Master Of Science in Informatics at Grenoble
Parallel, Distributed, and Embedded Systems

Parallel Systems

Vincent Danjean, Derrick Kondo, Arnaud Legrand

January 23, 2012

Important informations.
Read this before anything else!

- ▷ Any printed or hand-written document is authorized during the exam, even dictionaries. Books are not allowed though.
- ▷ Please write your answers to each problem on separate sheet of papers.
- ▷ The different problems are completely independent. You are thus strongly encouraged to start by reading the whole exam. You may answer problems and questions in any order but they have to be written in the order on your papers.
- ▷ All problems are independent and the total number of points for all problems exceeds 20. You can thus somehow choose the problems for which you have more interest.
- ▷ The number of points alloted to each question gives you an indication on the expected level of details and on the time you should spend answering.
- ▷ Question during the exam: if you think there is an error in a question or if something is unclear, you should write it on your paper and explain the choice you made to adapt.
- ▷ The quality of your writing and the clarity of your explanations will be taken into account in your final score. The use of drawings to illustrate your ideas is strongly encouraged.

I. Solving a Triangular System [4 points]

Consider the problem of a solving linear system $Ax = b$, where A is a lower triangular matrix of rank n and b is a vector with n components.

We recall that the following sequential algorithm can be used to solve such problem:

```
for i = 0 to n - 1 do
  s <- 0
  for j = 0 to i - 1 do
     s <- s - a[i,j] * x[j]
  x[i] <- (b[i] - s)/a[i,i]
```

We assume that we have at our disposal a computing platform that is a unidirectional ring of processors $P_0, P_1, \dots, P_{p-1}$. For simplicity we assume that p divides n.

Question I.1. *We wish to distribute columns of matrix A among the processors. What is likely the best strategy? Give the pseudo-code for the corresponding parallel algorithm.*

Question I.2. *Now we wish to distribute rows of A among processors. What is likely the best strategy? Give the pseudo-code for the corresponding parallel algorithm.*

II. Parallel Efficient 3D Mesh Algorithms [10 points + 4 bonus points]

Reminders and definitions

Cache and data locality

In order to efficiently use processor caches, algorithms should ensure that accesses to data respect two kinds of locality: spacial and temporal. Temporal locality ensures that a data loaded in the cache will be reused before being evicted of the cache. Spacial locality ensures that access to data stored nearby previously accessed data will occur. The last point is important because data are loaded block by block in the cache. The block size depends on the cache hardware. A block of cache data is also called a cache-line.

Algorithms classification

Efficient algorithms with respect to the cache can be classify in two category:

Cache-aware (CA) algorithms the value of the block size (B) used to load data in the cache is taken into account by the algorithm. This is the case for the Atlas BLAS library that splits matrix in sub-matrics that perfectly fit in the cache of the targeted processor;

Cache-oblivious (CO) algorithms the value of the block size (B) used to load data in the cache is not used by the algorithm. The same algorithm implementation will be efficient whatever the cache block size will be (so it will also work for any number of levels of caches).

3D meshes are used a lot in simulation and visualization. Big meshes represent solid objects or fluids. Physical simulations update meshes in order to predict the evolution of the system.

Mesh Data Structure. A mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). When the mesh is composed of cells of different types (using various number of points), an additional array allows random access to cells (Fig. 1). As the cache performance is similar whether meshes have identical type cells or not, you can consider here that only two arrays are used (ie meshes are homogeneous).

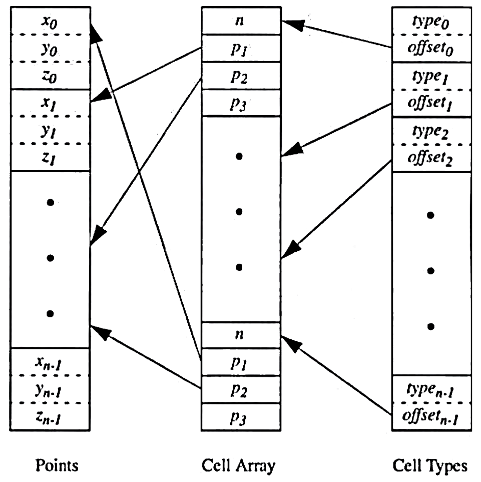

Figure 1: The vtkUnstructuredGrid data structure (from the VTK Textbook). The Points array contains point coordinates and the Cells array contains the indices of cell points. The Cell Types array contains the type of each cell and provides $O(1)$ random access to cells.

