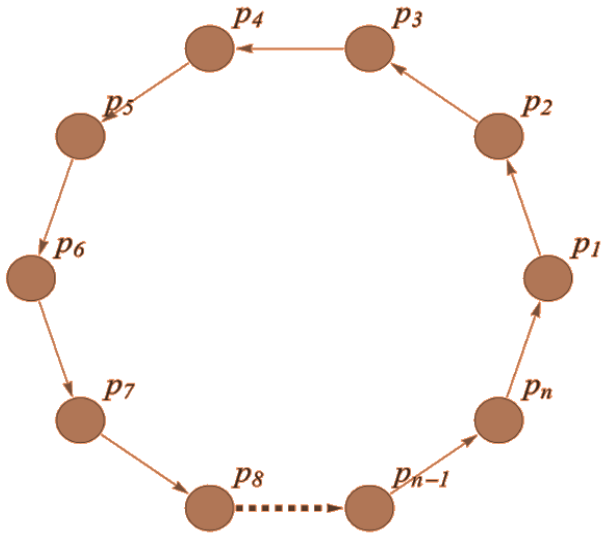

Processe Communication

Assignment 5

Ben-Gurion University of the Negev

Faculty of Engineering Science

Dept. of Communication Systems Engineering

Functional Programming in Concurrent and Distributed Systems

381-1-0112

Spring 2019

Maor Assayag 318550746

Table of contents

System design

Exported functions

The system will consist of 6 main functions that will be exported from the module **'exf_318550746.erl'** : mesh, mesh_serial, ringA, ringB.

receiveMsg & *receiveMsgB* will not be exported - I am using lambda (*fun()-> ..)* to enable using it on spawn.

ringA/2

Description

ringA(N , M) → time.

a. creates a circle of **N** processes that can communicate in the order of the circle (clockwise), every process creates its next neighbor, one after another.
b. When the construction of the circle is completed, **M** messages need to be transmitted by process number 1 (P1), The next message can be sent only when the previous message has arrived to P1.
c. When P1 receives the **M**th message, it should print the total time it took all of the **M** messages to pass through the circle.

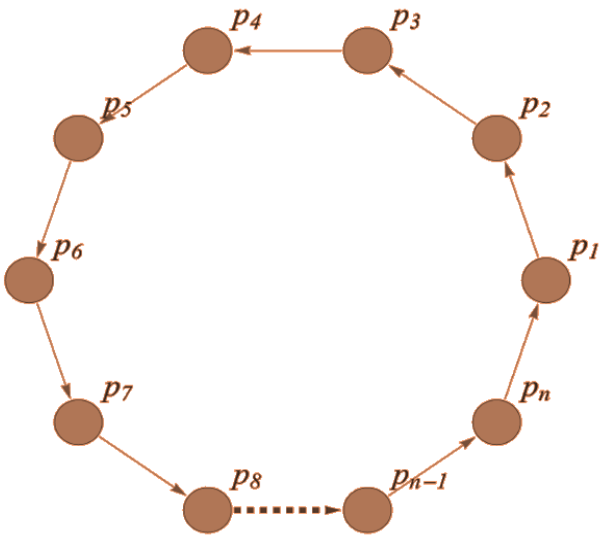

Figure 1.1 – example ring

Algorithm Flow

1. Get a new atom name for a new process P1 – pid1 and use *register* to attach the atom to the actual PID returned from *spawn* on the function in my module called *receiveMsg*, which used to receive messages with **receive** statements.

2. Send from the main process a message to P1 to start creating the circle of processes structure by each step sending the new process created a new request to create the next process etc.

3. Once the creation of the ring has been completed, the last process P_n will send new message with new format to P_1 to start sending M messages to the ring (to P2 etc).

4. When P1 receive the same message again from P_n it will send new message etc until this process is performed M times.

ringB/2

Description

ringB(N , M) → time.

a. creates a circle of **N** processes that can communicate in the order of the circle (clockwise), all processes are created by a central process.
b. When the construction of the circle is completed, **M** messages need to be transmitted by process number 1 (P1), The next message can be sent only when the previous message has arrived to P1.
c. When P1 receives the **M**th message, it should print the total time it took all of the **M** messages to pass through the circle.

