Parallel Cubic Gridsort with Heap Restoration using Transputers

S C Tay K P Tan G H Ong

Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge
Singapore 0511

email: {taysengc, tankp, onggh}@iscs.nus.sg

Abstract. The proposed parallel cubic gridsort consists of two phases: preprocess and restore. In the preprocessing phase, input data are randomly placed on a cubic grid and sorted in depth-wise order, followed by column-wise order and row-wise order. As each line of data in the preprocessing phase is independent, the line-sorts in the cubic grid can be processed in parallel. In the restoring phase the remaining of the unsorted data are repeatedly moved to their be-sorted position followed by heap restoration. This can be performed in parallel by synchronizing the start time of each data movement and embedding a look-ahead mechanism in the restoration. The transputer's implementation shows that reasonably good speedup can be achieved for sorting reverse-sorted data and sorted data. As for sorting random data, a semi-parallelized cubic gridsort significantly out-performs the fully-parallelized version due to strong data-dependency in restoring phase.

Keywords: parallel sorting algorithm, grid structure, look-ahead

1. Introduction

Parallel versions of grid-based sorting algorithms have been studied recently [TSC93]. Some parallel sorting algorithms imposed on square grid structures, such as parallel square gridsort [TOT93b], parallel hybridsort [TOT93c] and parallel blocksort [TOT94] have also appeared in literature. This paper extends the studies of parallel gridsort to a higher dimension. The proposed parallel cubic gridsort (PCGS) consists of two phases: preprocess and restore. The purpose of preprocessing phase is to establish heap property on the cubic grid structure. Such a characteristic is very useful because it significantly reduces the time required to search for the minimum of the unsorted data. In the restoring phase the unsorted data are repeatedly placed at their be-sorted positions followed by heap restoration on the cubic grid. The parallelism of these two phases are exploited in our studies. The rest of this paper is organized as follows. In section 2 we will describe the notations used for PCGS and discuss the approaches used to parallelize the serial cubic gridsort. Section 3 highlights the data collision problem in the restoring phase of PCGS. Section 4 discusses the number of processors required by PCGS. Section 5 derives the sort time and speedup metrics. Section 6 contains the implementation results on a network of transputers and section 7 contains the performance of a semi-parallelized version of the algorithm. Section 8 contains some concluding remarks.

2. Two Approaches used to Parallelize Cubic Gridsort

We first describe the notations. In our representation a cubic grid (CG) is a collection of square grids, where each plane of the square grids is arranged in depth-wise order (Figure 1). A grid point in a CG is denoted by $[k,i,j]$ where k is the depth index, i is the row index and j is the column index. $G[k,i,j]$ represents the data placed at $[k,i,j]$. For simplicity we let $n = m^3$, where m is the number of data along the edge of a cubic grid containing n data. It follows that $m = \sqrt[3]{n}$. The *root* of a cubic grid refers to the grid point at left-top corner of the first plane and the grid grows to the right and to the bottom from the first plane to the last plane. The *root of a subgrid* is also defined similarly. The *son(s)* of $G[k,i,j]$ refers to $G[k+1,i,j]$, $G[k,i+1,j]$ and $G[k,i,j+1]$ if existent. The *father(s)* of $G[k,i,j]$ refers to $G[k-1,i,j]$, $G[k,i-1,j]$ and $G[k,i,j-1]$ if existent.

A CG is the logical structure of a linear array which contains the actual data to be sorted, i.e., $G[k,i,j]$ refers to the $[(k-1)m^2+(i-1)m+j]$th element in the linear array. The purpose of the PCGS is to arrange the data in ascending order from the first plane to the last plane such that data in each plane are ordered from the left to the right and from the top row to the bottom row. The proposed PCGS in the following descriptions is based on its serial version in [LTOT89].

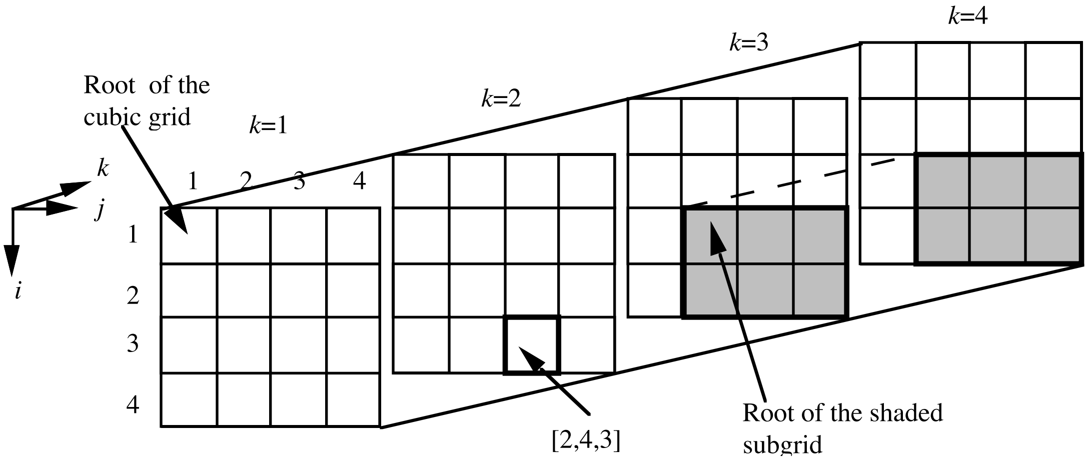


Figure 1 A cubic grid structure

2.1 Preprocessing Phase

In preprocessing phase, data on a CG are sorted in depth-wise order, followed by column-wise order and row-wise order. For simplicity we assume that bubblesort is used for each line-sort. There are m^2 lines of data in each direction. As each line of data are independent, they can be preprocessed in parallel. Let p be the number of processors. If $p = m^2$ (or $\sqrt[3]{n^2}$ equivalently), the depth-sort on each line can proceed in parallel, followed by the column-sort and row-sort. If $p < m^2$, we group the m^2 lines of data into p segments, where each segment contains either $\lfloor m^2/p \rfloor$ or $(\lfloor m^2/p \rfloor +1)$ lines of data (see Figure 2 where p=4 and m=5). Each of the segments is then assigned to one processor and all segments are then preprocessed in parallel.

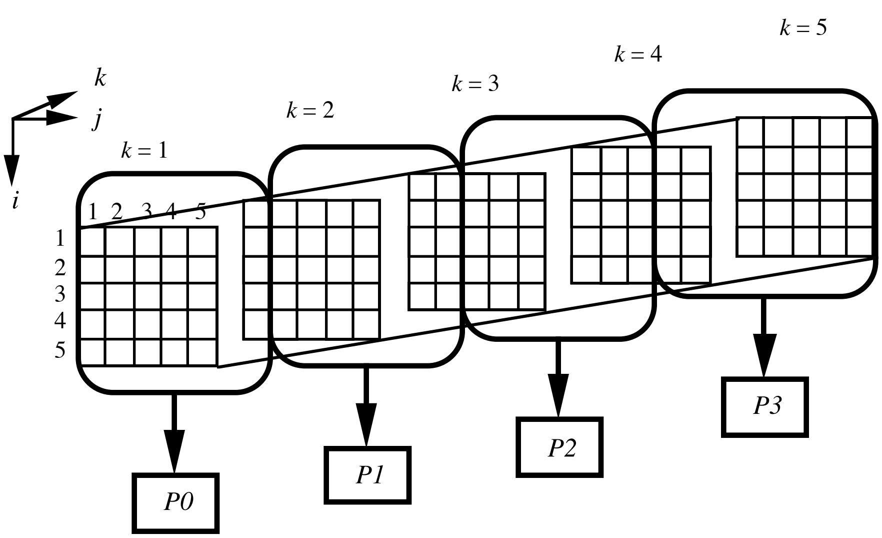


Figure 2 An example of workload distribution on 4 processors during the column-sort of a 5x5x5 cubic grid

2.2 Restoring Phase

In restoring phase, we proceed with the algorithm by repeatedly finding the minimum of the remaining data which are not sorted, moving it to its be-sorted position, and then restoring the heap property in the CG [LTOT89]. In finding the minimum we need not involve all the remaining data. Instead we find it from the roots of some subgrids, such that the union of these subgrids contains all the remaining unsorted data. Assume that all data before $[k',i',j']$ are already sorted. To fill $[k',i',j']$ with a sorted data we only have to find the minimum, denoted by u, of the following three data, if they exist:

(a) $G[k',i',j']$, which is the existing data at $[k',i',j']$
(b) $G[k',i'+1,1]$, which is the smallest data of the subgrid covering from the $(i'+1)$th row to the m-th row in the k'-th plane
(c) $G[k'+1,1,1]$, which is the smallest data of the subgrid covering from the $(k'+1)$th plane to the m-th plane

If $G[k',i',j']$ is smaller than $G[k',i'+1,1]$ and $G[k'+1,1,1]$, we proceed to the succeeding grid point at $[k',i',j'+1]$; otherwise we interchange u with $G[k',i',j']$. Let S be the subgrid originally rooted by u. As the data at the root of the subgrid S is altered, a heap restoration in S is necessary. Assume that after the data interchange, $G[k',i',j']$ is positioned at $[r,s,t]$. We proceed by repeatedly restoring the heap property of a cell of up to four data, consisting of $G[r,s,t]$ and its three sons, if existent. First we select the smallest son and then compare it to $G[r,s,t]$. If these two data are in order we stop the restoration process. Otherwise we interchange the data and perform the restoration for the next cell until no interchange operation is required in a cell, meaning that the heap property of the affected subgrid is restored. These interchange and restore operations are performed for the data in the CG but not including the grid points at the left-top corner of each plane and the bottom row in the last plane. The data are sorted when $[m,m-1,m]$ contains a sorted data and the heap property of the last row in the last plane is restored.

The restoring paths can be confined in the k'-th plane or extending to the other planes in k-direction. In restoring phase we can also process any two data in parallel if their restoring paths do not cross each other. For simplicity of descriptions, we assume that each data moves from plane to plane until it reaches the last plane. Finally, the data movements are confined in the last plane until the heap property of the affected subgrid is restored.

Let $S[k',i',j']$ be the start time of $G[k',i',j']$. Let $\tau=3c+e$, where c is the time required to compare two data, and e is the time required to interchange them. We say that a data is *active* in the restoring phase if it has started the first data movement but not completed its heap restoration yet. In PCGS, the start time refers to the time instance when two *corner data* of $G[k',i',j']$, namely $G[k',i'+1,1]$ and $G[k'+1,1,1]$ are compared. Synchronization is imposed on the first movement of each data during the restoring phase. As $G[1,1,1]$ is the root of the CG, $[1,1,1]$ has already contained a sorted data after the preprocessing phase. We therefore start the restoring phase from $[1,1,2]$. Let $S[1,1,2] = 0$ in Figure 5. We assume the worst case where data interchange is required after the data in each cell are compared. The first movement of $G[1,1,2]$ requires $2c+e$ duration, or $-c+\tau$ equivalently. Each of its subsequent movements requires $3c+e$ duration. When $t = (-c+\tau)+\tau$, $G[1,1,2]$ will have executed two data movements and therefore the grid locations $[1,2,1]$ and $[2,1,1]$ are released (Figure 3). It follows that $S[1,1,3] = -c+2\tau$. Similarly, the first movement of $G[1,1,3]$ requires $-c+\tau$ duration. When $G[1,1,3]$ is located at $[2,1,1]$, its subsequent movement will involve the selection of the smallest value of $G[3,1,1]$, $G[2,2,1]$ and $G[2,1,2]$. If the smallest value is $G[3,1,1]$, it will cause data collision with the previous active data $G[1,1,2]$, which is also positioned at $[3,1,1]$ when $t = S[1,1,3]+(-c+\tau)$ (Figure 4). Therefore $G[1,1,3]$ is delayed for ρ duration, where ρ is the delay time and $\rho = c$, to ensure that the grid location $[3,1,1]$ is released before it performs the next movement. The parallelism in the restoring phase starts when $t = S[1,1,3]$ (Figure 5). Since $G[1,1,2]$ and $G[1,1,3]$ are two steps apart when $t = S[1,1,3]+\tau$, the cells of data involved in heap restoration are independent. Therefore these two active data can proceed in parallel. Similarly, $G[1,1,4]$ is started 2τ after $S[1,1,3]$ and so on. It follows that $S[1,1,m] = -c+(m-2)2\tau$.

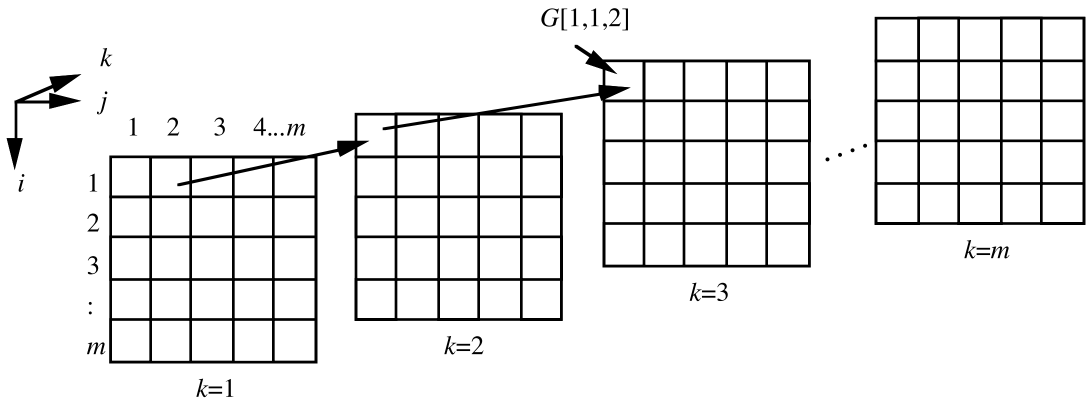


Figure 3 The location of $G[1,1,2]$ when $t = -c+2\tau$

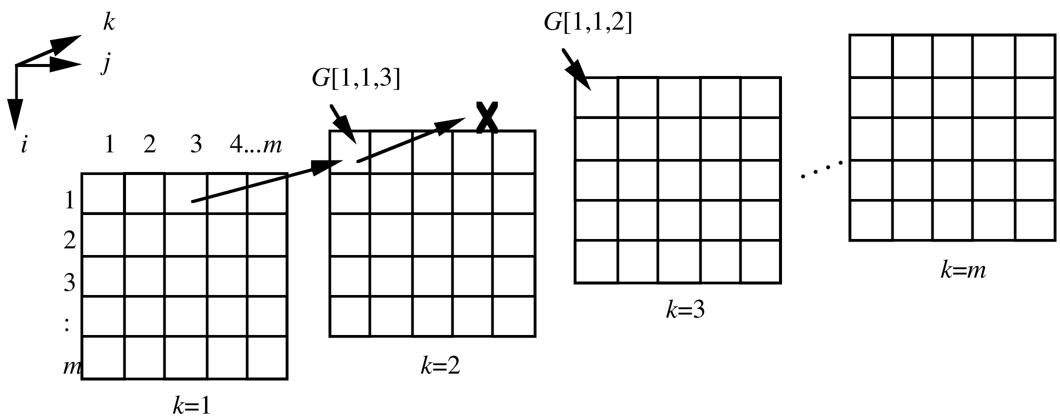


Figure 4 Data collision of $G[1,1,3]$ and $G[1,1,2]$ when $t = S[1,1,3] + (-c+\tau)$

