

Parallel Programing

With MATLAB Examples

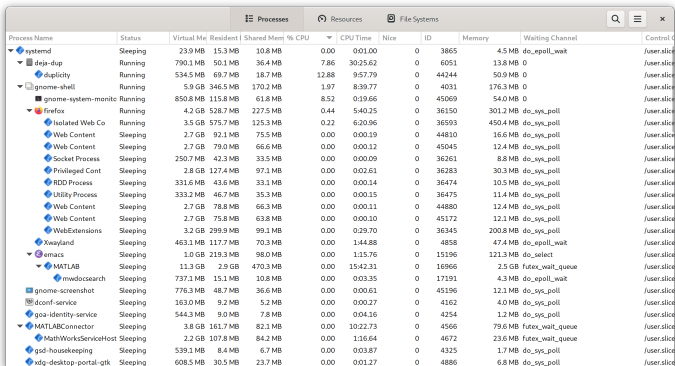
Marek Rychlik

Department of Mathematics
University of Arizona

February 18, 2023

Parallelism in the OS

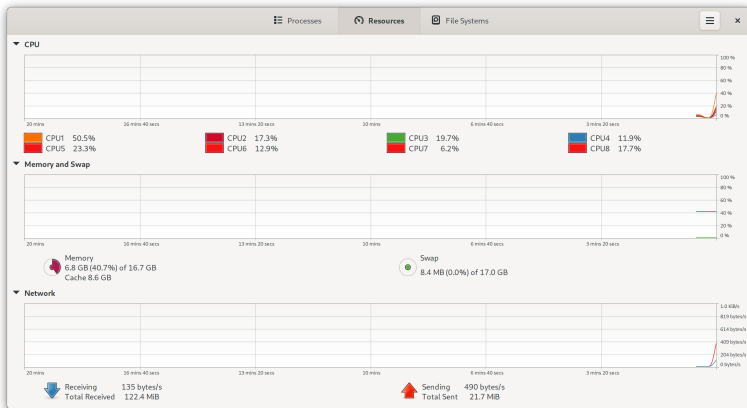
- A modern OS has a multitude of processes running, as shown by a **system monitor**
- OS creates **an illusion of parallelism** even if it runs on a single CPU not capable of multi-threading in hardware.



Process Name	Status	Virtual Mem	Resident	Shared Mem	% CPU	CPU Time	Nice	ID	Memory	Waiting Channel	Control
systemd	Sleeping	23.9 MB	15.3 MB	10.8 MB	0.00	0:01.00	0	3865	4.5 MB	do_epoll_wait	user.slice
deja-dup	Running	790.1 MB	50.1 MB	36.4 MB	7.86	30:25.62	0	6051	13.8 MB	0	user.slice
duplicit	Running	534.5 MB	69.7 MB	18.7 MB	12.88	9:57.79	0	44244	50.9 MB	0	user.slice
gnome-shell	Running	5.9 GB	346.5 MB	170.2 MB	1.97	8:39.77	0	4031	176.3 MB	0	user.slice
gnome-system-monitor	Running	850.8 MB	115.8 MB	61.8 MB	8.52	0:19.66	0	45069	54.0 MB	0	user.slice
firefox	Running	4.2 GB	528.7 MB	227.5 MB	0.44	5:40.25	0	36150	301.1 MB	do_sys_poll	user.slice
Isolated Web Co	Running	3.5 GB	575.7 MB	125.3 MB	0.22	6:20.96	0	36593	450.4 MB	do_sys_poll	user.slice
Web Content	Sleeping	2.7 GB	92.1 MB	75.5 MB	0.00	0:00.19	0	44810	16.6 MB	do_sys_poll	user.slice
Web Content	Sleeping	2.7 GB	79.0 MB	66.6 MB	0.00	0:00.12	0	45045	12.4 MB	do_sys_poll	user.slice
Socket Process	Sleeping	250.7 MB	42.3 MB	33.5 MB	0.00	0:00.09	0	36261	8.8 MB	do_sys_poll	user.slice
Privileged Cont	Sleeping	2.8 GB	127.4 MB	97.1 MB	0.00	0:02.61	0	36283	30.3 MB	do_sys_poll	user.slice
RDD Process	Sleeping	331.6 MB	43.6 MB	33.1 MB	0.00	0:00.14	0	36474	10.5 MB	do_sys_poll	user.slice
Utility Process	Sleeping	333.2 MB	46.7 MB	35.3 MB	0.00	0:00.15	0	36475	11.4 MB	do_sys_poll	user.slice
Web Content	Sleeping	2.7 GB	78.8 MB	66.3 MB	0.00	0:00.11	0	44880	12.4 MB	do_sys_poll	user.slice
Web Content	Sleeping	2.7 GB	75.8 MB	63.8 MB	0.00	0:00.10	0	45172	12.1 MB	do_sys_poll	user.slice
WebExtensions	Sleeping	3.2 GB	299.9 MB	99.1 MB	0.00	0:29.70	0	36345	200.8 MB	do_sys_poll	user.slice
Xwayland	Sleeping	463.1 MB	117.7 MB	70.3 MB	0.00	1:44.88	0	4858	47.4 MB	do_epoll_wait	user.slice
emacs	Sleeping	1.0 GB	219.3 MB	98.0 MB	0.00	1:15.76	0	15196	121.3 MB	do_select	user.slice
MATLAB	Sleeping	11.3 GB	2.9 GB	470.3 MB	0.00	15:42.31	0	16966	2.5 GB	futex_wait_queue	user.slice
mwdocsearch	Sleeping	737.1 MB	15.1 MB	10.8 MB	0.00	0:00.35	0	17191	4.3 MB	do_epoll_wait	user.slice
gnome-screenshot	Sleeping	776.3 MB	48.7 MB	36.6 MB	0.00	0:00.61	0	45196	12.1 MB	do_sys_poll	user.slice
iconf-service	Sleeping	163.0 MB	9.2 MB	5.2 MB	0.00	0:00.27	0	4162	4.0 MB	do_sys_poll	user.slice
goc-identity-service	Sleeping	544.3 MB	9.0 MB	7.8 MB	0.00	0:04.16	0	4254	1.2 MB	do_sys_poll	user.slice
MATLABConnector	Sleeping	3.8 GB	161.7 MB	82.3 MB	0.00	10:22.73	0	4566	79.6 MB	futex_wait_queue	user.slice
MathWorksServiceHost	Sleeping	2.2 GB	107.8 MB	84.2 MB	0.00	1:16.64	0	4672	23.6 MB	futex_wait_queue	user.slice
gsd-housekeeping	Sleeping	539.1 MB	8.4 MB	6.7 MB	0.00	0:03.87	0	4325	1.7 MB	do_sys_poll	user.slice
xdg-desktop-portal-gtk	Sleeping	608.5 MB	30.5 MB	23.7 MB	0.00	0:01.27	0	4886	6.8 MB	do_sys_poll	user.slice

