EE4-63
EE9-AO14

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2013

MSc and EEE/EIE PART III/IV: MEng, Beng and ACGI

HIGH PERFORMANCE COMPUTING FOR ENGINEERS

Thursday, 16 May 10:00 am

Time allowed: 2:00 hours

There are FOUR questions on this paper.

Answer THREE questions.

All questions carry equal marks.

Please use a SEPARATE ANSWER BOOK for each question.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible	First Marker(s) :	D.B. Thomas
	Second Marker(s) :	M.M. Draief

1. a) Consider a field of wheat with area F, which must be harvested by P harvesters (e.g. people with scythes, or mechanised combine-harvestors). Each harvester can independently harvest an area A per second. For each field the harvester must spend $O(\log P)$ time planning who will work where, to avoid covering the same area.

i) Use this example to illustrate Amdahl's law. [3]
Answer : Amdahl's law considers problems where there is a parallel fraction, which can be optimised or improved, and a serial fraction, which cannot be significantly improved. It states that there are diminishing returns to acceleration of the parallel fraction, as eventually performance will be limited by the serial fraction.

In the case of the harvesters, the planning stage is serial, while the harvesting is parallel. If we were to improve the performance of the harvesters (i.e. reduce A), then eventually the planning stage will come to dominate, limiting performance. Similarly, if we were to increase P instead of reducing A, then for the same size field the $\log P$ cost would come to dominate (and in fact would grow slightly).

ii) Use this example to illustrate Gustafson's law. [3]
Answer : Gustafson's law assumes that as performance increases (either through optimisation or through parallelism), then people try to process bigger problem sizes, unlike Amdahl's law, which assumes a fixed problem size.

In this case, the size of the problem is the area of the field F. The time taken in the harvesting stage is $O(F/(P \times A)$, while the time taken in planning is only $O(\log P)$. If we define $T(P,A)$ as the total time taken, then we have: for fixed P, as $F \to \infty$ then $O(T(P,A)) \to O(F/A)$; while for fixed A, as $F \to \infty$ then $O(T(P,A)) \to O(\log P + 1/P)$.

iii) Give examples of three metrics with ratio scales which could be applied to the harvesters. [2]
Answer :

- *Capital cost : Number of pounds to buy each harvester.*
- *Running cost : Number of money per hours running time.*
- *Harvesting performance (i.e. A) : Meters-squared harvested per second.*
- *Energy efficiency : Amount of fuel (e.g. liters of petrol) consumed per area harvested (metres squared).*

iv) Give an example of a categorical harvester metric. [1]
Answer :

v) Manufacturer : Which company provides the mower/scythe?

vi) Automatic vs manual : does it require a human operator?

b) An old-fashioned book publisher uses purely mechanical processes - no electricity, let alone computers - but is interested in maximising the number of books processed per year. For each w-word page of a p-page book the following steps are required:

- The managing editor of the publisher checks each book is appropriate, taking $O(\sqrt{p})$ time to scan the author's hand-written pages.

- Each word in the manuscript is manually spell-checked by an employee against a paper dictionary containing d words. Any unknown words are replaced with the closest word found in the dictionary.
- The manuscript is type-set into pages of fixed-width letters, by converting each hand-written word into a block of type, and sequentially filling up each page.
- The type-set pages are collated into sorted order and sent to the printer.

i) Given an unlimited supply of employees, what will be the bottleneck limiting the throughput of the publisher? [1]
Answer : The sequential bottleneck is the managing editors initial pass through the book, so the maximum throughput will be $O(p^{-1/2})$.

ii) If the publisher fires all employees apart from the managing editor, what is the maximum throughput of the publishing house? [2]
Answer : The complexity of the different stages is:

- *Editor : $O(\sqrt{p})$.*
- *Spell-check : $O(p)$.*
- *Type-setting : $O(p)$.*
- *Ordering : $O(1)$, as they are already be ordered.*

So the maximum throughput is $O(1/p)$.

iii) What is an appropriate design pattern for managing the publishing process? [2]
Answer : The pipeline design pattern will work well, as there is a source task (the managing editor), a sink task (the printer), and a number of intervening tasks, some of which can be handled in parallel. There is also the opportunity to nest the design pattern, as individual pages can also be treated as jobs within a larger book job.

iv) The managers have decided to put a huge number of employees to work on the spell-checking step, and have proposed two methods:

- Assign each word in the dictionary to one employee.
- Assign each word in the manuscript to one employee.