Algorithm Flow

1. Get a new atom name for a new process P1 – pid1 and use *register* to attach the atom to the actual PID returned from *spawn* on the function in my module called *receiveMsg*, which used to receive messages with **receive** statements.

2. All processes are created by a central process

3. Once the creation of the ring has been completed, the central process will send new message with new format to P_1 to start sending M messages to the ring (to P2 etc).

4. When P1 receive the same message again from P_n it will send new message etc until this process is performed M times.

mesh/3

Description

mesh(N, M, C) → time.

a. The function creates a mesh (grid) of NxN processes that are connected
   C is a number from 1 to N^2 that defines the master process.
   Sending M messages from the master process C to its neighbors.

b. Each node that accept the message from C returns an answer as a response to it by sending the response message to its neighbors. Each node passes an answer message to its own neighbors including C.

c. Each process is able to communicate in both directions with its neighbors.

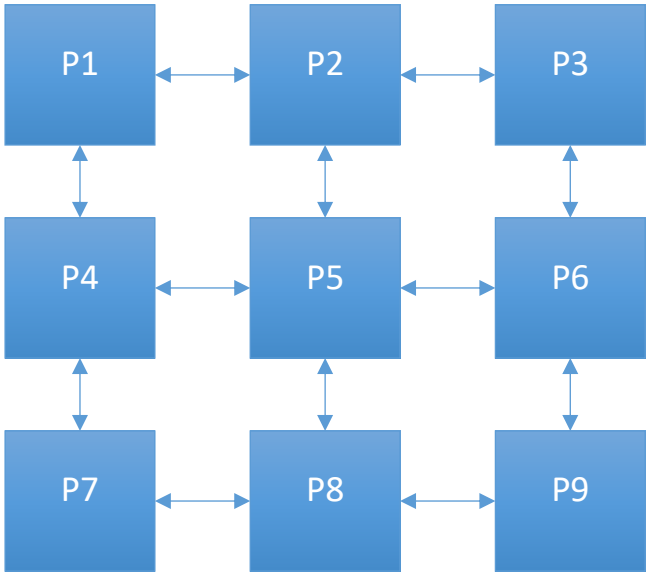

Figure 1.1 – example mesh grid of 3x3

Algorithm Flow

1. Register the main process as 'main', used later to initiate the kill sequence of the processes in the mesh.

2. Create the mesh grid by the main process using the aid function *createGrid.*

3. Once the mesh grid has been built, the main process send a **{"start"}** message to the C process (central process).

4. Process **C** is in *receiveMsgB* block-receive. Once the start message has been arrived, the process initializes some parameters in the process

dictionary (using *put*), e.g. total message, response etc, then it calles *sendMessages*.

5. In *sendMessages* C sends to its 4 neighbors (if exists) **M** messages in the format : **{SenderNode, MsgIndex, MsgType}** which in this case yields **{C, MsgIndex, "receive"}**, using *sendNeighbors*, sendNeighbor, *point2Node* & *node2Point* aid functions, and goes to *receiveMsgB* block-receive.

6. Each process will pass **"receive"** message to its neighbors and send a **"response"** message to its neighbors (if its not Process C and its not seen it already, using the process dictionary for tracking which message has already been arrived with pattern matching ang *get*),

7. Once Process C will get a total responses of $M * (N^2 - 1)$ messages, a final promotion will be displayed "The operation took ~p ns with ~p responses sent back to C and ~p total messages \n", and a **kill** message will be sent to the *main* process.
8. Once the main process will get **"done"** message – *killAll* function will be called which using exit to terminate the grid mesh processes.

Mesh_serial/3

Description

mesh_serial(N, M, C) → time.

d. Simulate a mesh (grid) of NxN processes that are connected, Sending M messages from the master process C to its neighbors.

e. Each node that accept the message from C returns an answer as a response to it by sending the response message to its neighbors. Each node passes an answer message to its own neighbors

including C.

f. The process communicates only with itself, simulating a grid of processes which leads to serial and non-parallel / distributed computing.

