Distributed Matlab (using MPI)

Introduction.

Distributed Matlab is MPI programming without knowledge of MPI. It is easy to use and looks like Matlab Parallel Computing Toolbox.

With Distributed Matlab you can run on many computers and cores in parallel, using same code.

All source codes are open and free, but with NTNU copy right ((Note! Matlab is not free and you need a license).

Note! It is also possible to add MPI calls into your program (see Matlab MPI).

You can not use Matlab parfor, but still; you can run one matlab instance on each core on the compute node(s). See job scripts below.

Matlab MPI and Distributed Matlab is installed on Fram, Vilje, Idun/Lille and Maur HPC clusters.

You need an account on Fram or Vilje :

Fram: Send an application to Sigma2, see here.

Vilje (local users): Send an application NTNU, see here.

User guides:

Fram, see here.

Vilje, see here.

Support: john.floan@ntnu.no

Available variables.

num_ranks: Number of ranks (or more exact MPI processes) that are selected in the job script (see Job scripts).

my_rank: MPI ID. The rank numbers are from 0 to num_ranks-1. 1 rank is 1 matlab instance.

Master_rank: Rank number 0.

Rank(s) can be one CPU core or more, and/or one compute node or more. (see Job scripts)

Distributed functions.

Overview

- parallel_start and parallel_end: Definition of the parallel region in your program. Note. The parallel_end must be in the end of the program.
- parsteps: Divides the for-loop iterations into chunks. (see example below). (Note! A parallel for-loop must be iterational independent).
- reduction and allreduction: Do a reduction of variables and arrays from all ranks, as eg summation. You can reduce one array or several variables
  Reduction operators: '+', '*', 'MAX', 'MIN' ,'AND' and 'OR' (+:Summation, *:multiplication,MAX:Maximum value, MIN:Minimum value, AND: Logial and, OR: logical or).
- reduce_array: Reduce multidim array.
- spread and collect:
  Spread several variables or one array from master to all ranks as: [a,b]=spread(a,b); array=spread(array);

Collect several variables or one array from all ranks to master as: [a,b]=collect(a,b): or [arrays{1:num_ranks}]=collect (array).
The array must be one dim.

- distrib and join.
  Distribute an array (1 to 3 dimensional) to all ranks, chunked in num_ranks coloumns.
  Join all chunked arrays (1 to 3 dimensional) to one array.
- mdisplay: Same as display but only master rank display the text.
- get_my_rank and get_num_ranks

See examples below.

Parallel_start and end

Example: Hello world.

```
% For Fram only (R2019a): Add link to MPI libraries
addpath('/cluster/software/MATLAB/2019a/NMPI/version13_intel2017b/');

% For Saga only (R2019a): Add link to MPI libraries
addpath('/cluster/software/MATLAB/2019a/NMPI/version13_intel2019a/');

% For Vilje only (R2017a):Add link to MPI Libraries
addpath('/sw/sdev/Modules/matlab/R2017a/NMPI');

% For Idun cluster only (2019a):
addpath('/lustre1/apps/software/Core/MATLAB/2019a/NMPI/version13_intel17b');

parallel_start;

display(['Hello world from rank number ',num2str(my_rank),' of total ',num2str(num_ranks)]);

parallel_end;
```

!!! Note that parallel_end shall always be in the end of the program.

Output (if 2 computers (2 ranks)):

Hello world from rank number 0 of total 2

Hello world from rank number 1 of total 2

Reduction and parsteps

Example: Parallel for-loop on 8 compute nodes.

Sequential code

```
n=64;
sum1=0;
for ii=1:n
    sum1=sum1+sin(ii);
end
display(['sum1 ',num2str(sum1)]); % Only the Master display text
```

Parallel code

```
% Remember addpath (see parallel_start above)
...
parallel_start;
n=64;
sum1=0;
for ii=parsteps(1:n)
    sum1=sum1+sin(ii);
end
sum1=reduction('+',sum1);
mdisplay(['sum1 ',num2str(sum1)]); % Only the Master display text
parallel_end;
```

What parsteps do; is to chunk the iterations (ii) to eg. 8 compute nodes, and all for-loops starts simultaneously:

rank 0	rank 1	rank 2	.....	rank 7
for ii = 1: 8	for ii = 9 :16	for ii = 17 : 24		for ii =57 : 64

NOTE! For all parallel for-loops; each iteration must be independet of the itereration before.

NOTE2! Number of iterations can not be less then number of ranks.

More advanced use of parallel for loop (double for-loop)

```
...

parallel_start;
n=64;
sum1=1;
sum2=1;
% Note! If sum1 and/or sum2 are zero, then you can skip sum1 and/or sum2 in parsteps; as parsteps(1:n)
% Only the outer for loop (ii) is parallel. Inner for loop (jj) is local.
for ii=parsteps(1:n,sum1,sum2)
   for jj=1:n
        sum1=sum1+sin(ii*jj);
        sum2=sum2+cos(ii*jj);
   end
end
[sum1,sum2]=reduction('+',sum1,sum2);
mdisplay(['sum1 ',num2str(sum1),' sum2 ',num2str(sum2)]); % Only the Master display text
parallel_end;
```

(Note! The inner loop is not in parallel. All ranks count jj from 1 to n).

