Parallel MATLAB at FSU: Task Computing

John Burkardt
Virginia Tech

....

https://people.sc.fsu.edu/∼jburkardt/presentations/...
matlab_tasks_2010_fsu.pdf

16 April 2010

Parallel MATLAB: Task Computing

Task Computing: not PARFOR or SPMD

In Monday's talk about **parfor**, a single program and a single set of data were involved. When the"client" reached a parallel loop, extra "workers" would assist.

On Tuesday, we saw **spmd** programs, with separate commands to the client and to the workers (in one program) and separate memory spaces. Moving data between the client and workers required explicit commands.

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

Task Computing: Description

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

For simplicity, assume each task will be carried out by the same MATLAB function.

Each task runs on a *single processor* (although there's no need to rule out parallelism) and has its *own memory*.

Tasks do not communicate while running; they start with input, they return *results* upon completion.

When all the tasks are completed, it is possible to gather, analyze and plot the combined results.

Task Computing: Description

You could set up 100 tasks, and run on 4 workers, for instance.

MATLAB ensures each task receives its input, gets executed somewhere, sometime, and stores the output for later review.

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.

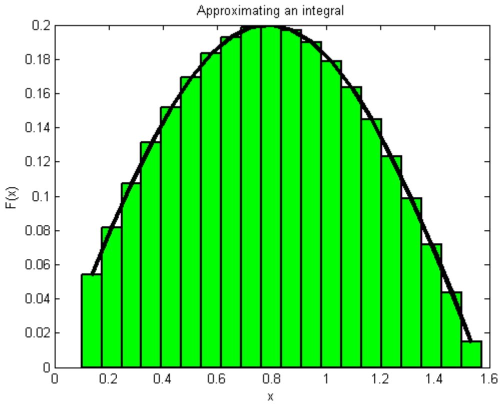
Task Computing: Examples

Task computing examples:

- **search**: if the search space can be divided up into subregions identified numerically;
- **Monte Carlo**: if each task can be sure of generating a separate stream of random numbers;
- **summation**: any task involving multiple summation, multiplication, or other operations (quadrature)
- **image processing**: operations that can be carried out over parts of the image; ray tracing;

Parallel MATLAB: Task Computing

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_fun ( 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: Tasks and Jobs

So we simply "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 );
```

(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 ('s'), and we have to create the array of input arguments:

```
n = 100001;
task_num = 10;
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

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

QUAD: Our simplest example

QUAD demonstrates the features of task programming.

The calculation is thought of as a job.

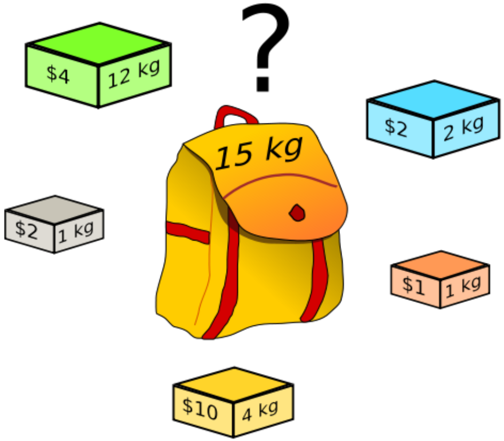
The job is thought of as multiple tasks, with each task executed by the same task function, using different inputs.

The job collects the output from each task function. The user can load all these output variables, or select a particular variable.

Once loaded, the output can be analyzed, manipulated, or plotted.

Parallel MATLAB: Task Computing

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 $S \subset W$ 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 $S \subset W$. Each **n**-digit binary string from 0 to $2^n - 1$ is a code for a possible solution.

For weights **W**$=\{15,11,10,8,3\}$, target **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 scheme 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 do ALL Codes

```
function [ code, indices ] = knapsack_all ( w, t )

  n = length ( w );

  for code = 0 : 2^n-1

    binary = bitget ( code, 1:n );
    indices = find ( binary );

    if ( sum ( w(indices) ) == t )
      return
    end

  end
  return
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 **code**. This creates the binary string that describes the subset.

The function **find** returns the indices of the 1's in the binary string.

It is exactly this string of indices that we need to use to index **W** and determine the sum of weights.

As soon as we find any solution, we return. We are not trying to find all solutions.

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 slighly modify our original code to make this new version.

KNAPSACK: Task Program to do SOME Codes

