Master Of Science in Informatics at Grenoble
Distributed Embedded Mobile Interactive Parallel Systems

Parallel Systems

Vincent Danjean, Derrick Kondo, Arnaud Legrand, Jean-Louis Roch

January 24, 2011

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 problem 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.

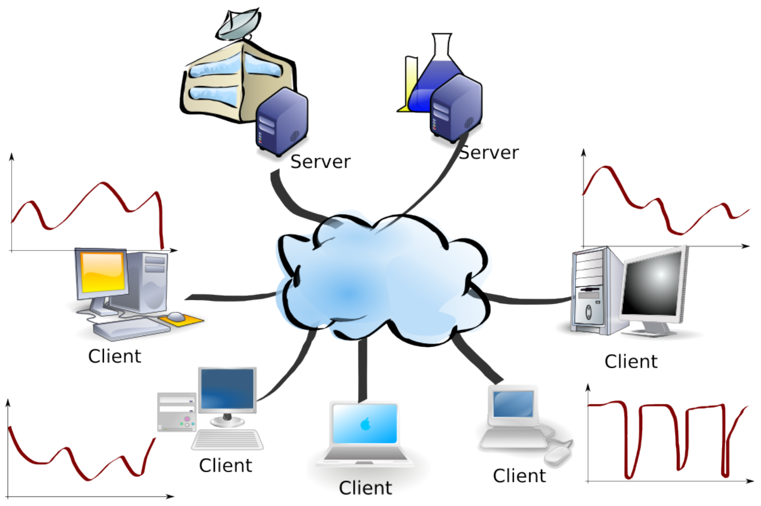

Figure 1: Volunteer computing architecture in a nutshell

I. Practical Scheduling Issues in Volunteer Computing Systems [8 points]

Volunteer computing (VC for short) is a type of distributed computing in which computer owners donate their computing resources (such as processing power and storage) to one or more "projects". BOINC is the most popular VC infrastructure today with over 580,000 hosts that deliver over 2,300 TeraFLOP per day.

Most of these systems have the same basic structure: a client program runs on the volunteer's computer. It periodically contacts project-operated servers over the Internet, requesting jobs and reporting the results of completed jobs (see Figure 1). This "pull" model is necessary because many volunteer computers are behind firewalls that don't allow incoming connections. The system generally keeps track of each user's "credit", a numerical measure of how much work that user's computers have done for the project.

VC systems must deal with several problematic aspects of the volunteered computers: their heterogeneity, their churn (that is, the arrival and departure of hosts, since users may reclaim their hosts at any time), their sporadic availability, and the need to not interfere with their performance during regular use.

Question I.1. *Handling Incorrect Results [2 points]*

VC systems must deal with several problems related to correctness:

- ▷ *Volunteers are unaccountable and essentially anonymous.*
- ▷ *Some volunteer computers (especially those that are overclocked) occasionally malfunction and return incorrect results.*
- ▷ *Some volunteers intentionally return incorrect results or claim excessive credit for results.*

One common approach to these problems is "replicated computing with majority voting", in which each job is performed on at least two computers. The results (and the corresponding *credit) are accepted only if they agree sufficiently. Otherwise, the job is executed on a third client to break the tie.*

a) *Even though the server selects on which clients it will replicate jobs, does the replication mechanism protects against malicious users that send falsified results ?*

b) *Note that the first replica is generally distributed a short time after the original so as to get the final result as soon as possible. How could a community of malicious users defeat a majority voting mechanism ?*

c) *In practice, only very few users are malicious and about 5% of machines are overclocked. What it the drawback of this replication approach in term of resource waste ? Propose mechanisms that improve resource usage while maintaining a good confidence in result correctness.*

d) *How would an additional set of reliable machines (e.g., a dedicated cluster or cloud resources) would help solving such issues ?*

Question I.2. *Job granularity [3 points]*

The Superlink-online *project provides online service to execute genetic linkage analysis. Genetic linkage analysis is a well-established statistical technique essential for identifying disease-provoking genetic mutations. The main goal is to determine the areas of the DNA where such mutated genes are likely to reside, thereby narrowing the search scope for microbiological study. Thus, each jobs consists in computing conditional probabilities on chunks of DNA sequences. The final probability is finally computed by summing the whole set of results.*

The ideal job duration ranges between a few minutes and two hours, because of the following. Indeed, since hosts may be reclaimed by their owners, the longer the job, the more likely it will be interrupted. On the other hand, jobs should not be too small either. Indeed, the job distribution overhead (sending the job and getting the result using a verbose XML protocol, keeping track of the job, . . .) would significantly outweights the effective payload of the jobs. Furthermore, typical work dispatch servers can handle up to 30 concurrent requests per second. Otherwise the firewall is likely to automatically blocks the respective server port considering such a load as a DDoS attack. Jobs should thus be automatically aggregated (sent in bulk) if their duration is too short. However, it is impossible to determine the runtime of a task without actually executing it.