Mesh algorithms. Lots of mesh algorithms do independent operations either on all cells or points of the whole mesh, or on all cells or points of a connected sub-part of the whole mesh. Moreover, an operation on a point or a cell generally involves the points or cells nearby (w.r.t. the 3D space) the current one.

This means that most of 3D mesh algorithms have an access pattern to the data that respect a good locality in the 3D space. To have a good behavior with respect to the cache, one should ensure that 3D locality leads, at least most of the time, to a good memory locality. The order in which each point and cell is put in their arrays has direct consequences on the efficient use (or not) of the processor caches.

Targeted platform. Unless otherwise specified, we consider that we want to run our algorithms on an SMP machine composed of one multiprocessor with 8 cores/16 threads (i.e. two threads per core). This processor has three level of caches. The first two levels L1 and L2 are private to each core (256 kB of L2 cache per core) but the L3 cache (20 MB) is shared between all cores.

The machine is dedicated to software experiments in high performance computing. After talking to the users (researchers), the administrators of the machine deactivated the hyperthreading, i.e. only one thread per core is used.

Question II.1. *[1 points] Give some reasons explaining why researchers working in HPC prefer to deactivate the hyperthreading feature.*

Question II.2. *[1 points] Draw a schema of the machine with all important hardware parts (do not forget to document (caption, etc.) your schema).*

A. Data order in mesh layout

A.1 Geometric layout

In order to decide about the order of points and cells in their array, a student propose to sort them geometrically[1]. It means that each cell/point is sorted first with its first coordinate (x), then the second one (y), and finally the third one (z) (as we do with the letters of a word for a alphabetic order).

Question II.3. *[2 points] What do you think of this data layout? When will it be good, when will it be bad, what property (with respect to the cache) can you expect, etc.?*

A.2 Cache-oblivious layout

Until the end of this exercise, we now consider that the order of cells and points is chosen so that we have a **cache oblivious (CO) layout**: 3D locality between points or cells ensures, with a good probability, a memory-locality in the corresponding arrays. More formally, the CO layout guarantees that a traversal by chunks of size $m \leq M$ of an N-size mesh induces less than $N/B + O\left(N/m^{1/3}\right)$ cache misses where B and M are the block and cache size, respectively.

B. Sequential algorithm

We consider an algorithm that will run on a 3D mesh. The algorithm will do the same kind of operation on each cell (updating values). In order to be able to create parallelism, we consider that each cell has two versions of the values (current and next ones) so that each update can be done in parallel.

The computation required to update the values can take more or less time: we can measure a factor 10 between the fastest and the slowest cell update. However, the time depends on physical property of the underlying simulated experiment, physical property that vary slowly. This means that adjacent cells (in the 3D mesh) can be expected to have a similar update time.

The sequential algorithm can be written like this :

```
for each cell c in the mesh
    datapoints={}
    for each point p in points_of_cell(c)
        datapoints += data_of_point(p)
    update_value(c, datapoints)
```

Due to the CO layout of the mesh, the sequential algorithm performs greatly, using very efficiently all cache levels: the CO layout ensures that all points related to a cell should be nearby, so the internal loop should access to nearby data (in memory), optimizing the cache use. Moreover, the CO layout ensures that cells are updated in an order related to the 3D locality. So, as points are

[1]The 3D position of a cell is the barycentre/center of gravity of its points.

shared between adjacent cells, one can expect that data loaded for a point of a cell will be reused for a next cell (before being evicted from the cache).

We will now think about parallel re-implementations of this algorithm.

C. Parallel algorithms

Let N be the number of cells in the mesh. Unless otherwise specified, all times are in millisecond and all numbers of cache misses are in million. All measures presented in this part are obtained by running an efficient implementation of the proposed algorithms on a big mesh with N cells.

When running the sequential algorithm, the execution time is $2400\,ms$ with $91.5\,M$ L2 cache misses and $7.6\,M$ L3 cache misses.

C.1 Static partitioning

At first, the sequential algorithm is parallelized by statically assigning one core to each cell. Two assignments are studied, leading to two parallel algorithm:

static-split algorithm: the cell array is divided 8 (equal) contiguous parts and each part is assigned to a core;

static-round-robin algorithm: each cell is assigned to a core in a round-robin way, i.e. core i will update cells $i + 8j$ for all $0 \leq j < N/8$.