```
function [ code, indices ] = knapsack_task ( w, t, range )

  n = length ( w );

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

    binary = bitget ( code, 1:n );
    indices = find ( binary );

    if ( sum ( w(indices) ) == t )
      return
    end

  end
  return
end
```

KNAPSACK: Define the Job and its Tasks

Once we've broken our calculation into pieces, we need to tell MATLAB what the original calculation was, and what the pieces are that constitute it.

We need to tell MATLAB we want to define a **job** that is going to carry out a computation. This is the **createjob** command.

We need to tell MATLAB that the job consists of a certain number of tasks; that each task is executed by a particular function, with particular input. This is done with the **createtask** command.

KNAPSACK: Define the Job and its Tasks

```
job = createJob ( 'configuration', 'local', ...
  'FileDependencies', { 'knapsack_task' } );

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

i2 = -1;
for task = 1 : 4
  i1 = i2 + 1;
  i2 = floor ( ( 2^n - 1 ) * task / 4 );
  createTask ( job, @knapsack_task, ...
    2, { w, t, [ i1, i2 ] } );
end
```

KNAPSACK: Submit the Job

With the following commands, we submit the job, and then pause our interactive MATLAB session until the job is finished.

We then retrieve the output arguments *from each task*, in a cell array we call **results**.

```
submit ( job );   <-- Sends the job
wait ( job );     <-- Waits for completion.
results = getAllOutputArguments ( job );
  or
results = load ( job );
  or
code = load ( job, 'code' ); <-- only load CODE
destroy ( job );  <-- Clean up
```

KNAPSACK: FSU HPC Submission

```
target = 2463098;
weights = [ 518533, 1037066, ... 1259008 ];
combos = 2^length(weights);
task_num = 4;
args = {};
hi = -1;
for task = 1 : task_num
  lo = hi + 1;
  hi = ( task / task_num ) * combos - 1;
  args{task} = { weights, target, [ lo, hi ] };
end

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

KNAPSACK: Examine the results

Because your job involved multiple tasks, the output must be returned to you in a *cell array*. To see output result 2 from task 3, you refer to **results**{**3,2**}.

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

KNAPSACK: Examine the results

There are technical reasons why MATLAB uses cell arrays for many of the objects in parallel computing.

If the stuff in your cell array is all of one numeric type, you can convert the cell array to a normal MATLAB array, on which you can do the usual kinds of indexing and arithmetic and so forth:

```
results = load ( job );
numbers = cell2mat ( results );
```
(numbers is now a (4,2) numeric array)
```
c = numbers(4,1) + numbers(4,2);
integral = sum ( numbers(:,1) );
plot ( numbers(:,1), numbers(:.2) );
```

Parallel MATLAB: Task Computing

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

CELLS: Problem Definition

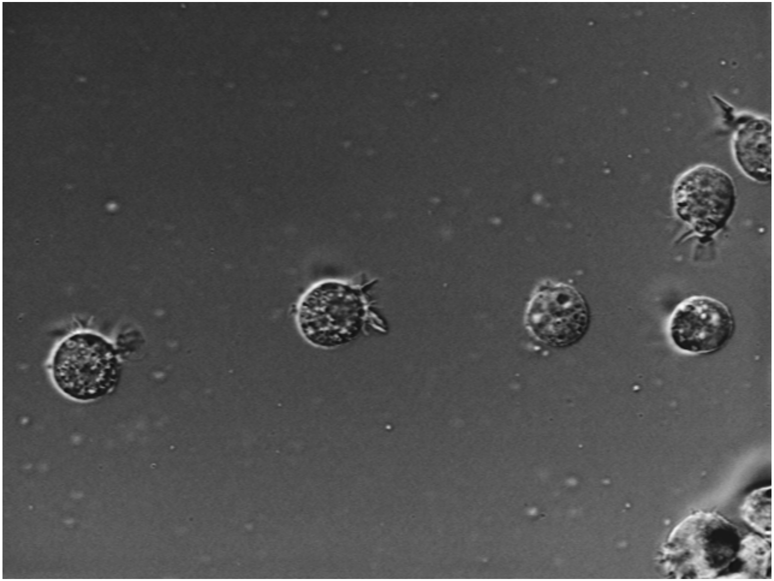
Medical researchers can film small groups of biological cells.