Figure: Explanations

How many CPUs/Hardware threads do I have?



Forking in Bash (&) — a minimal variant

```
1 (sleep 1e-2; echo -n "Hello , ") & \  
2 (sleep 1e-2; echo -n "World!")
```

Forking in Bash (&)

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1  #!/bin/sh
2  # EXAMPLE: Print 'Hello, ' and 'World!'
3  # in random order w/o a random number generator.
4  # HINT: We deliberately create a race condition.
5  if (($#)) ;then ntimes=$1 ;else ntimes=10; fi
6  function hello {
7      echo -n "Hello, "
8  }
9  function world {
10     echo -n "World!"
11 }
12 dlay=1e-2 # Change to 5 to see processes
13 for (( j=0; $j<$ntimes; j=$j+1 ))
14 do
15     # Fork with '&'
16     (sleep $dlay; hello) & (sleep $dlay; world)
17     echo " --Done with iteration: $j"
18 done
```

Forking in Bash (&) II

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
[marek@cannonball]$ ./forkme.sh
Hello, World! --Done with iteration: 1
Hello, World! --Done with iteration: 2
World!Hello, --Done with iteration: 3
World!Hello, --Done with iteration: 4
Hello, World! --Done with iteration: 5
World!Hello, --Done with iteration: 6
Hello, World! --Done with iteration: 7
Hello, World! --Done with iteration: 8
Hello, World! --Done with iteration: 9
```

A remarkable, more rare output

```
[marek@cannonball matlabmpi]$ ./forkme.sh
Hello, World! --Done with iteration: 1
Hello, World! --Done with iteration: 2
Hello, World! --Done with iteration: 3
Hello, World! --Done with iteration: 4
Hello, World! --Done with iteration: 5
Hello, World! --Done with iteration: 6
World! --Done with iteration: 7
Hello, World!Hello, --Done with iteration: 8
Hello, World! --Done with iteration: 9
```

A Glossary of Terms I

program counter The location (address) of the instruction currently being executed; a place in a program

process A running program with all necessary resources (program counter, open file descriptors, memory state)

fork, forking, clone The UNIX/Linux **system call** which allows one process to create another one

IPC, inter-process communication The protocol by which two distinct processes can exchange information

thread (of execution) Formerly known as a **light-weight process** directly shares the state of memory (variables) with other threads; threads have **separate program counters**; a modern process is a **collection of threads**

A Glossary of Terms II

process/thread synchronization Mechanisms by which one process tells another not to mess with some sensitive parts of its state; IPC can be used for process synchronization; threads are synchronized by **mutexes**

mutex, futex A mutually exclusive lock, which is a simple integer (logical) variable which is set/unset (=acquired/released) by a thread. What is important is the **interpretation** by another thread. A thread agrees not to do certain things when mutex is acquired by another, until it is released. **Semaphores** generalize mutexes to arbitrary integer values. **Futex** is a **fast mutex**, introduced by the Linux OS.

A Glossary of Terms III

atomicity Some operations need to be atomic, such as changing the value of a mutex/semaphore. Atomicity means that a thread that reads the value of a mutex does not get an inconsistent value while another thread is **in the process of changing it**. Normal variables cannot be used as mutexes because reading and writing to them **is not atomic**. Atomicity is implemented using hardware (special instructions) and compiler (awareness that some variables must be changed atomically).

MATLAB 'parfor' (parallel for) I

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 % FILE: parforEx.m
2 %p=gcp();
3 %mpilnit;
4 n=10;
5 % Evaluate x^2 for 1:n asynchronously and print
  results
6 parfor i=1:n
7     x=i^2
8     %pause(1);
9     disp([i,x]);
10 end
11 disp(' All done');
```

MATLAB 'parfor' (parallel for) II

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
parforEx
```

```
    1      1
```

```
    5     25
```

```
    9     81
```

```
    2      4
```

```
   10    100
```

```
    3      9
```

```
    4     16
```

```
    6     36
```

```
    7     49
```

```
    8     64
```

```
All done
```

```
>>
```

Question

Why does 'All done' print only once? Only at the end?