When running the experiments, we can observe the following results:

	Execution	Cache misses		Idle
Algorithm	time	L2	L3	time
sequential	2400	91.5	7.6	16800
static-split	500	92.3	64.3	1500
static-round-robin	400	503.5	30.2	700

Note: the idle time is the sum of all system idle time (usually spent in synchronization waiting functions) for all cores. Processor time lost when waiting for cache filling is **not** considered here. For the sequential algorithm, we consider that all the 7 other cores were blocked until the running one finished (hence the $16800 = 2400 \times 7$ value).

Question II.4. *[2 points] Propose some analysis of the results of the* static-* *parallel algorithms. Explain the parallel programs behavior, in particular with respect to the caches.*

C.2 Work stealing

In order to solve some problems revealed by the previous experiments, the algorithm will be rewritten using the work stealing paradigm.

Classical work stealing. In a first experiment, we ignore the fact that the L3 cache is shared (i.e. we consider 8 independent processors as in the classical model of work stealing).

Question II.5. *[2 points] Write the parallel algorithm using classical work-stealing. If you introduce some parameters/constants, discuss about their values and their influences. What is the critical path of your algorithm?*

> **Note:** *to create an independent task executing the function* `foo(bar, ...)`*, use the keyword* `spawn` *in front of the function call. Use the keyword* `sync` *in a function to wait for the end of all spawned tasks in this function.*

Work stealing within windows. In order to try to better use the shared L3 cache, two similar experiments are conducted. The idea is to use the work stealing, but all cores will work (with work stealing) on a small part the cell array before continuing on the next parts.

The first algorithm, *ws-static-window*, splits the array in x equal parts (windows) of size W. Then, the work-stealing algorithm will be called successively in each window.

The second algorithm, *ws-sliding-window*, also starts with a work-stealing algorithm working in a window of size W. But then, when the first cells are updated, the window is shifted (i.e. the following cells are added). So, the window in which the work-stealing algorithm is done has always the same size W, but the window in continuously changing: once the first cells of the window are updated, they are removed from the window and the same number of the next cells are added into the window.

When running the experiments, we can observe the following results:

	Execution	Cache misses		Idle
Algorithm	time	L2	L3	time
sequential	2400	91.5	7.6	16800
static-split	500	92.3	64.3	1492
static-round-robin	400	503.5	30.2	703
ws-classical	342	91.9	63.4	244
ws-static-window	332	93.2	8.3	250
ws-sliding-window	311	93.1	8.3	75

Question II.6. *[2 points] Analyze the results of the* ws-* *parallel algorithms. Explain the parallel programs behavior, in particular with respect to the caches. What do you think of the W parameter? How should it be chosen?*

D. Bonus

Question II.7. *[1 point] Now, we want to exploit the hyperthreads of the processor. Each core has two threads that share the L1 and L2 caches of the core. What can you suggest as parallel algorithms to efficiently use these hyperthreads?*

Question II.8. *[1 point] The machine becomes bigger: we now have a NUMA machine that has 8 of these 8 cores/16 threads processors (so 64 cores/128 threads in total). How would you modify the last parallel algorithm (*ws-sliding-window*) to efficiently use this machine?*

Question II.9. *[1 point] For each considered parallel algorithms, tell and justify whether it is a cache-aware algorithm, a cache-oblivious algorithm, or if it does not optimize cache accesses.*

III. N queens problem [8 points]

Consider the NQUEENS problem:

- ▷ Input: an integer $n \geqslant 2$.
- ▷ Output: NQUEENS(n), the number of different correct positions of n queens on a $n \times n$ chessboard. A position is correct if and only if all the following conditions are met:
  - n queens are mapped;
  - there is exactly one queen in each row and in each column;
  - any diagonal and anti-diagonal includes at most one queen.

We have NQUEENS(1)=1. For $n = 2$ and $n = 3$, there is 0 solution; NQUEENS(4)=2 ; NQUEENS(26)= 22,317,699,616,364,044.
The computation time is exponential; $n = 27$ uses years of computation on a single processor.

We consider the parallelization of the following sequential program `n_Queens_seq` that computes NQUEENS. It is not asked to optimize this sequential program!

The `n_Queens_seq` program explores all possible positions of n queens on the chessboard $n \times n$ by setting exactly one queens by line. Each of the n^2 cells of the chessboard is indexed by its row index i ($1 \leqslant i \leqslant n$) and column index j ($1 \leqslant j \leqslant n$). It uses the type `chessboard` and the following functions:

- ▷ `InitChessboard ( E, n )`: initializes a chessboard with size $n \times n$ and no queens. This function modifies E.
- ▷ `CopyChessboard ( E, E')`: copies the chessboard E in the chessboard E'. This function modifies E' but not E.
- ▷ `bool IsCorrect(E, i, j)`: returns true if and only if E is a correct position when adding a queen on cell (i, j). Does not modify E.
- ▷ `AddQueen(E, i, j)`: adds a queen in cell $(i; j)$ of E. Modifies E.
- ▷ `RemoveQueen(E, i, j)`: removes a queen on cell (i, j) of E. Modifies E.