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.

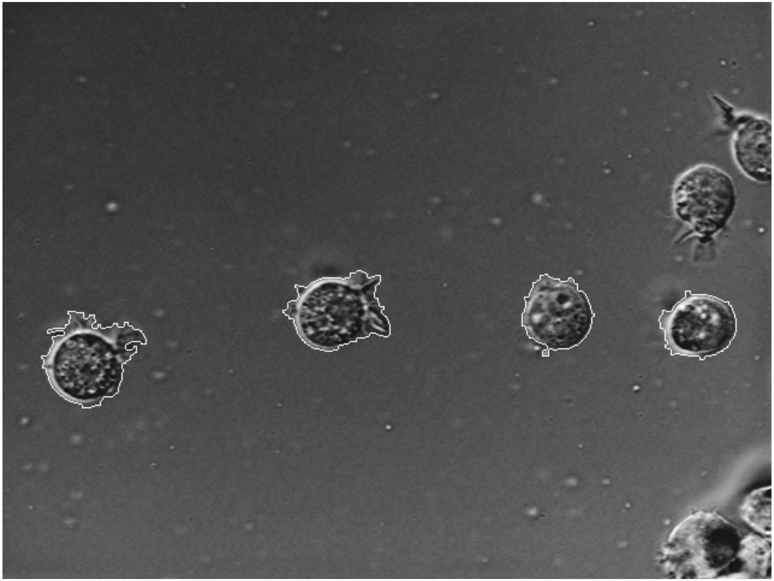
We will look at a simple application in which it is desired to identify cells by surrounding each one with a white boundary.

If we have 100 image files, each image can be processed independently.

CELLS: A Typical Image



CELLS: A Typical Image with Cells Identified



CELLS: Problem Definition

There are some significant differences between this problem and the KNAPSACK problem.

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 input files are indexed in a natural way, starting with **AT3_1m4_01.tif** and running up to **AT3_1m4_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_1m4_%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 create a job that sets up all the tasks.

Each task will need a copy of the task function, so that goes into our file dependencies.

Each task also needs a particular input file. Do we want to list all 99 of them? Let's try not to! Instead, we'll ensure that each task begins execution in the directory containing the input files.

This means we initialize the job with this simple statement:

```
job = createJob ( ...
  'configuration', 'local', ...
  'FileDependencies', { 'cell_task' } );
```

CELLS: Defining the Tasks

Now we must load up our job with tasks. But for this example, that's extremely simple!

```
for task = 1 : 99
  task_id = ...
    createTask ( job, @cell_task, 0, { task } );
end
```

Remember, the 0 indicates there are 0 output arguments, and { task } is the single input argument to the given task.

CELLS: Defining the Tasks

To run this job, we can simply type:

```
submit ( job );
```

```
wait ( job );
```

(no need for `load ( job );` *because there's no output!)*

```
destroy ( job );
```

Assuming things went OK, the output files are in our directory!

CELLS: HPC Execution

On the FSU HPC cluster, we must use the **fsuClusterMatlab()** command to set up and execute the tasks.

Recall that this command has the form

```
results = fsuClusterMatlab ( outputdir, moabopts,
  'p/m/s', 'w/n', workers, @function, {args} )
```

The only tricky part about this is the {args} object. For task computing, this contains a separate set of input values for *every task*, in other words, it's really { {args1}, {args2}, ... {argsn} }!

CELLS: HPC Execution

args is a cell array whose entries are cell arrays! It looks like this:

```
args = { {1}, {2}, {3}, ..., {98}, {99} }
```

and we can create and fill up args this way:

```
args = {}
for task = 1 : 99
  args{task} = {task};
end
```

Then we are ready to submit the job to run on 4 workers:

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

The **'s'** *means this is a "simple" job (multiple separate jobs), that is, what I've been calling a task computation.*

Parallel MATLAB: Task Computing

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.