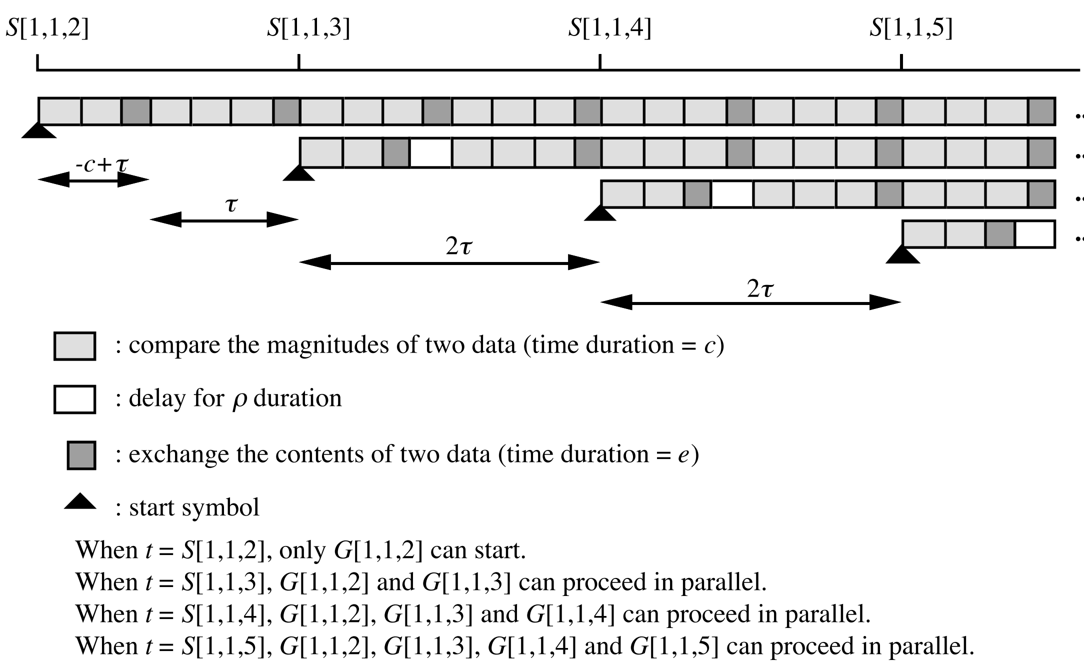


Figure 5 Activity chart for PCGS

We proceed to [1,2,1] when t=S[1,1,m]+2τ. The comparison and exchange operations involve only G[1,1,2] and the left-top corner data G[2,1,1] in the second plane, and the processing time is c+e. It is followed by a delay of 2c duration before G[1,2,1] makes the next move. The purpose of the delay is to ensure the synchronization in the restoring phase. Similarly, S[1,2,2] = S[1,2,1]+2τ. The start times of all the remaining data in the first plane follows the same descriptions in the previous paragraphs.

After the right-bottom data in the first plane is started, we proceed to G[2,1,2]. The start time of G[2,1,2] is 3τ, instead of 2τ, after S[1,m,m]. The reason is that when t = S[1,m,m]+2τ, G[1,m,m] may still occupy the grid location [3,1,1] (Figure 6). Therefore G[2,1,2] should wait for an additional τ duration so that [3,1,1] is released. In summary, the start times of the remaining data in the CG comply with the following three rules:

(a) The difference in the start times of two adjacent data in the same plane is 2τ.

(b) The difference in the start time of $G[k'',1,2]$ and $G[k''-1,m,m]$, $2 \leq k'' \leq m$, is 3τ.

(c) Each move of an active data requires τ duration; delay interval is added to the processing time to make up the τ duration if the data move requires less than three comparison operations.

Finally, the data in the CG are sorted when G[m,m-1,m] is started and the heap property in the last row of the last plane is restored.

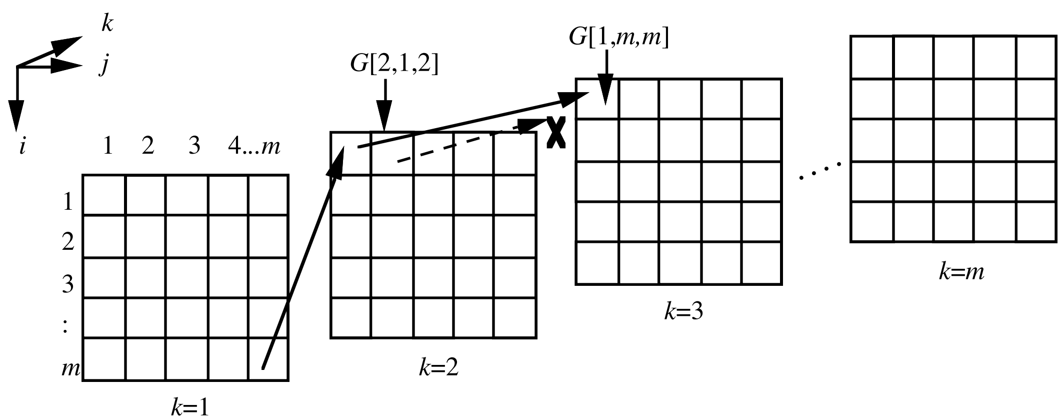


Figure 6 Data Collision of G[2,1,2] and G[1,m,m] when t = S[1,m,m]+2τ

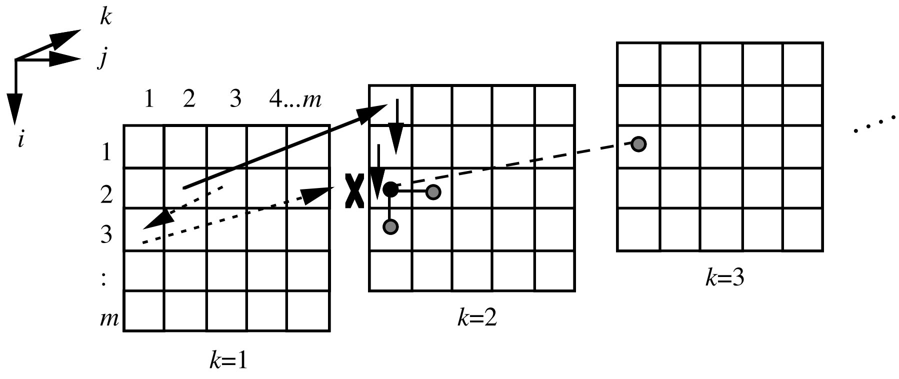
3. Data Collision Problem in the Restoring Phase of PCGS

Based on the assumption that each restoring data moves from plane to plane until it reaches the last plane, the synchronization in start times can ensure that data collision will not occur. In fact such an assumption is too strong. We observe that the correctness of the synchronous parallelism depends only on the first movement of each data. A collision-free heap restoration will be guaranteed in either of the following conditions:

(a) The first move of each active data, except those in the last plane, goes to the left-upper corner of next plane.

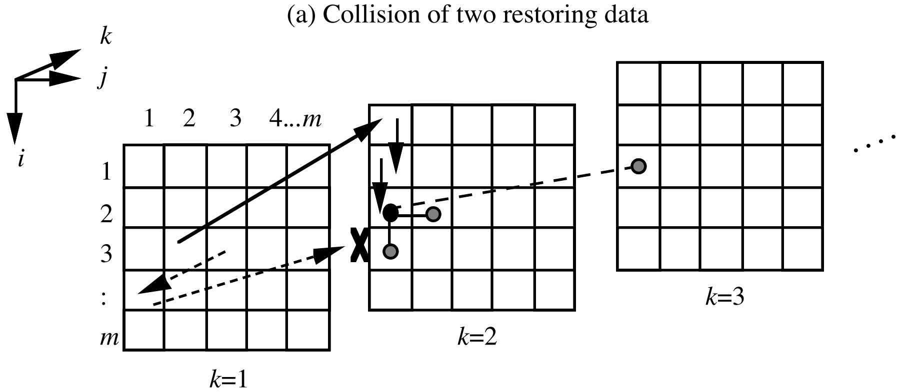
(b) The first move of each active data, except those in the last row of each plane, goes to the left corner of next row in the same plane.

In the actual sorting of random data, however, we cannot guarantee any conditions in the restoring phase. We discover that data are not sorted, and in the worst case the input data are corrupted, if the first movements of active data go to different corners (Figures 7(a), 7(b), 7(c)). To resolve this problem, we have to check for any interception of restoring paths before we process the active data in parallel.



