面向大数据的可扩展正则采样并行排序算法

王莹，陈志广，卢宇彤
中山大学计算机学院，广东 广州 510006

摘要
排序算法是计算机科学领域的一个基础算法，是大量应用的算法核心。在大数据时代，随着数据量的极速增长，并行排序算法受到广泛关注。现有的并行排序算法普遍存在通信开销过大、负载不均衡等问题，导致算法难以大规模扩展。针对以上问题，提出一种大规模可扩展的正则采样并行排序（scalable parallel sorting by regular sampling，ScaPSRS）算法，摒弃传统正则采样并行排序（parallel sorting by regular sampling，PSRS）算法中由一个进程负责采样的做法，转而让所有进程参与正则采样，选出$p-1$个分隔元素，将整个数据集划分成p个不相交的子集，然后实施并行排序，避免了单一进程的采样瓶颈。此外，ScaPSRS采用一种新的迭代更新策略选择$p-1$个分隔元素，保证划分的p个子集尽可能大小相同，从而确保p个进程对各自的子集进行本地排序时的负载均衡。在天河二号超级计算机上进行的大量实验表明，ScaPSRS算法能够成功地扩展到32 000个内核，性能比PSRS算法和Hofmann等人提出的分区算法分别提升了3.7倍和11.7倍。

关键词
并行排序；正则采样；负载均衡；大数据

中图分类号：TP399 文献标志码：A doi: 10.11959/j.issn.2096-0271.2024021

A scalable parallel sorting algorithm by regular sampling for big data

WANG Ying, CHEN Zhiguang, LU Yutong
School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China

Abstract
Sorting is one of the basic algorithms in computer science, and has been extensively used in a variety of applications. In the big data era, as the volumes of data increase rapidly, parallel sorting has attracted much attention. Existing parallel sorting algorithms suffered from excessive communication overhead and load imbalance, making it difficult to scale massively. To solve above problems, a scalable parallel algorithm sorting by regular sampling (ScaPSRS) was proposed, which sampled the $p-1$ pivot elements to divide the entire data set into p disjoint subsets by all parallel processes, rather than by only one given process as PSRS did. Furthermore, ScaPSRS adopted a novel iterative update strategy of selecting pivots to guarantee that the workloads and data were evenly scheduled among the parallel processes, thus ensuring

superior overall performance. A variety of experiments conducted on the Tianhe-Ⅱ supercomputer demonstrated that ScaPSRS succeeded in scaling to 32 000 cores and outperformed state-of-the-art works significantly.

Key words
parallel sorting, regular sampling, load balance, big data

0 引言

排序算法是一种计算机科学基础算法，并已广泛用于各种科学研究和应用中，而并行排序是计算科学中最基本的并行算法之一。常用的并行排序算法有基于归并和基于分区的排序。基于归并的并行排序算法主要分为预排序和归并，其核心是多次归并。在共享内存系统中进行数据交换的开销相对较小，而在分布式内存系统中进行频繁、大量的数据交换会成为并行排序算法性能提升的瓶颈，因此基于归并的并行排序算法主要用于共享内存系统。如图1所示，基于分区的并行排序算法主要由4个阶段组成：本地排序、确定数据分区、数据重分布、最后的本地排序。在该算法中，只需要进行一次数据重分布，远比基于归并的并行排序算法少，能在分布式内存系统中取得更好的性能。

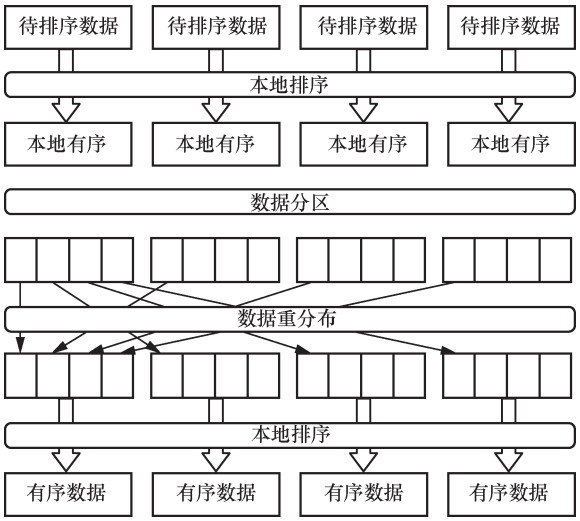

图1 基于分区的并行排序算法

在大数据时代背景下，各领域应用的数据量极速增长。随着超级计算机的发展，可用的计算资源越来越多，各领域的应用也期望依托超级计算机来处理更大规模的数据，算法并行化势在必行。然而，这受限于应用使用算法的可扩展性，比如并行排序算法。为充分利用计算资源，扩展排序算法的并行规模，不仅可以显著增加待排序的数据量，而且可以提高排序算法的性能，避免排序算法成为应用性能提升的瓶颈，从而加速各领域的应用。因此，为了对海量数据进行排序，排序算法的大规模并行扩展势在必行。

并行排序算法面临的主要问题是内存有限和耗时过长。共享内存系统上并行规模的限制以及归并排序算法的归并模式，都不利于算法的大规模扩展。显然，多节点、多进程的分布式并行算法可有效解决单节点共享内存不足的问题，分布式内存系统比共享内存系统更适合算法的大规模扩展。超级计算机是一种分布式内存系统，基于分区的并行排序算法只需要进行一次数据重分布，更适合在超级计算机上进行大规模扩展。

然而，大规模扩展的过程中，现有的基于分区的并行排序算法大多通过采样确定数据分区，并行排序算法的耗时主要由局部排序、确定数据分区和数据交换的耗时组成。扩展进程规模能减少每个进程的数

据量，进而显著减少局部排序的耗时。这会带来新的问题，如确定数据分区阶段的耗时和内存都会增加。此外，并行进程之间不平衡的工作负荷，例如真实世界的数据并不总是服从均匀分布，更多的是倾斜的、长尾的，这会影响随后的数据交换阶段。除此之外，甚至可能存在大量的重复元素，以至于用作分区的枢轴也存在重复元素。上述问题都对数据分区的并行排序构成挑战。在数据交换阶段以及最后的本地排序阶段，不均衡的分区会引入过多的开销，甚至出现内存溢出，从而导致并行排序算法无法高效实现。

为了解决上述问题，并行排序算法需要在进程之间进行有效的数据重分配。本文的贡献主要有以下4个方面。

（1）仍然采用从数据样本选出枢轴的方案。不同于现有的并行采样排序算法[1-4]将所有样本收集到主进程再由主进程独自选出所有枢轴元素的做法，本文提出的方法让每个进程都参与到生成候选枢轴的阶段中。这能有效缓解主进程存储和对所有数据样本进行排序的压力，避免出现内存溢出的情况，同时起到加速枢轴元素选取过程的作用，促进了排序算法实现大规模并行化。

（2）采用一种新颖的枢轴元素循环迭代更新策略，以保证工作负载在并行进程中相对均衡。在该迭代更新策略中，枢轴元素沿着负载均衡的方向更新，更新过程不仅考虑了数据样本，还参考了相邻的候选枢轴元素。

（3）优化了使用枢轴确定数据重分布的位移的算法。当相邻枢轴相差不足1时，对数据重分布的位移进行微调。因此，即使存在重复的枢轴元素，本文提出的并行排序算法ScaPSRS也能表现较好。

（4）在天河二号超级计算机上进行了不同规模的各种实验，证明了ScaPSRS能够成功扩展至32 000核，性能比PSRS[3]算法和Hofmann等人[5]提出的算法分别提升了3.7倍和11.7倍。

本文首先介绍了现有的并行排序算法的相关工作并阐述其优缺点；然后，介绍本文提出的可大规模并行的排序算法ScaPSRS的详细设计；最后，介绍使用ScaPSRS进行并行排序的实验细节，并对实验结果进行分析。