Parallel pools, workers, clusters

- TIP: first install **Parallel Computing Toolbox** and try its GUI to configure a cluster
- **Workers** are a MATLAB abstraction of threads, and they should directly map to hardware (CPU, hardware threads)
- A **parallel pool** is a collection of workers under the management of the **main thread**
- A parallel pool can live on one or more CPUs, and can be distributed across many computers; these details are abstracted away
- A **cluster** is defined by a configuration file (a **profile**, eg., 'local.settings') and it specifies computers and the number of CPU used on each machine. The configuration file must be placed in one of several standard places (see 'help parcluster').

Accumulating values, reduction variables I

```
1 % FILE: reductionVar.m
2 % This file demonstrates a useful notion of a '
   Reduction Variable '
3 % Makes it possible to accumulate values in a
   parfor without using
4 % spmd/gop.
5
6 p=gcp( 'nocreate' );
7 if isempty(p)
8     p = parpool( 'local' , 8)
9 end
10
11 disp( sprintf( 'Number of workers: %d' , p.
   NumWorkers) );
12
13 x=[];
```

Accumulating values, reduction variables II

14

15

```
parfor i = 1:10
    pause(rand());
```

16

17

```
disp(i);
```

18

```
x = [x, i];
```

19

```
end
```

20

21

```
x
```

```
>> reduction_var
Number of workers: 8
```

```
x =
```

```
1      2      3      4      5      6      7      8      9     10
```

Fact

Deterministic: the answer is always the same.

Accumulating values, reduction variables III

Reduction Variables

R2018b

MATLAB® supports an important exception, called reduction, to the rule that loop iterations must be independent. A *reduction variable* accumulates a value that depends on all the iterations together, but is independent of the iteration order. MATLAB allows reduction variables in `parfor`-loops.

Reduction variables appear on both sides of an assignment statement, such as any of the following, where `expr` is a MATLAB expression.

<code>X = X + expr</code>	<code>X = expr + X</code>
<code>X = X - expr</code>	See Associativity in Reduction Assignments in Requirements for Reduction Assignments
<code>X = X .* expr</code>	<code>X = expr .* X</code>
<code>X = X * expr</code>	<code>X = expr * X</code>
<code>X = X & expr</code>	<code>X = expr & X</code>
<code>X = X expr</code>	<code>X = expr X</code>
<code>X = [X, expr]</code>	<code>X = [expr, X]</code>
<code>X = [X; expr]</code>	<code>X = [expr; X]</code>
<code>X = min(X, expr)</code>	<code>X = min(expr, X)</code>
<code>X = max(X, expr)</code>	<code>X = max(expr, X)</code>
<code>X = union(X, expr)</code>	<code>X = union(expr, X)</code>
<code>X = intersect(X, expr)</code>	<code>X = intersect(expr, X)</code>

SPMD and SIMD

SPMD Stands for “Single program, multiple data”.

Multiple autonomous processors

simultaneously execute the same program at independent points (program counters). Can be implemented on general purpose CPUs (Intel, AMD)

SIMD Stands for “Single-instruction, multiple data”. A **vector processor** processes the same instruction on different data (example: coordinatewise addition or multiplication of two vectors).

Modern CPU(s) implements **both paradigms**:

- SIMD uses Intel/AMD **SSE instructions** and **vector registers**;
- SPMD uses multiple **threads, cores** and CPUs.

The MPI (Message Passing Interface)

- The most successful realization of SPMD; used in MATLAB; 40 years of history
- Implementations in C, C++, Fortran exist, with high-level language interfaces (e.g., **Python**).
- Worker becomes a **lab**
- Worker knows its identity, or **labindex**
- The main thread is now a lab with **labindex==1** (recall: MATLAB has 1-based arrays)
- Labs communicate by using **collective communications**: `labSend`, `labReceive`, `labSendReceive`;
- synchronization: `labBarrier`, `labBroadcast`
- Labs can be organized as a graph with variable topology, e.g. edges of a hypercube, for the purpose of communicating with some neighbors

Unintended blocking — a show stopper

- A **blocking operation** is one that stops the execution of the program (thread, process) until some condition is met
- Example: reading from a file. We wait for the data to be available (e.g., read from disk or network)
- Example: waiting for a mutex to be released
- **labReceive**, **labBarrier** are blocking operations
- A **non-blocking operation** does not wait for the condition to be met but immediately continues with the execution, reporting status to the caller
- Example: reading from a file in non-blockin mode reports the number of bytes successfully read. One repeatedly reads from the file, getting the file in chunks, until the end-of-file marker is found

Deadlock (deadly embrace)

Definition (Deadlock)

Deadlock (which is sometimes called **the deadly embrace**) occurs when two or more programs (threads, workers, labs) are each waiting for the others to complete a task before proceeding.

The programs act like the overly congenial gophers in some Looney Tunes cartoons:

"Oh please, you first," says one. "No no, I insist, you first," says the other. And nothing goes anywhere.

An 'spmd' example (WRONG!) I

```
1 % FILE: race1.m
2 % This file demonstrates a simple race condition
3 % when trying to share a value between all
  workers
4 p=gcp( 'nocreate' );
5 if isempty(p)
6     p = parpool( 'local' , 8)
7 end
8
9 disp( sprintf( 'Number of workers: %d' , p.
    NumWorkers) ) ;
10
11 value = Composite() ;
12
13 % An incorrect way to broadcast a value and
14 % receive it in all workers
15
16 spmd
```

An 'spmd' example (WRONG!) II

```

17 pause(rand() ./10);
18
19 if labindex == 1
20     for w=1:numlabs
21         display(sprintf('%d sending 7 to %d',
22                         ,labindex,w));
23         labSend(7,w);
24     end
25 end
26
27 value = labReceive;
28 display(sprintf('%d received %d from 1',
29                 ,labindex,value));
30
31 end
32 for w=1:p.NumWorkers

```

An 'spmd' example (WRONG!)