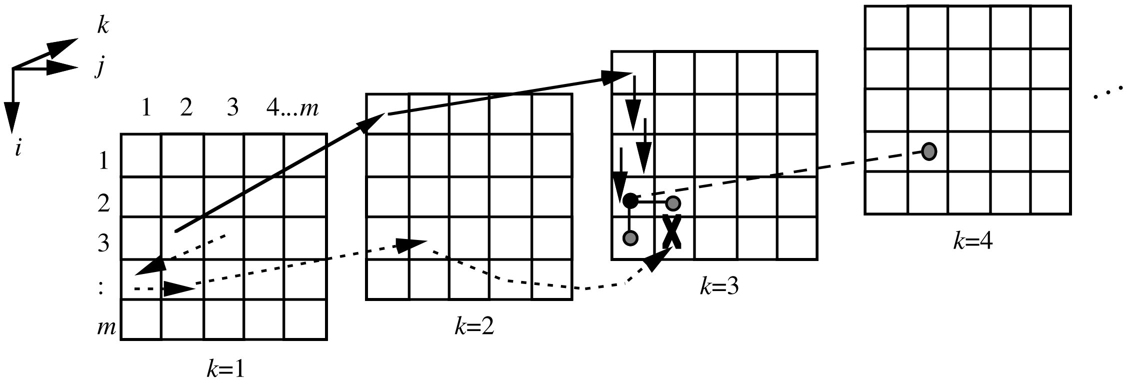
When t=S[1,2,2]+4τ, G[1,2,3] collides to G[1,2,2] at [2,3,1].

(a) Collision of two restoring data



When t=S[1,3,2]+4τ, G[1,3,3] collides to the *i*-son of G[1,3,2] at [2,4,1].

(b) Collision of a restoring data and an *i*-son



When t=S[1,3,2]+6τ, G[1,3,3] collides to the *j*-son of G[1,3,2] at [3,4,2].

(c) Collision of a restoring data and a *j*-son

Figure 7 Examples of data collision in synchronized data movements[1]

In the modified restore algorithm, the active data are also started synchronously but each restoring path is tagged with a *start time* (ST) and a *direction flag* (DIR). In the transputers' implementation, ST is assigned an increasing counter, while DIR is assigned a value, where 1 indicates that the active data moves to the next plane and 0 indicates the same plane. *Collision check* is required if there exists a mixture of 0 and 1 in the DIR flags. Otherwise we process all active data in parallel.

---

[1]The *j*-son of $G[k,i,j]$ refers to $G[k,i,j+1]$ if existent. Similarly, the *i*-son and *k*-son of $G[k,i,j]$ refer to $G[k,i+1,j]$ and $G[k+1,i,j]$ respectively if existent.

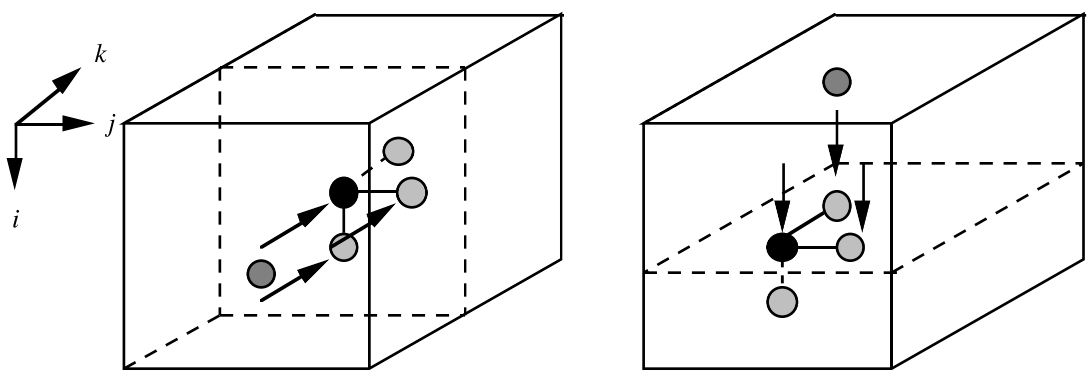
If two restoring paths are going to intercept, we resolve the problem in such a way that the older data is allowed to proceed and the younger data is delayed. The collision check proceeds as follows. First, we sort the start times of all the active data in ascending order. For discussion purpose, let $A[1,]$, $A[2]$, ..., $A[p']$, $p' \leq p$, be the active data, where $A[1]$ is the oldest and $A[p']$ is the youngest. $A[1]$ is always allowed to proceed because it is the oldest data, and therefore it will not intercept with any restoring paths before $A[1]$. We therefore check from $A[2]$ to $A[p']$. In general, for an active data, say $A[q]$, $2 \leq q \leq p'$, we check the grid locations of the cell of data rooted by $A[q]$ and the grid locations of the cells of data rooted by $A[1]$ to $A[q\text{-}1]$ for any occurrence of data collision.

We now include the additional notations used in the collision analysis. Let $A[r]$ be older than $A[q]$, i.e., $1 \leq r \leq q\text{-}1$. Let $L[A[r]]$ be the grid location of $A[r]$. Let $L[A[r]_k]$ be the grid location of the k-son of $A[r]$, and $L[A[r]_i]$, $L[A[r]_j]$, $L[A[q]_k]$, $L[A[q]_i]$ and $L[A[q]_j]$ are defined similarly. The notation $Collision_k$ indicates that two data collide in the k-direction, and $Collision_i$ and $Collision_j$ are also defined similarly. There are three ways in the k-direction where $A[q]$ can collide to $A[r]$. As shown in Figures 8(a), we have

$$Collision_k := (L[A[q]_k] = L[A[r]]) \text{ or } (L[A[q]_k] = L[A[r]_i]) \text{ or } (L[A[q]_k] = L[A[r]_j])$$

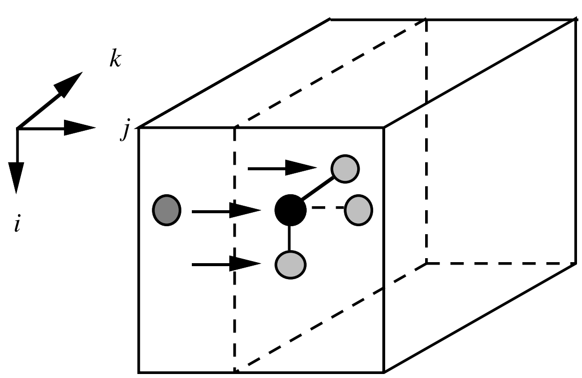
Similarly, there are also three ways of collision in the i-direction (Figure 8(b)) and j-direction (Figure 8(c)). We have

$$Collision_i := (L[A[q]_i] = L[A[r]]) \text{ or } (L[A[q]_i] = L[A[r]_k]) \text{ or } (L[A[q]_i] = L[A[r]_j])$$
$$Collision_j := (L[A[q]_j] = L[A[r]]) \text{ or } (L[A[q]_j] = L[A[r]_k]) \text{ or } (L[A[q]_j] = L[A[r]_i])$$



(a) Data collision in the k-direction

(b) Data collision in the i-direction



(c) Data collision in the j-direction

Figure 8 The data collision of two data $A[q]$ and $A[r]$

● : $A[r]$ ● : $A[q]$ ○ : son of $A[r]$

While checking the active data $A[q]$, where the index of q is changed from 2 to p', we stop the collision check immediately if any of the $A[q]$ indicates data collision to any of the older active data $A[r]$ in any of the three directions. Next, we delay $A[q]$ by τ duration, meaning that it remains in its current grid point. To maintain the synchronization in the restoring phase, all active data younger than $A[q]$, namely $A[q+1]$ to $A[p']$, also remain in their current location for τ duration. The restoring paths of all the older data, namely $A[1]$ to $A[q-1]$ are advanced in parallel during the τ duration. We call such a collision check as *look-ahead* mechanism as the active data are checked before the restoring paths are advanced in parallel.

4. Number of Processors Required by PCGS

In the preprocessing phase, PCGS requires m^2 (or $\sqrt[3]{n^2}$ equivalently) processors so that sorting on the m^2 lines of data in the k-direction can all proceed in parallel, followed by the i-direction and the j-direction.

In the restoring phase, each of the restoring data is also processed by one processor. Since parallelism is built up progressively, the number of processors required to sort the data , denoted by $N_{max}(n)$, is equal to the largest number of active data co-existing in the cubic grid.