1 并行排序算法相关工作

并行排序算法可以分为两大类：基于归并的和基于分区的。基于归并的并行排序算法要经过本地排序和归并两个阶段，其中归并阶段需要多步来完成p个进程的归并。例如，并行归并排序[6]每一步都将相邻的有序数组两两归并，共需要$\log_2 p$步，但每一步参与的进程数量连续减半，在最后一步减少到只有一个进程工作，其余进程空闲等待，所以并行归并排序的性能不佳。在负载均衡并行归并排序中[6]，数据被均匀地分配给所有的进程，每个进程都参与所有归并。在第一步中，一个进程作为一个进程组，两两匹配完成一步归并，组成一个新的进程组；往后每一步重复以上操作，进程组的大小也增长为前一步的两倍。同时利用索引互换最小化进程之间交换的数据量。分区并发归并算法[7]是通过优化奇偶交换并行排序得到的，奇偶进程并发对数据子序列进行归并和排序，共进行$O(p/2)$轮归并。双调排序[8]的基本思想是对先单调增再单调减的双调序列进行归并，与大部分基于归并的并行排序算法一样，每一步双调排序都需要进行进程间的数据交换，共$O(p)$次，这是大规模扩展的瓶颈。尽管基数排序[9]不是一个基于比较的排序算法，但它也需要进行多次进程间

所有数据的交换，这是基数排序难以大规模扩展的重要原因。

基于归并的并行排序算法在进程间具有相对较高的通信开销，不适合大规模扩展。近年来也有一些并行排序算法是针对系统特性[10]或在GPU上[11]进行改进的。而基于分区的并行排序算法只需要一次数据重分布，更易于扩展。因此，本文提出的可在超级计算机上进行大规模扩展的并行排序算法也是基于分区的。

基于分区的并行排序算法主要由4个阶段组成：本地排序、确定数据分区、数据重分布、最后的本地排序，如**图1**所示。首先，每个进程独立、并行地对自己拥有的数据进行本地排序，该阶段并没有进程间的通信，并且初始数据均匀分布在各个进程上，故该阶段各进程的开销相对均衡。然后，需要确定数据分区以保证数据在重分布后在进程间是有序的，即在编号小的进程上的数据一定不大于编号大的进程上的数据。接下来，根据数据分区进行数据交换。最后的本地排序也是每个进程独立进行本地排序，该阶段的耗时取决于耗时最久的一个进程，而本地排序的耗时与本地数据量呈正相关。因此，为了避免出现有的进程完成本地排序后空闲等待某一个被分配了大量数据的进程的情况，数据重分布后各进程的数据应满足负载均衡。因此，确定数据分区是并行排序算法的重中之重，主要分为按数据值和按数据位置确定分区。

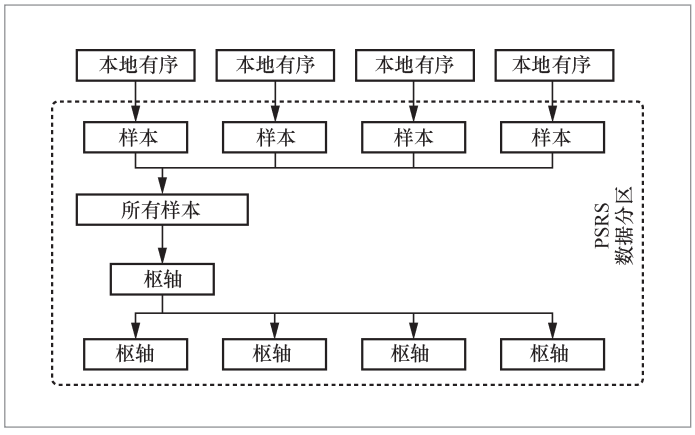

图 2 并行排序算法 PSRS 确定数据分区

按数据值确定数据分区是指，需要通过一个大小为$p-1$的有序枢轴元素数组将待排序的数据数组分成p份。待排序的数据值若落于相邻两个枢轴元素之间，则将被发送到对应进程。如此反复，将所有本地有序的待排数据划分为p个有序子集，再被分配到p个进程后，能保证数据在进程间是有序的，同时，期望数据交换后各进程间负载均衡。采样排序[1-3]是一种按数据值确定分区的并行排序算法。该算法的核心是每个进程对本地数据采样出q个样本，由一个主进程收集pq个样本并排序，从pq个有序样本中确定$p-1$个用于划分数据的枢轴元素并广播给所有进程，根据这$p-1$个枢轴元素的值对数据进行划分从而确定数据分区。常用的抽样方法有随机抽样、正则采样。

在正则采样并行排序[3]中，每个进程的本地采样和由主进程从收集到的所有样本中确定枢轴元素的采样都用了正则采样方法。在确定数据分区的过程中，主进程收集所有进程的样本并对收集到的样本进行排序，再从有序的样本中选出$p-1$个枢轴元素广播给其他进程。如**图2**所示，该过程中其他进程一直空闲等待，直到接收到由主进程广播的枢轴元素。假设系统有p个进程，需要将n个数据分成p份，那么至少需要$p-1$个枢轴元素，因此PSRS中的每个进程在本地数据上采样出$p-1$个样本点，那么主进程总共收集到$p(p-1)$个样本，显然主进程在这个阶段对收集到的样本进行本地排序的时空复杂度分别是$O(p^2\log p^2)$和$O(p^2)$。若想将PSRS扩展至16 384核，主进程需要2 GB的内存

来存储这$p(p-1)$个64位整数样本，并完成这2 GB样本数据的排序。随着进程规模的增加，主进程需要更多的内存来存储更多的样本，甚至超过存储待排序的数据所需内存，存在引起主进程所在节点内存溢出的风险。此外，主进程还需要花费更多的时间来完成样本数据的排序，而其他进程空闲等待。因此，PSRS只有在数据总量$n \geqslant p^3$的数据集上且所有样本数量不超过单个进程本地排序数量时才表现良好。但随着实际应用中数据量的极速增长，单个进程难以存储$O\left(\left(\sqrt[3]{n}\right)^2\right)$的数据量，无法高效地完成本地排序。这需要通过扩展进程规模来降低单个进程的排序数据量，而PSRS在大规模扩展时性能又较差，已然不适用。

Helman等人[4]提出的并行排序算法，用两轮正则通信取代了之前的单步非正则通信，以缓解负载不均衡的问题。这样能够以较低的采样成本保持较高的采样率，而且性能基本上不受输入数据的分布影响。HykSort[12]是基于超立方体的快速排序算法[13]，通过多路递归完成枢轴元素的迭代和并行选择。采样直方图排序HHS[14]每轮仅采样少量的数据样本，通过多轮全局归约的直方图统计所有样本数据的全局有序位置，直至找到满足要求的划分点。

然而，以上提到的采样排序算法，都需要主进程收集所有样本并从中选出所有的枢轴元素。主进程处于沉重的工作负荷中，而其他进程空闲等待，这并没有充分利用计算资源。除此之外，仅基于采样确定枢轴元素的并行排序算法一般只在均匀分布的数据集上表现良好，在倾斜分布的数据集上可能会导致严重的负载不均衡，从而表现欠佳。

按数据值选择枢轴元素，只能在具有不同数据值的位置对数据进行分割。Hofmann等人[5]提出了一种用于并行排序的分区算法。该分区算法使用分割位置来描述数据分区方案，而不是采用常用的根据数据枢轴值进行数据分区的方法。分割位置可以在任何位置对数据序列进行切分，即使数据存在重复元素，甚至所有数据值都相同。在该算法中，所有进程共同确定最终的数据重分布方案。对分割位置的搜索是分多轮进行的，每一轮从数据值的r位二进制表示的最高位开始进行t（$1 \leqslant t \leqslant r$）位的搜索。因此，最多需要进行$\frac{r}{t}$轮这样的搜索，相当于每一轮搜索中，每个进程都需要对$p-1$个位置使用二分搜索，确定2^{t+1}个候选分割数据值在本地数据序列的位置，然后将所有进程对应的分割位置归约计算总和，以找到满足负载均衡的分割位置。但该分区算法[5]的核心过程有3个循环嵌套，即最内层循环共进行了$\frac{r}{t} \times (p-1) \times 2^{t+1}$次，且最内层的循环包含二分搜索和全局同步归约操作，其开销将随着进程规模的增加而增加。因此Hofmann等人[5]的分区算法的耗时取决于3个循环嵌套和二分搜索花费的时间。

