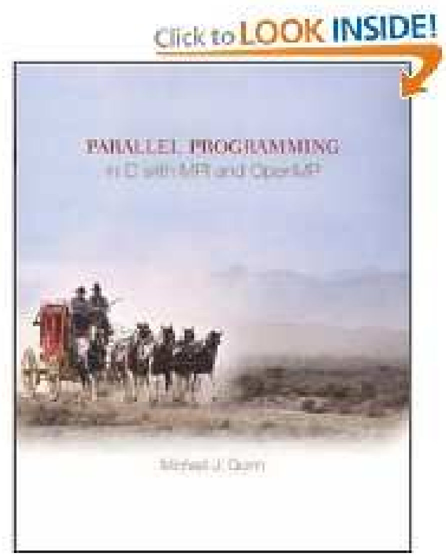
Parallel quicksort algorithms with isoefficiency analysis

Overview

- Sequential quicksort algorithm
- Three parallel quicksort algorithms
- Isoefficiency analysis
- Chapter 14 and Chapter 7.6 in *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**



Quicksort

- Given a list of numbers, we want to sort the numbers in an increasing order
  - The same as finding a suitable permutation
- Sequential quicksort algorithm: a recursive procedure
  - Select one of the numbers as pivot
  - Divide the list into two sublists: a “low list” containing numbers smaller than the pivot, and a “high list” containing numbers larger than the pivot
  - The low list and high list recursively repeat the procedure to sort themselves
  - The final sorted result is the concatenation of the sorted low list, the pivot, and the sorted high list

Example of quicksort

- Given a list of numbers: $\{79, 17, 14, 65, 89, 4, 95, 22, 63, 11\}$
- The first number, 79, is chosen as pivot
  - Low list contains $\{17, 14, 65, 4, 22, 63, 11\}$
  - High list contains $\{89, 95\}$
- For sublist $\{17, 14, 65, 4, 22, 63, 11\}$, choose 17 as pivot
  - Low list contains $\{14, 4, 11\}$
  - High list contains $\{64, 22, 63\}$
- $\ldots$
- $\{4, 11, 14, 17, 22, 63, 65\}$ is the sorted result of sublist $\{17, 14, 65, 4, 22, 63, 11\}$
- For sublist $\{89, 95\}$ choose 89 as pivot
  - Low list is empty (no need for further recursions)
  - High list contains $\{95\}$ (no need for further recursions)
  - $\{89, 95\}$ is the sorted result of sublist $\{89, 95\}$
- Final sorted result: $\{4, 11, 14, 17, 22, 63, 65, 79, 89, 95\}$