What are the advantages and disadvantages of each (remembering that we are limited to non electronic communication)? [3]
Answer : In the first case each word in the manuscript could be shouted out loud, and if it corresponds to the word of an employee they shout back. This requires $O(p \times w) = O(p)$ time to check all the pages. However, there is a big problem if a word is not recognised by any employee. There is also a scaling problem, due the need to perform a $1 \rightarrow d$ broadcast of information (i.e. loudness of voice).

At the other extreme, one could assign each word of the manuscript to one employee, who could then do an $O(\log d)$ lookup in the dictionary. However, there would be problems in distributing the work out to so many people, as a standard recursive splitting model only works down to the level of a page. Beyond this point it becomes necessary to copy pages, resulting, increasing the overhead of creating more work.

v) Suggest an efficient way of allocating the spell-check to many employees, explaining any design patterns involved. [3]

Answer : The first design pattern to apply is map reduce. Each page can be processed in parallel, but to achieve this a tree must be used to fan the pages out to p workers. Each worker then spell-checks each word on their page, which is effectively agglomeration as there is no communication cost or parallelism overhead. Once the page is finished, the modified page is returned back up the tree, using a merge at each stage to ensure the pages end up in the correct order.

A possible optimisation is to have a small number of workers sharing a page, who have extremely low overhead communication with each other. For example, two workers could start at the beginning and end of the page, and work towards the middle, incurring very little overhead.

2. a) Give two strengths each for Cilk and TBB.

Answer : One mark for each point.

- *Cilk : Because it is a language it can introduce new keywords, making it easier to use and less verbose.*
- *Cilk : The restricted set of parallelism primitives mean it is very difficult to write unsafe programs.*
- *Cilk : The same code can be used to produce both a parallel version and the serial elision.*
- *TBB : Using standard C++ libraries makes it easier to integrate into existing applications.*
- *TBB : It offers templated design-patterns for different types of parallel task, reducing the effort when dealing with things that are not pure spawn/sync.*
- *TBB : The availability of low-level primitives opens up the possibility of micro-optimisation (e.g. atomics).*

[4]

b) What names are used to describe c_1 and c_∞, and what are the mathematical definitions for each one? [4]

Answer : Two marks each for the two definitions.

c_1 *is the* work overhead *or* cost of work, *and represents the overhead needed to support parallel tasks, compared to a purely sequential program. It is defined as:*

$$c_1 = T_1/T_S \tag{2.1}$$

where T_1 is the time taken to execute the program on one processor, and T_S is the time taken for the serial elison of the program.

c_∞ *is the* critical overhead *or* cost of critical path, *and represents the cost of tracking dependency chains of tasks through the program in order to tell when tasks are ready to be executed. It is defined as:*

$$\min c_\infty : T_P \leq T_1/P + c_\infty T_\infty \tag{2.2}$$

where T_P is the time to execute on P processors and T_∞ is the time to execute on inifite processors.

