Parallel MATLAB at FSU: Task Computing

John Burkardt
Department of Scientific Computing
Florida State University

....

1:30 - 2:30
Thursday, 07 April 2011
499 Dirac Science Library

....

http://people.sc.fsu.edu/∼jburkardt/presentations/. . .
matlab_tasks_2011_fsu.pdf

- **Task Computing**
- QUAD Example
- KNAPSACK Example
- CELL DETECTION Example
- RANDOM WALK Example
- Conclusion

Task Computing: not PARFOR or SPMD

In last week's lecture, I talked about MATLAB's **parfor** and **spmd** commands.

With **parfor**, a single program and a single set of shared data were involved. When the"client" reached a parallel loop, extra "workers" would assist.

With **spmd**, we saw a single program, but one which was divided between commands to the client and command to the workers. Moreover, the client and workers had separate memory spaces. Data could be moved only by explicit commands.

Despite their differences, programming with **parfor** and **spmd** have something in common; the client and the workers are simultaneously executing a program.

Task Computing: Description

Today, we will consider a third technique, **task computing**, in which a big *job* is divided into very independent *tasks*;

Each task runs on the smallest addressable type of processor: a *single core* on a desktop or a *single node* on a cluster.

Tasks run in any order, at any time, on any available processor.

We'll assume each task executes the same MATLAB function.

Each task has its *own memory*.

Each task is given a set of input; it does no further communication until its computation is complete, when it returns its results.

When all the tasks are completed, the collection of results can be examined, analyzed or plotted.

Task Computing: Description

You could set up 100 tasks, for instance, as a single job.

For each task, MATLAB locates an available processor, ensures that the task receives its input, and recovers the output.

Once all the tasks are completed, the output can be examined.

Task computing allows you to organize a computation that has many independent parts. If many processors are available, many tasks will run at the same time, but you don't worry about the precise schedule.

Task computing is a handy way of filling up spare computer time with "bite sized" pieces of a computation whose final result can be determined once all the pieces have been completed.

Tasks can run on your desktop or on a cluster.

Task Computing: Examples

Task computing examples:

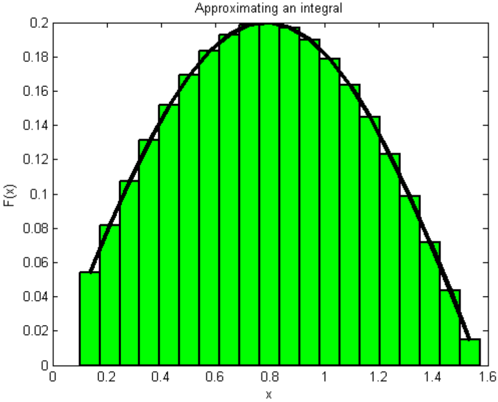
- **search**: divide the search space among tasks;
- **Monte Carlo**: give each task a unique random number seed;
- **summation**: any task involving multiple summation, multiplication, or other operations (quadrature)
- **image processing**: operations that can be carried out over parts of the image, or frames of an animation; ray tracing;

Parallel MATLAB: Task Computing

- Task Computing
- **QUAD Example**
- KNAPSACK Example
- CELL DETECTION Example
- RANDOM WALK Example
- Conclusion

QUAD: Approximate integration

Here we use evenly spaced sample points and equal weights.
Other schemes vary the spacing or weights, or randomize abscissas.



QUAD: Approximate integration

We recall our “favorite” problem, the approximation of an integral by a weighted sum of function values:

$$I = \int_a^b f(x)\,dx \quad \approx \quad Q = \sum_{i=1}^{N} w_i\, f(x_i)$$

We could easily regard this computation as, say, 4 tasks:

$$Q = Q_1 + Q_2 + Q_3 + Q_4$$

where each Q_i could be computed independently. The only communication required would come at the end, when the Q_i’s must be combined to form Q.

