Parallel & Distributed Computing

Zaeem Yousaf

Submitted To: Sir Haroon Mahmood

Contents

1 Q1

A number of examples of SIMD, Shared memory MIMD and Distributed memory MIMD systems is given in the last slide of Lecture 2. Your task is to choose at least 2 systems from this list, each from a different category and provide details of those systems (no. of cores, architecture, memory technology, communication network, its usage and applications etc.)

1.1 Ans

1: The Alenia Quadrics (Distributed-memory SIMD systems) 2: The IBM 9076 SP2 (Distributed-memory MIMD systems)

1.1.1 The Alenia Quadrics

Year of production: 1994-1996. **Machine type:** Processor array **Models:** Quadrics Qx, QHx, x = 1, 16 **Front-end:** Almost any Unix workstation **Operating system:** Internal OS transparent to the user, Unix on front-end **Connection structure:** 3-D mesh, (see remarks) **Compiler:** Fortran 77 compiler with some Fortran 90 and some proprietary array extensions

System parameters: **Model:** Qx QHx **Clock cycle:** 40 ns 40 ns **Per Proc:** (32-bits) 50 Mflop/s 50 Mflop/s **Memory:** 2 GB to 32 GB **No. of processors:** form 8-128 to 128-2048

The Quadrics is a commercial spin-off of the APE-100 project of the Italian National Institute for Nuclear Physics. Systems are available in multiples of 8 processor nodes in the Q-model where up to 16 boards can be fitted into one crate or in multiples of 128 nodes in the QH-model by adding up to 15 crates to the minimal 1-crate system. The interconnection topology of the Quadrics is a 3-D grid with interconnections to the opposite sides (so, in effect a 3-D torus). The 8-node floating-point boards (FPBs) are plugged into the crate backplane which provides point-to-point communication and global control distribution. The FPBs are configured as 2^3 cubes that are connected to the other boards appropriately to arrive at the 3-D grid structure.

1.1.2 The IBM 9076 SP2

Year of production: 1996-1997. **Machine type:** RISC-based distributed-memory multi-processor cluster **Models:** Quadrics Qx, QHx, x = 1, 16 **Front-end:** Almost any Unix workstation **Operating system:** AIX (IBMs Unix variant) **Connection structure:** switch **Compilers:** XL Fortran (Fortran 90), HPF, XL C, C++

System Parameter **Clock cycle:** 6.25 ns **Per Proc:** (64-bits) 640 Mflop/s **Memory/node:** 1/2 GB **Memory/maximal:** 1 TB **Point-to-point:** 150 MB/s **No. of processors:** from 8 to 512

Application:

Applications can be run using PVM or MPI. Also High Performance Fortran is supported, both a proprietary version and a compiler from the Portland Group. IBM uses its own PVM version from which the data format converter XDR has been stripped. This results in a lower overhead at the cost of generality. Also the MPI implementation, MPI-F, is optimised for the SP2 systems.

Presently, three types of node processors are available for the SP2. The P2SC thin nodes, P2SC wide nodes and the 604e “High node”. The latter is in fact a cluster of up to 8 604e processor connected to a common memory in an SMP fashion. The processors have a clock cycle of 5 ns, however, they can only deliver 2 floating-point results/cycle at maximum, while the P2SC nodes can deliver 4 floating-point results per clock cycle. as the fastest of the P2SC nodes has a clock cycle of 6.25 ns, it has a peak performance of 640 Mflop/s while a single 604e processor can attain 400 Mflop/s at maximum. It seems that the 604e processor-based systems are more targetted to the commercial market. The P2SC-based systems are primarily meant for the technical/scientific market. In the parameter list above we included the presently fastest P2SC processor.

The SP2 configurations are housed in columns that each can contain 8–16 processor nodes. This depends on the type of node employed: a thin nodes occupies half of the space of a wide node. Although the processors in these nodes are basically the same there are some differences. At the time of writing no 6.25 ns clock wide nodes were available yet. The fastest in this class feature a clock cycle of 7.4 ns giving a peak speed of 540 Mflop/s. Also thin nodes with a clock cycle of 8.3 ns are still around, having a peak speed of 480 Mflop/s.

2 Q2

2.1 Q2(a)

```
max degree of concruncy = 5
task1, task2, task3, task4, task5
```

2.2 Q2(b)

critical path

task1 -> task6 -> task8-> task10-> task12-> task13

2.3 Q2(c)

critical path length

task1(10) -> task6(7) -> task8(6)-> task10(5)-> task12(9)-> task13(4)
=> 10 + 7 + 6 + 5 + 9 + 4 = 41

2.4 Q2(d)

max possible speedup assuming large number of processors

= sequentialTimeExecution/parallelTimeExecution

= (10+10+10+10+10+7+7+5+6+4+5+9+4) / (critical path length)
= 97 / 41 = 2.37

2.5 Q2(e)

Minimum number of processes needed to obtain the maximum possible speedup
at least 5 processors