2 ScaPSRS

并行排序的问题可以描述为：在p个核心的分布式内存系统上，对数据总量为n的待排序数据序列进行排序，最后得到全局有序的数据序列$\overline{\mathcal{X}}$。假设并行排序前，第i个进程拥有一个大小为n_i的待排序子序列$\mathcal{X}_i$，其中$i \in \{0,\cdots,p-1\}$，且满足$\sum_{i=0}^{p-1} n_i = n$；并行排序后，第i个进程拥有

一个大小为 $\overline{n}_i$ 的待排序子序列 $\overline{\mathcal{X}}_i$，其中 $i\in\{0,\cdots,p-1\}$，且满足 $\sum_{i=0}^{p-1}\overline{n}_i=n$。假设 $x_{i,j}$ 和 $\overline{x}_{i,j'}$ 分别代表序列 $\mathcal{X}_i$ 和 $\overline{\mathcal{X}}_i$ 中的第j个元素，其中 $j\in\{0,\cdots,n_i-1\}$，$j'\in\{0,\cdots,\overline{n}_i-1\}$。符号 ≤ 表示两个元素之间的顺序，如 a≤b 表示序列a、b是有序的。在并行排序后，数据序列全局有序，表现为各数据元素在进程内和进程间都是有序的，即 $\overline{x}_{i,j}\leqslant\overline{x}_{i,j+1}\in\overline{x}_i$ 以及 $\overline{x}_{i,s}\leqslant\overline{x}_{i+1,t}$，其中 $j\in\{0,\cdots,n_i-2\},s\in\{0,\cdots,\overline{n}_i-1\},t\in\{0,\cdots,\overline{n}_{i+1}-1\}$，$i\in\{0,\cdots,p-2\}$。

本文提出的适用于大规模超级计算机的可扩展正则采样并行排序算法ScaPSRS，背后的原理与PSRS类似，是一种基于分区的采样并行排序算法。

2.1 ScaPSRS算法概述

基于分区的并行排序算法的框架如**图1**所示，主要总结为以下4阶段。

- 本地排序阶段：每个进程都对本地数据序列 $\mathcal{X}_i$ 进行一次本地排序，得到一个新的有序序列 $\mathcal{X}_i'$，使得 $\forall j\in\{0,\cdots,n_i-2\}$ 都满足 $x'_{i,j}\leqslant x'_{i,j+1}$。

- 确定数据分区阶段：每个进程都确定如何分割数据，用一个位移数组 $S_i=\{s_{i,j}|j=0,\cdots,p-1\}$ 来表示数据分区情况，将本地有序的数据序列 $\mathcal{X}_i'$ 分割成p份互不相交的子序列 $\mathcal{X}'_{i,j}=\{\mathcal{X}'_{i,j}|s_{i,j}\leqslant k<s_{i,j+1}\}$。根据定义，$S_i$ 是一个递增数组，满足 $s_{i,0}=0$，$s_{i,p}=n_i$，$s_{i,j}\leqslant s_{i,j+1}$。注意，如果 $s_{i,j}=s_{i,j+1}$，那么 $\mathcal{X}'_{i,j}$ 为空。

- 数据重分布阶段：每个进程根据代表数据分区情况的位移数组 S_i，向其他$p-1$个进程发送不同数量的数据元素，并通过集合通信操作MPI_Alltoallv从其他$p-1$个进程接收不同数量的数据元素，得到p段局部有序的数据序列 $\mathcal{X}_i''$。

- 最后的本地排序阶段：每个进程都对数据重分布后接收到的数据序列 $\mathcal{X}_i''$ 进行一次本地排序，得到一个新的有序序列 $\overline{\mathcal{X}}_i$。完成这一阶段操作后，每个进程上和进程间的数据都是有序的。

以PSRS为代表的基于分区的并行排序算法可以很快地从数据抽样得到一个可接受的枢轴元素序列，但这类抽样排序算法也有致命的弱点，导致其难以进行大规模扩展。首先，需要将所有进程的采样样本都收集到一个进程中，只有一个进程忙于对所有样本进行排序，而其他进程空闲等待。如此高开销且负载极度不均衡的采样排序策略，使得这些采样排序算法不适合大规模的并行化。其次，即使样本在某种程度上反映数据的分布，但鲁莽地从样本中再采样选择枢轴序列，难以生成负载均衡的数据分区，特别是对于偏态分布的数据。此外，对于存在大量重复元素的数据集，选出的枢轴元素也存在重复元素，这可能会恶化数据分区情况。Hofmann等人[5]的分区算法也会在最后一轮搜索后，出现不能确定$p-1$个分割位置的情况，需要额外增加一轮循环选出最终的分割位置。

针对以上并行排序算法的问题，提出了可大规模扩展的优化方案，包括正则通信下采样阶段的优化（regular sampling under regular communication）、枢轴的迭代更新策略（selection and update of pivots）、数据划分与微调（dividing data and fine-tuning），如**图3**所示。与**图2**中PSRS算法不同，在ScaPSRS算法中，所有进程参与数据分区的确定，缓解了单一进

程的数据存储压力，同时充分利用可用计算资源加速并行排序。此外，在迭代更新枢轴的阶段中，引入了进程间的代理策略以加速枢轴更新过程，找到满足负载均衡的数据分区。

2.2 正则通信下的正则采样

每个进程都从本地有序的数据 $\mathcal{X}_i'$ 中正则采样出 $p-1$ 个样本，组成一个子序列 $\mathcal{Y}_i$，如式（1）所示。$\mathcal{X}_i'$ 是有序的，$\mathcal{Y}_i$ 也是有序的。

$$\mathcal{Y}_i=\begin{cases}\mathcal{X}_i', n_i<p\\ \left\{x_j'\in\mathcal{X}_i', j\equiv 0 \bmod \frac{n_i}{p}\right\}, n_i\geqslant p\end{cases} \quad (1)$$

在之前的采样排序算法中，主进程需要提供 $O(p^2)$ 的空间存储从所有进程收集到的样本，并花费 $O(p^2\log p^2)$ 的时间对所有样本进行排序。

本文提出了一个正则通信下优化的正则采样算法。为了缓解主进程在存储所有样本的压力和加速样本排序的过程，本文利用正则的集合通信操作MPI_Alltoall收集并发送样本。进程 i 发送 $y_{i,j}\in\mathcal{Y}_i$ 给进程 j 并从进程 j 接收 $y_{j,i}\in\mathcal{Y}_j$，其中 $i\in\{0,\cdots,p-1\}$，$j\in\{0,\cdots,p-1\}$。因此，每个进程都接收到一个大小为 $O(p)$ 的样本序列 $\mathcal{Y}_i'=\{y_{i,j}'=y_{j,i}\}$，对 $\mathcal{Y}_i'$ 进行排序得到 $\mathcal{Y}_i''$。有序序列 $\mathcal{Y}_i''$ 是第 i 个枢轴元素 z_i 的候选序列，将第 $\frac{p}{2}$ 个值（$y_{i,\frac{p}{2}}''$），作为第 i 个枢轴 z_i 的初始候选值。每个进程负责确定一个枢轴元素的候选值，组成枢轴序列 $\mathcal{Z}=\{z_1,\cdots,z_{p-1}\}$，用于将数据序列 $\mathcal{X}_i'$ 划分成 p 个互不相交的子序列，期望进行重分布后的每个进程的数据量相对均衡。算法1展示了正则通信下正则采样算法的伪代码。