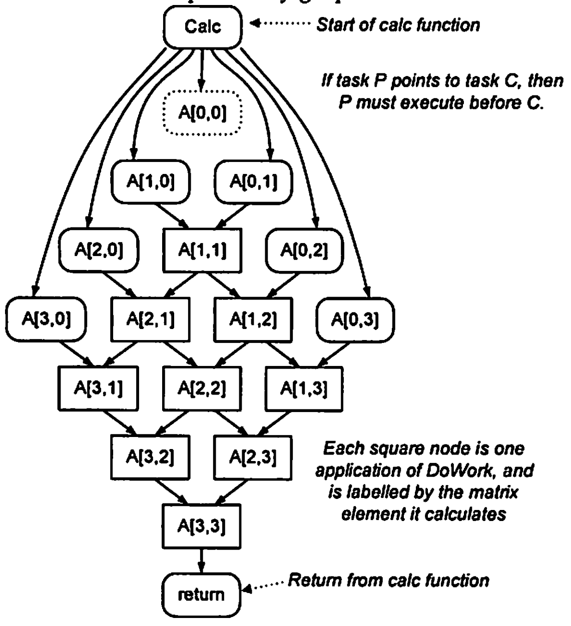
c) You have been given the following function to accelerate:

```
float DoWork(float x, float y); // Safe to call in parallel with self

float Calc(unsigned N, const float *boundary)
{
  // Create a matrix with N rows and N columns
  matrix2d<float> A(N,N);
  // supports A[y][x] to access element in row y and col x

  // Set up boundary conditions on left and top edge
  for(unsigned i=0;i<n;i++){
    A[i][0]=boundary[i];
    A[0][i]=boundary[i];
  }

  // Update each cell using neighbours on left and above
  for(unsigned x=1;x<N;x++){
    for(unsigned y=1;y<N;y++){
      A[y][x]=DoWork(A[y][x-1], A[y-1][x]);
    }
  }

  // Return bottom right element
  return A[N-1][N-1];
}
```

The plan is to parallelise the execution of "Calc" across hundreds of cores. Based on profiling it appears that "DoWork" takes much longer than spawn or sync, so agglomeration of calls to it are not necessary.

i) Draw a dependency graph showing the true data dependencies in Calc for N=4, using labels or annotations to indicate which nodes correspond to which parts of the code. [3]

Answer : The dependency graph is:



Calc
Start of calc function
If task P points to task C, then P must execute before C.
A[0,0]
A[1,0]
A[0,1]
A[2,0]
A[1,1]
A[0,2]
A[3,0]
A[2,1]
A[1,2]
A[0,3]
A[3,1]
A[2,2]
A[1,3]
A[3,2]
A[2,3]
Each square node is one application of DoWork, and is labelled by the matrix element it calculates
A[3,3]
return
Return from calc function

ii) What is the critical path and total work of the dependency graph (using any appropriate assumptions)? [2]

Answer : Counting each box in the graph as a step, then the critical path is $2N-2$ through the array, plus the function call and return, for a total critical path of $2N$ (the value of A[0][0] is never actually used). The total work is N^2+1, due to the quadratic array size and the fixed startup over-head.

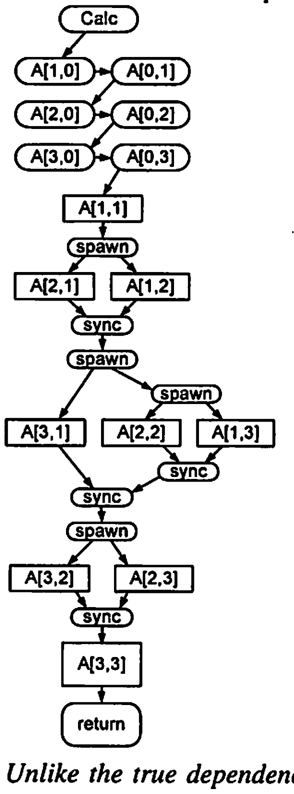
iii) Use Cilk to parallelise the code. [5]