To compute $N_{max}(n)$, we assume that all active data follow the same restoring path in the restoring phase. The length of restoring path is longer if the first data movement proceeds to the left-top corner of the next plane. The separation between two active data is at least one grid point because synchronization is imposed on the first move of each data. Therefore a rough estimate of $N_{max}(n)$ is $\lfloor (3m-3)/2 \rfloor$, which is obtained by counting the number of data placed on the alternate grid points along the three edges in the k-direction, the i-direction and the j-direction respectively. To obtain a closer estimate of $N_{max}(n)$, we impose the following 2 rules:

(a) The active data co-existing in a CG during the peak of parallelism come from more than one row.
(b) The number of active data co-existing in a CG is less than $\lceil (3m-2)/2 \rceil$, i.e., the active data can come from two or three consecutive rows only.

The reason for imposing rule (a) is that it requires more than m data to fill in the alternate grid points on three edges, and that for rule (b) is that the longest path length in a CG is less than $(m-1)+(m-1)+m$ on the three edges. The restoring path which provides the largest number of active data co-existing in the CG is not unique. By assuming that the active data come from the last two rows in the first plane and the fact that each active data cannot enter the dotted subgrid[2] shown in Figure 9, a closer estimate of the number of processors required in the restoring phase is

$$\begin{aligned} N_{max}(n) &= \lceil ((m-1)+(m-1)+(m-3))/2 \rceil \\ &= \lceil (3m-5)/2 \rceil \\ &= \lceil (3\sqrt[3]{n}-5)/2 \rceil \qquad \text{...(1)} \end{aligned}$$

---

[2]The active data come from the first plane of the dotted subgrid (see Figure 9) and therefore are not greater than all the data in the entire subgrid (data exhibit heap property after preprocessing phase [LTOT89]). During the heap restoration these data should not re-enter the subgrid or else they will not be sorted in order.

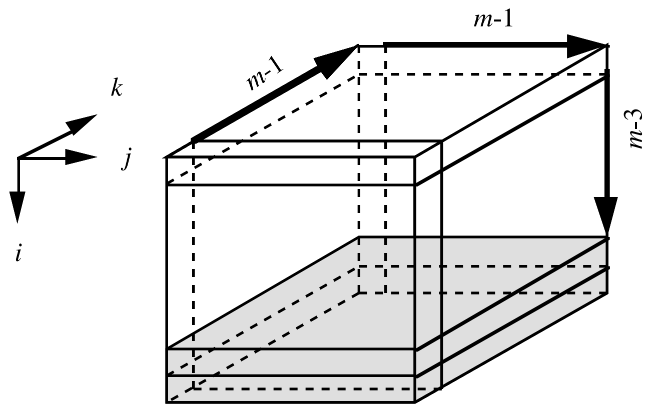


Figure 9   An example of the longest restoring path in a CG

5. Performance Analysis Constrained by *p* Processors

In this section the time complexities in preprocessing and restoring the data are analyzed separately. In addition, we use the speedup metric, which is the ratio of the sort time required by the serial version of the algorithm to the sort time required by its p-processor counterpart, to measure the performance of the parallel algorithm. In the following analysis, superscript of all notations represents either the preprocessing phase (P) or the restoring phase (R). The number of processors used by the algorithm is indicated in the subscript. The time complexity in preprocessing and restoring the data using p processors is represented by t_p^P and t_p^R respectively. SP_p^P and SP_p^R are the speedup in the preprocessing phase and the restoring phase respectively when p processors are used.

5.1 Preprocessing Phase

The preprocessing of data are performed in three directions. When $p < m^2$, the time complexity in preprocessing the data by bubblesort is

$$\begin{aligned} t_p^P &= 3 * \lceil m^2/p \rceil * m(m-1)/2 \\ &\simeq 3m^4/2p \\ &= 1.5n^{4/3}/p \qquad \text{...(2)} \end{aligned}$$

From [LTOT89] we have

$$t_1^P = 1.5n^{4/3} \qquad \text{...(3)}$$

By eqns. (2) and (3), the speedup in preprocessing the data is

$$\begin{aligned} SP_p^P &= t_1^P / t_p^P = \frac{1.5n^{4/3}}{\square 1.5n^{4/3}/p} \\ &= p \qquad \text{...(4)} \end{aligned}$$

Eqn (4) shows that the ideal-speedup in the preprocessing phase is linear provided m^2 is a multiple of p, meaning that the workload distribution in all processors is balanced. The upper bound of the speedup in preprocessing the data is m^2, or $\sqrt[3]{n^2}$ equivalently.

5.2 Restoring Phase

Assume $p \geq \lceil (3\sqrt[3]{n}-5)/2 \rceil$. There are two components in the restoring time, namely the time duration used in the synchronized data movements and the time duration used in collision checks. The following analysis is divided into two parts for different presortedness.

5.2.1 Sorting Random Data

From Figure 5, we have

$$S[1,1,2] = 0$$
$$S[1,1,3] = -c + 2\tau$$
$$\vdots$$
$$S[1,1,m] = -c + (m\text{-}2)(2\tau)$$
$$\vdots$$
$$S[1,m,m] = -c + (m^2\text{-}2)(2\tau)$$

$$S[2,1,2] = S[1,m,m] + 3\tau = -c + (m^2\text{-}2)(2\tau) + 3\tau$$
$$\vdots$$
$$S[2,m,m] = -c + 2(m^2\text{-}2)(2\tau) + 3\tau$$

$$S[3,1,2] = -c + 2(m^2\text{-}2)(2\tau) + 2(3\tau)$$
$$\vdots$$
$$S[m\text{-}1,m,m] = -c + (m\text{-}1)(m^2\text{-}2)(2\tau) + (m\text{-}2)(3\tau)$$
$$\vdots$$
$$S[m,m\text{-}1,m] = S[m\text{-}1,m,m] + (m(m\text{-}1)\text{-}2)(2\tau)$$

Once $G[m,m\text{-}1,m]$ is started, it restores the heap property along the last row in the last plane. It follows that the time duration incurred in synchronized data movements is

$$\begin{aligned} S[m,m\text{-}1,m] + (m\text{-}1)(\tau) &\simeq 2m^3\tau \\ &= 6nc + 2ne \end{aligned} \qquad \text{...(5)}$$

The time duration incurred in the collision check comprises four components.

(i) The DIR flags are checked before active data are processed. Altogether we have to check p DIR flags in each iteration, therefore the first component incurs p comparison operations.

(ii) The starting times of p active data are sorted in ascending order in each iteration. Since $p \leq 8$ in our machine platform, we use bubblesort to sort them. In the worst case we assume that the DIR flags are different in each iteration so that the sorting of start times is necessary. It follows that the second component incurs $p(p\text{-}1)/2$ comparison operations.

(iii) For each data x, we have to check all the data which are older than x for any occurrence of data collision. In the worst case data collision occurs on the last active data. Therefore, we have to scan over all the older data in each collision check, yielding $p(p\text{-}1)/2$ iterations. Each collision check incurs three comparison operations on the grid coordinates. As data collision may occur in three directions, and in each direction collision may occur on three data, the third component incurs $3*3*3*p(p\text{-}1)/2$ comparison operations.

(iv) Delay of τ duration is imposed on some active data if restoring paths cross each other, meaning that the start times of the remaining un-sorted data are delayed as well. In the worst case of each iteration, the start time is delayed by τ duration and therefore the fourth component incurs three comparison operations.

