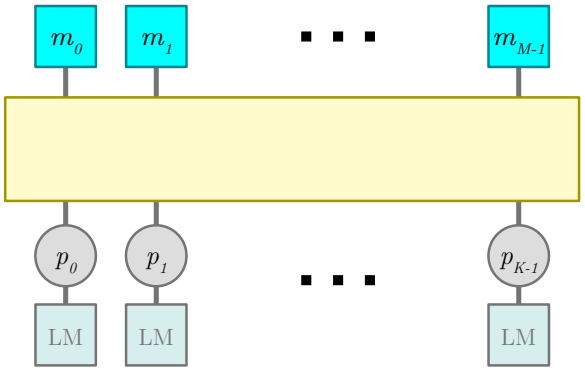
Parallel and Distributed Computing: Homework 1

Objectives : Use computer simulation to characterize the performance of a shared-memory distributed computing system under assumed workloads. The student is challenged to determine the best number of memory modules connected to a fixed number of processors to work on a given computational workload characterized by their memory access patterns. Using the C programming language, a computer simulation program is developed and models the computer's resources (processors, memory modules and interconnection network), models simulated time, models the computational workload, and implements the measuring system (metrology). Interpreting the results concludes the exercise.



System Definition: Consider a system, as the one depicted in the figure above, with K processors (with their local memories) and a shared memory subsystem consisting of M single-ported memory modules. Processors and memory modules are interconnected by a network capable of pairing any processor to any free memory module: at any given time, a non-paired requesting processor connects to one memory module if the module is free, otherwise the processor remains non-paired. A free memory module is one that is not currently paired with any processor. During normal operation, processors run code and request access to data in the shared memory.

The purpose of this assignment is to use computer simulation to characterize the access time from processors to memory modules under different configurations (number of processors and memory modules). This characterization is done different workloads exhibiting different memory access patterns, namely the manner in which a concurrent program accesses memory.

Memory cycles: For the purposes of simulation, global time is divided into equal successive intervals called "memory cycles".

System processing: At the beginning of each memory cycle, processors request access to memory modules. Following a priority scheme, each requesting processor is connected to its requested memory module if and only if the module is free. A memory module is free if and only if no other processor has previously been connected to that module during the memory pairing mechanism at the beginning of the current memory cycle.

Processor access to data is assumed to occur during the remainder of the memory cycle. All memory modules become free again at the end of every memory cycle.

Processors that are not granted a connection to their requested memory modules wait for the next memory cycle to again request the same memory module.

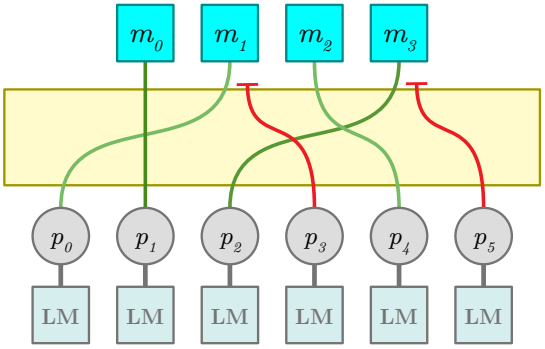
At the beginning of the next memory cycle, all processors that were granted access generate a new request while those who waited retry access to their same previously denied memory module.

Access-Time: If a processor p_k is connected to a memory module m_j, no other processor

p_i can connect to m_j during the same memory cycle. **Processor p_i has to wait** and attempt access to m_j in the next memory cycle.

The number of cycles that a processor p_i waits for the connection to its requested memory module is the *"wait-time"*, $W(p_i)$, of that connection. When a processor accesses a memory module on its first attempt (without waiting), $W(p_i) = 0$.

Counting the extra cycle during which a processor p_i accesses its requested memory module, the *"access-time"*, $T(p_i)$, is $W(p_i)+1$.