QUAD: Tasks and Jobs

Our basic **task** estimates the integral over a subinterval $[a_i, b_i]$ using n_i points. We can write a MATLAB function to do this:

```
qi = quad_task ( ni, ai, bi )
```

Then our **job** is made up of four tasks (assume $[a, b] = [0,1]!$):

- task 1 integrates from 0/4 to 1/4
- task 2 integrates from 1/4 to 2/4
- task 3 integrates from 2/4 to 3/4
- task 4 integrates from 3/4 to 4/4

Each task can be expressed by specifying the appropriate input arguments to **quad_task**.

QUAD: QUAD_TASK

```
function qi = quad_task ( ni, ai, bi )

  x = linspace ( ni, ai, bi );

  qi = h * ( 0.5 * f ( x(1) ) ...
           + sum ( f ( x(2:ni-1) ) ) ...
           + 0.5 * f ( x(ni) ) );

  return
end
```

QUAD: Local Execution of a Job

On a desktop, we "create" a job, define its tasks, and submit it:

```
job = createJob ( 'configuration', 'local' );
n = 100001;
ni = 25001;
for task = 1 : 4
  ai = ( task - 1 ) / 4;
  bi =   task       / 4;
  task_id = createTask ( job, ...
    @quad_task, 1, { ni, ai, bi } )
end
submit ( job );
wait ( job );   <-- Wait until all tasks have completed.
```

(The third argument to **createTask**, ("1"), reports the number of outputs produced by **quad_task**).

QUAD: Collecting Results

Our final integral estimate **Q** is the sum of the individual results.

If **qi** is the name of the output from the *quad_task* function, the load command can return a cell array containing the value of **qi** returned by each task;

```
qi = load ( job, 'qi' );

qi = cell2mat ( qi );

q = sum ( qi );
```

QUAD: FSU HPC Cluster

To run on the FSU HPC cluster, we run as a "simple" job, and we have to create the cell array args of input arguments:

```
n = 100001;
task_num = 4;
ni = 1 + floor ( ( n - 1 ) / task_num );

args = {};
for task = 1 : task_num
  ai = ( task - 1 ) / task_num;
  bi =   task       / task_num;
  args{task} = { ni, ai, bi };
end

qi = fsuClusterMatlab ( [], [], 's', 'w', ...
  task_num, @quad_task, args )
```

QUAD: FSU HPC Cluster

We have asked MATLAB to execute the four commands:

```
qi{1} = quad_task ( 0.00, 0.25, 25001 );
qi{2} = quad_task ( 0.25, 0.50, 25001 );
qi{3} = quad_task ( 0.50, 0.75, 25001 );
qi{4} = quad_task ( 0.75, 1.00, 25001 );
```

in any order, on any available cores, while we wait.

Although these commands execute independently, when they are done, we have access to the results.

In particular, the quadrature value is:

```
qi = cell2mat ( qi );
q = sum ( qi );
```

QUAD: Our simplest example

QUAD demonstrates the features of task programming.

The calculation is thought of as a job.

The job is thought of as multiple tasks, with each task executed by the same task function, using different inputs.

The job collects the output from each task function.

The user can wait for the job to complete (the 'w' option), or log out and check back in later (the 'n' option).

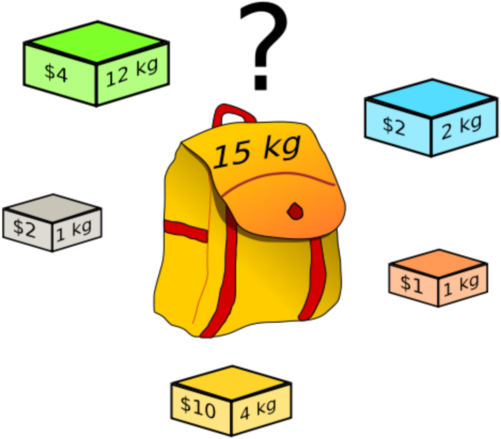
When the job is complete, the user can resume an interactive MATLAB session to analyze, manipulate or plot the results.