All those functions lost a bounded time, independent of n, but unknown and that may depend on the state of the chessboard.
N.B.: Only the functions `InitChessboard`, `AddQueen`, and `RemoveQueen` modify the chessboard E (effective parameter).

```
void n_Queens_seq ( Chessboard& E, int n, int i, int& nb_sol) {
  /* On each of the (i-1) first rows of E, a queen has been
   * previously added. The function then adds a new queen on row i,
   * and then recursively on the rows with index > i.
   */
  if (i > n)
   nb_sol = nb_sol + 1 ;
  else {
    int j ;
    for ( j = 1; j <= n ; j = j + 1 ) {
      if ( IsCorrect ( E, i, j ) ) {
        AddQueen ( E, i, j ) ;
        n_Queens_seq ( E, n, i+1, nb_sol ) ;
        RemoveQueen ( E, i, j ) ;
      }
    }
  }
}

main()  {  /* main call */
  int n = .... ;
  int nb_solutions = 0 ;
  Chessboard E ;
  InitChessboard ( E, n ) ;
  n_Queens_seq ( E, n, 0, nb_solutions ) ;
}
```

Figure 2: Sequential program for NQUEENS

Figure 2 is the C++ program `n_Queens_seq`: the parameters `n` and `i` are passed by value and the parameters `E` and `nb_sol` are passed by reference (indicated by the "`&`" symbol).

Important. It is assumed that:

▷ the work $W_{seq}(n)$ of the main call is equal to a function $\Gamma(n) = \Theta(n!)$, so with the same order of magnitude as $n!$.

▷ the cost of the call n_Queens_seq (E, n, i, nb_solutions) is $\Gamma(n-i) = \Theta\left((n-i)!\right)$.

Question III.1. *[3 points] In this question, it is assumed that the creation of a new ready task (for instance, by using operation* `spawn` *in Cilk,* `Fork` *in Athapascan/Kaapi,* `pthread_create` *in Posix, by unlocking a blocked task, ...) has the same cost in sequential and in parallel, and communications are not considered.*

a) Propose a fine grain parallelization of `n_Queens_seq` *whose parallel depth $D(n)$ is $O\left(n^{O(1)}\right)$ and $W(n) = O(\Gamma(n))$: from the sequential program, write a parallel program (in pseudo-Cilk or Athapascan/Kaapi or OpenMP or ...), making the parallelism explicit as well as the new data that need to be allocated, ...*

NB: *You have to use the type* `Chessboard` *and the primitives* `InitChessboard, CopyChessboard, IsCorrect, AddQuueen, RemoveQueen` **without providing their implementation!**.

b) *Analyze the parallel depth $D(n)$ of your program.*

Question III.2. *[4 points] We want now to obtain a program with a very small depth $D(n)$ and whose work $W(n)$ is close to $W_{seq}(n) = \Gamma(n)$:*

$$W(n) = \Gamma(n) + o(\Gamma(n)) \text{ and } D(n) \ll \Gamma(n). \tag{1}$$

a) *What are the instructions that introduce overheads in your previous parallel program?*

b) *Modify your previous parallel program to verify Equation (1); the depth $D(n)$ should be as small as possible.*

Hint: *To adapt the granularity, you may use in your program the approximate reciprocal of $\Gamma(n)$, assumed known and denoted $\Gamma^{-1}(n)$: the value of $\Gamma^{-1}(n)$ is a positive integer and verifies: $\Gamma(\Gamma^{-1}(n)) = \Theta(n)$.*

c) *Analyze the parallel depth $D(n)$ of your program.*

d) *What scheduling algorithm will you use to execute the program on a machine with p identical processors? What will be the corresponding execution time?*

Question III.3. *[1 point] We consider now the execution for a large n (eg $n = 27$). on a large global computing system with about $p = 10^5$ processors (e.g. a computing grid); the network is slow (the average latency is about 1 s). Without considering communication, the global computation speed Π_{tot} of the machine is such that $\frac{W_{seq}(n)}{\Pi_{tot}} > 1$ month.*
Question: *What scheduling algorithm will you use to execute the program? What will be the order of the whole number of communications performed in function of n and p? What do you conclude?*

Would it make sense to use the same technique on a global computing system such as BOINC?