Answer : Two marks for how appropriate the parallelism pattern is; three marks for getting the details correct.

```
// This is effectively a parallel_for, specialised for
// diagonals
cilk CalcDiag(matrix2d<float> &A, int x, int y, int len)
{
  if(len==1){
    // The base-case is identical to sequential version
    A[y][x]=DoWork(A[y][x-1],A[y-1][x]);
  }else{
    // Otherwise split the diagonal into two
    spawn CalcRow(A, x, y, len/2);
    spawn CalcRow(A, x+len/2, y+len/2, len-len/2);
  }
}

float Calc(unsigned N, const float *boundary)
{
  matrix2d<float> A(N,N);

  // Don't do A[0][0], as it isn't needed
  for(unsigned i=1;i<n;i++){
    A[i][0]=boundary[i];
    A[0][i]=boundary[i];
  }

  for(unsigned i=1;i<N;i++){
    // Calculate diagonal of increasing size
    CalcDiag(A, i,1,i);
  }
  for(unsigned i=2;i<N;i++){
    // Calculate diagonals of decreasing size
   CalcDiag(A, N-1,i,N-i);
  }
  return A[N-1][N-1];
}
```

iv) Draw the actual dependency graph of your parallel program due to the spawns/syncs and sequential statements (for N=4), and contrast it to your true dependency graph from i). [2]

Answer : The actual dependence graph is:



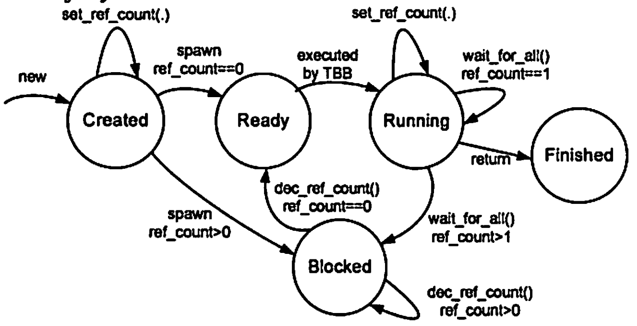
Unlike the true dependency graph, there is a sequential ordering on the initialisation of the left and top rows, but this should be extremely fast.

There is also a binary tree of spawns needed to issue each diagonal, followed by a tree of syncs. This is because it is difficult to express two *a task with two dependencies in Cilk, so we instead find an ordering which is guaranteed to be a correct scheduling of the true dependency graph.*

3. a) Draw a diagram showing the different states in a TBB task's lifecycle, and annotate the transitions to show how changes in reference count are used to manage changes in state. [5]

Answer : Three marks for describing the different states and arcs; two marks for annotations linking them to the reference counts.

The lifecycle is:



b) The following structure is a binary tree, used to provide a dictionary data-structure (a mapping between unique keys and non-unique values).

```
struct Node{
    Key key; // Type Key supports <, ==, <=
    Value value; // Type Value only supports ==
    Node *left; // Contains all nodes with key less than this node's key
    Node *right; // Contains all nodes with key greater than this node's key
 };
```

i) When finding a given key within the binary tree, how does the total work compare to the critical path? [2]

Answer : Performing a key-based lookup reduces to a standard traversal through a binary tree. Depending on the shape of the tree and the specific key, the amount of work will vary. However, the critical path will asymptotically be the same as the total work.

ii) For some classes of key it is *much* faster to perform k1==k2 than it is to perform k1<k2. Show how this property could be used to make parallel key lookup faster than sequential key lookup, and define the circumstances under which this would work. [4]

Answer : Two marks to identify why the parallel version might be faster; two marks for a working solution.

Let T_{eq} be the time for equality testing and T_{lt} be the time for less than. If we assume that the tree is approximately balanced with n nodes, then the sequential version will take $\approx \log_2(n) \times (T_{eq} + T_{lt})$ to do a simple binary search. However, we could define a parallel version:

```
cilk const Node *lookup(const Node *tree, const Value &value)
{
  if(tree==NULL) return NULL;
Node *left=spawn lookup(tree->left, value);
Node *right=spawn lookup(tree->right, value);
if(tree->value!=value)
tree=NULL;
sync;
return (tree | left | right); // Only one can be non-zero
}
```