Example: Reduction array with Max, and find the maximum value of the local arrays from all ranks.

```
data=rand(n,1);
m=reduction ( 'MAX' , data); % m is a variable with the max value
```

Example: Reduction of an array with '+' and '*': Summation/multiplication of all arrays.

```
data=rand(n,1);
data=reduction ( '+' , data);
```

Exampel : Reduction of array with '+': Summation of all arrays (element for elements) to master rank (0).

Input

rank 0: data = [1 2 3];

rank 1: data = [2 3 4];

data = reduction('+',data):

After reduction:

rank 0: data = 3 5 7 (Master rank)

rank 1: data = 0 0 0 (Other ranks)

Exampel : Reduction of array with '*': Multiplication of all arrays (element for elements) to master rank.

Input

rank 0: data = [2 3 4];

rank 1: data = [3 4 5];

data = reduction('*',data):

After reduction:

rank 0: data = 6 12 20 (Master rank)

rank 1: data = 0   0   0 (Other ranks)

Example: Update a array for each rank (2 ranks in this case)

```
data=zeros(6,1);
for parsteps(ii=1:6)
   data (ii)=ii
end
data=reduction('+',data);
```

Input

rank 0: data = [1 2 3 0 0 0]

rank 1: data = [0 0 0 4 5 6]

After reduction

rank 0 : [1 2 3 4 5 6] (Master rank)

rank 1: [0 0 0 0 0 0] (Other ranks

Example: Reduction of multi-dim array.

```
M=ones(n,m);
for j=parsteps(1:n)
 for i=1:m
    M(i,j)=M(i,j)*i*j;
  end
end
%Reshape M to 1 dim array before Reduction
d=reshape( M , n*m , 1 ); % Reshape to 1 dim array:
d2=reduction ( '+' , d);  % Reduce all elements
M=reshape(d2,[n,m]);      % Reshape back to 2 dim array
```

Allreduction

Same as reduction but all ranks get same reduction value(s)

Reduce multi-dim array

Reduce multi-dimensional array (max 3 dim) beetween all ranks, and all ranks get same values

Operators: '+' (default), '*' , 'MAX', 'MIN', 'AND' and 'OR'

Syntax:

arrayout = reduce_array (arrayin); %Operator is default '+'

arrayout = reduce_array (operator , arrayin);

Example code 1 (2 ranks): 2 dim array

```
if my_rank==0
   array = [1,2;0,0];
else
   array = [0,0;3,4];
end
array = reduce_array(array);  % Default operator is '+'
```

Input array (2x2)

Rank 0 Rank 1

```
1  2    0  0
```

```
0  0    3  4
```

Output array from all ranks (after reduce_array)

```
1  2
```

```
3  4
```

Example code 2 (2 ranks): 2 dim array

```
if my_rank==0
   array = [1,2;2,2];
else
   array = [2,2;3,4];
end

array = reduce_array('*',array);
```

Input to array:

Rank 0 Rank 1

```
1  2    2  2
```

```
2  2    3  4
```

Output from all ranks (after reduce_array)

```
2  4
```

```
6  8
```

Spread and collect.

Example code for spread.

Array:

```
n=3;
array=ones(n,1)*my_rank;
array=spread(array);
```

All ranks receive the same array, created by the master rank. All receive: (0,0,0) (Note, my_rank for master is 0)

Variables:

```
a=my_rank;b=my_rank+1;
[a,b]=spread(a,b);
```

All ranks receive variables a and b, created by the master rank: a=0, b=1. (Note, my_rank for master is 0)

Example code for collect:

Array:

```
n=3;
array=ones(n,1)*my_rank;
[arrays{1:num_ranks}]=collect(array);
```

Input to collect (2 ranks): Rank 0: (0,0,0), Rank 1: (1,1,1).

Output of collect (master): arrays is a cell array with 2 cells (1 cell each rank): arrays{1}=(0,0,0), arrays{2}=(1,1,1)

Variables:

```
a=my_rank;b=my_rank+10;
[a,b]=collect(a,b);
```

Input to collect for 2 ranks: Rank 0: a = 0 , b = 10, Rank 1: a = 1 , b = 11.

Output of collect (master): a = (0 , 1), b = (10 , 11)

Distribute and join (distrib and join)

Distrib divide a array in "number of ranks" chunks. Join gather the local array to master rank.