Parallel MATLAB: Task Computing

- Task Computing
- QUAD Example
- **KNAPSACK Example**
- CELL DETECTION Example
- RANDOM WALK Example
- Conclusion

KNAPSACK: Problem Definition

In the classical problem, the objects have both weights and values. For our problem, we'll just worry about the weights!



KNAPSACK: Problem Definition

Suppose we have a knapsack with a limited capacity, and a number of objects of varying weights. We want to find a subset of the objects which exactly meets the capacity of the knapsack.

(This is sometimes called the *greedy burglar's problem!*)

Symbolically, we are given a *target value* **t**, and a set **W** of **n** weights. We want a subset $\mathbf{S} \subset \mathbf{W}$ so that:

$$t = \sum_{s \in S} s$$

We don't know if a given problem has 0, 1, or many solutions.

KNAPSACK: Encoding

A solution of the problem is a subset $\mathbf{S} \subset \mathbf{W}$. Each **n**-digit binary string from 0 to $2^n - 1$ is a code for a possible solution.

For weights $\mathbf{W}$={15,11,10,8,3}, target $\mathbf{t}$=24, we have:

Code	Binary Code	Indices	S	$\sum s$
0	00000	{}	{}	0
1	00001	{1}	{3}	3
2	00010	{2}	{8}	8
3	00011	{2,1}	{8,3}	11
4	00100	{3}	{10}	10
5	00101	{3,1}	{10,3}	13
6	00110	{3,2}	{10,8}	18
...	...	...	...	...
31	11111	{5,4,3,2,1}	{15,11,10,8,3}	47

KNAPSACK: Algorithm

A simple search chooses a value of **code** in the range 0 to $2^n - 1$, decodes the subset **S**, adds the weights, and compares to **t**.

On the 23rd step of the search, we have a code of 22 = binary 10110 = subset $\{5,3,2\}$, so a weight of 15+10+8=33, too high.

The process of checking one code is completely independent of checking any other.

One program could check all codes, or we could subdivide the range, and check the subranges in any order and at any time.

KNAPSACK: Program to Search Entire Range

```
function [ i_choose, w_choose ] = knapsack ( w, t )

  i_choose = [];
  w_choose = [];
  n = length ( w );

  for code = 0 : 2^n - 1

    choose = find ( bitget ( code, 1:n ) );

    if ( sum ( w(choose) ) == target )
      i_choose = choose;
      w_choose = w(choose);
      return
    end
  end
```

KNAPSACK: A Couple Mysterious MATLAB Functions

The MATLAB function **bitget** returns a vector of 0's and 1's for positions 1 to **n** in the counter **code**.

Each such binary string describes a unique subset.

The function **find** indexes the 1's in the binary string.

This string of indices, called **choose**, selects the subset of **w** that we must compare to **t**.

If the subset has the right weight, we found a solution, and return.

KNAPSACK: The Calculation is “Divisible”

It’s easy to see that we could divide this problem up into smaller problems that are worked on independently.

For this problem, it’s clear that the key is to take the original range of **code**, from 0 to $2^n - 1$, and break it into subranges.

A single function can work on the problem over the restricted subrange. In fact, we only have to slightly modify our original code to make this new version.

KNAPSACK: Program to Search Selected Range

```
function [ i_choose, w_choose ] = knapsack ( w, t, rnge )

  i_choose = [];
  w_choose = [];
  n = length ( w );

  for code = rnge(1) : rnge(2)

    choose = find ( bitget ( code, 1:n ) );

    if ( sum ( w(choose) ) == target )
      i_choose = choose;
      w_choose = w(choose);
      return
    end
  end
```

KNAPSACK: Set up for Local Execution

Once we have mentally divided our calculation into independent subcalculations, we need to be able to express this logical fact to MATLAB.

Of course, just as for a sequential calculation we need to define the general problem parameters;

However, now, we also need to specify the number of tasks, identify the function that will execute the tasks and create a cell array containing a separate copy of the input to each task.