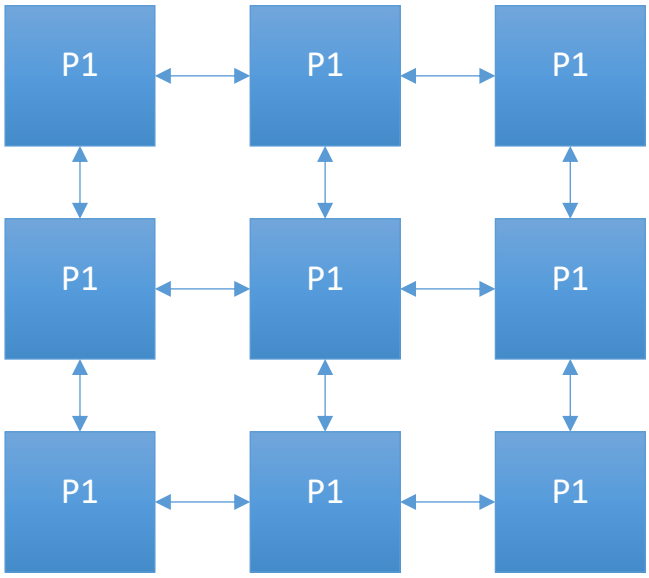

Figure 1.1 – example mesh grid of 3x3

Algorithm Flow

1. the main process send a **{"start"}** message to itself, then proceed to *receiveMsgB* block-receive.

2. The main process is in *receiveMsgB* block-receive. Once the start message has been arrived, the process initializes some parameters in the process dictionary (using *put*), e.g. total message, response etc, then it calles *sendMessages as from node C*.

3. In *sendMessages* C sends to its 4 neighbors (if exists) **M** messages in the format : **{SenderNode, MsgIndex, MsgType}** which in this case yields **{C, MsgIndex, "receive"}**, using *sendNeighbors*, sendNeighbor,

point2Node & *node2Point* aid functions, and goes to *receiveMsgB* block-receive.

4. The process will pass **"receive"** message to its neighbors and send a **"response"** message to its neighbors (if its not Process C and its not seen it already, using the process dictionary for tracking which message has already been arrived with pattern matching ang *get*),

5. Once Process C will get a total responses of $M * (N^2 - 1)$ messages, a final promotion will be displayed "The operation took ~p ns with ~p responses sent back to C and ~p total messages \n", and the local dictionary will be erase (to free resources).

Aid functions

receiveMsg/3

Description

receiveMsg (N, M, InitialTime) → receiveMsg loop.

a. Receiving messages in a loop, a message can create a new process, send forward messages & finish the operation.

b. Message Format 1 - **{"create", Index}**
if Index < N : create a new process, send it "create" message and keep looping this function, used in ringA.

Message Format 2 - **{CurrIndex, CurrMsg}**
its $P_{CurrIndex}$ when **CurrIndex** > 1, pass the message to the next process in the ring and keep looping this function (block/receive).

receiveMsgB/4

Description

receiveMsgB (N, M, C, CurrNode, IsSerial) → receiveMsg loop.

a. Receiving messages in a loop, a message can create a new process, send forward messages & finish the operation.

b. Message Format 1 - **{"start"}**
Initialize some parameters about this request (total messages, response , initial time) in the process dictionary and start sending M messages as from node C to the neighbors of C (as described above).