As data at the left-top corner of each plane and data at the last row of the last plane are not involved in the restoring phase, altogether we have to process $n-2\sqrt[3]{n}$ data. The total time complexity of the look-ahead mechanism becomes

$$
\begin{aligned}
&(p + p(p-1)/2 + 27p(p-1)/2 + 3)*(n-2\sqrt[3]{n}\,) \\
&\simeq 14p^2 n \qquad \text{...(6)}
\end{aligned}
$$

By eqns. (5) and (6), the time complexity of the restoring phase is $6n+14p^2n$. If the number of processors available is less than $N_{max}(n)$, we have to divide the active data into $\lceil N_{max}(n) / p \rceil$ blocks, where each block contains p active data. All the data in each block are processed in parallel but the blocks are processed serially. It follows that the time complexity in the restoring phase using p processors is

$$
\begin{aligned}
t_p^R &\simeq \lceil 3\sqrt[3]{n}/2p \rceil*(6n+14p^2n\,) \\
&= 9n^{4/3} / p + 21pn^{4/3} \qquad \text{...(7)}
\end{aligned}
$$

From [LTOT89], we have

$$t_1^R = 6.5n^{4/3} \qquad \text{...(8)}$$

By eqns. (7) and (8), the speedup in restoring phase is

$$
\begin{aligned}
SP_p^R = t_1^R / t_p^R &\simeq \frac{6.5n^{4/3}}{9n^{4/3}/p+21pn^{4/3}} \\
&= \frac{6.5}{9/p+21p} \qquad \text{...(9)}
\end{aligned}
$$

By eqns. (2) and (7), the total sort time on the CG, given p processors, is

$$
\begin{aligned}
t_p^P + t_p^R &= 1.5n^{4/3}/p + 9n^{4/3}/p + 21pn^{4/3} \\
&= \frac{10.5n^{4/3}}{p} + 21pn^{4/3} \qquad \text{...(10)}
\end{aligned}
$$

By eqns. (3), (8) and (10), the overall speedup is

$$
\begin{aligned}
(t_1^P + t_1^R)/(t_p^P + t_p^R) &= \frac{8n^{4/3}}{10.5n^{4/3}/p+21pn^{4/3}} \\
&= \frac{8}{10.5/p+21p} \qquad \text{...(11)}
\end{aligned}
$$

5.2.2 Sorting Reverse-Sorted Data and Sorted Data

For revere-sorted data and sorted data, the input data become sorted after the preprocessing phase. Therefore, there is no data interchange after the data comparisons in the cell containing $G[k,i,j]$,

$G[k,i+1,1]$ and $G[k+1,1,1]$. As altogether we have to process $n-2\sqrt[3]{n}$ data[3], it follows that the sort time on a uniprocessor is

$$t_1^R = c_1(n-2\sqrt[3]{n}) \qquad ...(12)$$

where $c_1 = 2c$ [TSC93].

Due to the delays of 2τ and 3τ in the start times, the time duration used in the synchronization of start times is $c_2(n-2\sqrt[3]{n})$, where $6c < c_2 < 9c$.

As at any time instance of the restoring phase we process only 1 active data, the time complexity of the look-ahead mechanism becomes

$$(1+1+1+1)*(n-2\sqrt[3]{n}) \simeq 4n$$

It follows that the sort time on p processors is

$$t_p^R \simeq c_2(n-2\sqrt[3]{n}) + 4n \qquad ...(13)$$

By eqns. (12) and (13), the ideal-speedup in the restoring phase is

$$\begin{aligned} SP_p^R &= t_1^R / t_p^R \\ &\simeq \frac{c_1(n-2\sqrt[3]{n})}{c_2(n-2\sqrt[3]{n}) + 4n} \\ &\simeq \frac{c_1}{c_2 + 4} \end{aligned} \qquad ...(14)$$

By eqns. (2) and (13), the total time complexity of the PCGS is

$$t_p^P + t_p^R = 1.5n^{4/3} / p + c_2(n-2\sqrt[3]{n}) + 4n \qquad ...(15)$$

By eqns. (3) and (12), the total sort time on a uniprocessor is

$$t_1^P + t_1^R = 1.5n^{4/3} + c_1(n-2\sqrt[3]{n}) \qquad ...(16)$$

By eqns. (15) and (16), the overall speedup is

$$\begin{aligned} (t_1^P + t_1^R)/(t_p^P + t_p^R) &= (1.5n^{4/3} + c_1(n-2\sqrt[3]{n})) / (1.5n^{4/3} / p + c_2(n-2\sqrt[3]{n}) + 4n) \\ &\simeq p \end{aligned} \qquad ...(17)$$

As shown in eqn. (17), the ideal-speedups of PCGS for reverse-sorted data and sorted data are close-to-linear for large n. This is due to the fact that the workload in the preprocessing phase outweighs the workload in the restoring phase.

---

[3]The total number of data, subtracted by the total number of data on the left-top corner of each plane and the total number of data on the last row of the last plane.

6. The Transputer's Implementation Results

The PCGS was tested using one, four and eight processors on a network of transputers. The data size in each test ranged from 512 to 373,248, i.e., 8^3 to 72^3. Three types of uniformly distributed data, namely reverse-sorted data, random data and sorted data are used to evaluate the performance of algorithm.

In the following analysis, we refer to the sum of sort time and inter-processors communication overhead as elapsed time. As for the speedup metric, we refer to the ratio of sort times as ideal-speedup, and ratio of elapsed times as actual-speedup. In figures ξ indicates the asymptotic value of the speedup metric obtained from the test runs.

6.1 Overall Sort Times and Overall Ideal-Speedups

This subsection shows the theoretical performance of the proposed PCGS. As shown in Figure 10, the overall sort times for a fixed n are decreased for the three types of input when more processors are used. This phenomenon shows that the serial cubic gridsort can be parallelized to reduce the sort time.

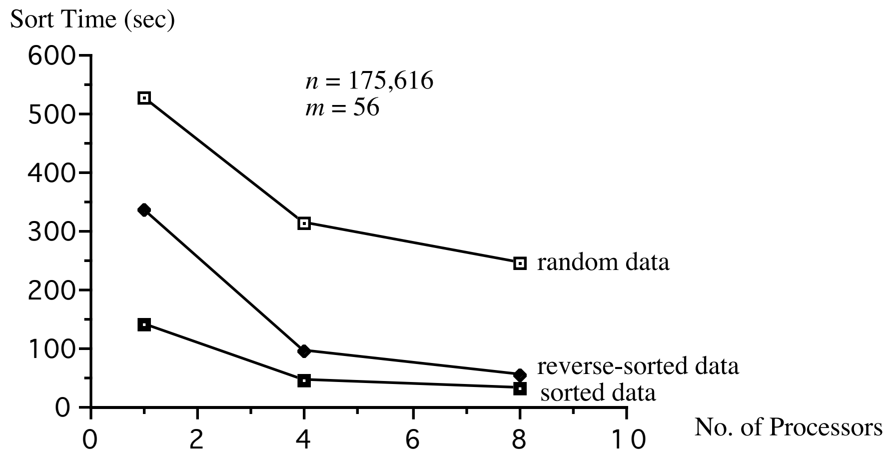


Figure 10 Comparison of overall sort time versus the number of processors

Figure 11 shows that the overall ideal-speedups in sorting random data are low for the 4-processor and 8-processor test runs. This phenomenon is due to the algorithm penalties inherent in the synchronous parallelism and the look-ahead mechanism. For reverse-sorted data and sorted data, we observe that as the number of data is increased so is the ideal-speedup.

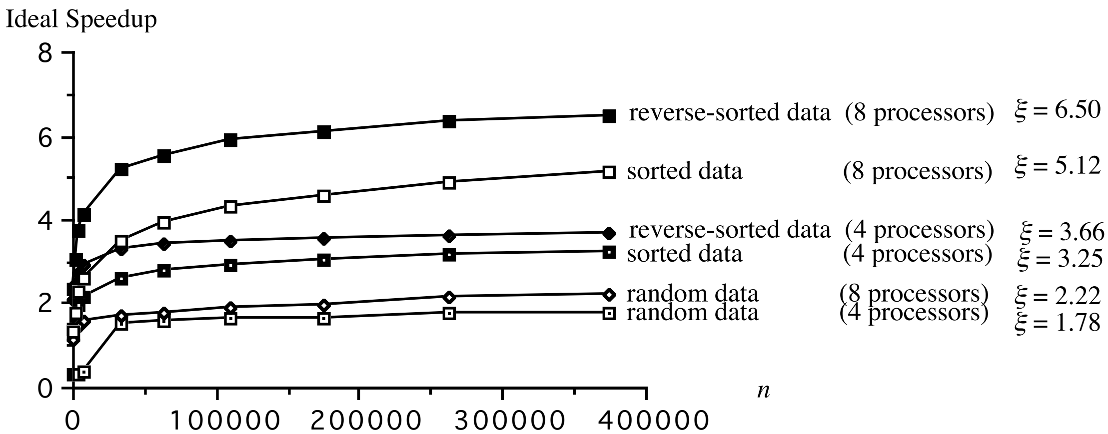


Figure 11 Comparison of overall ideal-speedups on 4 and 8 processors

6.2 Overall Elapsed Times and Overall Actual-Speedups

The discussions in this sub-section show the actual performance of the PCGS implemented on transputers as implementation overheads are included in the analysis.

