Two hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Parallel Programming for Numerical Applications

Date: Friday 27th January 2012
Time: 09:45 - 11:45

Please answer any TWO questions from the FOUR questions provided.

This is an OPEN book examination.

The use of electronic calculators is permitted provided they are not programmable and do not store text.

[PTO]
Question 1.

Indirection of array accesses, via an intermediate index array, is a technique that is commonly encountered in scientific simulation codes. For example, in the following sequential loop nest (a fragment from a typical Fortran program), \( j(n) \) is the integer index array for accesses to the real array \( a(n) \):

\[
\text{C} \quad \text{pertinent variable declarations}
\]
\[
\text{C} \quad \text{INTEGER } k, n, ix
\]
\[
\text{PARAMETER } (n=1000000)
\]
\[
\text{INTEGER } j(n)
\]
\[
\text{REAL } a(n), b(n), c(n)
\]
\[
:\quad :\quad :\quad :
\]

\[
\text{C} \quad \text{loop nest code fragment starts here}
\]
\[
\text{C} \quad \text{DO } k = 1, n
\]
\[
\text{DO } ix = 1, k
\]
\[
\quad a(j(ix)) = a(j(ix)) + b(k) + c(ix)
\]
\[
\text{END DO}
\]
\[
\text{END DO}
\]
\[
:\quad :\quad :
\]
\[
\text{END}
\]

The integer index array \( j(n) \) effectively holds pointers to the elements of the real array \( a(n) \), the pointers themselves being accessed in ascending order by the innermost loop index \( ix \). Note that the value of \( j(ix) \) need not be unique. That is, \( j(ix) \) may take the same integer value, say \( v \), for several distinct values of \( ix \). However, \( v \) always satisfies \( 1 \leq v \leq n \). By this means, selected elements of \( a(n) \) are updated according to the values held in \( j(n) \).

a) Using diagrams where appropriate, describe the patterns of access to the arrays \( a \), \( b \) and \( c \) as the loop indices \( k \) and \( ix \) vary, and comment on the nature of any computational work in the loop nest that might be performed in parallel. Remember that \( n \) is large (a million).

(4 marks)

b) The following is one possible parallel implementation of the above loop nest, using a data-sharing programming model (OpenMP directives – the \textsc{Atomic} directive ensures that each element \( a(j(ix)) \) is updated by only one thread at a time):

Page 2 of 8
C

DO k = 1, n
!$OMP PARALLEL DO
   DO ix = 1, k
!$OMP ATOMIC
   a(j(ix)) = a(j(ix)) + b(k) + c(ix)
   END DO
!$OMP END DO
END DO

Describe, for a NUMA architecture like that of chronos, the behaviour of this implementation, and discuss its potential performance in terms of overheads incurred due to synchronisation, load imbalance and remote memory accesses. State clearly any assumptions you make about the contents of the array \( j(n) \).

(8 marks)

c) Suggest a more efficient parallel implementation for the given loop nest. Explain how your revised implementation reduces the overheads identified in your answer to part b).

(8 marks)
Question 2.

a) Explain what is meant by the execution time ‘overheads’ of a parallel program (you should clearly identify each different kind of overhead you might expect to occur). Describe how these overheads affect execution of the parallel program. (6 marks)

In the following OpenMP/Fortran program, any subroutine parameters which get written into during a CALL are shown underlined. The subroutines INITIALISE and DO_WORK both write to entirely independent elements of their array parameters.

```c
PROGRAM examQ2
  REAL a(100000), b(100000), c(100000)
  INTEGER i

  DO I = 1, 100000
    CALL INITIALISE(a, b)
  END DO

!$OMP PARALLEL DO
  DO I = 1, 100000
    CALL DO_WORK(a, b, c)
  END DO
!$OMP END DO

CALL FINALISE(c)
END
```

Timers were inserted around the whole program, and the code was executed on a NUMA architecture multi-core computer, one thread per core, with various numbers of active threads/cores. This yielded the following execution times (in milliseconds):

<table>
<thead>
<tr>
<th>Number of active threads/cores</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1005.28</td>
</tr>
<tr>
<td>2</td>
<td>506.31</td>
</tr>
<tr>
<td>5</td>
<td>206.91</td>
</tr>
<tr>
<td>10</td>
<td>107.11</td>
</tr>
<tr>
<td>20</td>
<td>57.21</td>
</tr>
<tr>
<td>50</td>
<td>27.27</td>
</tr>
<tr>
<td>100</td>
<td>17.29</td>
</tr>
<tr>
<td>200</td>
<td>12.30</td>
</tr>
</tbody>
</table>

The scheduling and synchronisation overheads were measured by experiment and were found to be constant at 0.01 ms per thread. Another experiment established that the sequential calls to INITIALISE and FINALISE altogether took a total of 5.27 ms to execute, thus defining the non-parallel code overhead for the program.
b) There are clearly other sources of overhead contributing to less-than-ideal parallel performance for this program. **Quantify** the amount of overhead observed as the number of threads varies, **explain** possible sources of the additional overhead, and **give details of** any experiments you would perform in order to establish which of these potential sources actually gives rise to any of the observed overhead.