Message Format 2 - **{SenderNode, MsgIndex, MsgType}**
Calles *analyzeMsg* to further analyze the current message, if the node need to pass it/ response to it etc.

ringB_loop/4

Description

ringB_loop (*CurrIndex, N, M, InitialTime*) simply spawns N process using spawn on the function receiveMsg.
register(getProcessName(CurrIndex), spawn(ex5_318550746, receiveMsg, [N, M, InitialTime])),

getTime/0

Description

getTime( ) Return the current Time stamp using *os : system_time(nanosecond)* in nanosecond time resolution.

getProcessName/1

Description

getProcessName (*Index*) Return a name represent a process with the index Index, used later to register the PID with this name.

list_to_atom("pid" ++ integer_to_list(Index))

e.g. getProcessName(1) -> "pid1".

killAll/2

Description

killAll (*N, C*) kill all **N*N** processes, when **C** is the central process. Using the dictionary to store the actual PIDs of the processes, and the BIF functions ***exit***, ***whereis*** and the aid function ***getProcessName***. Used for ***mesh*** function.

killAll(N,C,Curr) -> exit(whereis(getProcessName(Curr)), kill), killAll(N,C,Curr+1).

createGrid/5

Description

createGrid (*N, M, C, Curr,IsSerial*) Create a grid of N^2 processes and spawns them using *spawn* on the function *receiveMsg*B. Upon completion, the main process will send a **{"start"}** to process **C**.

sendMessages/4

Description

sendMessages (*N, M, C, IsSerial*) C (central) process is sending M 'receive' messages to the grid. Each message added to C dictionary and been sent to its neighbors.

sendNeighbors/8

Description

sendNeighbors (*CurrNode, N, MsgIndex, MsgType, SenderNode, IsSerial, M, C*)

send to all neighbors of CurrNode a message MsgIndex of type MsgType from SenderNode.

First get the current node index on the grid using *node2Point* and then send all 4 neighbors(if exists) this message **{SenderNode, MsgIndex, MsgType}** using *sendNeighbor* and *point2Node*.

sendNeighbor/8

Description

sendNeighbor (*CurrNode, N, MsgIndex, MsgType, SenderNode, IsSerial, M, C*)

Send *CurrNode* the message **{SenderNode, MsgIndex, MsgType}**, if *CurrNode* exists.

If IsSerial = true, send the message to self().

point2Node/3

Description

point2Node (I, J, N) -> I*N + J+1, Number from 1 to N^2 represent the process on the NxN grid.

node2Point/2

Description

node2Point (Node, N) -> {(Node-1) div N , (Node-1) rem N}, tuple of 2 indexes represent the process location on the NxN grid.

getMsgName/4

Description

getMsgName (SenderNode, MsgIndex, MsgType, CurrNode) -> convert message index and MsgType to list to be registerd in the process dictionary.

analyzeMsg/2

Description

analyzeMsg
(undefined,{SenderNode,MsgIndex,MsgType},N,C,M,C,IsSerial) ->
analyze the current message received in *receiveMsgB*. *Undefined* is the results of *get* on the message tuple in the dictionary.

a. If current node is C, and the message is "response" that didn't been received so far, add it to the total response received so far. If **C** got $M * ((N * N) - 1)$ responses, end the operation with the promotion print.

b. If current node isn't **C**, and the message is **"response"** that didn't been received so far, pass it to the current node neighbors.

c. If current node isn't **C**, and the message is **"receive"** that didn't been received so far, pass it to the current node neighbors and also send a **"response"** message with the **MsgIndex** and the *getMsgName*(**self()**) as **SenderNode**.

d. If current node is C, and the message is **"receive",** add it to the total messages (**"receive"** messages sent only from C, so its already in its dictionary).

Analysis

Part A - rings

Comparing ringA and ringB system runtime using N,M from 10 to 1000 are summarized in the following table :

Parameters		Time took [ns]	
N processes	M messages	ringA	ringB
10	10	744931	632653
100	100	157824878	139564278
1000	1000	9908881803	10316047627

Table 3.1 – Analysis of different *ring* systems

Figure 3.1 – evaluating on linux

We can see the effect of the distribution of the task to create the processes vs creating the processes of the cycle in the main process.

The differences become dominant as scale goes up. It's not surprising that even if the computing task itself (passing M messages in the ring) is done by multiple processes at once, the creation of the environment itself has an effect on total performance and runtime .