Yet, recent studies report that, in many cases, many of the tasks are far shorter than the others (see Figure 2), and their numerical result is zero. Therefore, these tasks do not contribute to the final result. It turns out that these tasks correspond to some of combinations of the values of conditioning variables used for the parallelization which are impossible in a given probabilistic models, and thus have zero probability. Hence, their result can safely be completly ignored in the final result.

Such short tasks are so short that their execution is very inefficient. Furthermore, they also increase the load on the dispatch server thereby increasing the dispatch latency for other, more useful tasks.

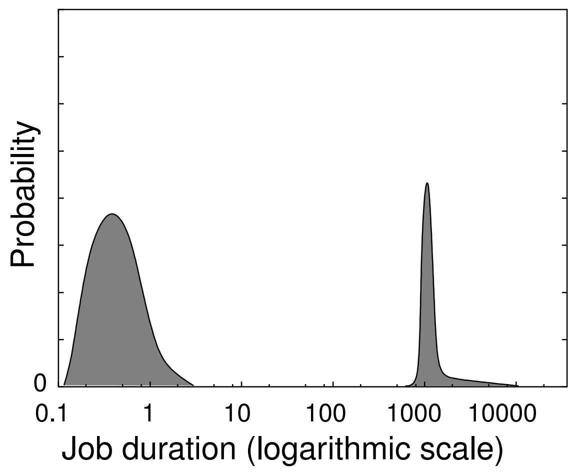

Figure 2: Typical distribution of job duration (on a dedicated machine). This distribution clearly shows two modes. A non negligible fraction of the tasks have a very short duration (smaller than a few seconds). Larger jobs have a duration between 10 minutes and 8 hours with a majority around 20 minutes.

- **a)** *Propose a mechanism to handle short jobs. Note that such a linkage study easily comprises 1 million tasks and that all tasks share some common parts (e.g., memory allocation and initialization of the algorithm).*
- **b)** *How would an additional small set of reliable machines (e.g., a dedicated cluster or cloud resources) would help solving such issues ?*
- **c)** *How would you handle the set of larger jobs ?*

Question I.3. *Turn-around of Bags of Tasks [3 points] In a typical situation, projects have thousands of independent CPU-bound jobs and they are interested in optimizing their throughput, i.e., maximize the total number of tasks completed per day.*

However, the popularization of BOINC has lead to the emergence of new projects with different characteristics, in particular, projects whose workload is made of batches of a small number of work units (jobs) arriving periodically, in contrast to regular and continuous throughput projects. This is for example the case of the Superlink-online *project. Such projects are usually interested in optimizing the response time of each Bag of Tasks and have no interest in the response time of each individual job. Yet, this kind of objective is much harder to optimize than optimizing throughput.*

Figure 3 depicts, for various batch sizes, the number of jobs completed since the batch arrival. Interestingly, the completion time of whole batch (i.e. the completion time of the last finishing job) is around 80 minutes whatever the size of the batch.

- **a)** *Explain why the completion rate of jobs is rather high at the beginning of the batch and then always drops toward the end of the batch.*
- **b)** *Propose mechanisms to keep this rate high and thus improve the batch completion time (e.g., so that the completion time of 100 tasks batches is around 12 minutes and the completion time of 200 tasks batches is around 24 minutes).*

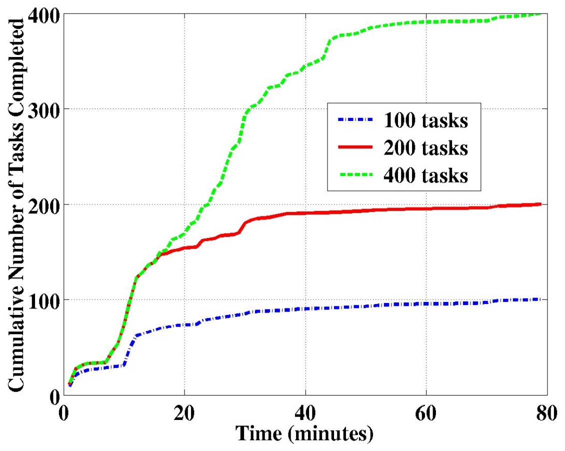

Figure 3: Illustrating job completion rate evolution for various batch sizes when using a simple first-come first-served strategy with no replication on a set of heterogeneous workers.