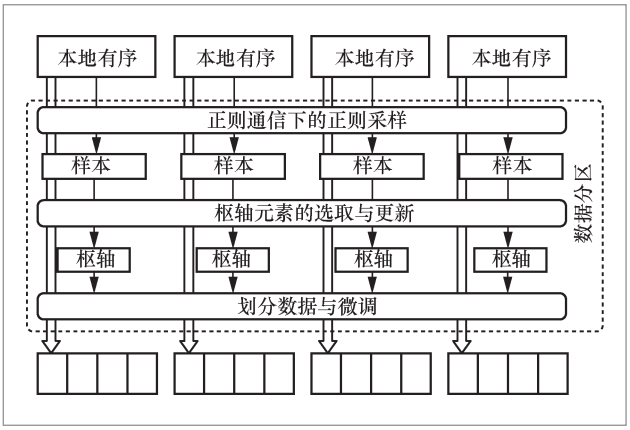

图 3 并行排序算法 ScaPSRS 中的数据分区

算法1： 正则通信的正则采样

输入： 进程数 p，当前进程编号 i，本地有序数据序列 $\mathcal{X}_i'=\{x_{i,0}',\cdots,x_{i,n_i-1}'\}$ 及其大小

输出： 有序的样本序列 $\mathcal{Y}_i''=\{y_{i,0}'',\cdots,y_{i,p-1}''\}$

```
1:  y_{i,j} = 0  for  j = 0,...,p-1
2:  if n_i < p then
3:      for j = 0  to  n_i - 1 do
4:          y_{i,j} = x'_{i,j}
5:      end for
6:  else
7:      q = n_i / p
8:      for j = 1 to  p - 1 do
9:          y_{i,j} = x'_{i,j×q}
10:     end for
11: end if
12: 通过MPI_Alltoall通信操作交换枢轴元素，确定 y'_{i,j} = y_{j,i}  for  j ∈ {0,...,p-1}
13: Y''_i = LocalSort(Y'_i)
```

在该采样模块中，每个进程只需要提供 $O(p)$ 的空间存储收集到的样本，并花费 $O(p\log p)$ 的时间对收集到的样本进行排

序。显然，采用正则通信下的正则采样模块的并行排序算法，在大规模并行情况下性能优于PSRS。

2.3 枢轴元素的选择与迭代更新

尽管采用优化后的采样方法的并行排序算法的运行速度明显比PSRS快，但在倾斜分布的数据上表现往往不尽如人意。这是因为使用只从样本中取样一次的枢轴，难以生成一个负载均衡的数据分区。因此，本文引入枢轴元素的迭代更新策略，在一个迭代循环中更新枢轴元素的选择，期望选出的枢轴元素能生成一个负载均衡的数据分区。

在数据分区中，位移 $s_{i,j} \in S_i$ 不仅代表进程i要发送给进程j的数据子序列在数据序列 $\mathcal{X}_i'$ 中的起始位置，还代表了进程i一共要给前j个进程发送的数据总量。为此，计算所有进程的位移数组 S_i 的纵向和 $V=\left\{v_j|v_j=\sum_{i=0}^{p-1}s_{i,j},j=0,\cdots,p-1\right\}$。根据定义，$v_0=0$，$v_p=n$，且 $v_j \leqslant v_{j+1}$。元素 v_j 表示根据该数据分区进行数据重分布后，前j个进程的数据总量，即 $v_j=\sum_{i=0}^{j}\overline{n_i}$。不难得出，$v_{j+1}-v_j$ 的值与进程j在数据重分布后的数据量相等。因此，通过计算位移数组的纵向和V，可以在没有进行任何实质数据重分布的情况下，计算出根据当前数据分区方案进行数据重分布后每个进程应有的数据量，这有助于选出负载均衡的枢轴元素。

理想情况下，进行数据重分布后，每个进程拥有完全相同的数据量，即 n/p，那么位移纵向和V应是 n/p 的倍数。用 $\overline{V}$ 来表示理想的位移纵向和，即 $\overline{V}=\left\{\overline{v}_j=j\times\frac{n}{p},j=0,\cdots,p-1\right\}$。但要在可接受的时间内找到满足理想的位移数组$S$几乎是不可能的。更新$S$使得$V$沿着理想的方向靠近。显然，当$V$越来越接近理想时，这个靠近速度会减慢。

因此，在理想 $\overline{V}$ 的邻域内定义纵向和V的上下界，通过一个可接受的偏差率b来控制上下界的范围，$j=0,\cdots,p-1$ 的上界 v_j^{u} 和下界 v_j^{l} 如下。

$$\begin{aligned} v_j^{\mathrm{u}}&=\overline{v}_j+b\times\frac{n}{p}=(j+b)\times\frac{n}{p}\\ v_j^{\mathrm{l}}&=\overline{v}_j-b\times\frac{n}{p}=(j-b)\times\frac{n}{p}\end{aligned} \tag{2}$$

根据定义，$v_0^{\mathrm{l}}=v_0^{\mathrm{u}}=0$，$v_p^{\mathrm{l}}=v_p^{\mathrm{u}}=n$。为保证上界 v_j^{u} 和下界 v_j^{l} 满足以下情况：

$$\begin{aligned} 0\leqslant v_j^{\mathrm{l}}\leqslant\overline{v}_j\leqslant v_j^{\mathrm{u}}\leqslant n\\ v_j^{\mathrm{l}}<v_{j+1}^{\mathrm{l}}\\ v_j^{\mathrm{u}}<v_{j+1}^{\mathrm{u}}\end{aligned} \tag{3}$$

偏差率b必须满足 $0<b<0.5$。如果b=0，那么V就是理想 $\overline{V}$；如果b=0.5，则会出现 $v_j^u=v_{j+1}^l$，可能导致数据重分布后进程j的数据量为0。

接下来，将描述如何利用 v_j 和 v_j^{l} 与 v_j^{u} 的关系来确定数据分区负载均衡的方向，进而为枢轴元素 z_j 接下来的迭代更新提供参考。首先，枢轴元素 z_j 是由对应进程j收集到的样本 $\mathcal{Y}_j''$ 初始化的。在收集所有的枢轴元素 $\mathcal{Z}=\{z_1,\cdots,z_{p-1}\}$ 后，需要对枢轴数组 $\mathcal{Z}$ 进行排序，以确保 $\mathcal{Z}$ 是有序的。然后，每个进程独立使用枢轴数组对本地排序后的数据 $\mathcal{X}_i'$ 进行划分，确定代表数据分区的位移数组 S_i，计算位移纵向和 $v_j=\sum_{i=0}^{p-1}s_{i,j}$。如果 v_j 正好落在 v_j^{l} 和 v_j^{u} 之间，表明枢轴元素

z_j 满足负载均衡数据分区的第j个枢轴；否则，需要根据 v_j 和 v_j^{l} 与 v_j^{u} 的关系来确定枢轴元素 z_j 在接下来的迭代中的更新方向。如果 $v_j < v_j^{\mathrm{l}}$，那么枢轴元素 z_j 应该沿着变大的方向更新；如果 $v_j > v_j^{\mathrm{u}}$，那么枢轴元素 z_j 应该沿着变小的方向更新。在更新枢轴元素 z_j 过程中，不仅在样本 $\mathcal{Y}_j''$ 上进行二分搜索，还考虑邻近的枢轴元素。这个更新策略可以弥补样本偏差，更快找到满足负载均衡数据分区的枢轴元素。算法2展示了以上枢轴元素的迭代更新的伪代码。为了更好地阐述，用 $z_j^{(r)}$ 表示在第r轮的枢轴元素 z_j。一般情况下，对样本 $\mathcal{Y}_j''$ 进行二分搜索来更新枢轴元素 z_j，因此，最多只需要进行 $\lceil \log p \rceil$ 轮的枢轴元素更新。

算法2： 枢轴元素的选择与迭代更新

输入： 进程数p，当前进程编号i，本地有序数据序列 $\mathcal{X}_i' = \{x_{i,0}', \cdots, x_{i,n_i-1}'\}$ 及其大小 n_i，数据总量n，有序的样本序列 $\mathcal{Y}_i'' = \{y_{i,0}'', \ldots, y_{i,p-1}''\}$，可接受的偏差率$b$