(I am using nanosecond resolution for the time unit as described above, and each run I get a range of results depending on the resources & compute power – all around the amount of time mention in the table).

Part B – mesh

Comparing mesh and mesh_serial system runtime using with N in range from 2 to 5 , 3 different centers for each and with 10,100,1000 messages are summarized in the following table :

Parameters			Time took [ns]		Total Messages	
N processes	M messages	C center	mesh	mesh_serial	mesh	mesh_serial
2	10	1	2866894	2112058	80	98
2	10	2	1932385	1967144	70	147
2	10	3	2821830	3669987	70	147
2	100	1	138060707	95741678	975	998
2	100	2	127842693	91606592	843	1497
2	100	3	176545581	95713908	900	1497
2	1000	1	911349911	933077661	9973	9998
2	1000	2	494256967	805191652	9906	14997
2	1000	3	915116198	921800417	10177	14997
3	10	1	69415339	48502978	199	198
3	10	5	53966021	40173014	313	396
3	10	7	76696695	41976586	265	297
3	100	1	781654467	445646225	1997	1998
3	100	5	518458219	461425874	2812	3996
3	100	7	817467321	485697571	2887	2997
3	1000	1	8631365104	4757212505	19947	19998
3	1000	5	5603323887	4685313425	28316	39996
3	1000	7	8761419588	4573561461	28858	19998
4	10	1	276676337	159118833	338	338
4	10	10	207818935	142977950	590	676
4	10	16	261158010	126258816	331	338
4	100	1	4076397459	734217299	3360	3398
4	100	10	2290302283	700922943	6536	6796

4	100	16	2908464971	687221037	3371	3398
4	1000	1	34162881918	14068501213	33981	33998
4	1000	10	27541301029	10514525957	63784	67996
4	1000	16	32834923411	8266029903	33887	33998
5	10	1	377490013	228476138	516	518
5	10	12	307134613	253494487	892	1036
5	10	25	351494430	233433776	515	518
5	100	1	3232103397	1546496801	5189	5198
5	100	12	2706189712	1628213865	9202	10396
5	100	25	3269249546	1719180970	5198	5198
5	1000	1	42151324268	20938083759	51968	51998
5	1000	12	32572107832	20684194595	90961	103996
5	1000	25	43286896607	19806751306	51955	51998

Table 3.1 – running analysis with varies parameters

```
53> ex5_318550746:mesh_serial(2,100,1).
The operation took 95741678 ns with 300 responses sent back to C and 998 total m
essages
finished
54> ex5_318550746:mesh_serial(2,100,2).
The operation took 91606592 ns with 300 responses sent back to C and 1497 total
messages
finished
55> ex5_318550746:mesh_serial(2,100,3).
The operation took 95713908 ns with 300 responses sent back to C and 1497 total
messages
finished
```

Figure 3.1 – evaluating on linux snapshot

(I am using nanosecond resolution for the time unit as described above, and each run I get a range of results depending on the resources & compute power – all around the amount of time mention in the table).

1. I can see how the location of C in the grid is affecting the runtime. Its logical because the location of C changed how many messages get through it and how faster response messages can get to him (taking into account how the system grant computing to processes , task scheduler etc.). I can

see correlation between how much C is centered in the grid to faster runtime / increasing amount of message through.

2. There is no absolute winner on runtime among mesh / mesh_serial, this is due to the process scheduling that most likely interfering with optimizing mesh architecture, and sometimes is noticeable that the frequent context switching slowing down the performance (taking into account the local hardware, how the system grant computing to processes , task scheduler etc.), and this become dominant as scale goes up. As hardware improvement will be present (e.g. more cores) mesh performance will improve as well (and maybe dominant over mesh_serial).

3. The linear correlation between the **total number of messages** and **M** in **mesh_serial** architecture is noticeable, when increasing **M** 10 times causing the total message that goes through C been incresead10 times as well.

4. It's important to remember the advantage of parallel / distributed computing (mesh) as scale and data size goes up. Even if in this particular implementation and **testing (on singular PC)** we don't see it.

Thank you.