Illustration of quicksort

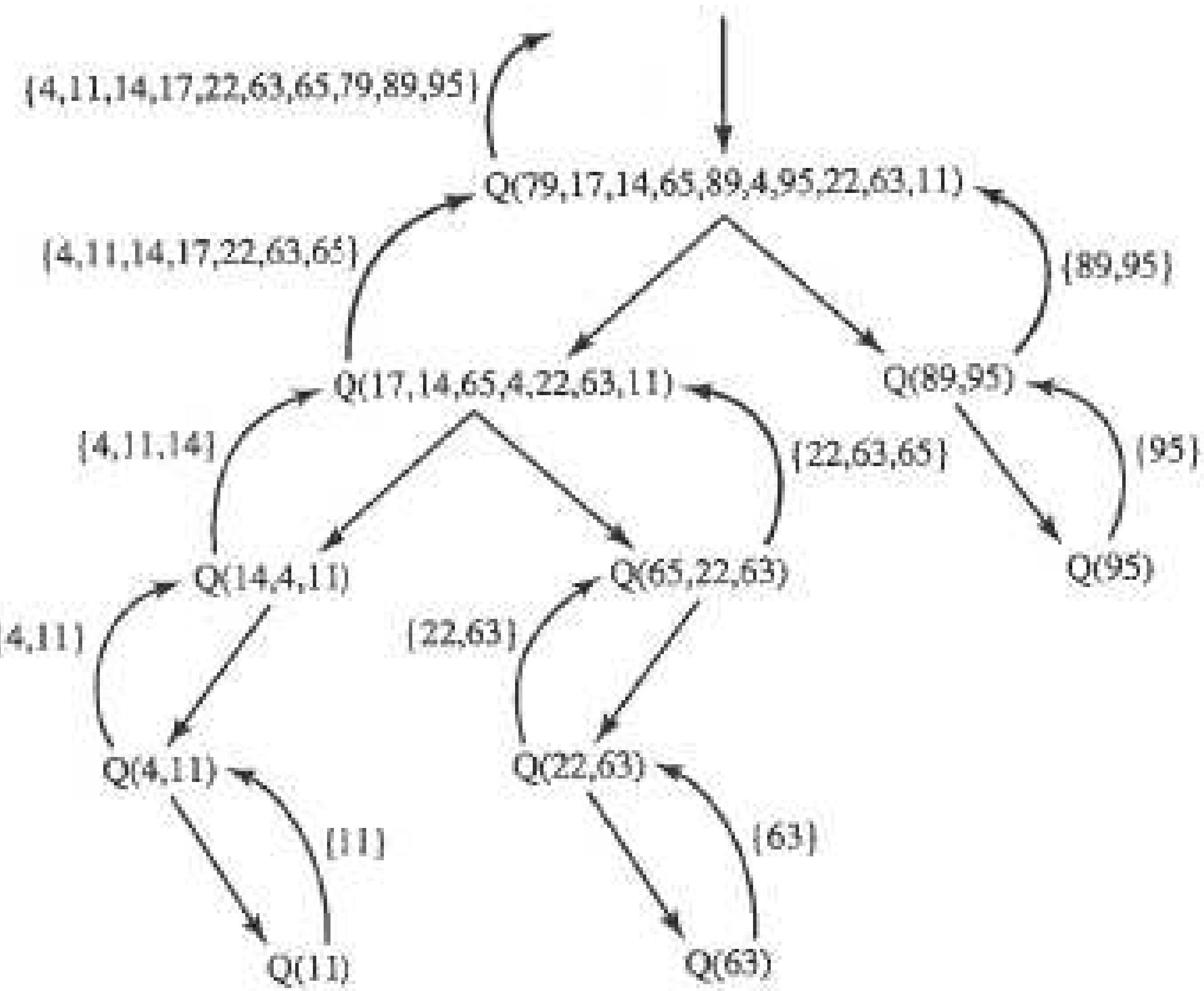


Figure 14.1 from Parallel Programming in C with MPI and OpenMP

Two observartions

- Quicksort is generally recognized, in the average case, as the fastest sorting algorithm
- Quicksort has some natural concurrency
  - The low list and high list can sort themselves concurrently

Parallelizing quicksort

- We consider the case of distributed memory
- In the beginning, each process holds a segment of the unsorted list
  - The unsorted list is evenly distributed among the processes
- When a parallel quicksort algorithm is finishied, we want
  - Each process holds a segment of the list (length may vary from process to process)
  - The list segment stored on each process is sorted
  - The last element on process i's list is smaller than the first element on process $i+1$'s list

Parallel quicksort algorithm 1

- We randomly choose a pivot from one of the processes and broadcast it to every process
- Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot
- Each process in the upper half of the process list sends its “low list” to a partner process in the lower half of the process list and receives a “high list” in return
- Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.
- Thereafter, the processes divide themselves into two groups and the algorithm recurses.
- After $\log P$ recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes.
  - The largest value on process i will be smaller than the smallest value held by process $i+1$.
  - Each process finally sorts its list using sequential quicksort