III

OS

Parallelism

Trivial

Parallelism

An Intro to
MPITroop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

33

```
disp(value{w});
```

34

```
end
```

```
>> race1
Number of workers: 8
Worker 1:
  1 sending 7 to 1
Error using race1
Error detected on worker 1.

Caused by:
  Error using race1
  Destination (1) is same as source, would cause deadlock.

>>
```

An 'spmd' example (CORRECT!) I

```
1 % FILE: race1Fixed.m
2 % This file demonstrates a simple race condition
3 % when trying to share a value between all
  workers.
4 % NOTE: We avoid sending to ourselves. This
5 % avoids the race condition.
6
7 p=gcp( 'nocreate' );
8 if isempty(p)
9     p = parpool( 'local' , 8)
10 end
11
12 disp( sprintf( 'Number of workers: %d' , p.
    NumWorkers) );
13
14 value = Composite();
```


An 'spmd' example (CORRECT!) II

```
% An incorrect way to broadcast a value and  
% receive it in all workers
```

```
spmd
```

```
    pause(rand() ./10);
```

```
    if labindex == 1
```

```
        for w=2:numlabs
```

```
            display(sprintf('%d sending 7 to %d',  
                            labindex,w));
```

```
            labSend(7,w)
```

```
        end
```

```
        value = 7;
```

```
    else
```

```
        value = labReceive;
```

An 'spmd' example (CORRECT!) III

```
30         display ( sprintf ( '%d received %d from 1 ',  
31                        end    labindex , value ) ) ;  
32  
33     end  
34  
35     for w=1:p.NumWorkers  
36         disp ( [w, value {w}] ) ;  
37     end
```

An 'spmd' example (CORRECT!) IV

```
>>
Number of workers: 8
Worker 1:
  1 sending 7 to 2
  1 sending 7 to 3
  1 sending 7 to 4
  1 sending 7 to 5
  1 sending 7 to 6
  1 sending 7 to 7
  1 sending 7 to 8
Worker 2:
  2 received 7 from 1
Worker 3:
  3 received 7 from 1
Worker 4:
  4 received 7 from 1
Worker 5:
  5 received 7 from 1
Worker 6:
  6 received 7 from 1
Worker 7:
  7 received 7 from 1
Worker 8:
  8 received 7 from 1
  1      7

  2      7
```

An 'spmd' example (CORRECT!) V

3	7
4	7
5	7
6	7
7	7
8	7

>>

Broadcasting (another fix) I

Marek Rychlik

OS

Parallelism

Trivial

Parallelism

An Intro to
MPITroop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 % FILE: race1FixedBroadcast.m
2 % This file demonstrates a simple race condition
3 % when trying to share a value between all
   workers
4 % NOTE: In this version, we avoid the race
   condition
5 % by using labBroadcast.
6
7 p=gcp( 'nocreate' );
8 if isempty(p)
9     p = parpool( 'local' , 8)
10 end
11
12 disp( sprintf( 'Number of workers: %d' , p.
   NumWorkers) );
13
14 value = Composite();
15
```

Broadcasting (another fix) II

```
16 % An incorrect way to broadcast a value and
17 % receive it in all workers
18
19 root=1;
20 spmd
21     pause(rand() ./10);
22     if labindex == root
23         value = 7;
24         value = labBroadcast(root, value);
25         display(sprintf('Root==%d broadcast %d',
26                           labindex, value));
27     else
28         value = labBroadcast(root, 666); %
29         Second inut ignored on root
30         display(sprintf('%d received %d from
31                           root==%d', labindex, value, root));
32     end
33 end
```

Broadcasting (another fix) III

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```

31
32 for w=1:p.NumWorkers
33     disp([w,value{w}]);

```

```

34 end

```

```

>>
Number of workers: 8
Worker 1:
    Root==1 broadcast 7
Worker 2:
    2 received 7 from root==1
Worker 3:
    3 received 7 from root==1
Worker 4:
    4 received 7 from root==1
Worker 5:
    5 received 7 from root==1
Worker 6:
    6 received 7 from root==1
Worker 7:
    7 received 7 from root==1
Worker 8:
    8 received 7 from root==1
    1      7

    2      7

    3      7

```

Broadcasting (another fix) IV

OS

Parallelism

Trivial

Parallelism

**An Intro to
MPI**

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

4 7

5 7

6 7

7 7

8 7

>>

Linear Troop Topology

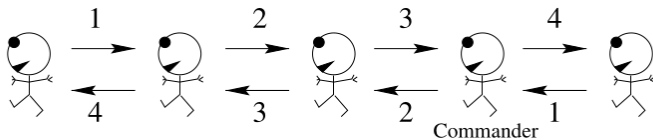


Figure: A line of soldiers counting themselves using message-passing rule-set A. The commander can add “3” from the soldier in front, “1” from the soldier behind, and “1” for himself, and deduce that there are 5 soldiers in total.

Message-passing rule-set A (parallel pseudo-code).

- 1 If you are the front soldier in the line, say the number **one** to the soldier behind you.
- 2 If you are the rearmost soldier in the line, say the number **one** to the soldier in front of you.
- 3 If a soldier ahead of or behind you says a number to you, add one to it, and say the new number to the soldier on the other side.