c) *How would an additional set of reliable machines (e.g., a dedicated cluster or cloud resources) would help solving such issues ?*

II. Cascading divide&conquer and applications (7 points)

Preamble. In this exercise, algorithms should be preferably written in a pseudo parallel language, expliciting parallelism: for instance using parallel instructions similar to Cilk (`spawn`, `sync`, `parfor`, ...) and/or, Athapascan/Kaapi (`Fork`, `Shared`) and/or Posix (`pthread_create`) and/or others (OpenMP, Posix, MPI, ...) However, you may explicit the principle of your algorithm in a pseudo language, but clearly exhibiting the sequential steps and the parallel operations performed.

Question II.1. *Merge and cascading [2 points]*

The MERGE problem is defined as follows:

Input: *two* sorted *arrays $A = [a_0, \dots, a_{n-1}]$ and $B = [b_0, \dots, b_{m-1}]$ (by increasing order). Moreover, all elements a_i are b_j assumed* **distincts** *: $a_i \neq b_j$ for any $0 \leqslant i < n$ and $0 \leqslant j < m$. Thus: $a_0 < a_1 < \dots < a_{n-1}$ and $b_0 < b_1 < \dots < b_{m-1}$.*

Output: *a sorted array $X = [x_0, \dots, x_{n+m-1}]$ (i.e. $x_0 < x_1 < \dots < x_{n+m-1}$) that contains the elements of both A and B.*

We consider the following parallel Divide&Conquer algorithm for MERGE(B, A, X), assuming $n \geqslant m > 0$:

1. *The array A is split into two sub-arrays $A_1 = [a_0, \dots, a_{n/2-1}]$ and $A_2 = [a_{n/2}, \dots, a_{n-1}]$.*
2. *Let $\alpha = a_{n/2}$; B is split into two subarrays B_1 and B_2 : $B_1 = [b_0, \dots, b_{j-1}]$ contains the elements of B lesser than α and $B_2 = [b_j, \dots, b_{m-1}]$ the elements of B larger than α, i.e.,*
   - *if $b_0 > \alpha$ then B_1 is empty and $B_2 = B$;*

- *else if $b_{m-1} < \alpha$ then $B_1 = B$ and B_2 is empty;*
- *else: j is the unique index such that $b_{j-1} < \alpha < b_j$; j is computed by a binary search with $\log m$ operations.*

3. *A_1 and B_1 are recursively merged in $X[0, \ldots, n/2 + j - 1]$;*
   and A_2 and B_2 are recursively merged in parallel in $X[n/2 + j, \ldots, n + m - 1]$.

This algorithm is programmed with a parallel programming interface (such as Cilk, Kaapi, OpenMP, ...).

a) *What is the work and depth of this algorithm (justify very briefly) ?*

b) *Is it possible to obtain a parallel MERGE algorithm with a smaller depth ? If yes, indicate only the main idea and principle of the algorithm, without detailing its different steps.*

Question II.2. *Independent tasks and cascading [2.5 points]*

We now consider n independent tasks $t_1, \ldots, t_n$, with n very large. For $1 \leqslant i \leqslant n$, the task t_i performs w_i operations. The values w_i are unknown, but are assumed to be bounded by two constants c and c': $c \leqslant w_i \leqslant c'$.

We consider a multicore machine with p identical cores.

a) *We first assign $\frac{n}{p}$ tasks to each core. Prove that the execution time T_p verifies $\frac{n}{p} \cdot c \leqslant T_p \leqslant \frac{n}{p} \cdot c'$. Exhibit a worst case for the w_i together with an assignment such that the execution time T_p is close to $\frac{c'}{c} \frac{\sum_{i=1}^{n} w_i}{p}$.*

b) *Write a program (in a language like Cilk, Athapascan/Kaapi, OpenMP, MPI...) such that the execution time T_p verifies $T_p \leqslant \frac{\sum_{i=1}^{n} w_i}{p} + O(\log n)$ with high probability: note that this time should be achieved whatever the values of the constants c, c' and the w_i are.*
Analyze the work and depth of this algorithm (justify very briefly).

c) *Is it possible to write a program that achieves a depth $O(\log \log n)$? Argue briefly.*