2.6 Q2(f)

Maximum speed up if number of processes are limited to 2
p_1 (t_1) p_2 (t_2) -> 10
p_1 (t_4) p_2 (t_5) -> 10
p_1 (t_3) p_2 (t_6) -> 10
p_1 (t_9) p_2 (t_7) -> 7
p_1 (t_8) p_2 (t_{11}) -> 6
p_1 (t_{10}) p_2 (idle) 5
p_1 (t_{12}) p_2 (idle) -> 9
p_1 (t_{13}) p_2 (idle) -> 4
total sequential execution time = 10 + 10 +10 + 7 + 6 + 5 + 9 + 4
= 61
speedup = 97 / 61 = 1.6

3 Q3

```
#include<iostream>
#include<time.h>
using namespace std;

void printArray(int A[], int size)
{
  for (auto i = 0; i < size; i++)
    cout << A[i] << " ";
}

class Matrix{
public:
  int rows;
  int cols;
  int **matrix;
```

```
  Matrix(int r, int c){
    rows = r;
    cols = c;
    matrix = new int*[rows];
    for(int i=0; i < rows; i++){
      matrix[i] = new int[cols];
    }
  }
  ~Matrix(){
    for(int i=0; i < rows; i++){
      delete(matrix[i]);
    }
  }

  void print(){
    for(int i=0; i< rows; i++){
      for(int j=0; j < cols; j++){
 cout << matrix[i][j] << " ";
      }
      cout << endl;
    }
  }

  void populate(){
    for(int i=0;i<rows;i++){
      for(int j=0;j<cols;j++){
 matrix[i][j]=rand() % 50;
      }
    }
  }
};

struct ArrayData{
  int *array;
  int begin;
  int end;
  int sum;
  ArrayData(int *arr, int b, int e, int s){
    array = arr;
    begin = b;
```

```
    end = e;
    sum = s;
  }
  void print(){
    cout << "Array\n";
    printArray(array,end-begin+1);
    cout << endl;
    cout << "begin: " << begin << " end " << end << " sum: " << sum <<endl;
  }
};

void merge(int array[], int left, int mid, int right)
{
  auto const subArrayOne = mid - left + 1;
  auto const subArrayTwo = right - mid;

  // Create temp arrays
  auto *leftArray = new int[subArrayOne],
    *rightArray = new int[subArrayTwo];

  // Copy data to temp arrays leftArray[] and rightArray[]
  for (auto i = 0; i < subArrayOne; i++)
    leftArray[i] = array[left + i];
  for (auto j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

  auto indexOfSubArrayOne = 0, // Initial index of first sub-array
    indexOfSubArrayTwo = 0; // Initial index of second sub-array
  int indexOfMergedArray = left; // Initial index of merged array

  // Merge the temp arrays back into array[left..right]
  while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
    if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
    }
    else {
      array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
    }
```

```
      indexOfMergedArray++;
    }
    // Copy the remaining elements of
    while (indexOfSubArrayOne < subArrayOne) {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
      indexOfMergedArray++;
    }
    // Copy the remaining elements of
    while (indexOfSubArrayTwo < subArrayTwo) {
      array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
      indexOfMergedArray++;
    }
}

void *mergeSort(void * args)
{
  ArrayData *mat = (ArrayData *) args;
  if (mat->begin >= mat->end){
    int *element = new int(mat->array[mat->begin]);
    return (void *) element;
  }

  auto mid = mat->begin + (mat->end - mat->begin) / 2;
  ArrayData *left = new ArrayData(mat->array, mat->begin, mid,0);
  ArrayData *right = new ArrayData(mat->array, mid+1, mat->end,0);

  pthread_t *twoThreads = new pthread_t[2];
  pthread_create(&twoThreads[0],NULL,mergeSort,left);
  pthread_create(&twoThreads[1],NULL,mergeSort,right);

  int  *left_sum = new int;
  *left_sum = 0;
  int *right_sum = new int;
  *right_sum= 0;
  pthread_join(twoThreads[0],(void **) &left_sum);
  pthread_join(twoThreads[1],(void **) &right_sum);

  int *total = new int;
```

```
  *total = (*left_sum) + (*right_sum);

  merge(mat->array, mat->begin, mid, mat->end);
  return (void *) total;
}


int main()
{
  srand(time(0));

  cout << "enter rows: ";
  int rows;
  cin >> rows;

  cout << "enter cols: ";
  int cols;
  cin >> cols;

  Matrix *mat = new Matrix(rows,cols);
  mat->populate();

  mat->print();

  int total = 0;
  int *row_sum = 0;
  pthread_t *threads = new pthread_t[rows];
  for(int i=0; i< mat->rows; i++){
  ArrayData *data = new ArrayData(mat->matrix[i],0,mat->cols-1,0);
  pthread_create(&threads[i], NULL, mergeSort, data);
  }

  for(int i=0; i< mat->rows; i++){
    pthread_join(threads[i],(void **)&row_sum);
    total = total + *row_sum;
  }


  cout << "\nSorted array is \n";
  mat->print();
```

```
    cout << "\ntotal sum: " << total << endl;

    return 0;
}
```