Implementation I

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 % FILE:      soldiers.m
2 % Mackay, algorithm 16.1: to count soldiers
3 % marching in line
4 %
5 % NOTE: If you are running the program on a
6 % processor with n=4 cores and 2*n
7 % hyperthreads, the setting for the 'local'
8 % cluster is used, and the number of workers
9 % in a parpool is set automatically to n,
10 % ignoring hyperthreading. You can modify
11 % the number of worker threads using Matlab
12 % GUI, using Home > Parallel > Manage Cluster
13 % Profiles > Edit. So, if you request 8
14 % workers, make sure to first edit the local
15 % profile and increase the number of allowed
16 % workers to >=8. I changed it to 64.
17 %
18 p = gcp('nocreate');
```

Implementation II

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
19 numSoldiers=5;
20 % Must have at least numSoldiers workers
21 if ~isempty(p) && p.NumWorkers < numSoldiers
22     delete(p);
23     p=[];
24 end
25 if isempty(p)
26     % Create a local parpool with num. workers
27     % == num. soldiers
28     p = parpool('local',numSoldiers);
29 end
30 mpilnit;
31 commander=2;
32 % Must have at least commander+1 workers
33 assert(p.NumWorkers > commander);
34 spmd
35     me=labindex;
36     value=0;
```

Implementation III

```
if me==commander
    [value1 , source1 , tag1]=labReceive ;
    fprintf( '%d=commander got %d from %d\n' ,
             me,value1 , source1 ) ;
    [value2 , source2 , tag2]=labReceive ;
    fprintf( '%d=commander got %d from %d\n' ,
             me,value2 , source2 ) ;
    value=value1+value2+1;
    fprintf( '%d=commander says: count is %d\
             n' ,me,value ) ;
elseif me==1
    value=1;
    dest=2;
    fprintf( '%d sending %d to %d\n' ,me,value
             ,dest) ;
    labSend( value , dest ) ;
elseif me==numlabs
    value=1;
```

Implementation IV

OS

Parallelism 50

Trivial 51

Parallelism

An Intro to MPI 52

Troop 53

Counting Example 54

Line graph topology 55

Tree graph topology Implementation 56

Wrapping up 57

```
dest=me-1;
fprintf( '%d sending %d to %d\n',me,value
        ,dest);
labSend( value , dest );

else
[ value , source ,~]=labReceive ;
value=value+1;
if source==me-1
    dest=me+1;
elseif source==me+1
    dest=me-1;
end
fprintf( '%d sending %d to %d\n',me,value
        ,dest);
labSend( value , dest );

end

end
```

General topology of the troop

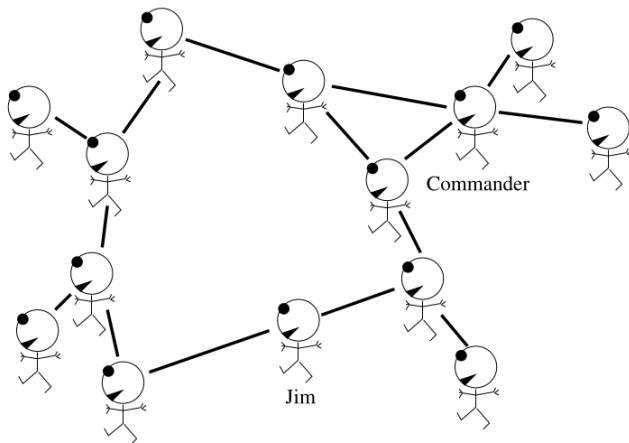


Figure: A swarm of guerillas.

Arranging workers/labs into a graph I

```
1 % FILE:      buildTroop.m
2
3 % Efficient graph encoding (often used in MPI
  programs)
4 % Adjacency matrix; passed as 1-d array, in which
  each vertex is followed
5 % Row format: node followed by neighbors ,
  sentinel 0.
6 % A final zero is added to terminate the
  structure .
7 adj=[1 ,2 ,0 ,...
8      2 ,1 ,3 ,11 ,0 ,...
9      3 ,2 ,4 ,5 ,0 ,...
10     4 ,3 ,0 ,...
11     5 ,3 ,0 ,...
12     6 ,8 ,0 ,...
```


Arranging workers/labs into a graph II

OS
Parallelism

Trivial
Parallelism

An Intro to
MPI

Troop
Counting
Example

Line graph topology
Tree graph topology
Implementation

Wrapping up

```

13     7,8,0,...
14     8,6,7,9,0,...
15     9,8,10,12,0,...
16     10,9,11,0,...
17     11,2,10,0,...
18     12,9,13,14,0,...
19     13,12,0,...
20     14,12,0,...
21     0];
22
23     % Automatically determine the number of soldiers
      (nodes)
24     numSoldiers = numel(find(adj==0))-1;
25     commander = 9; %
      Designate the commander
26
27     % Start the parpool (thread pool)
  
```

Arranging workers/labs into a graph III

```
28 p = gcp( 'nocreate' );
29 if ~isempty(p) && p.NumWorkers < numSoldiers
30     delete(p);
31     p=[];
32 end
33 if isempty(p)
34     p = parpool( 'local' , numSoldiers );
35 end
36
37
38 mpilnit;
39
40 % Convert nb to cell array
41 nb=Composite();
42 start=1;
43 for s=1:numSoldiers;
44     me=adj( start );
```

Arranging workers/labs into a graph IV

```
45 neighbors = [];  
46 n = start + 1;  
47 % Make a list of neighbors  
48 while adj(n) ~= 0  
49     neighbors = [neighbors, adj(n)];  
50     n = n + 1;  
51 end  
52 nb{me} = neighbors;  
53 start = n + 1;  
54 end  
55 assert(adj(start) == 0);  
56  
57 % A normal function call to let soldiers report  
58   the neighbors  
59 report(nb);  
60  
61 % A consistency check for graph data
```