We “feed” this information to the **fsuClusterMatlab** command.

KNAPSACK: Execution on FSU HPC Cluster

```
w = [ 518533, 1037066, (...more...), 1259008 ];
t = 2463098;

args = {};
i2 = -1;
for task = 1 : 4
  i1 = i2 + 1;
  i2 = floor ( ( 2^n - 1 ) * task / 4 );
  args{task} = { w, t, [ i1, i2 ] };
end

results = fsuClusterMatlab ( [], [], 's', 'w', ...
  4, @knapsack_task, args )
```

KNAPSACK: Examine the results

The output arguments from each task are returned as a cell array.

Task 3's second output (the weights) is in **results**{**3,2**}.

```
for task = 1 : 4
  if ( isempty ( results{task,1} ) )
    fprintf ( 1, 'Task %d found no solutions.\n', task );
  else
    fprintf ( 1, 'Task %d found a solution:\n', task );
    disp ( 'Indices Chosen:' );
    disp ( results{task,1} );
    disp ( 'Weights Chosen:' );
    disp ( results{task,2} );
  end
end
```

KNAPSACK: Sample Run

```
>> knapsack_fsu

results =                 []              []
                    [1x3 double]    [1x3 double]
                          []              []
                          []              []

  Task 1 did not find a solution.
  Task 2 found the following solution:
weight indices
     2     5    20
weight_values
     1037066      796528      629504
  Task 3 did not find a solution.
  Task 4 did not find a solution.
```

Parallel MATLAB: Task Computing

- Task Computing
- QUAD Example
- KNAPSACK Example
- **CELL DETECTION Example**
- RANDOM WALK Example
- Conclusion

CELLS: Problem Definition

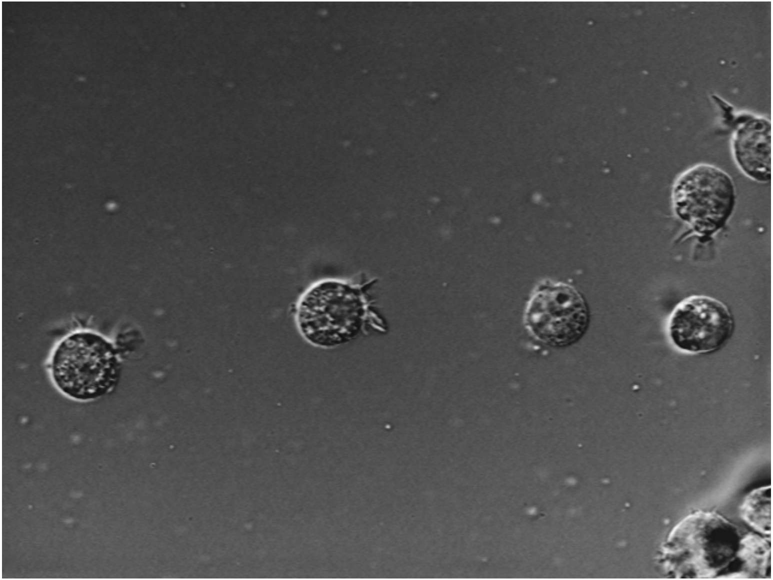
Medical researchers can film small groups of biological cells.

In some cases, an enormous number of such records are created. Rather than having a lab worker view each frame of film, it is possible to automatically process the images and, for most part, detect the cells, determine the position of any given cell over a sequence of images, and monitor the area, shape and average separation of the cells.