Exampel code; distribute:

```
n=2;
A=zeros(n,n);
A(1,1)=1;A(1,2)=2;A(2,1)=3;A(2,2)=4;

A=distrib(A);
```

Input from master (rank 0) (A: 2x2 elements):

```
 A = 1    2

    3    4
```

Output (for eg. 2 ranks): (A: 2x1 elements)

```
   Rank 0       Rank 1

   A =  1       A  =  2

        3            4
```

Example code: join

```
n=2;
A=zeros(n,1);
if my_rank==0
   A(1,1)=1;A(2,1)=3;
else
   A(1,1)=2;A(2,1)=4;
end
A=join(A);  % The array size and dimension is set by function distrib.
            % If join is used without distrib; the output array A is one dim.
            % Note that if you need 2 dim, set the dim to 2 as A=join(A,2);
```

Input join:

```
 Rank 0       Rank 1

 A  =  1      A  =  2
```

```
     3          4
```

Output join (master)

```
A  =  1    2

      3    4
```

Job scripts.

For more information about job scripts and job execution; see here.

1. To run one Matlab job on each compute node.
   Example: One Matlab job on each of 4 nodes. That is 4 ranks (Note! You still have 32 cores (Fram) available (16 cores on Vilje) each node. Use operators and function that support multicore running)

FRAM

```
#!/bin/bash
#SBATCH --account=myaccount
#SBATCH --job-name=jobname
#SBATCH --time=0:30:0
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
module purge
module load intel/2017b
module load MATLAB/2019a
module list

srun --mpi=pmi2 matlab -nodisplay -nodesktop -nojvm -r "myprogram"
```

SAGA

```
#!/bin/bash
#SBATCH --account=myaccount
#SBATCH --job-name=jobname
#SBATCH --time=0:30:0
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
module purge
module load intel/2019a
module load MATLAB/2019a
module list

srun --mpi=pmi2 matlab -nodisplay -nodesktop -nojvm -r "myprogram"
```

VILJE:

```
#!/bin/bash
#  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#PBS -N job
#PBS -l walltime=00:30:00
#PBS -l select=4:ncpus=32:mpiprocs=1:ompthreads=16
#PBS -q workq
#PBS -A myaccount

cd $PBS_O_WORKDIR
module load gcc/4.9.1
module load mpt
module load matlab/R2017a

mpiexec_mpt -n 4 omplace -nt 16 matlab -nodisplay -nojvm -nodesktop -nosplash -r "myprogram"
```

Typical use of this configuration is use of eg Matlab linear algebra functions and operators. Matlab support multicore running for several functions and operators..

2. To run Matlab jobs on each core on the compute node.

FRAM:

Example: 4 nodes and 32 MPI processes each node, that is 128 ranks (or cpu cores).

```
#!/bin/bash
#SBATCH --account=myaccount
#SBATCH --job-name=jobname
#SBATCH --time=0:30:0
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=32
module purge
module load intel/2017b
module load MATLAB/2019a
module list

srun --mpi=pmi2 matlab -nodisplay -nodesktop -nojvm -r "myprogram"
```

VILJE:

Example: 2 nodes and 16 MPI processes each node, that is 32 ranks (or cpu cores).

```
#!/bin/bash
#  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#PBS -N job
#PBS -l walltime=00:30:00
#PBS -l select=2:ncpus=32:mpiprocs=16
#PBS -q workq
#PBS -A myaccount

cd $PBS_O_WORKDIR
module load gcc/4.9.1
module load mpt
module load matlab/R2017a
mpiexec_mpt -n 32 matlab -nodisplay -nojvm -nodesktop -nosplash -r "myprogram"
```

For the parallel for-loop example above, the iterations of ii are as:

Rank 0	Rank 1	...	Rank 32
for ii=1:2	for ii=3:4		for ii=31:32

Typical use of this configuration is, similar as parfor; for-loops with non multicore supported function and operators; as eg calculation with variables (a = b + c);

Object Oriented Matlab.

Example code: AverageOO

```
parallel_start;

n=100;
a=averageOO(n);
avg=a.Calculate();
mdisplay(['avg ',num2str(avg)]); % Master display text

parallel_end;
```

AverageOO Class:

```
classdef averageOO
    properties
        n         % Size of Array
        data      % Array
    end
    methods
        % Constructor
        function obj = averageOO ( n )
            obj.n = n;
            obj.data = rand(1,obj.n);
        end
        function dataAvg = Calculate( obj )
            dataAvg=0;
            for ii=parsteps(1:obj.n) % Parallel for-loop
                dataAvg=dataAvg+obj.data(ii);
            end
            dataAvg=reduction('+',dataAvg);
            dataAvg=dataAvg/obj.n;
        end
     end % methods
end %classdef
```

Save to file.

Normally you let the Master rank save to file as: (or else use different file name for each rank):

```
parallel_start;
...
if my_rank==Master_rank
    save('mydata.mat','mydata');
end
...
parallel_end;
```

!!! Note that parallel_end shall always be in the end of the program.

Job script Idun

```
#!/bin/bash
#SBATCH -J job               # sensible name for the job
#SBATCH -N 2                 # Allocate 2 nodes for the job
#SBATCH --ntasks-per-node=20 # 20 ranks/tasks per node (see example: job script)
#SBATCH -t 00:10:00          # Upper time limit for the job (HH:MM:SS)
#SBATCH -p WORKQ

module load intel/2017b
module load MATLAB/2019a
mpirun matlab -nodisplay -nodesktop -nosplash -nojvm -r "test"
```