Assignment no. 5

Implementing the Parallel Quick-sort algorithm in parallel distributed memory environment

DCAMM PhD Course
Scientific Computing
DTU

January 2008

1 Problem setting

Given a sequence of n integer numbers, the task is to implement the Quick-sort algorithm in parallel using Fortran/C/C++ and MPI, and compare the parallel performance results for various pivot selection strategies.
The implementation must be independent of n and the number of processors p to be used, thus, n and p must be input parameters.

2 Outline of the algorithm

Algorithm 2.1 The Parallel Quick-sort algorithm

1 *Divide the data into p equal parts, one per processor*
2 *Sort the data locally for each processor*
3 *Perform global sort*
  - 3.1 *Select pivot element within each processor set*
  - 3.2 *Locally in each processor, divide the data into two sets according to the pivot (smaller or larger)*
  - 3.3 *Split the processors into two groups and exchange data pairwise between them so that all processors in one group get data less than the pivot and the others get data larger than the pivot.*
  - 3.4 *Merge the two lists in each processor in one sorted list*

4 *Repeat 3.1 - 3.4 recursively for each half.*

The algorithm converges in $\log_2 p$ steps.
For steps 2 and 3.4 you can use the `qsort` routine provided in the system libraries (see the appendix).

3 Data generation

Generate your sequence of integers to be sorted using some random number generation procedure. Some examples follow.

- Wichmann-Hill formula, $x_{n+1} = (171x_n) mod 30269$, will give uniformly distributed numbers between 0 and 30269.
- Exponentially distributed numbers can be computed using $y_n = -\frac{1}{\lambda} \ln x_n$, where $\{x_n\}$ are uniformly distributed between 0 and 1.
- Normally distributed numbers can be computed using the formula $y_n = \mu + \sigma\sqrt{-2 \ln x_i} \cos(2\pi x_i)$, where x_i and x_j are two independent uniformly distributed random numbers between 0 and 1.

You can also use the inbuilt random number generators in a proper way (see the appendix also).
One of your test runs has to be on a *backward* sorted sequence as input. You can try also almost sorted input data.
Clearly, one should use the same random number sequence for all pivot strategies in order to get fair comparisons.

4 Pivot strategies

For the numerical tests implement the following pivot strategies:

1. Select the median in one processor in each group of processors.
2. Select the median of all medians in each processor group.
3. Select the mean value of all medians in each processor group.
4. (Non-compulsory) Select all pivots (p-1) at once:
   (a) each processor selects evenly l evenly distributed elements within its data set;
   (b) all lp elements are sorted globally;
   (c) choose $p - 1$ evenly distributed elements in the above sequence and use as pivots.

You are free to test other pivot selecting techniques as well.

5 Writing a report on the results

The report has to have the following issues covered (see also the general instructions on the course web-page):

1. Describe the theoretical problem setting, the algorithm, the performance figures and expectations.
2. Describe the parallel implementation - the logical structure of the MPI machine you have in mind, data structures and other related issues.
3. Include table(s) of results for various values of n, varying number of processors p, corresponding speedup and efficiency figures.

4. Present plots of the speedup (show also the ideal speedup on the plot). The plots cam be done using Matlab, Maple or any other graphical tool you are familiar with.

5. Analyse the speedup and efficiency results in comparison with the theoretical expectations. Does the scalability of your formulation depend on the the architectural characteristics of the machine? Make relevant comparisons with the results you obtained from the experiments.

6. Add observations, comments how the particular computer architecture influences the parallel performance. Does the pair *algorithm - computer system* scales? Pay attention how to balance properly the workload per processor and the number of processors used.

7. Conclusions, ideas for possible optimizations

The report must be written in English. A listing of the program code has to be attached to the report. Standard requirements are put on the design of the code, namely, structure and comments.
You are encouraged to work in pairs. However, all topics in the assignment should be covered by each student.

6 Deadlines and credit points

The full report (paper copy!) must be submitted no later than January 30, 2008. Assignments submitted after this date will not be approved.

Success!

Maya Neytcheva

Any comments on the assignment will be highly appreciated and will be considered for further improvements. Thank you!

Appendix: Calling serial qsort

From C: qsort()

The qsort() function is an implementation of the quick-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

The base argument points to the element at the base of the table. The nel argument is the number of elements in the table. The width argument specifies the size of each element in bytes. The compar argument is the name of the comparison function, which is called with two arguments that point to the elements being compared.

The function must return an integer less than, equal to, or greater than zero to indicate if the first argument is to be considered less than, equal to, or greater than the second argument.