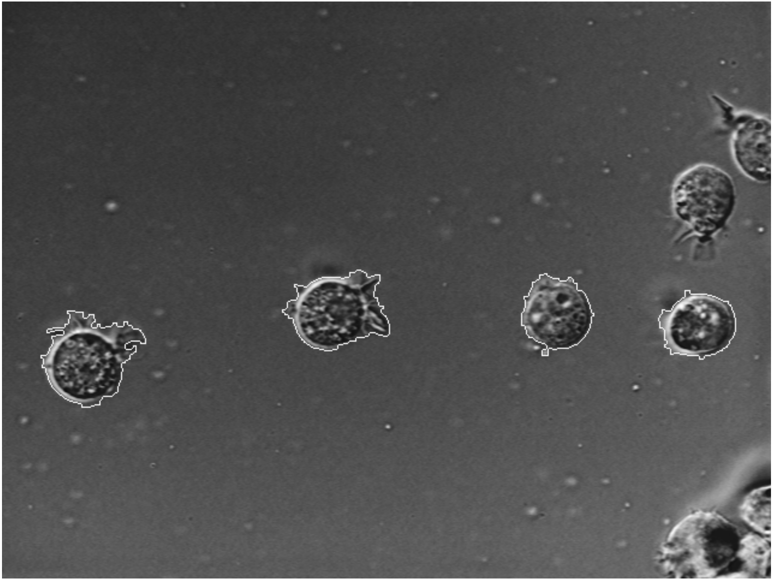
We will look at a simple application in which it is desired to identify cells by surrounding each one with a white boundary.

If we have many image files, each can be processed independently.

CELLS: A Typical Image



CELLS: A Typical Image with Cells Identified



CELLS: Problem Definition

In the KNAPSACK problem, the input and output for each task was a short list of numbers.

For the CELLS problem, the input is really a graphics file, and the output is a transformed graphics file.

This means that each task, when it runs, has to be able to determine which unique file it should open; it needs to be able to find that file; and it needs to know how to name the output file and where to place it.

So while the KNAPSACK tasks used the input and output arguments of the MATLAB function for their communication, the CELLS example will almost entirely be dealing with external files.

CELLS: File Names

Our 99 input files are indexed in a natural way, starting with **AT3_01.tif** and running up to **AT3_99.tif**.

The output files will follow a similar convention, but their names will start with **BT3**.

The input filename for a given task can be computed:

```
  filename_input = sprintf ( 'AT3_%02d.tif', task );
```

Using the format %02d means that small integers will be left-padded with zeros.

CELLS: The Task Function

For this problem, the task function will have a simple interface:

```
function cell_task ( task )
```

It only needs to know the index of the file it should open, and it doesn't return any output - at least, not via output arguments. So the body of this function will be:

```
generate input filename based on task number
open input file
process data
generate output filename based on task number
write output file
```

and we won't worry about the details of processing the data!

CELLS: Initializing the Job

Now we need to set up the tasks.

Notice that there are no output results...

The “output” of **cell_task** will be the modified version of the image file.

```
task_num = 99;    <-- there are 99 images to process;

args = {};
for task = 1 : task_num
  args{task} = { task };  <-- task i works on image i.
end

fsuClusterMatlab ( [], [], ’s’, ’w’, ...
  task_num, @cell_task, args );
```

CELLS: Defining the Tasks

To keep things simple, we assume that the file **cell_task.m** and all the input image files are in the current directory.

Then, once we issue the **fsuClusterMatlab** command, each task, when it executes, will start in this directory, be able to “see” the input image, and will leave its modified version here as well.

CELLS: Sample Output for 10 Images

```
>> cell_fsu

CELL_FSU:
  Use MATLAB's task computing on the FSU HPC cluster.
  Here, we want to apply an image processing operation
  (identify edges) to each of 10 biological images.

  Here is a current directory listing:

AT3_01.tif AT3_04.tif AT3_07.tif AT3_10.tif
AT3_02.tif AT3_05.tif AT3_08.tif cell_fsu.m
AT3_03.tif AT3_06.tif AT3_09.tif cell_task.m

  Call fsuClusterMatlab
```

CELLS: Sample Output for 10 Images

After fsuClusterMatlab executes, we see:

```
  Here is a current directory listing:

AT3_01.tif  AT3_09.tif  BT3_07.tif  Job1.in.mat
AT3_02.tif  AT3_10.tif  BT3_08.tif  Job1.jobout.mat
AT3_03.tif  BT3_01.tif  BT3_09.tif  Job1.out.mat
AT3_04.tif  BT3_02.tif  BT3_10.tif  Job1.state.mat
AT3_05.tif  BT3_03.tif  cell_fsu.m  matlab_metadata.mat
AT3_06.tif  BT3_04.tif  cell_task.m
AT3_07.tif  BT3_05.tif  Job1
AT3_08.tif  BT3_06.tif  Job1.common.mat
```

Job files are MATLAB log and data information.