Illustration of parallel quicksort 1

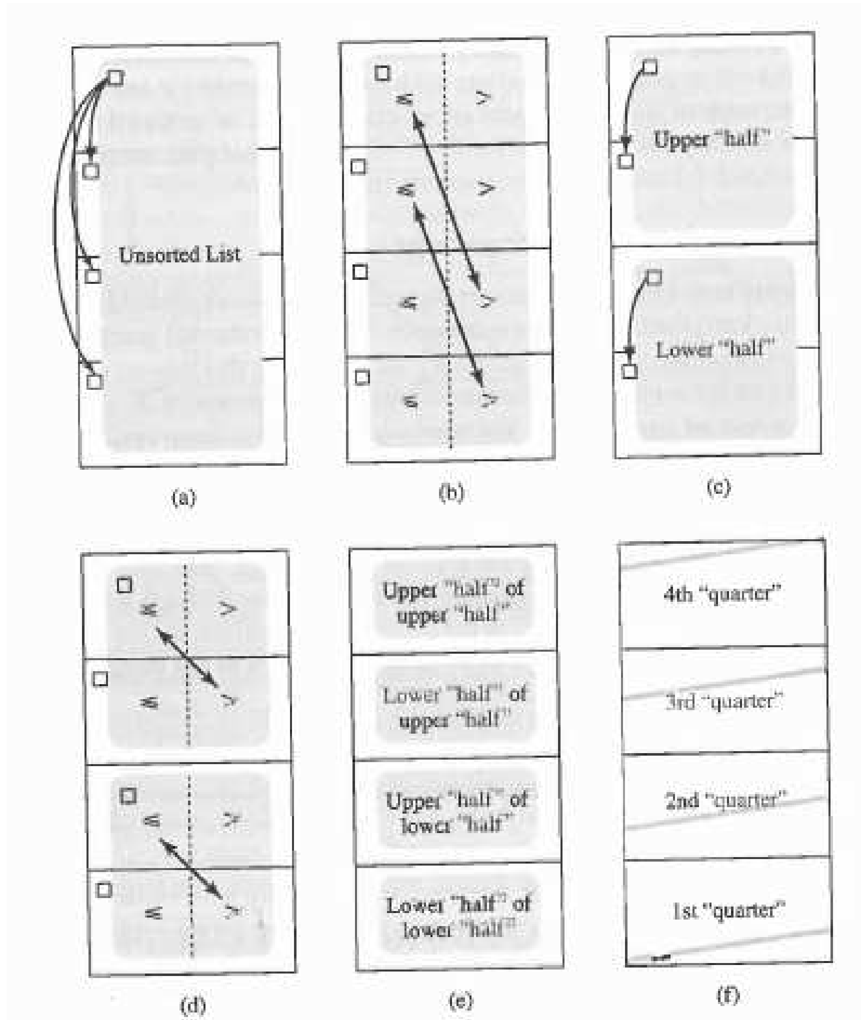


Figure 14.2 from Parallel Programming in C with MPI and OpenMP

Analysis of parallel quicksort 1

- This parallel quicksort algorithm is likely to do a poor job of load balancing
  - If the pivot value is not the median value, we will not divide the list into two equal sublists
  - Finding the median value is prohibitively expensive on a parallel computer
- The remedy is to choose the pivot value close to the true median!

Hyperquicksort – parallel quicksort 2

- Each process starts with a sequential quicksort on its local list
- Now we have a better chance to choose a pivot that is close to the true median
  - The process that is responsible for choosing the pivot can pick the median of its local list
- The following three next steps of hyperquicksort are the same as in parallel quicksort 1
  - broadcast
  - division of “low list” and high list”
  - swap between partner processes
- The 4th step is different from parallel quicksort 1
  - One each process, the remaining half of local list and the received half-list are merged into a sorted local list
- Recursion within upper-half processes and lower-half processes . . .