Figure 12 shows that the overall elapsed time decreases as the number of processors increases for sorting reverse-sorted data and sorted data. Due to the high implementation overheads incurred in the restoring phase, the overall elapsed time for sorting random data increases when more processors are used for sorting.

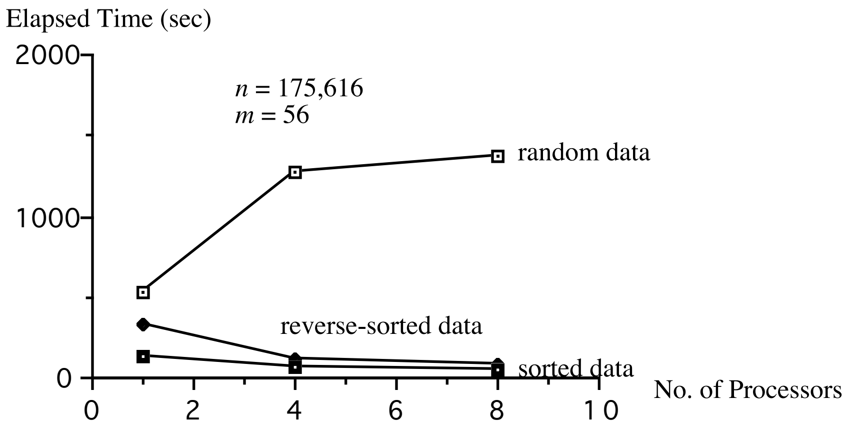


Figure 12 Comparison of overall elapsed time versus the number of processors

As shown in Figure 13, the actual-speedup in sorting random data is less than 1, meaning that it is not worthwhile to run PCGS on transputers for sorting random data. The actual-speedup for sorting reverse-sorted data is better than that for sorted data. This phenomenon is due to the heavy workload in the preprocessing of reverse-sorted data, making the total implementation overhead insignificant as compared to the overall elapsed time. On the other hand, the implementation overheads are significant for sorted data due to the light workload in the preprocessing phase.

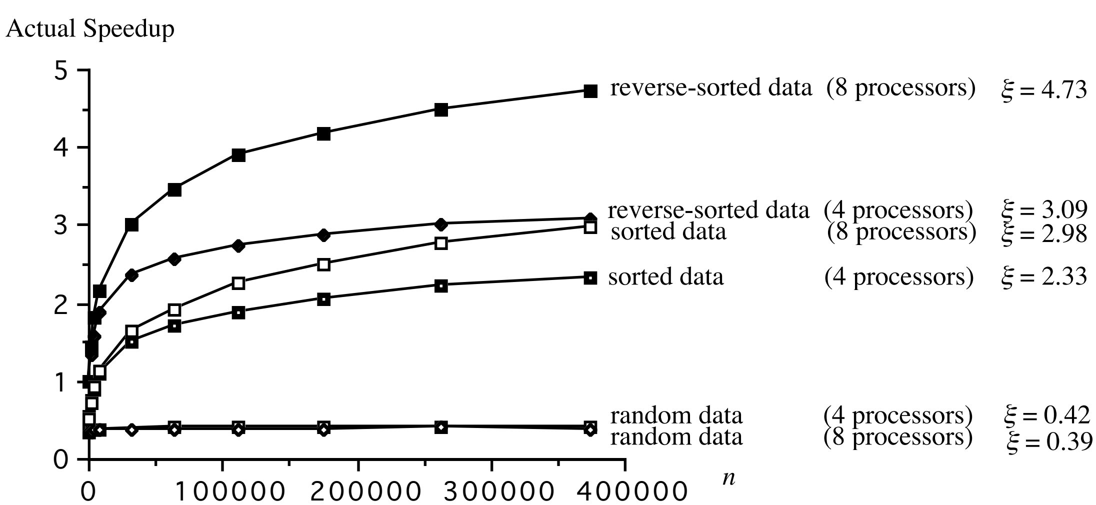
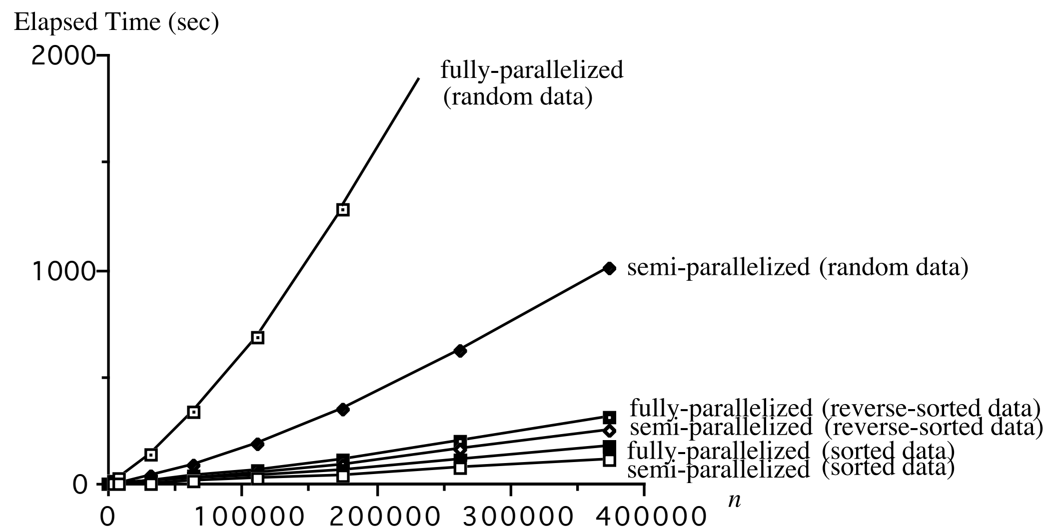


Figure 13 Comparison of overall actual-speedups versus the number of data on 4 and 8 processors

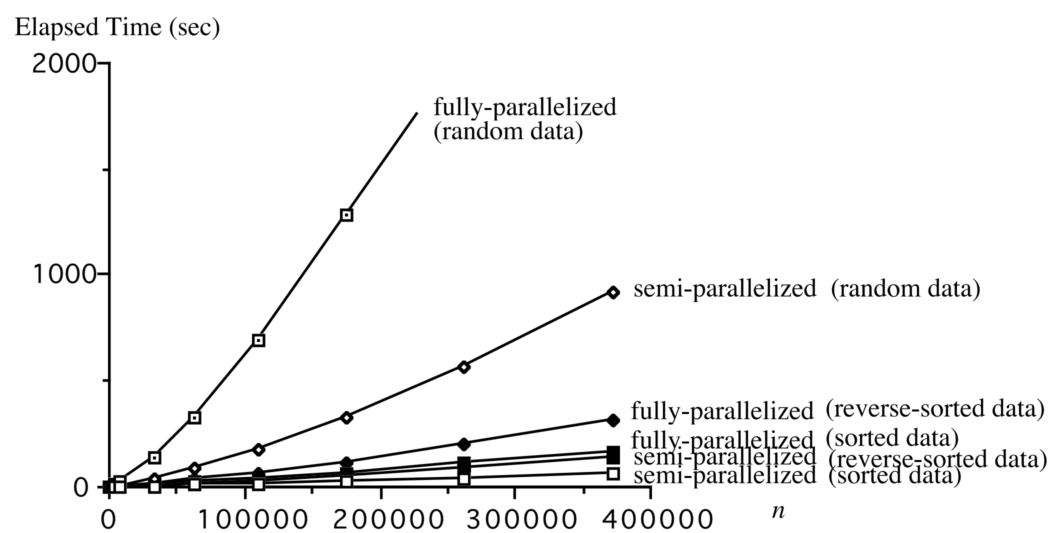
7. Semi-Parallelized Version of Cubic Gridsort

The overheads incurred in sorting random data outweighs the sort time, making the actual speedup on transputers less than 1. To eliminate such overheads we have designed a semi-parallelized cubic gridsort, where the preprocessing of data is performed by p processors, but heap restoration by only one processor. Our implementation results show that the semi-parallelized cubic gridsort performs better than the fully-parallelized version, in terms of elapsed time and actual-speedup on transputers, for all the three types of data.