Parallel MATLAB: Task Computing

- Task Computing
- QUAD Example
- KNAPSACK Example
- CELL DETECTION Example
- **RANDOM WALK Example**
- Conclusion

WALKS: Problem Definition

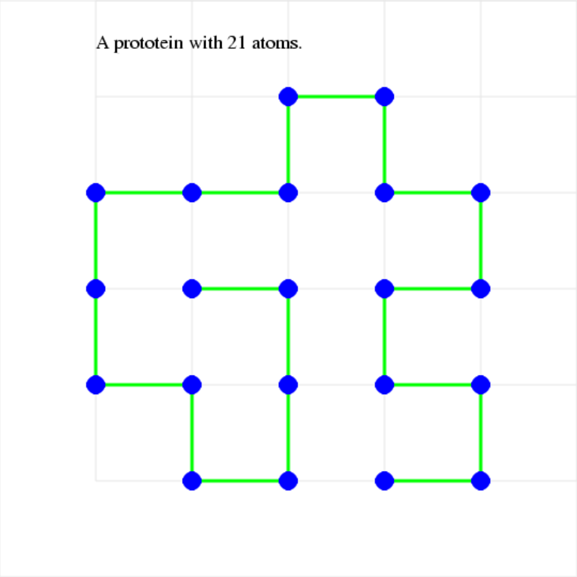
The classic example of a random walk places a person at the origin. The person then flips a coin, and takes one step to the left or right, repeating this process as long as desired.

Surprisingly, random walks are models of many physical processes, and their simulation and analysis can give insight to situations where there are no exact methods available.

A variation of this problem is the **self avoiding walk in 2D**, in which the person is allowed to move over a 2D lattice, but can never visit the same place twice.

Because the path never crosses, this walk can be thought of as a simple model of a protein folding. Self avoiding walks of a fixed length could represent possible shapes of the protein.

WALKS: A Self Avoiding Walk / Abstract Protein



WALKS: Sampling a Huge, Unruly Space

Our idea is to generate lots of different self avoiding walks. Since the set is extremely large, we rely on the random number generator to make our choices. A different initial seed should almost always give a different walk. The actual number of possible walks can easily exceed the number of possible seeds!

Our samples might give us

- an estimate for the typical distance between the start and end;
- the typical distance of the starting point to the "boundary";
- number of empty lattice points in the convex hull of the walk;
- the likelihood a walk will terminate early;

WALKS: The Task Code

```
function [ step_num, dist ] = walk_task ( step_max, seed )
```

The task uses **seed** to initialize **rand()** by:

```
  rand ( 'twister', seed );
```

Then it tries taking **step_max** self-avoiding steps from the origin.

The walk terminates early if all four neighbors have already been visited. Otherwise, we move to a random unvisited neighbor.

The function returns the actual number of steps taken, and the final distance from the origin.