This will explore every single node, so will execute the main part of the *body for n nodes. If we assume that equality testing outweighs the cost of spawn/sync, then this will require $\approx (n \times (2 * T_{spawn} + T_{eq} + T_{sync}))/P$ operations on P processors. If this ends up taking less time, then the parallel version would be a better choice.*

iii) The function `reverse_lookup` finds the key corresponding to a given value, and if multiple nodes have the same value, then the smallest key should be returned. If the value does not exist, then a sentinel key `MAX_KEY` is returned, where `k < MAX_KEY` is true for all valid keys. A sequential version of the function is:

```
Key reverse_lookup(const Value &value, const Node *node)
{
  if(node!=NULL){
    Key left=reverse_lookup(value, node->left);
    if(left<MAX_KEY)
      return left;

    if(node->value==value)
      return node->key;

    Key right=reverse_lookup(value, node->right);
    if(right<MAX_KEY)
      return right;
  }
  return MAX_KEY;
}
```

Parallelise the `reverse_lookup` function using Cilk, and briefly explain how efficient it is. [5]

Answer : Three marks for a solution that can exploit parallelism; two marks for discussion of efficiency.

A simple solution uses the fact that the reverse_lookup function has no side-effects, so it is always safe to lookup the right tree even if we haven't finished with the left one:

```
cilk Key reverse_lookup(const Value &value, const Node *node)
{
  if(node!=NULL){
  bool meDone=false;

Key left=spawn reverse_lookup(value, node->left);
Key right=spawn reverse_lookup(value, node->right);

if(left<MAX_KEY)
  return left;
if(node->value==value)
  return node->key;
if(right<MAX_KEY)
  return right;
  }
  return MAX_KEY;
}
```

A more efficient version would notice that if something is found in the left tree, then nothing found in the right tree will ever be returned, so could use a shared variable to indicate that all tasks can exit. However, if we assume a balanced tree and that the value occurs infrequently, then the extra complexity is high for a relatively small payoff.

iv) The data-structure is difficult to process in parallel, as the tree can have many different shapes, and be very unbalanced. Suggest a single data member which could be added to each node, and how it could be used to reduce the cost of work. [3]

Answer : The problem is that the parallel code has no idea how many nodes there are in a given tree, so as it recursively explores the tree it is very difficult to apply agglomeration.

One solution would be to add a "depth" field to each node, which contains the number of levels below the node, so the algorithm would switch to sequential recursion when the depth is less than a threshold. Another alternative would be an "nchildren" node, which counts the total number of child nodes below the current node, and could be used in the same way.

4. a) i) While most CPUs contain some sort of SIMD capabilities, they are not considered to be GPUs. Suggest three distinguishing characteristics between contemporary GPUs over CPUs. [3]
Answer :

- *GPUs typically do not allow instructions for the same thread to overtake each other (i.e. they are not out-of-order issue). Some modern GPUs are starting to exploit parallel issue, but this is far less aggressive than in a super-scalar CPU.*
- *Instead of large caches, GPUs have per-core scratch-pad memories.*
- *The proportion of silicon area dedicated to ALUs is far higher in a GPU than in a CPU.*
- *GPUs have built-in architectural support for SIMT (single-instruction multiple thread), which allows SIMD lanes to be enabled and disabled. GPUs have SIMD, but need to explicitly enable and disable each lane using extra instructions.*

ii) For modern GPUs, what are the approximate bandwidth and latency ranges, of: registers, local (shared) memory, and global memory? [3]
Answer : Global memory is usually implemented in DDR3 or up, and so has bandwidth measured in the range of 10 to 300 GB/sec per GPU, with latency of hundreds of clock cycles.

Local memory bandwidth is dependent on the number of bank conflicts, so in the ideal case the bandwidth is typically 1 word per thread per cycle in the best case, and up to 16-32 words per cycle in the worst case. Latency will depend on both bank conflicts and also any additional architectural delays, and so will vary from approximately 1 to 100 cycles.