Example of using hyperquicksort

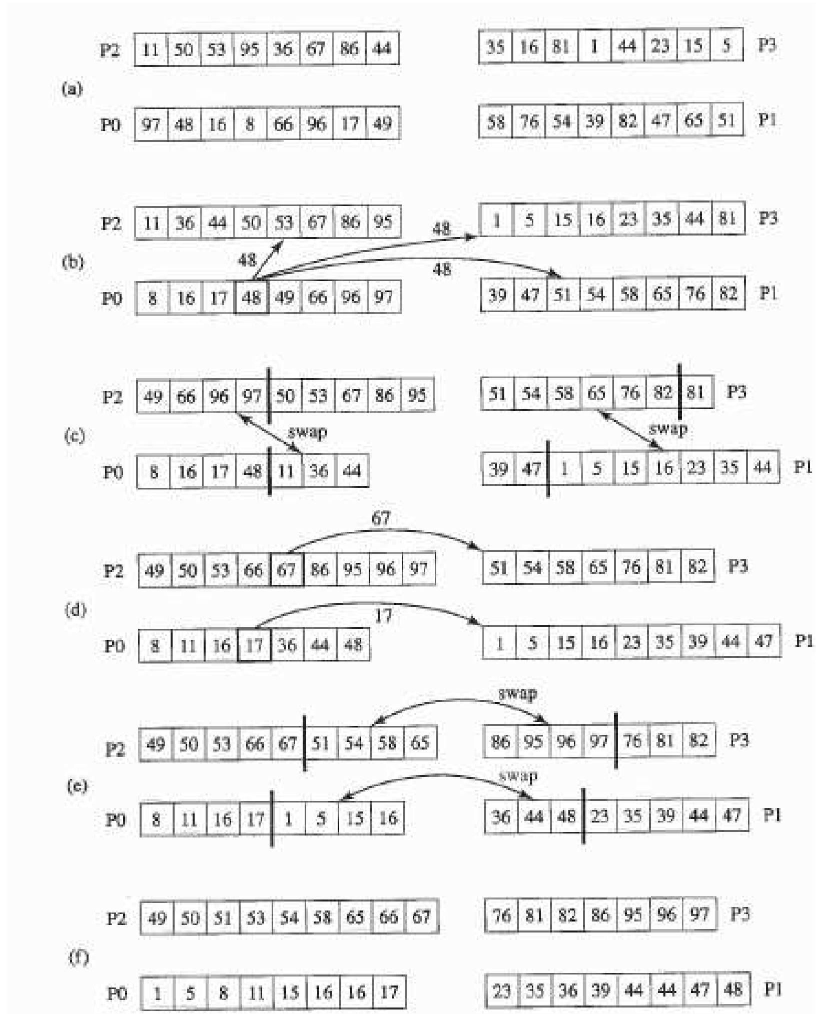


Figure 14.3 from Parallel Programming in C with MPI and OpenMP

Some observations about hyperquicksort

- $\log P$ steps are needed in the recursion
  - The average number of times a value is passed from one process to another is $\frac{\log P}{2}$, therefore quite some communication overhead!
- The median value chosen from a local segment may still be quite different from the true median of the entire list
  - Load imbalance may still arise, although better than parallel quicksort algorithm 1
  - How good exactly is hyperquicksort?
  - We can make use of the so-called *isoefficiency* relation

Isoefficiency relation

- Purpose: To find the scalability of a parallel program
- Definition of scalability: The ability to maintain efficiency when the number of processes is increased
- Recall that due to the inherently sequential work and communication overhead, parallel efficiency of a fixed-sized problem will decrease when more processes (p) are used
- To maintain the same level of efficiency, the problem size must increase together with p
- Chapter 7.6 of the textbook

Isoefficiency relation (2)

- Recall the definition of speedup

$$
\begin{aligned}
\psi(n,p) &\leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \\
&= \frac{p\left(\sigma(n)+\varphi(n)\right)}{p\sigma(n)+\varphi(n)+p\kappa(n,p)} \\
&= \frac{p\left(\sigma(n)+\varphi(n)\right)}{\sigma(n)+\varphi(n)+(p-1)\sigma(n)+p\kappa(n,p)}
\end{aligned}
$$

Isoefficiency relation (3)

- We denote $T_o(n,p)$ to be the total time spent by all processes doing work that is above $\sigma(n)+\varphi(n)$
- That is

$$T_o(n,p) = p\,T(n,p) - (\sigma(n)+\varphi(n)) = (p-1)\sigma(n) + p\kappa(n,p)$$

- Therefore

$$\psi(n,p) \leq \frac{p\,(\sigma(n)+\varphi(n))}{\sigma(n)+\varphi(n)+T_o(n,p)}$$

Isoefficiency relation (4)

- Recall efficiency $\varepsilon(n,p)$ is $\psi(n,p)$ divided by p
- Therefore

$$\begin{aligned}
\varepsilon(n,p) &\leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)+T_o(n,p)} \\
&= \frac{1}{1+\frac{T_o(n,p)}{\sigma(n)+\varphi(n)}}
\end{aligned}$$