(9 marks)

c) Assuming that there is only one source for the additional overhead, and making reasonable assumptions about this source and the internal nature of the two subroutines, suggest ways in which you might change the program so that it executes more quickly for larger numbers of active threads/cores.

(5 marks)
Question 3.

a) In OpenMP, the DO work-sharing directive allows the programmer to specify one of four distinct kinds of schedule, as follows:

- simple static (block scheduling)
- interleaved (block-cyclic scheduling)
- simple dynamic (chunk self-scheduling)
- guided dynamic (guided self-scheduling)

For the two code fragments given below, analyse the behaviour of each of the available kinds of schedule and suggest a suitable choice for the kind of schedule, plus chunk size, if appropriate, for each DO directive, giving reasons for your choice. Assume that the target architecture is a quad quad-core Opteron system, like chronos, and do not consider any restructuring of the code.

(i) 

```fortran
!$OMP DO PRIVATE (j) SHARED (a)
   DO j = 1, 100000
     IF isSquare(j) THEN
       CALL subr(a(j))
     END IF
   ENDDO
!$OMP END DO
```

In this code fragment, `isSquare` is a logical function that returns `.TRUE.` if and only if its argument is a perfect square, and `subr` is a subroutine which writes to its argument and whose execution time is independent of its argument and is significantly longer than the execution time of `isSquare`. (6 marks)

(ii) 

```fortran
PARAMETER (n = 40000)
REAL a(n,n), b(n)
!$OMP DO PRIVATE (i, j) SHARED (a, b, n)
   DO j = 1, n
     b(j) = 0.0
     DO i = 1, n
       a(i,j) = i + j
     END DO
   END DO
!$OMP END DO
```

```fortran
!$OMP DO PRIVATE (i, j) SHARED (a, b, n)
   DO j = 1, n
     DO i = 1, j-1
       b(j) = b(j) + a(i,j)
     END DO
     DO i = j, n
       b(j) = b(j) + a(j,i)
     END DO
   END DO
```

(7 marks)
b) Locally adaptive quadrature is described by the following fragment of Fortran code:

```fortran
SUBROUTINE QUAD(f, a, b, tol, result)
    CALL INTEGRATE(f, a, b, area, error)
    IF (error < tol) THEN
        result = result + area
    ELSE
        CALL QUAD(f, a, (a+b)/2.0, tol/2.0, result)
        CALL QUAD(f, (a+b)/2.0, b, tol/2.0, result)
    END IF
END QUAD
```

We assume that INTEGRATE calculates an approximate value, area, for the definite integral $\int_{a}^{b} f(x) \, dx$ and an estimate, error, of the (absolute) error of this approximation.

Discuss the difficulties that would need to be addressed if P stacks (as opposed to a single shared stack) were used in a dynamic scheduling, P-fold parallel implementation of this algorithm. Consider carefully the correct termination of the algorithm.

(7 marks)
Question 4.

Note: It is not necessary to use correct MPI syntax in your answer to this question, but you should make it clear what each part of your pseudo-code is intended to achieve by adding appropriate comments, plus diagrams where these help the explanation.

a) A message-passing parallel implementation of a finite difference solution to Laplace’s equation acts over a 1000 × 1000 square grid with constant boundary conditions. In a sequential Fortran implementation, the square grid would be represented by a single 2-dimensional array, x(1000, 1000). The message-passing implementation is to be executed on P MPI processes, each identified by a unique integer in the range 0 to (P-1), inclusive. Assume that P exactly divides 1000. The square grid is to be distributed onto the P processes in a 1-dimensional block partition such that elements x(k*p+1:k*p+k, 1:1000) lie in process p, where k = ⌊1000/P⌋ and 0 ≤ p < P. Assume that code already exists to establish and start executing the P processes, and that a local array A(k+2, 1000) has been allocated within each process to hold the elements of x that are assigned to lie in that process together with the ‘halo’ values in the columns to the left and right of the corresponding block of the partition.

Write appropriate MPI/Fortran pseudo-code to initialise the values of A within each process so as to achieve the equivalent of setting the gridpoints around the edge of the square (i.e. points x(1, j), x(1000, j), x(i, 1), x(i, 1000) for all 1 ≤ i, j ≤ 1000) equal to 2.0, and all other x(i, j) (i.e. the gridpoints in the interior of the square) equal to 0.0.

Hint: MPI functions MPI_COMM_SIZE() and MPI_COMM_RANK() yield the value P and the integer identifier of the current process, p, respectively.

(5 marks)

b) Once initialised, the edge gridpoint values remain constant until the program terminates. The values of the interior gridpoints are iteratively recalculated until all the current values are within a constant value, eps, of their values at the end of the previous iteration. The new value, nx(i,j), of x(i,j) is calculated using the old values, ox(1000, 1000), as follows:

\[
x(i,j) = 0.25 \times (ox(i,j-1)+ox(i-1,j)+ox(i+1,j)+ox(i,j+1))
\]

Write MPI/Fortran pseudo-code to compute the successive gridpoint values, and, when appropriate, to print the final gridpoint values and finish execution.

(15 marks)

END OF EXAMINATION