输出： 枢轴元素序列 $\mathcal{Z} = \{z_1, \cdots, z_{p-1}\}$

1: $n_{\mathrm{avg}} = \dfrac{n}{p}$

2: 纵向和的下界 $v_i^{\mathrm{l}} = n_{\mathrm{avg}} \times (i-b)$

3: 纵向和的上界 $v_i^{\mathrm{u}} = n_{\mathrm{avg}} \times (i+b)$

4: for $r = 0$ to $\lceil \log p \rceil$ do

5: 二分搜索 $\mathcal{Y}_i''$，结合 $z_j^{(r-1)}$，确定 $z_j^{(r)}$

6: LocalSort($\mathcal{Z}$)

7: 结合 $\mathcal{Z}$ 确定 S_i

8: 通过Allreduce-Sum通信操作计算 S 的纵向和，确定 $v_i = \sum_{k=0}^{p-1} s_{k,i}$，$s_{k,i} \in S_k$

9: if $v_i^{\mathrm{l}} \leqslant v_i \quad v_i^{\mathrm{u}}$ then

10: 标记枢轴元素 z_i 完成更新

11: end if

12: end for

在整个确定数据分区阶段，每个进程样本量为 $O(p)$，通过MPI_Alltoall进行一次样本交换的通信开销为 $O(p)$；在每一轮枢轴元素迭代更新中，通过Allreduce-Sum通信操作计算纵向和的通信开销为 $O(p)$，最多有 $\log p$ 轮。因此，确定数据分区阶段的通信开销为 $\mathrm{O}(p + p\log p)$。在数据重分布阶段，每个进程都需要根据确定的数据分区将数据量为 $O\left(\frac{N}{p}\right)$ 的本地数据发送给其他进程，并从其他进程接收数据，总量也约为 $O\left(\frac{N}{p}\right)$，通信开销为 $O\left(\frac{N}{p}\right)$。因此，ScaPSRS算法的通信开销为 $O\left(p + \frac{N}{p} + p\log p\right)$。

2.4 数据划分与微调

在选出$p-1$个枢轴元素 $\mathcal{Z}$ 后，每个进程i通过在数据 $\mathcal{X}_i'$ 上对枢轴元素 $\mathcal{Z}$ 进行二分搜索来确定数据分区的位移数组 S_i。**图4（a）**是枢轴元素 $\mathcal{Z}$ 不存在重复值的理想情况，能对数据进行均衡的划分。但当枢轴元素 $\mathcal{Z}$ 存在重复值时，这种划分方法会导致负载极度不均衡，大量的数据将被发送到一个特定的进程，而有部分进程没有任何数据，如**图4（b）**所示。

为了解决枢轴元素存在重复值时划分不均衡的问题，Khatami等人[10]提出只对不重复的枢轴元素执行二分搜索来确定

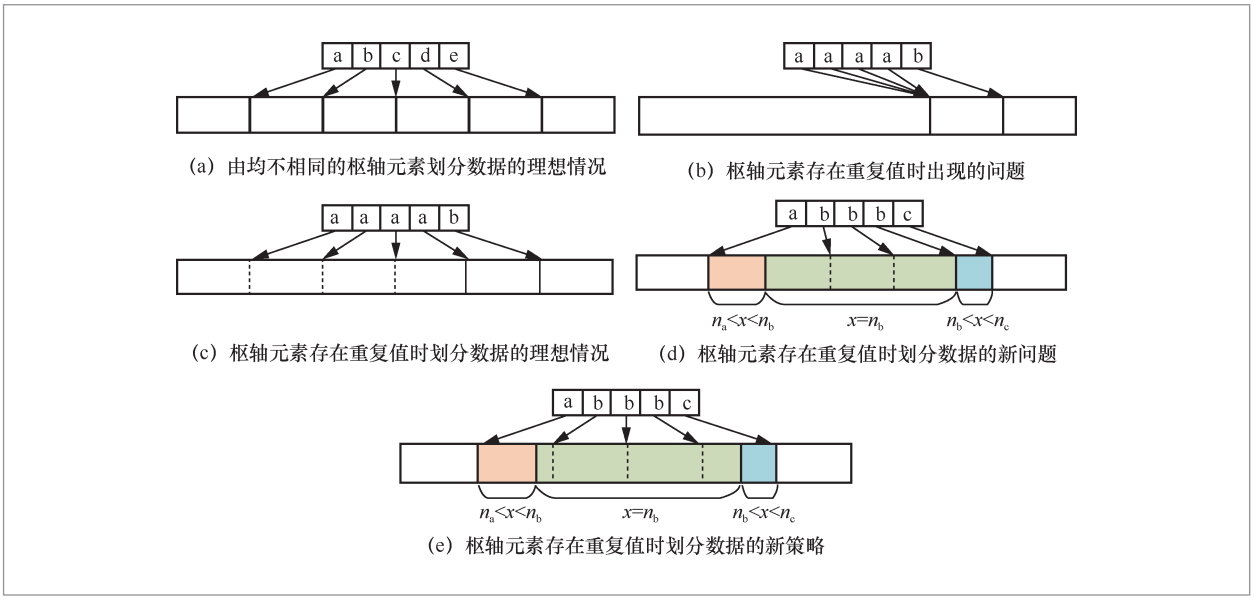

图 4 对本地数据进行二分搜索来确定数据分区

对应的数据分区位移，然后在具有相同值的枢轴元素之间平均分配确定的范围。例如，在**图4（b）**中划分给第一个a值枢轴元素的范围，在**图4（c）**中被平均分配给4个具有a值的重复枢轴元素。尽管这种划分方法避免了部分进程无法分到任何数据的情况，但仍然会得到一个不均衡的数据分区。在**图4（d）**中，存在3个具有相同b值的枢轴元素，如果直接对具有b值的数据进行平均分配，值在a和b之间的数据可能使得第一个b值的枢轴元素对应的进程接收过多的数据，单独把值在b和c之间的数据分配给值为c的枢轴元素对应的进程使其接收极少的数据，那么这种划分方式得到的数据分区也是不均衡的。如果每个进程都贸然地将值在a和c之间的本地数据进行平均分配，由于每个进程存在不同数量的值在a和b之间或者值在b和c之间的数据，每个进程的3个实际枢轴元素并不一定仍是b，极有可能在a和b之间或b和c之间产生，这种情况下，各个进程的枢轴元素无法统一数据重分布后无法满足数据在进程间保持有序的要求，因此该数据分区是不可接受的。

在数据重分布后数据在所有进程之间都是有序的前提下，为了保持数据分区的负载均衡，提出了一种考虑重复枢轴元素的前后确定位移的数据划分方法。如**图4（e）**中，存在3个具有b值的枢轴元素，其前后值为a或c的枢轴元素在数据序列上已确定定位，需要对值在a和c之间的数据进行尽可能均匀的划分，但同时要保证划分位点都落在值为b的数据上，以确保数据重分布后对应划分位点涉及的进程间的数据除了b值外值域互不相交，从而保证所有数据在进程间都是有序的。

尽管以上操作能解决大部分的数据分区不均衡的问题，但对于存在大量重复元素的数据集，仍会有数据分区不均衡的情况。尤其是在两个相邻的枢轴元素之间不存在任何其他元素，对于整数来说，即两者差值不超过1。本文提出对位移数组进行一轮额外的微调，以实现更好的负

载均衡。以整数为例，当 $v_j > v_j^{\mathrm{u}}$ 且 $z_j - 1 \leqslant z_{j-1} \leqslant z_j$，或 $v_j < v_j^{\mathrm{l}}$ 且 $z_j \leqslant z_{j+1} \leqslant z_j + 1$ 时，式(4)可以计算离负载均衡有多远。

$$\mathrm{rate} = \frac{\frac{1}{2}(v_j^{\mathrm{l}} + v_j^{\mathrm{u}})}{v_j} \tag{4}$$