WALKS: Setting up the Job

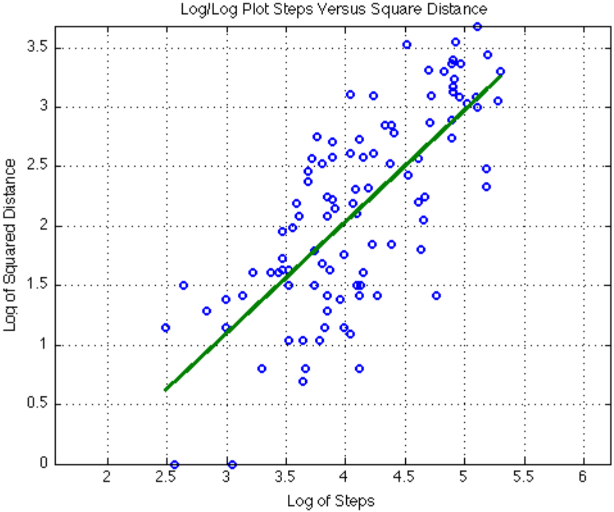
The job code must decide how many walks are to be generated, and how many steps they are to take. It chooses different random number seeds for each task.

```
task_num = 100;
step_max = 200;

args = {};
for task = 1 : task_num
  seed = 123456789 + task;
  args{task} = { step_max, seed };
end

results = fsuClusterMatlab ( [], [], 's', 'w', ...
  task_num, @walk_task, args );

results = cell2mat ( results );
```

WALKS: Plotting the Results

```
results = cell2mat ( results );  <-- make numeric

step_num = results(:,1);
dist2 = results(:,2).^2;

x = log ( step_num );      <-- X = log steps
y = log ( dist2 );         <-- Y = log dist^2

c = polyfit ( x, y, 1 ); <-- Seek linear fit
log_fit = c(1) * x + c(2);

plot ( x, y,        'bo', ...
       x, log_fit, 'r-' )
```

WALKS: Dist Squared = c * Steps



Parallel MATLAB: Task Computing

- Task Computing
- QUAD Example
- KNAPSACK Example
- CELL DETECTION Example
- RANDOM WALK Example
- **Conclusion**

CONCLUSION: Summary of Examples

Our examples suggest the computations suitable for task programming.

QUAD is so simple we have used it with parfor, spmd and tasks.

In **KNAPSACK**, we had to figure out an encoding, and we gave each task a subrange of the full problem. We had thousands of codes to check, but we assigned a large subrange of codes to each task (rather than generating thousands of tiny tasks!)

In **CELL**, each task worked on a file, and there was no input or output, except that each task knew its index.

In **RANDOM WALK**, we had to ensure that each task received a unique random seed, and we had to gather the data up at the end and plot it.

CONCLUSION: Summary of Task Computing

Task computing is significantly different from the usual kind of parallel computing, where the computations happen concurrently. Here, each task is assigned a single processor. Tasks can run sequentially on the same processor, simultaneously on different processors, or in many other combinations.

Tasks do not communicate, except that they receive an initial input from the job, and send their output results back to it.

In each of our examples, the same MATLAB function executed each task, but a job is free to collect an arbitrary set of tasks.

Conclusion: Desktop Experiments

If you are interested in parallel MATLAB, the first thing to do is get access to the Parallel Computing Toolbox on your multicore desktop machine, so that you can do experimentation and practice runs.

You can begin with some of the sample programs we have discussed today.

You should then see whether the **job** and **task** approach would help you in your own programming needs.

Conclusion: FSU HPC Cluster

If you are interested in serious parallel MATLAB computing, you should consider requesting an account on the FSU HPC cluster, which offers MATLAB on up to 16 cores.

To get an account, go to www.hpc.fsu.edu and look at the information under **Apply for an Account**.

Accounts on the general access cluster are available to any FSU faculty member, or to persons they sponsor.

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 5.0
  www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/...
  distcomp.pdf
- http://people.sc.fsu.edu/∼jburkardt/presentations/. . .
  fsu_2011_matlab_tasks.pdf *these notes*;
- Gaurav Sharma, Jos Martin,
  MATLAB: A Language for Parallel Computing,
  International Journal of Parallel Programming,
  Volume 37, Number 1, pages 3-36, February 2009.
- http://people.sc.fsu.edu/∼jburkardt/m_src/m_src.html
  - **quad_tasks**
  - **knapsack_tasks**
  - **cell_detection_tasks**
  - **random_walk_2d_avoid_tasks**