Isoefficiency relation (5)

- Recall the sequential execution time: $T(n, 1) = \sigma(n) + \varphi(n)$
- That is,

$$\varepsilon(n, p) \leq \frac{1}{1 + \frac{T_o(n,p)}{T(n,1)}}$$

- Therefore, we can derive

$$\frac{T_o(n, p)}{T(n, 1)} \leq \frac{1 - \varepsilon(n, p)}{\varepsilon(n, p)}$$

- Consequently,

$$T(n, 1) \geq \frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)} T_o(n, p)$$

- In other words, if we want $\varepsilon(n, p)$ to remain constant (when both p and n increase), we must have $T(n, 1) \geq C\, T_o(n, p)$, where $C = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$

What has isoefficiency relation told us?

- Suppose we know the formulas for $T(n,1)$ and $T(n,p)$
- Then, $T_o(n,p) = p\,T(n,p) - T(n,1)$
- So we can find out whether $T(n,1) \geq C\,T_o(n,p)$ is satisfied

Consideration of memory usage

- Memory usage $M(n)$ (as function of n) can be a limiting factor
- Suppose the isoefficiency relation says $n \geq f(p)$
- This in fact says that
$$n \geq M(f(p))$$
- Hence $M(f(p))/p$ indicates how the memory usage per processor must increase as a function of p, in order to maintain the same level of efficiency
- We call $M(f(p))/p$ the *scalability function*
  - For example, $M(f(p))/p = \mathcal{O}(1)$ means perfect scalability
  - However, if $M(f(p))/p = \mathcal{O}(p)$, the parallel efficiency can not be maintained forever

Applying isoefficiency analysis to hyperquicksort

- Assume there are n values to be sorted, using p processes, and that $n \gg p$.
- In the beginning, each process has $\lceil n/p \rceil$ values
- Cost of initial sequential quicksort per process: $\mathcal{O}((n/p)\log(n/p))$
- In every split-and-merge step, each process keeps on average $n/2p$ values and transmits $n/2p$ values
  - Cost of communication per process: $\mathcal{O}(n/p)$
- There are $\log p$ iterations, so total communication cost per process: $\kappa(n,p) = \mathcal{O}(n \log p/p)$ (cost of broadcast is neglected because $n \gg p$)
- In addition, we have $\sigma(n) = 0$ (no inherently sequential work)
- Therefore,

$$T_o(n,p) = (p-1)\sigma(n) + p\kappa(n,p) = \mathcal{O}(n \log p)$$

More analysis of hyperquicksort

- Isoefficiency relation requires $T(n,1) \geq CT_o(n,p)$
- We have now $T(n,1) = \mathcal{O}(n \log n)$ and $T_o(n,p) = \mathcal{O}(n \log p)$
- Therefore, the following condition must satisfy:

$$n \log n \geq Cn \log p \quad \Rightarrow \quad \log n \geq C \log p \quad \Rightarrow \quad n \geq p^C$$

- Memory requirement is simple: $M(n) = n$
- Therefore the scalability function $M(p^C)/p = \frac{p^C}{p} = p^{C-1}$
  - If $C > 2$, scalability is low

Algorithm 3 – parallel sorting by regular sampling

- Parallel sorting by regular sampling (PSRS) has four phases
  1. Each process uses sequential quicksort on its local segment, and then selects data items at local indices $0, n/P^2, 2n/P^2, \ldots, (P-1)(n/P^2)$ as a regular sample of its locally sorted block
  2. One process gathers and sorts the local regular samples. The process then selects $P-1$ pivot values from the sorted list of regular smaples. The $P-1$ pivot values are broadcast. Each process then partitions its sorted sublist into P disjoint pieces accordingly.
  3. Each process i keeps its ith partition and sends the jth partition to process j, for all $j \neq i$
  4. Each process merges its P partitions into a single list