The test runs show that the elapsed time of semi-parallelized cubic gridsort for sorting random data reduces tremendously as compared to that of fully-parallelized sorting algorithm (Figures 14(a) and 14(b)). The reduction in the elapsed time is also observed for reverse-sorted data and sorted data because the inherent overheads in the restoring phase are removed.



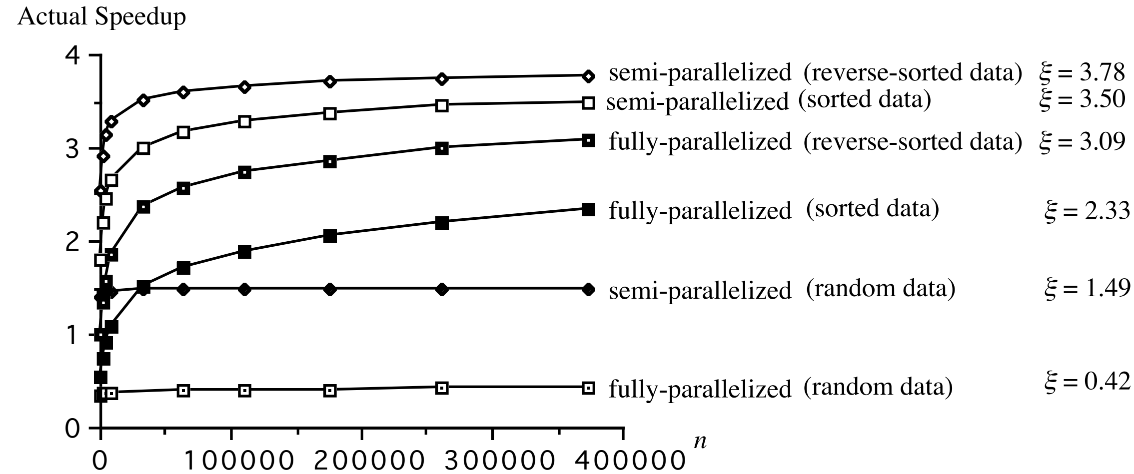
(a) 4 processors



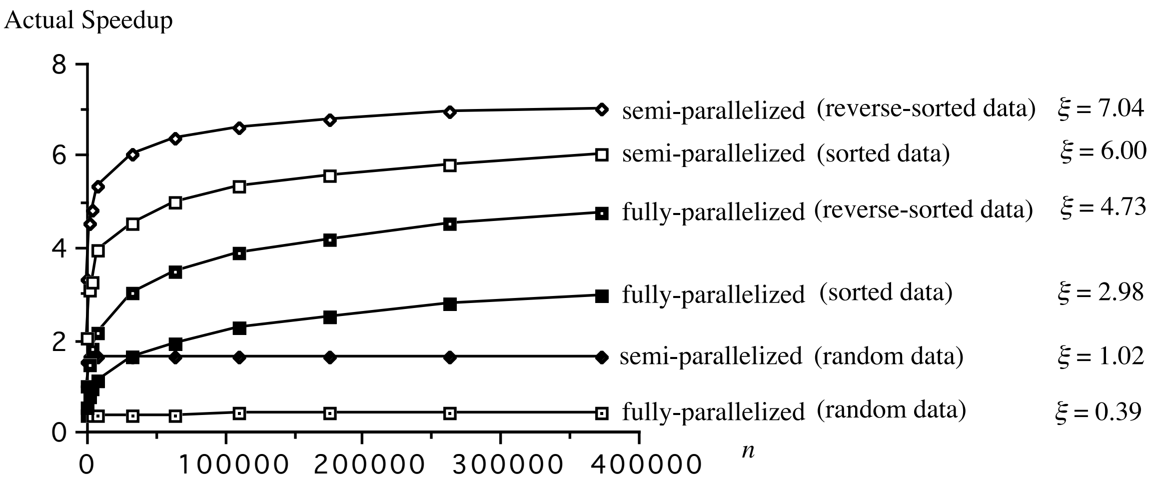
(b) 8 processors

Figure 14 Comparison of the elapsed times for fully-parallelized and semi-parallelized algorithms

The actual-speedups of the semi-parallelized algorithm for all the three types of data are also improved as shown in Figures 15(a) and 15(b). In particular, the actual-speedup in sorting random data changes from 0.42 to 1.49 in the 4-processor test run and, 0.39 to 1.62 in the 8-processor test run when n=373,248. Figures 15a and 15b also show that the actual-speedups for sorting reverse-sort data and sorted data can be close to linear for large n.



(a) 4 processors



(b) 8 processors

Figure 15 Comparison of actual-speedups of the fully-parallelized and semi-parallelized algorithms

8. Conclusion

Unlike parallel square gridsort, the synchronized data movements in the restoring phase of PCGS cannot guarantee a collision-free parallel heap restoration. Therefore we include a look-ahead mechanism in the restore algorithm, where all the active data are checked before the restoring paths are advanced in parallel. As aggravated by the delay-time in the synchronized data movements and the time duration used in the collision check, the total inherent overhead of PCGS greatly degrades the ideal-speedup of the parallel algorithm for sorting random data. Nonetheless, we have shown that reasonably good speedup is still achievable for sorting reverse-sorted data and sorted data.

The massive implementation overheads of PCGS on the transputers are obstacles to achieving linear speedup. Therefore a semi-parallelized cubic gridsort is designed. The implementation results show that such an algorithm is able to achieve good speedup on transputers for sorting reverse-sorted data, random data and sorted data.

REFERENCES

[Akl85] Selim G. Akl, *Parallel Sorting Algorithm*, 1985, Academic Press.

[Inmo91] Inmos Limited, *3L Parallel C User Guide*, July 1991.

[LTOT89] H.W. Leong, K.P. Tan, G.H. Ong, T.S. Tan, Generalized Grid-based Sorting Algorithms, *Proceeding of the International Conference on Computing and Information,* Toronto, Canada, 1989, North-Holland Publisher, Amsterdam, pp. 97-103.

[Mark91] Mark Ware Associate*, TBX05 Transputer Module Motherboard Users Guide*, October 1991.

[Rich86] Dana Richards, Parallel Sorting - A Bibliography, *ACM SIGACT News*, Summer 1986, pp. 28-48.

[TaLe85] K. P. Tan and H. W. Leong, Hybrid Sorting Technique in Grid Structure, *Proceedings of the International Conference on Foundation of Data Organization*, May 21-24, 1985, Kyoto, Japan, pp. 320-326.

[TaTa91] K.P. Tan and T.C. Tan, Hypercubic Sort with Minimal Storage, *Proceedings of International Conference for Young Computer Scientist*, July 18-20,1991, International Academic Publishers, Beijing, pp. 288-291.

[TSC93] Tay Seng Chuan, *Parallel Sorting Algorithms on Grid Structures*, MSc Dissertation, Supervisors : Dr Tan Kok Phuang and Dr Ong Ghim Hwee, Department of Information Systems and Computer Science, National University of Singapore, May 1993.

[TOT93a] Kok-Phuang Tan, Ghim-Hwee Ong, Seng-Chuan Tay, An $O(n \log_2 n)$ Hybrid Sorting Algorithm on 2-*D* Grid, *Proceedings of Fifth International Conference on Computing and Information ,* 27-29 May, 1993, Sudbury, Ontario, Canada, pp. 60-64.

[TOT93b] ———, Performance Analysis of Parallel Square Grid Sort on Transputers*, Proceedings of World Transputer Congress'93,* 20-22 September, 1993, Aachen, Germany, pp. 555-563.

[TOT93c] ———, A Parallel Sorting Algorithm based on Quick Sort using a 2-D Grid*, Proceedings of the 8-th International Symposium on Computer and Information Sciences*, 3-5 November, 1993, Instanbul, Turkey, pp. 192-200.

[TOT94] ———, Parallel Block Sort for Uniformly Distributed Data using Grid Heap, *Proceedings of International Computer Symposium*, 12-15 December, 1994, Hsinchu, Taiwan, Republic of China (forthcoming).