Arranging workers/labs into a graph V

```
61 l=adjacency_matrix(nb);
62 % Check symmetry of the adjacency relation
63 assert(all(all(l==l')));
64 % Check for 'no loops' (loop=connection of edge
    to itself)
65 assert(all(diag(l)==0));
66 % Check for 'no cycles'; upper triangular
    portion of l should be nilpotent
67 assert(all(all(triu(l)^numSoldiers==0)));
68
69 %g = graph(l);
70 %plot(g,'LineWidth',4,'NodeFontSize',44,'
    MarkerSize',5,'NodeLabelColor','blue','
    NodeFontWeight','bold');
```

A tree topology of the troop

OS

Parallelism

Trivial

Parallelism

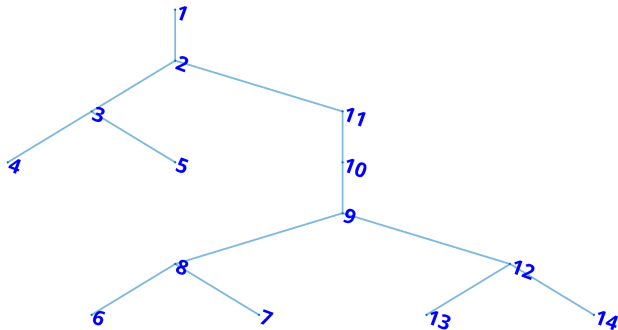
An Intro to
MPITroop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up



Message-passing rule-set B. I

```

1: procedure MESSAGEPASSING(Graph)
2:    $N \leftarrow$  the count your neighbours in Graph
3:    $m \leftarrow 0$  ▷ count of messages received from
     neighbours
4:   for  $j$  from 1 to  $N$  do
5:      $v_j \leftarrow -1$  ▷ initial value of message is invalid
6:   end for
7:    $V \leftarrow 0$  ▷ running total of messages you have
     received
8:   if  $m == N - 1$  then
9:     Find neighbor  $j$  such that  $v_j == -1$  ▷ the only one
     who has not sent you a message
10:    Tell them the number  $V + 1$ 
11:  end if

```

Message-passing rule-set B. II

OS
ParallelismTrivial
ParallelismAn Intro to
MPITroop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
12:   if  $V == N$  then
13:       the number  $V + 1$  is the required total.
14:       for each neighbour  $n$  do
15:           say to neighbour  $n$  the number  $V + 1 - v_n$ 
16:       end for
17:   end if
18: end procedure
```

Implementation I

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 % FILE: soldiers2.m
2 % NOTE: Make sure to modify the parpool to allow
   14 workers.
3
4 buildTroop;
5
6 % Main course: count the soldiers by message
   passing
7 spmd
8     me=labindex;
9     N=length(nb); % Neighbor
   count
10    m=0; % Message
   count
11    v=-ones(N,1); % Message
   values
12    V=0; % Running
   total of messages
```


Implementation II

```

labBarrier;                                     % Not needed
, harmless, for demo purposes

% Receive first N-2 messages
while m < N-1
    [isDataAvail, source]=labProbe;
    if isDataAvail                             % If
        available, get the data
        n=find(source==nb,1);                 % Find which
            neighbor sent the mssage
        assert(~isempty(n));                   % Otherwise
            it is not a neighbor
        fprintf( '%d sees data available from
            %d...\n ',me,source);
        value=labReceive(source);
        fprintf( '%d received value %d from %
            d.\n ',me,value, source);

```

Implementation III

OS	25
Parallelism	26
Trivial	26
Parallelism	27
An Intro to MPI	28
Troop	29
Counting Example	30
Line graph topology	31
Tree graph topology	32
Implementation	33
Wrapping up	34
	35
	36
	37
	38

```

m=m+1;
v(n)=value;
V=V+value;

end
end

%labBarrier;                                % Will
break the code!!!

assert(m==N-1);                               % Check
number of messages

% Send the message to who has not send us a
message
n=find(v==-1,1);                             % Identify who has
not send us a message
dest=nb(n);
value_to_send=V+1;

```

Implementation IV

OS

Parallelism 39

Trivial

Parallelism 40

An Intro to

MPI 41

Troop

Counting 42

Example

Line graph topology 43

Tree graph topology

Implementation

Wrapping up 44

45

46

47

48

49

50

```

fprintf( '%d sending %d to %d...\n', me,
        value_to_send, dest );
labSend( value_to_send, dest );
%labSendReceive( value_to_send, dest );
fprintf( '%d completed sending %d to %d.\n',
        me, value_to_send, dest );
fprintf( '%d waiting for message from %d...\n
        ', me, dest );
[ value, last_source ] = labReceive( dest );
fprintf( '%d received %d from %d.\n', me, value
        , last_source );
assert( last_source == dest );           % Last
        message source
v(n) = value;
V = V + value;
m = m + 1;

```

Implementation V

OS Parallelism	51	%labBarrier ;	% Will
		break the code!!!	
Trivial Parallelism	52		
An Intro to MPI	53	assert(m==N) ;	% Check
		message count	
Troop Counting Example	54		
Line graph topology	55	% Send message to everyone except the one	
Tree graph topology		who was the last to send us	
Implementation		% a message.	
Wrapping up	56	for l=1:N	
	57	if l==n	
	58	continue ;	% Do not
	59	send again	
	60	end	
	61	value_to_send=V+1-v(l) ;	
	62	fprintf('%d sending %d to %d...\n',me,	
		value_to_send,nb(l)) ;	
	63	labSend(value_to_send,nb(l)) ;	