Example of using PSRS

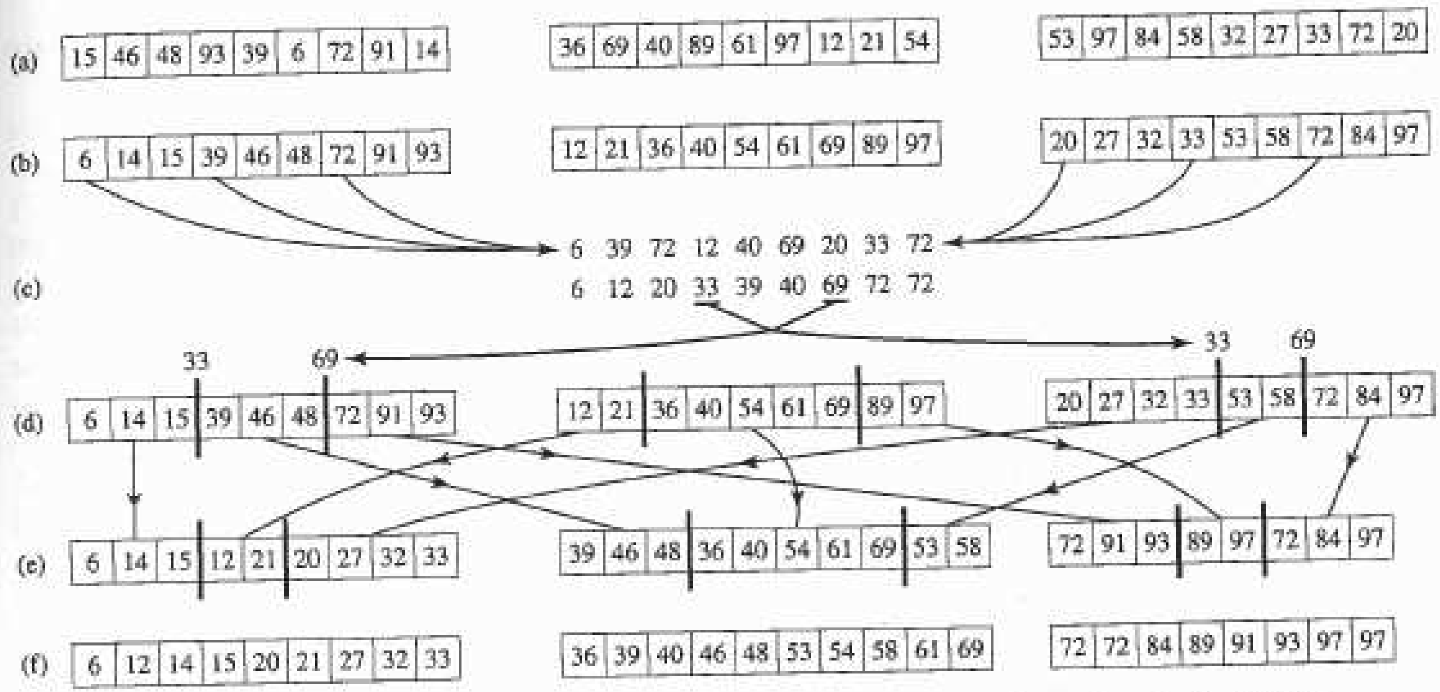


Figure 14.5 This example illustrates how three processes would sort 27 elements using the PSRS algorithm. (a) Original unsorted list of 27 elements is divided among three processes. (b) Each process sorts its share of the list using sequential quicksort. (c) Each process selects regular samples from its sorted sublist. A single process gathers these samples, sorts them, and broadcasts pivot elements from the sorted list of samples to the other processes. (d) Processes use pivot elements computed in step (c) to divide their sorted sublists into three parts. (e) Processes perform an all-to-all communication to migrate the sorted sublist parts to the correct processes. (f) Each process merges its sorted sublists.

Figure 14.5 from Parallel Programming in C with MPI and OpenMP

Three advantages of PSRS

- 1. Better load balance (although perfect load balance can not be guaranteed)
- 2. Repeated communications of a same value are avoided
- 3. The number of processes does not have to be power of 2, which is required by parallel quicksort algorithm 1 and hyperquicksort
- Additional comment: The isoefficiency analysis of PSRS is similar to hyperquicksort