Memory Access Schemes: To pair processors and memory modules at the beginning of a memory cycle, any non-starving assignment scheme is acceptable.

As an example, consider a priority scheme such processors are granted access to memory modules in their natural index order (processors with lower index are granted access first). In the figure above, this is equivalent to scanning the processing elements from left to right and grant, in that order, memory access to those processors for which the requested memory module is free. In the example of the figure above, p_3 and p_5 will have to wait to another cycle to gain access to their requested modules. It is clear that this scheme suffers from starvation, some processors may never gain access to requested memory modules.

To fix the problem and avoid starvation, processors can be re-labeled at the beginning of a memory cycle such that the first "waiting" processor is re-labeled with index 0 and the remaining processors are cyclicly re-labeled in the natural order. The relabeling can be done dynamically while scanning, or a circular ordered data structure can be kept with a dynamic pointer to the first processor to be considered (which is the first processor with a denied access in the current scan of the processor list).

Assignment

You are asked to create a simulator in C with only standard libraries, to characterize the access-time of processors to memory modules under different configurations and workload characteristics.

Workload assumptions To represent the memory access demands of a workload, different memory access patterns can be used. Access patterns of an application represent the extent to which locality of reference is present in the workload. For the purposes of this assignment, two distributions of memory requests will be assumed:

- **Uniform distribution** Each processor issues a memory request to a random memory module at the beginning of each memory cycle using a uniform distribution.
- **Normal distribution** In a system with M memory modules, before the main simulation execution, every processor π selects one uniformly distributed random memory module μ_π. All subsequent memory requests of processor π will be given by $mod(X, M)$, where X is a discrete random variable normally distributed around μ_π with standard deviation 5.

Computation Let $S(P, M, D)$ be a system with P processors, M memory modules, and a workload distribution D. And let $S_c(P, M, D)$ be the system $S(P, M, D)$ at memory cycle c of its simulation.

- Given a fixed number of processors, for each number of memory modules $M \in [1, 512]$, a simulation shall be run and will be limited to a maximum of $C = 10^6$ cycles.

- As the simulation runs, each processor p_i requests and is granted access to memory modules, therefore, registering multiple access-times $T(p_i)$.

  During the simulation, at every cycle c, you are to compute each processors' time-cumulative average access-time $\overline{T}_c(p_i)$, and to obtain the arithmetic average of all processors' time-cumulative averages

  $$\overline{T}_c(S) = \frac{\Sigma_{i=0}^{i<P}(\overline{T}_c(p_i))}{P}$$

  Each processor's $\overline{T}_c(p_i)$ can be computed as the total number of simulated memory cycles c divided by the total number of memory access requests granted to that processor so far.

  Note that if by cycle c a processor p_i has never been granted memory access, then its only access-time $T(p_i)$ is still **undefined**, therefore $\overline{T}_c(p_i)$ is also undefined. This means that p_i is in a starving situation. Consider only defined access-times to compute $\overline{T}_c(S)$. (The equation above applies only if all processors have a defined $\overline{T}_c(p_i)$.)

- It is expected that the average system memory access-time $\overline{T}_c(S)$ settles at some stable value. Therefore, two termination conditions for the simulation of $S(P, M, D)$ will be considered.

  - The change between the current $\overline{T}_c(S)$ and previous $\overline{T}_{c-1}(S)$ is less than a certain tolerance ε

    $$Abs\left(1 - \frac{\overline{T}_{c-1}(S)}{\overline{T}_c(S)}\right) < \varepsilon$$

    Consider a small value for *varepsilon*, such as 0.02.

  - The maximum number of cycles is reached $c = C$

If your simulation reaches any of these termination conditions, but there are still processors with undefined $\overline{T}_c(p_i)$, then your system's memory assignment policy is **unacceptable** due to allowing starving processors.

Program specification: Your program will accept 2 command line arguments. You can assume that the given command line arguments will always be correct.

- A positive integer p specifying the number of processors to simulate.