Question II.3. *Cascading and Ladner-Fisher parallel prefix [2.5 points]*

For the sake of simplicity, we assume that n is a power of 2. The Ladner-Fisher algorithm is the following recursive algorithm that computes the prefix sequence with an associative binary operator $\star$:

```
(1) LadnerFisher($a_0, \ldots a_{n-1}$)
(2)   if $n == 1$ return $a_0$
(3)   parfor $i = 0, \ldots, n/2 - 1$ do $b_i \leftarrow a_{2i} \star a_{2i-1}$
(4)   $a_0, a_2, a_4, \ldots, a_{n-2}$ = LadnerFisher( $b_0, \ldots, b_{n/2-1}$ )
(5)   parfor $i = 0, \ldots, n/2 - 1$ do $a_{2i+1} \leftarrow a_{2i} \star a_{2i+1}$
(6)   return $(a_0, a_1, \ldots a_{n-1})$
```

The two parallel loops (parfor) in lines(3) and (5) are performed in depth $\Theta(\log n)$.

a) *Briefly explain that a direct programming of this algorithm in Cilk (or Athapascan/Kaapi or OpenMP, ...) has a depth $O(\log^2 n)$.*

b) *Develop a parallel implementation of this algorithm in Cilk (or Athapascan/Kaapi or OpenMP, ...) that has a depth $O(\log n)$ and work $O(n)$.* **Hint:** *use the cascading divide&conquer strategy to decrease the depth induced by the recursive calls to* $\log \log n$.

III. Gauss-Jordan Algorithm (7 points)

The Gauss-Jordan method is an algorithm for solving a linear system $Ax = b$, where A is a matrix of rank n, b is a right-hand side vector with n components, and x is a vector of unknowns with n components. The method consists in eliminating all the unknowns but x_i from equation i. The sequential algorithm, assuming that indices start at 0, is easily written as follows:

```
void GaussJordan(double $A$[][], double $b$[], double $x$[], int $n$) {
  for $k = 0$ to $n-1$        /* $A[k,k]$ is the selected pivot */
    for $i = 0$ to $n-1$        /* Update every line $L_i$ */
      If $i \neq k$ {             /* $L_i \leftarrow L_i - \frac{A[i,k]}{A[k,k]} L_k$ */
        for $j = k$ to $n-1$ {
          /* No need to update the $j < k$ since they all have been zeroed
               in the previous iterations */
          $A[i,j] \leftarrow A[i,j] - (A[i,k]/A[k,k]) \times A[k,j]$
        }
        $b[i] \leftarrow b[i] - (A[i,k]/A[k,k]) \times b[k]$
      }
  for $i = 0$ to $n-1$
    $x[i] \leftarrow b[i]/A[i,i]$
}
```

Question III.1. *Preliminary question. [1 point]*

a) *[0.5 point] What sub-matrix of A is updated at step k?*

b) *[0.5 point] What is the complexity of the Gauss-Jordan algorithm?*

Question III.2. *In this series of questions, we aim at designing a parallel version of this algorithm for a ring of p processors. [4 points]*

a) *[0.5 point] Identify the parallelism of the previous algorithm (i.e., which loops are sequential, which loops are parallel).*

b) *[0.5 point] Assume now that we give each processor $i \in [0, p-1]$ a vertical slice $A[0, n-1][i\frac{n}{p}, (i+1)\frac{n}{p}-1]$ of the matrix A. What would be bad with such a distribution? Hint: Check the load balancing for $k = n/2$.*

c) *[0.5 point] Assume we give each processor $i \in [0, p-1]$ an horizontal slice $A[i\frac{n}{p}, (i+1)\frac{n}{p} - 1][0, n-1]$ of the matrix A. Would we have the same issue as previously?*

d) *[1 point] Describe a parallel algorithm using the previous distribution. We denote by ω the time needed to perform an update (i.e., $b[i] \leftarrow b[i] - (A[i,k]/A[k,k]) \times b[k]$), by λ the latency, and by β the inverse of the bandwidth. Therefore, sending a message of size m from P_i to P_{i+1} takes $\lambda + m.\beta$ time units and performing m updates takes $m.\omega$ time units.*

What is the execution time of your algorithm ?

e) *[1 point] Assume now that λ is rather large compared to β. How would you improve the previous algorithm ?*

f) *[0.5 point] Assume now that we give each processor $k \in [0, p-1]$ an horizontal cyclic slice of the matrix A, i.e., processor k has all the values $A[k+ip][j]$ for $i \in [0, n/p]$ and $j \in [0, n-1]$.*

Would we have the same load balancing issue as previously? Would this distribution require to heavily modify the implementation of the previous algorithm?

Question III.3. *Perform the same analysis with a grid of processors instead of a ring. [1 point]*

Question III.4. *Assume you need to implement a parallel version of the Gauss-Jordan algorithm (not necessarily relying on one of the previous parallel algorithm). What kind of parallel architecture and of programming library would you recommend ? [1 point]*