对应的位移 $s_{i,j}$ 可以更新为 $s_{i,j} = s_{i,j} \times \mathrm{rate}$。注意，$s_{i,j}$ 不应比 $s_{i,j-1}$ 小，也不应比 $s_{i,j+1}$ 大，其中 $j = 1, \cdots, p-1$。位移微调使得在j位置的数据划分满足负载均衡，保证数据交换后进程$j-1$与j、j与$j+1$之间的数据是有序的。因为当 $v_j > v_j^{\mathrm{u}}$ 且 $z_j - 1 \leqslant z_{j-1} \leqslant z_j$ 时，rate<1，表示位移 $s_{i,j}$ 会往数值变小的方向变化，更新为 $s'_{i,j}$，即进程i的 $s'_{i,j}$ 到 $s_{i,j}$ 之间本地数据不再划分到进程$j-1$而改为划分到进程j，这部分数据值落在区间 $[z_{j-1}, z_j]$ 上。如果 $z_{j-1} = z_j$，那么这部分被微调的数据的值也与 z_{j-1} 和 z_j 相等；如果 $z_{j-1} = z_j - 1$，那么这部分被微调的数据的值与 z_{j-1} 或 z_j 相等，微调后进程$j-1$的数据最大值为 z_{j-1}，进程j的数据最小值为 z_{j-1}，即微调后仍能保证进程$j-1$与进程j之间的数据是有序的。同理可得，当 $v_j < v_j^{\mathrm{l}}$ 且 $z_j \leqslant z_{j+1} \leqslant z_j + 1$ 时，rate>1，表示位移 $s_{i,j}$ 会往数值变大的方向变化。

2.5 枢轴元素迭代更新的代理策略

在枢轴元素的迭代更新过程中，每个进程都只负责确定一个枢轴元素。由于枢轴元素是由样本初始化的，并在样本上通过二分搜索来更新的，有的进程可能在前几轮就确定了满足负载均衡的枢轴元素。为了避免已经确定枢轴元素的进程空闲等待并加速其他枢轴元素的确定，本文提出一个枢轴元素迭代更新的代理策略。

在该代理策略中，安排已经确定枢轴元素的进程帮助一个邻居进程确定它的枢轴元素。假设进程i已完成枢轴元素 z_i 的确定，而进程j是离进程i最近的一个未确定枢轴元素的进程。用d表示i和最近的j之间的距离，即 $d = \max k$，其中$i-k$到$i+k$的进程均已完成。进程i为 z_j 提供一个枢轴候选元素，朝着 v_i^l 或 v_i^u 方向以与d有关的步长更新。如果进程i为j找到了一个合适的枢轴元素，即值落在 v_i^l 和 v_i^u 之间，那么标记枢轴元素 z_j 完成更新，且进程i和j继续为下一个最邻近的未确定枢轴元素的进程代理，直至所有枢轴元素的迭代更新结束。

3 实验

通过在不同数据分布的数据集上对比ScaPSRS与其他基于分区的并行排序算法来分析ScaPSRS的性能。基于分区的并行排序算法的耗时主要由始末两个本地排序、确定数据分区和数据重分布的时间组成，本文也主要从这几个方面进行对比。

3.1 实验设计细节

所有并行排序算法的实验都在天河2号超级计算机上进行。每个计算节点都配备了两个12核的英特尔至强E5 CPU，更多的环境细节可见**表1**。

随着进程规模的增加，在最终本地排序阶段的待排序序列数量也在增加。许多实验表明，对于较大的p值，简单地对p个序列进行归并排序的表现不如C语言标准库中的排序算法std::sort。因此，本文实验在最终本地排序阶段均使用std::sort。

表1 实验环境

类别	项目	细节
软件	操作系统	Linux 3.10.0
	编译器	gcc 4.8.5 MPICH version 3.2.1
CPU	类型	IntelXeonE5
	主频	2.20 GHz
	并行	2个12核的socket
DRAM	类型	DDR3
	大小	64 GB
网络	带宽	TH2 Express-2+14 Gbit/s*8lane

3.2 正则通信下的正则采样的实验结果

本文将只采用了2.2节中的正则通信下的正则采样模块而没采用迭代更新枢轴元素来进行负载均衡优化的并行排序算法记为ScaPSRS_without_loadbalance。因此，ScaPSRS_without_loadbalance与PSRS[3]之间只有采样方式不同。通过对比这两个算法在相同进程规模下对相同数据分布、相同数据规模的数据集进行并行排序的耗时，观察正则通信下的正则采样模块对并行排序算法的性能提升作用。基于分区的并行排序算法的耗时主要由始末两个本地排序阶段、确定数据分区和数据交换的耗时共同组成。其中，始末两个本地排序阶段的耗时只与进程本地待排序数据量有关。在数据集大小不变的情况下，始末两个本地排序阶段通过增加进程数量来减少单进程数据量，从而降低本地排序耗时，但同时也会增加确定数据分区的耗时。图5展示了ScaPSRS_without_loadbalance和PSRS[3]分别在32~8 192个不同进程规模下对服从均匀分布的总共$2^{14}\times10^5$个整数的数据集进行并行排序的运行时间。相同数据规模、相同进程规模下，每个进程的本地数据量相同，本地排序时间也相同，ScaPSRS_without_loadbalance和PSRS的性能差距主要表现在本文提出的正则通信下的正则采样模块。从图5可以看到，在所有进程规模下ScaPSRS_without_loadbalance都比PSRS耗时短，尤其是在进程规模较大的时候。这是因为在PSRS中需要一个主进程对$O(p^2)$个样本进行本地排序；而在ScaPSRS_without_loadbalance中，每个进程只需要对$O(p)$个样本进行排序。即ScaPSRS_without_loadbalance在确定数据分区阶段的耗时远比PSRS短，两者之间的差距随着进程数p的增加而加大。因此，正则通信下的正则采样模块对并行排序算法有正向的性能提升效果，且进程规模越大，性能提升效果越明显，为大规模并行排序提供了可能性。

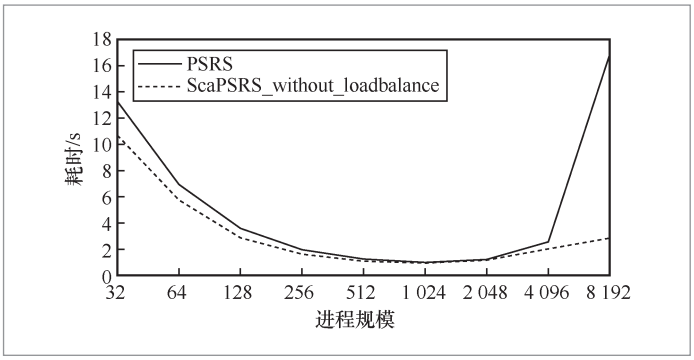

图5 并行排序算法在均匀分布数据集上的运行时间

3.3 ScaPSRS的实验结果

本文使用服从指数分布的数据集来模拟倾斜数据的并行排序。图6（a）和图6（b）展示了ScaPSRS分别在均匀分布和指数分布的数据集上的运行时间，两个数据集都有$2^{14}\times10^5$个整数。结合这两张图能发现，ScaPSRS在指数分布数据集上的表现和在均匀分布上的表现一样好。当进程规模较小时，ScaPSRS的总耗时

主要取决于始末两个本地排序阶段，且随着进程规模的增加，待排序的数据总量不变，分配到各个进程的数据量减少，始末两个本地排序耗时也随之下降。当进程规模增加到一定程度，分配给各个进程的数据量过少，并行排序的总耗时由确定数据分区阶段和数据重分布的耗时占主导，始末两个本地排序的耗时几乎可以忽略。这是因为在数据总量固定的前提下，本地排序的耗时与进程规模成反比，确定数据分区和数据重分布的耗时与进程规模成正比。因此，在实际应用中，需要根据待排序数据总量选择并行规模，平衡本地排序与确定数据分区和数据重分布的耗时。