- A lowercase character d specifying the distribution of the memory requests with the only possible values `'u'` (uniform) or `'n'` (normal).

Using the code template provided, your programming task is to complete the function `simulate()` present in the file `simulator.c` such that it fills the array `avg_access_time` with the **last $\overline{T}_c(S)$ of each one of the 512 simulations**. Put any needed functions and data structures in `simulator.c`, which is **the only source file to be submitted**.

Do not use libraries other than the ones found in GCC, ($9 \leq$ GCC version < 12). Use the uniform and normal random number generators given in the code template. The final results are printed in `main.c`, use this to export and plot the values obtained.

Verification of feasibility As long as your code is contained in `simulator.c`, and the signature of your function is the one we present to you in `simulator.h`, you are free to design your simulation program in any which way you like. However, you can use the following considerations/specifications:

- Using 3 equal size arrays to represent each processors' request, access counter, and priority of connection. Think of this as being a data structure containing K entries, one per each processor, and each entry having, for each processor, a request, access counter, and a priority of connection.

- Another array to represent the memory modules. Each element in this array has a 1 if the represented memory module is already connected to a processor or 0 if it is free.

- To avoid processing element starvation, you can use the cyclic priority index assignment described above to effect processor-memory pairings. There may be better stratergies than this one.

Report

You are to digitally produce an individual report of two pages. *Do not forget to include the student ID number and name of all the students in this team in the first page of your report.*

Charts The first page of your report will contain two charts: The first showing results for the uniform distribution of memory requests, and the second showing the results for normal distribution. Make sure the charts are clear and easy to read.

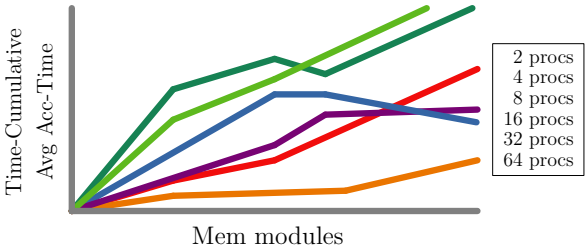
Each chart consists of an X-Y cartesian drawing in which the results of the simulations are plotted. A plot is a curve showing the values on the Y-axis of a variable as a function of values in the X-axis.

- In each chart the X-axis represents the number of memory modules varying from 1 to 512 while the Y-axis shows the last time-cumulative average access-times $\overline{T}_c(S)$ from each simulation.

- X-axis must be linear and Y-axis logaritmic.

- Generate one plot (curve) for each fixed number of processors $\{2, 4, 8, 16, 32, 64\}$. Therefore, there must be 6 curves per chart. The data for the plots is generated by your simulator as a set of $\overline{T}_c(S)$ that can be captured on a file by redirecting the command line output, you can use any available tool to generate the plots/charts, such as LibreOffice, Python+Matplotlib, Google Sheets, Microsoft Office, etc.

In summary, parameterizing each chart on the number of processors, six plots (curves) will be superimposed into each chart. An example is given in the follwoing figure. Note that the shape of the plots in this example are not representative.



Discussion In the second page, you will include the analysis and interpretation of the obtained results. Feel free to explain any important aspect of your findings. Also, include a discussion to these questions:

- How does the memory request distribution affects the behavior of the system? Which one is more realistic?

- If this simulation was in the context of making a decision to buy expensive memory modules for a given number of powerful processors, what would you recommend? Why?

Submission

Submit **one** zip file named `hw1.zip` on Canvas, containing **exactly 3 files**. Make sure the zip file does **not** include sub-directories (as Mac's default compression tool does) or extra files:

1. `simulator.c`: The well commented C source code file implementing the function `simulate()`.
2. `report.pdf`: The digitally produced report of 2 pages.
3. `team.txt`: Text file where each line contains, separated by commas, the UCINetID, first name, and last name of each student, for example:
   `ptanteater, Peter, Anteater`