Implementation VI

```
fprintf( '%d completed sending %d to %d.\n', me, value_to_send, nb(l) );
```

```
end
```

```
if me==commander
```

```
    fprintf( 'COMMANDER %d reporting count of %d.\n', me, V+1 );
```

```
end
```

```
fprintf( '%d is done.\n', me );
```

```
end
```

```
% A demonstration that a Composite is a kind of cell array
```

```
% Print the totals of all soldiers.
```

```
for n=1:numSoldiers
```

```
    fprintf( 'Running total of %d is %d.\n', n, V{n} );
```

```
end
```

A modified implementation I

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 % FILE: soldiers3.m
2 % This is like soldiers2, but uses
  labSendReceive instead
3 % of 2 calls labSend/labReceive. It is preferred
  to do it this way,
4 % as there is a smaller chance of programming
  error causing a race
5 % condition (labReceive when there is noone
  sending, or labSend when
6 % there is noone waiting to labReceive).
7
8 buildTroop;
9
10 % Main course: count the soldiers by message
   passing
11 spmd
12     me=labindex;
```

A modified implementation II

Marek Rychlik

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

13

14

15

16

17

18

19

20

21

22

23

24

```

N=length (nb) ;                                % Neighbor
    count
m=0;                                            % Message
    count
v=-ones (N,1) ;                                % Message
    values
V=0;                                            % Running
    total of messages

fprintf( '%d reached barrier.\n' , me) ;
labBarrier;                                    % Not needed
    , harmless , for demo purposes
fprintf( '%d crossed barrier.\n' , me) ;

% Receive first N-2 messages
while m < N-1
    [isDataAvail , source]=labProbe ;

```

A modified implementation III

```

if isDataAvail% If available , get the
data
    n=find ( source==nb ,1 ) ;    % Find which
    neighbor sent the mssage
    assert (~isempty (n)) ;    % Otherwise
    it is not a neighbor
    fprintf ( '%d sees data available from
    %d...\n ',me,source ) ;
    value=labReceive (source) ;
    fprintf ( '%d received value %d from %
    d.\n ',me,value , source ) ;
    m=m+1 ;
    v(n)=value ;
    V=V+value ;
end
end

```


A modified implementation IV

OS Parallelism	37	%labBarrier ;	% Will
		break the code!!!	
Trivial Parallelism	38		
An Intro to MPI	39	assert(m==N-1);	% Check
		number of messages	
Troop Counting Example	40		
	41	% Send the message to who has not send us a	
Line graph topology		message	
Tree graph topology			
Implementation	42	n=find(v==-1,1);	% Identify who has
		not send us a message	
Wrapping up	43	dest=nb(n);	
	44	fprintf('%d noticed not receiving from %d',	
		dest);	
	45	value_to_send=V+1;	
	46	fprintf('%d sending %d to %d...\n',me,	
		value_to_send,dest);	
	47	value = labSendReceive(dest, dest,	
		value_to_send);	

A modified implementation V

OS

Parallelism 48

Trivial

Parallelism 49

An Intro to

MPI 50

Troop

Counting 51

Example 52

Line graph topology 53

Tree graph topology

Implementation

Wrapping up 54

55

56

57

58

59

60

```
fprintf( '%d received %d from %d.\n', me, value
        , dest);
```

```
v(n)=value;
```

```
V=V+value;
```

```
m=m+1;
```

```
%labBarrier;
```

```
break the code!!!
```

```
% Will
```

```
assert(m==N);
```

```
message count
```

```
% Check
```

```
% Send message to everyone except the one
   who was the last to send us
```

```
% a message.
```

```
for l=1:N
```

```
if l==n
```

A modified implementation VI

```

61         continue ;                                % Do not
           send again
62     end
63     value_to_send=V+1-v(l) ;
64     fprintf( '%d sending %d to %d...\n', me,
               value_to_send, nb(l) ) ;
65     labSend( value_to_send, nb(l) ) ;
66     fprintf( '%d completed sending %d to %d.\n', me, value_to_send, nb(l) ) ;
67 end
68 if me==commander
69     fprintf( 'COMMANDER %d reporting count of
               %d.\n', me, V+1 ) ;
70 end
71 fprintf( '%d is done.\n', me ) ;
72 end

```

A modified implementation VII

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
75 % A demonstration that a Composite is a kind of  
    cell array  
76 % Print the totals of all soldiers.  
77 for n=1:numSoldiers  
78     fprintf('Running total of %d is %d.\n', n, V  
            {n});  
79 end
```

