

CSE 457 Assignment II

Due: 10/24/06

This assignment concerns writing efficient “single program multiple data” parallel programs using C/C++ and the MPI library for message passing. You should submit your report and any program outputs that were used for the data and analysis in your report. This assignment will be graded out of 100 points and is worth 10% of the total score for this class.

In your report, you must cite all reference materials (other than the text book and lecture notes for the class) used in preparing your solution.

You are encouraged to work with a partner and submit a single joint project report.

You are to write a program for ‘Quicksort’ in parallel using the ‘pqsort2’ version of ‘HyperQuicksort’ discussed in class (documented in a supplementary sorting worksheet). **A skeleton of the code is provided. Please use that as a starting point.**

Your goal is to study its performance empirically to identify factors that affect an efficient implementation and the scaling of performance.

Your code should be correct with respect to the following definition of ‘sorting.’ A list of size N is assumed to be sorted (in increasing order) on P processors if (1) each local list on a processor is sorted, and (2) the largest (rightmost) number on a processor i is less than or equal to the smallest (leftmost) number on processor $i + 1$.

Your program should run correctly when P is a power of 2, i.e., $P = 2^k$ for some integer k . Assume that each processor starts out with a `mylist` of size $m = N/P$; the numbers in the original list of size N are divided arbitrarily into these P lists of size $m = N/P$. You can also assume that $P < N$ and that $N = cP$ for some integer constant c .

The function `pqsort2(startprocessor, endprocessor)` is a sorting function to sort in parallel the global list comprising all local lists (`mylist`) at processors with ranks `startprocessor` to `endprocessor`.

The function `quicksort(mylist)` is the standard serial Quicksort function which returns `mylist` in sorted order.

The function `split(mylist, pivot, leftlist, rightlist)` partitions elements in `mylist` so that `leftlist` contains all numbers smaller than or equal to `pivot` and `rightlist` contains the remaining numbers.

The function `merge(mylist, list1, list2)` merges the elements in sorted lists `list1` and `list2` into `mylist` which then contains numbers from both lists in sorted order.

HyperQuicksort

```
    get number of processors
    get processor rank etc

    if leader processor
        get size of list from user
    broadcast to all processors

    initialize mylist

    quicksort (mylist)

    pqsort2 (0, number of processors -1)

end HyperQuicksort

pqsort2(startprocessor, endprocessor)

    processorgroupsize = endprocessor - startprocessor +1
    partnerdistance = processorgroupsize/2
    midprocessor = startprocessor+partnerdistnace

    if ( partnerdistance >= 1)

        if (myprocessorid = startprocessor)
            pick pivot as the median of mylist
        end if
        broadcast pivot from startprocessor to processors in the range
        startprocessor to endprocessor

        split(mylist, pivot, leftlist, rightlist)
        if (myrocessorid >= midprocessor)
            mypartner = myprocessorid - partnerdistance
            send and receive to exchange leftlist with partners rightlist
            pqsort2(midprocessor, endprocessor)
        else
            mypartner = myprocessorid + partnerdistance
            send and receive to exchange rightlist with partners leftlist
            merge(mylist, leftlist, rightlist)
            pqsort2(startprocessor, midprocessor-1)
        end if

    end if

end pqsort2
```

1. Implement your parallel sorting code.
2. Provide the expression for the memory used in your implementation in terms of N and P . Include in this count the size of `mylist` and other lists used in `split/merge/send/receive` operations. (15 points)
3. Now you will empirically test and report on the performance of your parallel code with respect to your serial implementation of Quicksort.
Place a synchronization call after initialization of `mylist` and before the sorting process begins; call this `sync1`. Likewise, place a synchronization call upon end of sorting but before verification; call this section `sync2`. Time the section beginning immediately after `sync1` and ending after `sync2`. This is the time for sorting; we will refer to this as $T(N, P)$. Using the synchronization calls, the time observed at any processor should be the same; you therefore need not time the section on every processor and then gather and report the maximum value.
Initialize `mylist` to reflect a partition of a global random list. If you call a random number generator initialized to the same seed on every processor, then it will generate the same random sequence on each processor. You could set up `mylist` at processor rank i by picking every $(i + 1)$ element in this sequence. This would ensure a reasonable random initial global list.
 - (a) Run your parallel sorting scheme for values of:
 $P = 1, 2, 4, 8, 16$ and $N = 32, 768$ for $P = 1$, $N = 65, 536$ for $P = 2$ and so on Report observed $T(N, P)$ in a table with 5 rows (one for each value of P) and 5 columns, one for each value of N . (25 points)
 - (b) Provide a table of speedups (with 25 elements) using the table above. How do the observed speedups vary across a row and a column? Explain clearly using analytic expressions for speedups. (30 points)
 - (c) Provide a table of efficiencies (with 25 elements) using the table above. How do the observed speedups vary across a row and a column? Explain clearly using analytic expressions for efficiencies. Indicate how you can vary the problem size with the number of processors to maintain iso-efficiency based on the analysis done in class. (30 points)