Each instruction is usually able to read at least two registers and write one register, so bandwidth is of the order of three words per thread per cycle. Latency is effectively zero.

b) Many modern GPUs have support for atomic memory operations, which provide similar functionality to the `tbb::atomic` objects. Assume the existence of a function `atomic_add`, which performs the following operation atomically for pointers to all types of storage:

```
void atomic_add(int *acc, int value)
{
  (*acc) += value;
}
```

At compile-time, the compiler will automatically work out the type (register, local memory, or global memory) of the pointer `acc`, and select the correct underlying code.

i) How could `atomic_add` be implemented for pointers to registers? [2]
Answer : Operations on registers are automatically atomic, as only a single thread has access to them, so the reference implementation given above can be used directly.

ii) Most GPUs provide architectural support for 32-bit integer atomics, but none support atomic addition on double-precision numbers. Suggest why this is the case. [2]

Answer : Integer operations such as add can usually be implemented as very low-latency logic in a single cycle, minimising the amount of time that a memory location must be locked. A double-precision addition requires multiple cycles, so in principle the memory implementing the atomic addition may need to block for the same number of cycles, or alternatively it will need complicated logic to support multiple atomic operations in progress at the same time, and resolve conflicts.

Another explanation is that floating-point operations are not associative, so there is no way of writing deterministic programs that use floating-point additions. If a set of threads execute $2^{100} + 2^{-100} - 2^{100} - 2^{-100}$ using atomic addition, then the result could legally be 0, 2^{-100}, or -2^{-100}, depending on how the operations are scheduled.

iii) Implement `atomic_add` for pointers to local memory, *without* any architectural support. If necessary, you can assume that there will be no warp divergence, so all threads in a block will call `atomic_add`. [4]

Answer : Two marks for the basic idea of iterative blocking; two marks for implementing correctly.

As long as only one thread is active at once, then we can safely read and write to one local memory item. With careful use of synchronisation points we can make sure that only one thread in the block is active, though we will need to make sure that all *threads call* `atomic_add`*.*

So a correct (but extremely inefficient) solution is:

```
__kernel void atomic_add_impl(__local int *acc, int value)
{
  barrier(CLK_LOCAL_MEM_FENCE);
for(int i=0;i<get_local_size(0);i++){
  if(i==get_local_id(0)){
  (*acc)+=value;
}
barrier(CLK_LOCAL_MEM_FENCE);
}
}
```

iv) A user wishes to implement a kernel which calculates the sum of a set of integers. The kernel takes in an array of numbers in global memory, where the array size is a multiple of the total threads in the grid, and after execution the sum of all the numbers must be gathered in a single global memory location. The kernel needs to scale across multiple GPUs, so it must be possible to have multiple blocks within the grid.

Give code for a kernel (using CUDA or OpenCL syntax) which can efficiently perform this task, with the following prototype:

```
// N – Total number of data–items, will be a multiple of grid size
// data – N data–items to be summed
// sum – Single global memory location to receive sum
__kernel void parallel_sum(unsigned N, __global int *data,
    __global int *sum);
```

[6]

Answer : Three marks for realising the importance of a hierarchy of atomicity; three marks for correct solution.

The important things are: to minimise the number of atomic operations, particularly those to global memory; to maximimise the amount of serial work done per thread; and do ensure coalesced memory reads from global memory. One solution is:

```
__kernel void parallel_sum( unsigned N, __global int *data,
    __global int *sum)
{
  unsigned K=N/get_global_size(0);

  __local int localAcc;

  localAcc=0; // All threads write 0, result is well-defined
  barrier(CLK_LOCAL_MEM_FENCE);

  int privateAcc=0;
  for(unsigned i=0;i<K;i++){
   // Create coallesced memory access within warp
   privateAcc += data[i*get_global_size(0)+get_global_id(0)];
  }

  // All threads in local work-group update sum
  atomic_add(&localAcc, privateAcc);

  // Make sure everyone has updated
  barrier(CLK_LOCAL_MEM_FENCE);

  // Only one thread in local work-group updates global sum
  if(get_local_id(0)==0){
    // And has to do it atomically
    atomic_add(sum, localAcc);
  }
}
```