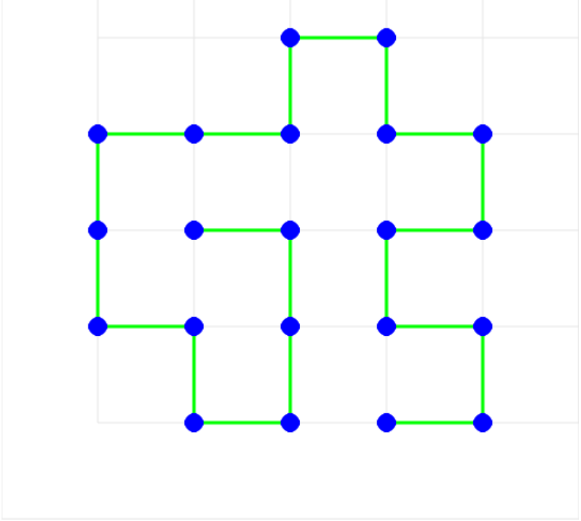
Surprisingingly, random walks are models of many physical processes, and their simulation and analysis can give insight to situations where there are no exact methods available.

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

A prototein with 21 atoms.



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: The Job Code (1)

The job code must decide how many walks are to be generated, and how many steps they are to take. It chooses random number seeds for each task, so there is no repetition of computation.

```
walk_num = 100;
step_max = 200;
for task = 1 : walk_num
  seed = 123456789 + task;
  task_id = createTask ( job, ...
    @walk_task, 2, { step_max, seed } );
end
```

The value 2 indicates each task returns 2 output arguments.

WALKS: The Job Code (2)

The job code submits the job, and waits for it to finish and collects the results:

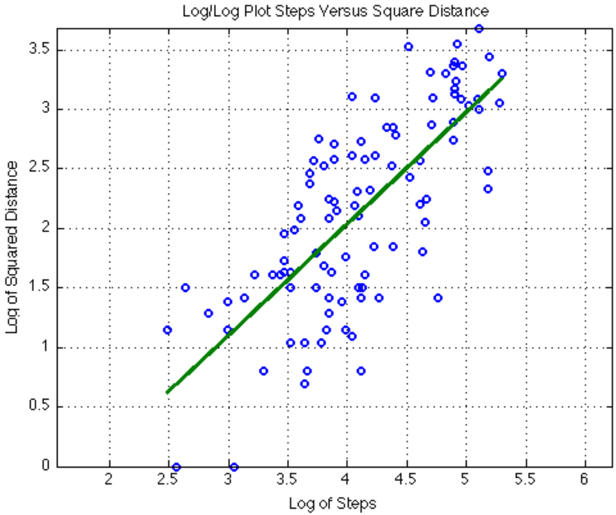
```
results = getAllOutputArguments ( job );
```

Now we want to plot the results. This is tricky, because results is a 100 x 2 cell array. We'll convert it into a standard MATLAB array:

```
results = cell2mat ( results );
```

Once **results** is a normal numeric array, we can do a scatter plot of steps versus distance, or even a least squares fit to the data.

WALKS: The Job Code (3)



CELLS: HPC Execution

On the FSU HPC, we need to make a cell array, containing **walk_num** pairs of arguments:

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

and then we submit the job:

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

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

This kind of parallel programming is quite different from the usual kinds. Each task is assigned to a single processor. Tasks could 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 our examples, every task had the same “signature”, but a job could include tasks with different task functions, different numbers and types of input and output. That’s why a cell array is necessary to deal with arbitrary output!

It’s even possible for any task to use **parfor** and run in parallel on several processors, but that’s well beyond our concerns right now!

CONCLUSION: Where is it?

The Parallel Computing Toolbox is installed on the HPC login nodes (2 licenses each) and there are 16 DCE licenses on the HPC compute nodes.

For the first presentation of this talk, FSU's Department of Scientific Computing received 20 extra, temporary licenses for the Parallel Computing Toolbox.

It was available on classroom machines **class01** through **class10** and the public machines **hallway-b** through **hallway-f**, and valid through 16-21 April 2010.

It was run from the commandline by typing
/scratch/R2010aTrial/bin/matlab

CONCLUSION: Where is it?

The temporary license includes lots of extras!:

- Curve Fitting Toolbox
- Image Processing Toolbox
- Optimization Toolbox
- Parallel Computing Toolbox
- Signal Processing Blockset
- Signal Processing Toolbox
- Statistics Toolbox
- Symbolic Math Toolbox

CONCLUSION: Where is it?

- MATLAB Parallel Computing Toolbox User's Guide 5.0
  www.mathworks.com/access/helpdesk/help/pdf_doc/distcomp/...
  distcomp.pdf
- FSU HPC web site: www.hpc.fsu.edu/
- 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**