The contents of the table are sorted in ascending order according to the user supplied comparison function.

```
Example 1 - Program sorts.

The following program sorts a simple array:

#include #include
static int
intcompare(const void *p1, const void *p2)
{
    int i = *((int *)p1);
    int j = *((int *)p2);
    if (i > j)
        return (1);
    if (i < j)
        return (-1);
    return (0);
}
int
main()
{
    int i;
    int a[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
    size_t nelems = sizeof (a) / sizeof (int);
    qsort((void *)a, nelems, sizeof (int), intcompare);
    for (i = 0; i < nelems; i++) {
        (void) printf("%d ", a[i]);
    }
    (void) printf("\n");
    return (0);
}
```

From Fortran: qsort

```
qsort,qsort64: Sort the Elements of a one-dimensional array

qsort( array, len, isize, compar )

array   array         Input    Contains the elements to be sorted
len     INTEGER*4     Input    Number of elements in the array.
isize   INTEGER*4     Input    Size of an element, typically: 4 for
                               integer
compar  function name Input    Name of a user-supplied INTEGER*2 function.
The compar(arg1,arg2) arguments are elements of array, returning:
Negative  If arg1 is considered to precede arg2
Zero      If arg1 is equivalent to arg2
Positive  If arg1 is considered to follow arg2

demo% cat tqsort.f

          external compar
          integer*2 compar

          INTEGER*4 array(10)/5,1,9,0,8,7,3,4,6,2/, len/10/, isize/4/

          call qsort( array, len, isize, compar )

          write(*,'(10i3)') array
          end

          integer*2 function compar( a, b )
          INTEGER*4 a, b

          if ( a .lt. b ) compar = -1
          if ( a .eq. b ) compar = 0
          if ( a .gt. b ) compar = 1

          return
          end

      demo% f77 -silent tqsort.f
      demo% a.out

        0 1 2 3 4 5 6 7 8 9
```

From Fortran90: qsort

```
SUBROUTINE qsort(a, n, t)

!     NON-RECURSIVE STACK VERSION OF QUICKSORT FROM N.WIRTH'S PASCAL
!     BOOK, 'ALGORITHMS + DATA STRUCTURES = PROGRAMS'.

!     SINGLE PRECISION, ALSO CHANGES THE ORDER OF THE ASSOCIATED ARRAY T.

IMPLICIT NONE

INTEGER, INTENT(IN)    :: n
REAL, INTENT(INOUT)    :: a(n)
INTEGER, INTENT(INOUT) :: t(n)

!     Local Variables

INTEGER                :: i, j, k, l, r, s, stackl(15), stackr(15), ww
REAL                   :: w, x

s = 1
stackl(1) = 1
stackr(1) = n

!     KEEP TAKING THE TOP REQUEST FROM THE STACK UNTIL S = 0.

10 CONTINUE
l = stackl(s)
r = stackr(s)
s = s - 1

!     KEEP SPLITTING A(L), ... , A(R) UNTIL L >= R.

20 CONTINUE
i = l
j = r
k = (l+r) / 2
x = a(k)

!     REPEAT UNTIL I > J.

DO
  DO
    IF (a(i).LT.x) THEN               ! Search from lower end
      i = i + 1
      CYCLE
    ELSE
      EXIT
    END IF
  END DO
```

```
  DO
    IF (x.LT.a(j)) THEN                ! Search from upper end
      j = j - 1
      CYCLE
    ELSE
      EXIT
    END IF
  END DO

  IF (i.LE.j) THEN                     ! Swap positions i & j
    w = a(i)
    ww = t(i)
    a(i) = a(j)
    t(i) = t(j)
    a(j) = w
    t(j) = ww
    i = i + 1
    j = j - 1
    IF (i.GT.j) EXIT
  ELSE
    EXIT
  END IF
END DO

IF (j-l.GE.r-i) THEN
  IF (l.LT.j) THEN
    s = s + 1
    stackl(s) = l
    stackr(s) = j
  END IF
  l = i
ELSE
  IF (i.LT.r) THEN
    s = s + 1
    stackl(s) = i
    stackr(s) = r
  END IF
  r = j
END IF

IF (l.LT.r) GO TO 20
IF (s.NE.0) GO TO 10

RETURN
END SUBROUTINE qsort
```

From C++:

```
/*
  *  QuickSort, algorithm by C.A.R. Hoare (1960)
  *
  *  01.06.1998, implemented by Michael Neumann (neumann@s-direktnet.de)
  */

 # ifndef __QUICKSORT_HEADER__
 # define __QUICKSORT_HEADER__

 # include <algorithm>

 template <class itemType, class indexType=int>
 void QuickSort(itemType a[], indexType l, indexType r)
 {
   static itemType m;
   static indexType j;
   indexType i;

   if(r > l) {
     m = a[r]; i = l-1; j = r;
     for(;;) {
       while(a[++i] < m);
       while(a[--j] > m);
       if(i >= j) break;
       std::swap(a[i], a[j]);
     }
     std::swap(a[i],a[r]);
     QuickSort(a,l,i-1);
     QuickSort(a,i+1,r);
   }
 }

 # endif
```

Random number generation:
In Fortran: `r=irand(i)`
`irand` returns positive integers in the range 0 through 2147483647.
Similarly in C. More can be found on `http://docs.sun.com`.