3.4 负载均衡的实验结果

并行排序算法ScaPSRS有一个算法ScaPSRS_without_loadbalance没有的枢轴元素的迭代更新过程，该过程能保证选出的枢轴元素使数据分区满足负载均衡。**表2**展示了并行排序算法ScaPSRS_without_loadbalance和ScaPSRS的整体运行时间。这组实验仍然是在有$2^{14}\times10^{5}$个整数的均匀分布数据集上执行。可以发现，ScaPSRS并没有比ScaPSRS_without_loadbalance多花费很多时间，甚至在某些情况下ScaPSRS比ScaPSRS_without_loadbalance耗时更短。这是因为最后的本地排序阶段的耗时取决于接收数据量最多的进程的本地排序时间，而负载均衡的数据分区使各个进程接收到的数据量相对均衡，那么各个进程在最后的本地排序阶段的耗时也就相对平均，减少了进程之间的空闲等待。

图7呈现了两种并行排序算法，即ScaPSRS_without_loadbalance和ScaPSRS，在不同进程规模下消耗的总运行时间，可以直观地看到，两者耗费的时间差不多。

结合**图8（a）**和**图8（b）**，可以看出ScaPSRS和ScaPSRS_without_loadbalance在确定数据分区和进程间的

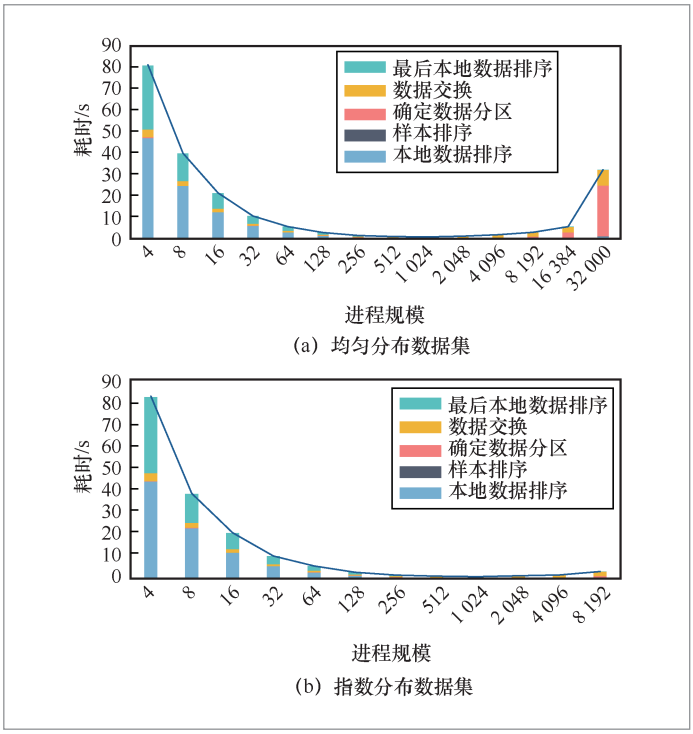

图 6 在不同分布数据集上 ScaPSRS 的运行时间

表 2 并行排序算法的运行时间

进程数量	并行排序算法运行时间/s	
	ScaPSRS_without_loadbalance	ScaPSRS
32	10.6884	10.6208
64	5.750138	5.69849
128	2.867554	2.91613
256	1.620544	1.56447
512	1.089102	1.06383
1 024	0.9441702	0.889724
2 048	1.171596	1.25763
4 096	2.02978	1.90966
8 192	2.85293	3.04061

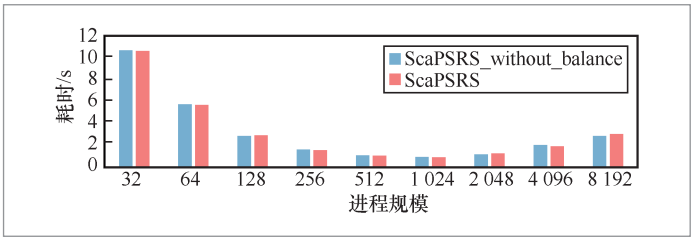

图 7 并行排序算法在均匀分布数据集上的运行时间

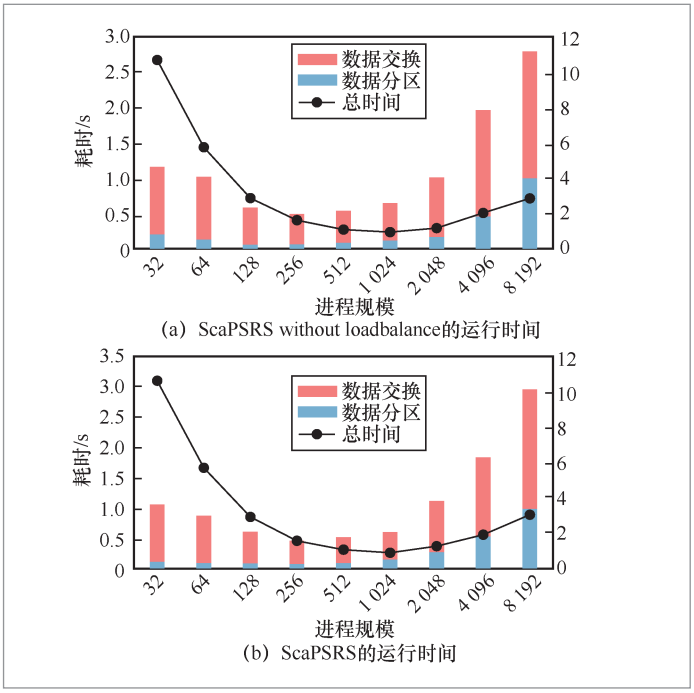

图 8 并行排序算法在均匀分布数据集上数据分区和数据交换阶段的运行时间

数据传输阶段分别花费的时间占比是相似的，变化规律也是相似的，说明增加枢轴元素的迭代更新过程并没有在确定数据分区和数据重分布阶段增加额外的耗时。

一个数据集的不平衡率[15]通常被定义为 $\mathrm{IR}=N_{\mathrm{maj}}/N_{\min}$，其中 N_{maj} 是数据集中多数类的大小，$N_{\min}$ 是少数类的大小。本文沿用不平衡率来评估数据重分布后 p 个进程中数据的负载均衡情况，具体如下。

$$\mathrm{IR}=\frac{\max\limits_i \overline{n}_i}{\min\limits_i \overline{n}} \tag{5}$$

在可接受偏差率 b 的负载均衡理想情况下，有 $\max\limits_i \overline{n}_i = n_{\mathrm{avg}}\times(1+b)$ 和 $\min\limits_i \overline{n}_i = n_{\mathrm{avg}}\times(1-b)$。那么，理想的不平衡率为 $\mathrm{IR}=(1+b)/(1-b)$。

本文实验都使用 b=0.1的可接受偏差率。那么理想情况下不平衡率IR应该不超过 $(1+0.1)/(1-0.1)=1.2222$。并行排序算法ScaPSRS_without_loadbalance和ScaPSRS分别在数据总量为 $2^{14}\times10^5$ 的均匀分布数据集上进行实验，**表3**展示了数据重分布后的实际不平衡率。从**表3**中的数据可知，当进程规模比较小时，ScaPSRS的不平衡率略优于ScaPSRS_without_loadbalance。但当进程规模变大时，ScaPSRS_without_loadbalance的不平衡问题就变得更严重了，如2 048个进程时。本文提出的算法，即能生成负载均衡数据分区的ScaPSRS，可以在不增加太多时间的前提下很好地解决这个问题。这也对最后的本地排序阶段极其有利，每个进程在数据重分布后，待本地排序的数据负载均衡。各进程的本地排序耗时相近，避免了等待一个特殊进程来对大量数据进行排序。由于并行排序算法被广泛地用于各个领域应用，这也有利于实际应用在并行排序后的阶段维持数据负载均衡。

3.5 对比实验

根据PSRS[3]和Hofmann等人[5]论文中的伪代码实现了两个并行排序算法，进行对比实验。由于Hofmann等人[5]论文并未对其算法进行命名，本文将其标注为HPCC2011。在总共有个整数的均匀分布数据集上进行了3种并行排序算法的对比实验。**图9**呈现了这3个算法的时间消耗情

