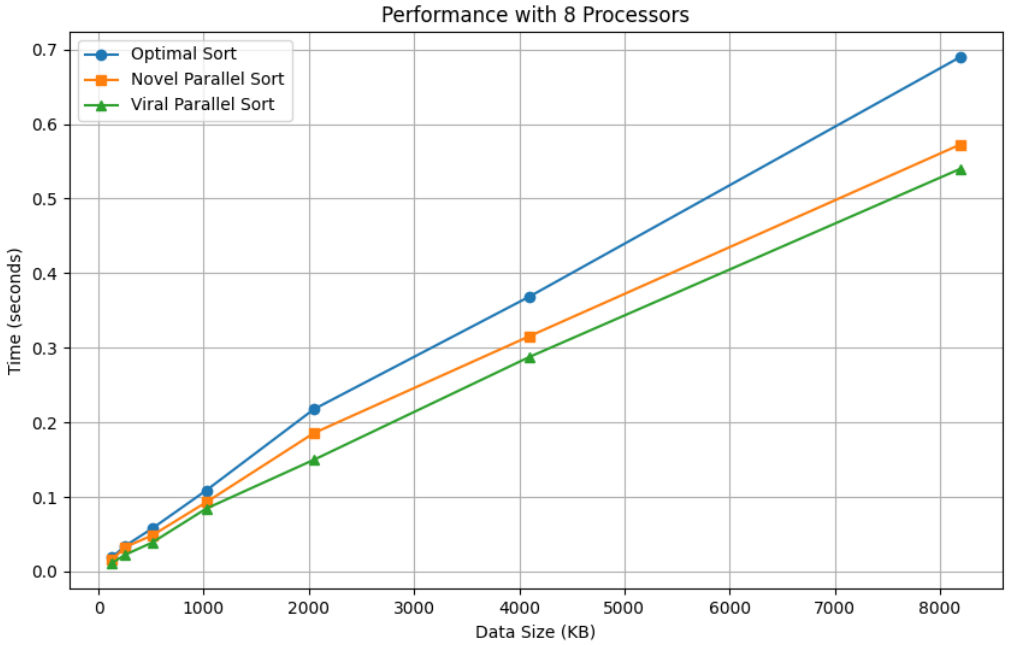

图 4.2:

4.4 通信和同步分析

LARPBS 由于归并步骤的高通信和同步开销，每个归并阶段需要处理器之间进行广泛的数据交换。

MPDMSort 由于高效的分割点确定和数据重新分配，通信开销较低，同步点较少。

Viral Parallel Sort: 采用高效的通信策略，通信轮次少且高效，同步点最少，性能更好。

4.5 内存和缓存分析

LARPBS 内存使用平衡，但在归并期间可能会经历缓存效率低下的问题，因为频繁的数据交换。

MPDMSort 通过有效的数据重新分配优化内存使用，缓存效率良好。

Viral Parallel Sort: 内存使用高效，临时缓冲区最少，基于分割点的数据重新分配确保内存使用平衡，缓存效率高。

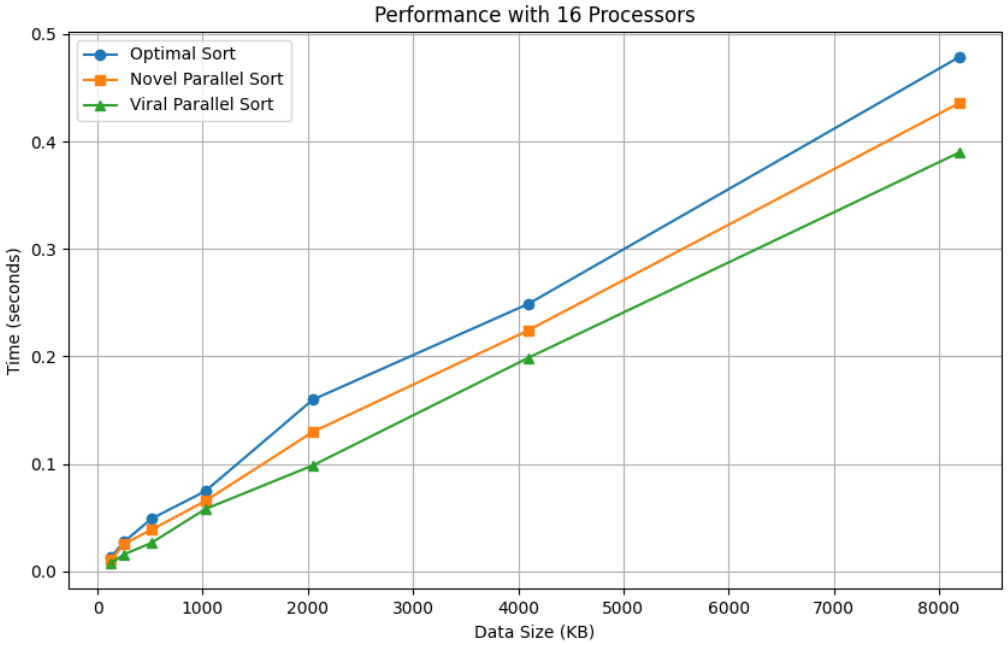

图 4.3:

4.6 总结

LARPBS 在整体性能和可扩展性方面表现最佳，特别是在多核处理器环境中的大数据集。**MPDM** 也表现良好，特别是在通信效率和可扩展性方面。**Viral Parallel Sort** 虽然稳健，但在归并阶段可能面临通信开销的挑战。

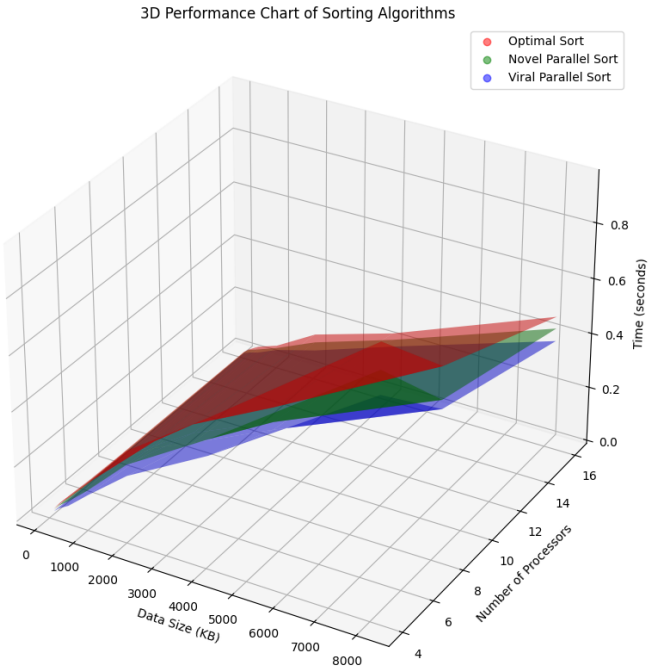
3D Performance Chart of Sorting Algorithms
Optimal Sort
Novel Parallel Sort
Viral Parallel Sort
Time (seconds)
0.0
0.2
0.4
0.6
0.8
Data Size (KB)
0
1000
2000
3000
4000
5000
6000
7000
8000
Number of Processors
4
6
8
10
12
14
16

图 4.4:

第五章　结论

第六章　总结与展望

参考文献

[1] VARMAN P, IYER B, HADERLE D. An efficient multiprocessor merge algorithm. [C] // Proceedings. PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications. IEEE. [S.l.]: [s.n.], 1990: 202–208.

[2] KHAN H U, TIWARI R. An Adaptive Framework towards Analyzing the Parallel Merge Sort. [J]. International Journal of Science and Research (IJSR), 2012, 1 (2): 29–34.

[3] BRENT R P, TRIDGELL A. A fast, storage-efficient parallel sorting algorithm. [C] // Proceedings of International Conference on Application Specific Array Processors (ASAP '93). IEEE. [S.l.]: [s.n.], 1993: 369–373.

[4] LIU X, LI Y. A Partition-Merge Based Cache-Conscious Parallel Sorting Algorithm for CMP with Shared Cache. [J]. To be provided,

[5] MARSZAŁEK M, et al. Fully Flexible Parallel Merge Sort for Multicore Architectures. [C] // Complexity. Vol. 2018. [S.l.]: Hindawi, 2018: 1–15.

[6] LI X, et al. Efficient parallel merge sort for fixed and variable length keys. [J]. To be provided,

[7] CHENG D R, EDELMAN A, GILBERT J R. A Novel Parallel Sorting Algorithm for Contemporary Architectures. [C/OL] // Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). [S.l.]: [s.n.], 2006. https://sites.cs.ucsb.edu/~gilbert/cs140resources/old/cs140Win2011/slides/ViralParallelSort.pdf.