Sample output I

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
>> soldiers3
Worker 1:
    Soldier 1 reporting, sir! My neighbors are 2, sir!
Worker 2:
    Soldier 2 reporting, sir! My neighbors are 1, 3, 11, sir!
Worker 3:
    Soldier 3 reporting, sir! My neighbors are 2, 4, 5, sir!
Worker 4:
    Soldier 4 reporting, sir! My neighbors are 3, sir!
Worker 5:
    Soldier 5 reporting, sir! My neighbors are 3, sir!
Worker 6:
    Soldier 6 reporting, sir! My neighbors are 8, sir!
Worker 7:
    Soldier 7 reporting, sir! My neighbors are 8, sir!
Worker 8:
    Soldier 8 reporting, sir! My neighbors are 6, 7, 9, sir!
Worker 9:
    Soldier 9 reporting, sir! My neighbors are 8, 10, 12, sir!
Worker 10:
    Soldier 10 reporting, sir! My neighbors are 9, 11, sir!
Worker 11:
    Soldier 11 reporting, sir! My neighbors are 2, 10, sir!
Worker 12:
    Soldier 12 reporting, sir! My neighbors are 9, 13, 14, sir!
Worker 13:
    Soldier 13 reporting, sir! My neighbors are 12, sir!
Worker 14:
    Soldier 14 reporting, sir! My neighbors are 12, sir!
Worker 1:
    1 reached barrier.
```

Sample output II

OS
Parallelism

Trivial
Parallelism

An Intro to
MPI

Troop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
1 crossed barrier.
2 noticed not receiving from 1 sending 1 to 2...
1 received 13 from 2.
1 is done.
Worker 2:
2 reached barrier.
2 crossed barrier.
2 sees data available from 3...
2 received value 3 from 3.
2 sees data available from 1...
2 received value 1 from 1.
11 noticed not receiving from 2 sending 5 to 11...
2 received 9 from 11.
2 sending 13 to 1...
2 completed sending 13 to 1.
2 sending 11 to 3...
2 completed sending 11 to 3.
2 is done.
Worker 3:
3 reached barrier.
3 crossed barrier.
3 sees data available from 4...
3 received value 1 from 4.
3 sees data available from 5...
3 received value 1 from 5.
2 noticed not receiving from 3 sending 3 to 2...
3 received 11 from 2.
3 sending 13 to 4...
3 completed sending 13 to 4.
3 sending 13 to 5...
3 completed sending 13 to 5.
```

Sample output III

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
3 is done.
Worker 4:
4 reached barrier.
4 crossed barrier.
3 noticed not receiving from 4 sending 1 to 3...
4 received 13 from 3.
4 is done.
Worker 5:
5 reached barrier.
5 crossed barrier.
3 noticed not receiving from 5 sending 1 to 3...
5 received 13 from 3.
5 is done.
Worker 6:
6 reached barrier.
6 crossed barrier.
8 noticed not receiving from 6 sending 1 to 8...
6 received 13 from 8.
6 is done.
Worker 7:
7 reached barrier.
7 crossed barrier.
8 noticed not receiving from 7 sending 1 to 8...
7 received 13 from 8.
7 is done.
Worker 8:
8 reached barrier.
8 crossed barrier.
8 sees data available from 7...
8 received value 1 from 7.
8 sees data available from 6...
```

Sample output IV

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
      8 received value 1 from 6.
      9 noticed not receiving from 8 sending 3 to 9...
      8 received 11 from 9.
      8 sending 13 to 6...
      8 completed sending 13 to 6.
      8 sending 13 to 7...
      8 completed sending 13 to 7.
      8 is done.
Worker 9:
      9 reached barrier.
      9 crossed barrier.
      9 sees data available from 12...
      9 received value 3 from 12.
      9 sees data available from 8...
      9 received value 3 from 8.
      10 noticed not receiving from 9 sending 7 to 10...
      9 received 7 from 10.
      9 sending 11 to 8...
      9 completed sending 11 to 8.
      9 sending 11 to 12...
      9 completed sending 11 to 12.
COMMANDER 9 reporting count of 14.
      9 is done.
Worker 10:
      10 reached barrier.
      10 crossed barrier.
      10 sees data available from 9...
      10 received value 7 from 9.
      11 noticed not receiving from 10 sending 8 to 11...
      10 received 6 from 11.
      10 sending 7 to 9...
```


Sample output V

OS
Parallelism

Trivial
Parallelism

An Intro to
MPI

Troop
Counting
Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
10 completed sending 7 to 9.
10 is done.
Worker 11:
11 reached barrier.
11 crossed barrier.
11 sees data available from 2...
11 received value 5 from 2.
10 noticed not receiving from 11 sending 6 to 10...
11 received 8 from 10.
11 sending 9 to 2...
11 completed sending 9 to 2.
11 is done.
Worker 12:
12 reached barrier.
12 crossed barrier.
12 sees data available from 14...
12 received value 1 from 14.
12 sees data available from 13...
12 received value 1 from 13.
9 noticed not receiving from 12 sending 3 to 9...
12 received 11 from 9.
12 sending 13 to 13...
12 completed sending 13 to 13.
12 sending 13 to 14...
12 completed sending 13 to 14.
12 is done.
Worker 13:
13 reached barrier.
13 crossed barrier.
12 noticed not receiving from 13 sending 1 to 12...
13 received 13 from 12.
```

Sample output VI

OS

Parallelism

Trivial

Parallelism

An Intro to

MPI

Troop

Counting

Example

Line graph topology

Tree graph topology

Implementation

Wrapping up

```
    13 is done.  
Worker 14:  
    14 reached barrier.  
    14 crossed barrier.  
    12 noticed not receiving from 14 sending 1 to 12...  
    14 received 13 from 12.  
    14 is done.  
Running total of 1 is 13.  
Running total of 2 is 13.  
Running total of 3 is 13.  
Running total of 4 is 13.  
Running total of 5 is 13.  
Running total of 6 is 13.  
Running total of 7 is 13.  
Running total of 8 is 13.  
Running total of 9 is 13.  
Running total of 10 is 13.  
Running total of 11 is 13.  
Running total of 12 is 13.  
Running total of 13 is 13.  
Running total of 14 is 13.  
>>
```

What has been left out?

- **OpenMP** (e.g., GOMP=GNU OpenMP); a high level interface to threads (SPMD) and vectorization (SIMD); realized as C/Fortran compiler **pragmas** (annotations)
- **POSIX threads** (“pthreads”); a C library available on most OS which allows direct access to multi-threading and thread synchronization
- Extensive **C++ language** constructs supporting parallelism
- **Building hardware**; hardware is inherently parallel; the most straightforward hardware to build is FPGA (Field-Programmable Gate Arrays); programming languages **Verilog** and **VHDL**
- University of Arizona **HPC facilities**