况，能直观地看出ScaPSRS要比另外两个并行排序算法表现得好。这是因为本文提出的基于正则通信的正则采样策略可以有效地减少采样阶段的开销。为了保证生成的数据分区满足负载均衡，Hofmann等人[5]的分区算法在更新分割位点的过程中嵌套了3个循环，而且开销随着进程规模的增加而增加。在ScaPSRS中，枢轴元素由数据样本初始化，由于数据样本一定程度上反映了数据集的整体分布，大部分的枢轴元素已经满足负载均衡，而其他的则由每个进程同步更新，这远比HPCC2011表现得好。

表4展示了上述3个并行排序算法在不同进程规模下的具体运行时间，加粗表示同一规模下性能最佳（即耗时最短）的运行时间。由**表4**可知，ScaPSRS在大多数情况下都优于其他两种并行排序算法，尤其是在大规模下。由此可见，ScaPSRS更适合大规模扩展。

4 结束语

本文提出了一个可大规模扩展的并行排序算法ScaPSRS，以下3个改进点使得应用能够控制进程间的数据不平衡比例。第一，在采样阶段，所有进程不仅参与样本的存储与排序工作，而且通过正则通信参与枢轴元素的确定，使得工作开销和数据负载均衡；第二，ScaPSRS采用一种新颖的枢轴元素迭代更新策略，经过 $O(\log p)$ 轮沿着负载均衡方向的迭代，实现了PSRS在倾斜分布的数据中满足负载均衡的数据分区；第三，ScaPSRS改进了对存在重复值的枢轴或相邻值差为1的枢轴对数据的划分方案，加入了一步微调，能够更好地实现负载均衡。综上所述，上述所有因素使得ScaPSRS的性能超过了

表 3 并行排序算法的不平衡率

进程数量	并行排序算法不平衡率	
	ScaPSRS without loadbalance	ScaPSRS
32	1.00123	1.00083
64	1.00238	1.00214
128	1.00492	1.00447
256	1.00778	1.00793
512	1.01636	1.01637
1 024	1.50715	1.2932
2 048	569.794	1.47838

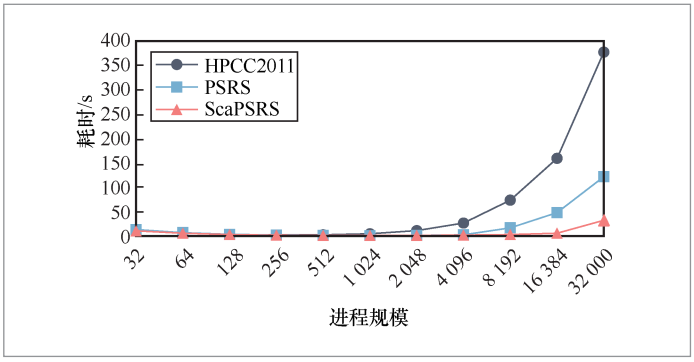

图 9 并行排序算法的对比

表 4 并行排序算法的运行时间

进程数量	并行排序算法运行时间/s		
	PSRS	HPCC2011	ScaPSRS
32	13.2805	10.4133	10.6208
64	6.9433	5.68862	5.69849
128	3.589884	3.048818	2.91613
256	1.964592	2.147516	1.56447
512	1.245028	2.437384	1.06383
1 024	0.9912062	4.558566	0.889724
2 048	1.222952	10.91112	1.25763
4 096	2.55476	26.46832	1.90966
8 192	16.8989	73.46036	3.04061
16 384	48.10605	159.2938	5.6674
32 000	121.621	377.3762	32.2569

现有并行排序算法，特别是在大规模情况下。在天河二号超级计算机上进行的大量实验表明，ScaPSRS可以成功地扩展到32 000核，其表现超越了现有先进算法，比PSRS[3]和Hofmann等人[5]提出的分区算法分别提升了3.7倍和11.7倍。

在未来的工作中，进一步优化枢轴元素的迭代更新过程，对确定数据分区和数据重分布阶段进行计算和通信重叠的优化。此外，计划在具有科学意义的数据集上与更多的并行排序算法进行更大规模的比较。

参考文献:

[1] FRAZER W D, MCKELLAR A C. Samplesort: a sampling approach to minimal storage tree sorting[J]. Journal of the ACM, 1970, 17(3): 496–507.

[2] HUANG J S, CHOW Y C. Parallel sorting and data partitioning by sampling[C]// Parallel sorting and data partitioning by sampling. Chicago: IEEE, 1983.

[3] SHI H, SCHAEFFER J. Parallel sorting by regular sampling[J]. Journal of Parallel and Distributed Computing, 1992, 14(4): 361–372.

[4] HELMAN D R, JÁJÁ J, BADER D A. A new deterministic parallel sorting algorithm with an experimental evaluation[J]. ACM Journal of Experimental Algorithmics, 1998, 3: 4.

[5] HOFMANN M, RUNGER G. A partitioning algorithm for parallel sorting on distributed memory systems[C]// Proceedings of 2011 IEEE International Conference on High Performance Computing and Communications. Piscataway: IEEE Press, 2011: 402–411.

[6] JEON M, KIM D. Parallelizing merge sort onto distributed memory parallel computers[C]//International Symposium on High Performance Computing. Berlin: Springer, 2002: 25–34.

[7] HERRUZO E, RUIZ G, BENAVIDES J I, et al. A new parallel sorting algorithm based on odd–even mergesort[C]//Proceedings of 15th EUROMICRO International Conference on Parallel, Distributed and Network–Based Processing (PDP'07). Piscataway: IEEE Press, 2007: 18–22.

[8] BATCHER K E. Sorting networks and their applications[C]//Proceedings of the Spring Joint Computer Conference. New York: ACM, 1968: 307–314.

[9] SOHN A, KODAMA Y. Load balanced parallel radix sort[C]//Proceedings of the 12th International Conference on Supercomputing. New York: ACM, 1998: 305–312.

[10] KHATAMI Z, HONG S, LEE J, et al. A load–balanced parallel and distributed sorting algorithm implemented with PGX.D[C]//Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Piscataway: IEEE Press, 2017: 1317–1324.

[11] SHAMOTO H, SHIRAHATA K, DROZD A, et al. Large–scale distributed sorting for GPU–based heterogeneous supercomputers[C]//Proceedings of 2014 IEEE International Conference on Big Data (Big Data). Piscataway: IEEE Press, 2015: 510–518.

[12] SUNDAR H, MALHOTRA D, BIROS G. HykSort: a new variant of hypercube quicksort on distributed memory architectures[C]//Proceedings of the 27th international ACM conference on International conference on supercomputing. New York: ACM, 2013: 293–302.

[13] WAGAR B. Hyperquicksort: a fast sorting algorithm for hypercubes[J]. Hypercube Multiprocessors, 1987, 1987: 292–299.

[14] HARSH V, KALE L, SOLOMONIK E.

Histogram sort with sampling[C]//The 31st ACM Symposium on Parallelism in Algorithms and Architectures. New York: ACM, 2019: 201–212.

[15] ZHU R, GUO Y W, XUE J H. Adjusting the imbalance ratio by the dimensionality of imbalanced data[J]. Pattern Recognition Letters, 2020, 133: 217–223.

作者简介

王莹（1995– ），女，中山大学计算机学院硕士生，主要研究方向为算法并行。

陈志广（1984– ），男，博士，中山大学计算机学院副教授，主要研究方向为大数据存储与处理、并行与分布式计算、高性能计算与超级计算机。

卢宇彤（1969– ），女，博士，中山大学计算机学院教授，主要研究方向为高性能并行系统软件、大数据处理技术、HPC+AI技术。

收稿日期： 2023–06–12

通信作者： 卢宇彤，yutong.lu@nscc–gz.cn

基金项目： 国家重点研发计划项目（No. 2021YFB0300103）；国家自然科学基金项目（No.62272499，No.U1911401）

Foundation Items: The National Key Research and Development Program of China（No.2021YFB0300103），The National Natural Science Foundation of China（No.62272499，No.U1911401）