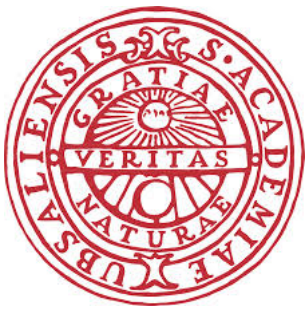


Uppsala University

Parallel Programming for Scientific Computing

Chunks and Tasks & Matlab PCT

Anastasia Kruchinina [1]
Timofey Mukha [2]

[1] anastasia.kruchinina@it.uu.se
[2] timofey.mukha@it.uu.se

Contents

Part I
Chunks and Tasks

1 Introduction

The Chunks and Tasks programming model [1] can be used for parallelization of dynamical algorithms, with a hierarchical and recursive structure. The idea comes from the area of large scale electronic structure calculations, where algorithms usually have dynamic structure and therefore it is hard to parallelize them. The user should divide data into smaller pieces, called *chunks* and specify the work to perform, called *tasks*. The user does not need to care how to distribute these data and tasks among the physical resources. It is done by the library.

The chunk class should be derived from base `cht::Chunk` class and implement a few mandatory functions needed for packing/unpacking data. After that the chunk needs to be registered:

```
class A : public cht::Chunk {...};

ChunkID id = registerChunk<chunk_type>(data, ...);
```

After registration the user gets the chunk identificator `ChunkID` and the modification of the chunk is not possible any more. `ChunkID` contains the size of the chunk (it is read only, so we know it at the start), the MPI rank of the worker where chunk is stored and the chunk type identificator, needed to reconstruct the chunk on another node.

The task class must be derived from base `cht::Task` class. To define a task the programmer needs to specify the input and output chunk types and write the `execute` function, which defines the work to be performed:

```
class A : public cht::Task
  {
  public:
    cht::ID execute(chunk_type const &, ...);
    CHT_TASK_INPUT(chunk_type const &, ...));
    CHT_TASK_OUTPUT((chunk_type));
    CHT_TASK_TYPE_DECLARATION;
  };
```

After registration the task scheduler will take care of task's execution.

2 Main program

The main program starts executing serially. Then when it is required the worker processes are spawned. Every worker starts the task and chunk schedulers and a few threads which are responsible for communication, fetching data and execution (see Figure 1). There are two listening threads, one for listening for messages of the chunk schedulers and the second is listening for the messages of task schedulers of other workers . The number of workers and threads can be specified in the program.

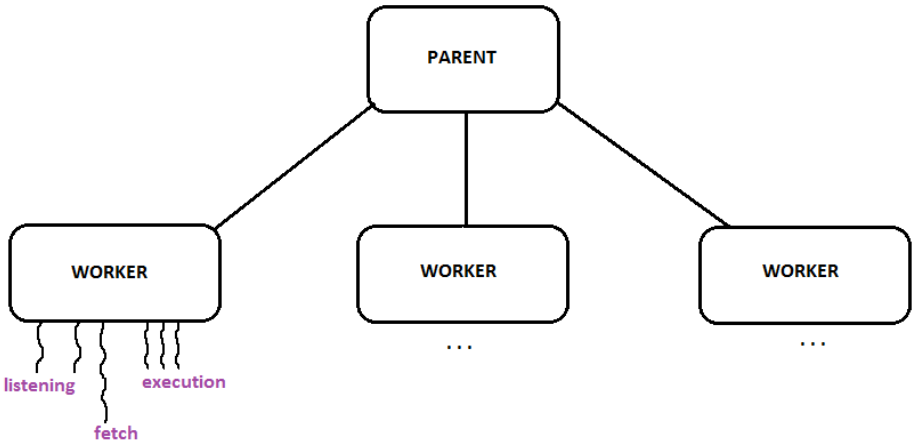


Figure 1: The main program

3 Implementation and results

As mentioned before the model is good for parallelization of hierarchical algorithms. The vector constructed as a binary tree, where each node is represented by a vector-chunk class. Every such chunk can contain two `ChunkID`s which refer to the parts of the vector or contain data if it is leaf chunk (see Figure 2). The maximum number of elements in the leaf chunks can be specified in the program (we refer to it as the block size).

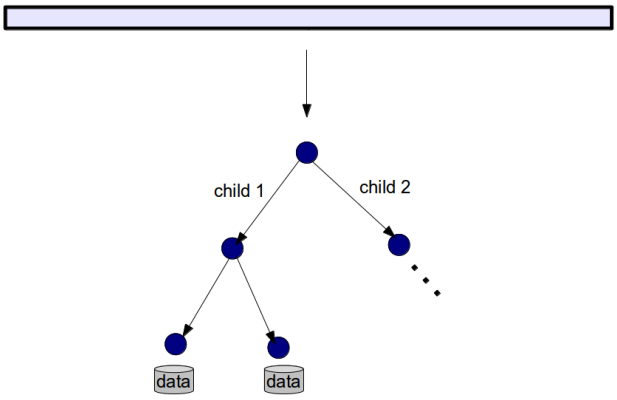


Figure 2: Vector construction as a binary tree

The main idea of the algorithm is to go down through the tree and define the tasks recursively. When the leaf-chunks are reached, the computations are performed. After that the results of leaf-chunks are combined to get the final result.

The matrix is constructed in a way similar the construction of a vector. The matrix chunk is implemented as quad tree, i.e. one stores four child `ChunkID`s (see Figure 3). When the chunk contains data which are just zeros, then it is not needed to store it and such a `ChunkID` is replaced with `CHUNK_ID_NULL`. Once the matrix chunk class is implemented, the user can use it for different representations of dense and sparse matrices. Such representations will differ just in the way data is stored in the leaf-chunks. A matrix-chunk class and a few leaf-chunk classes are implemented in the library.

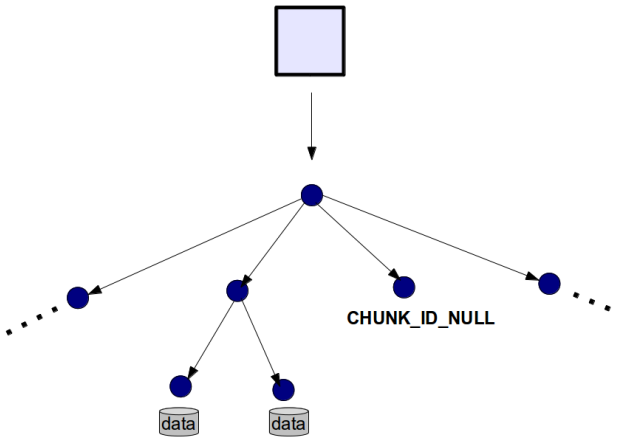


Figure 3: Matrix construction as a quad tree

All the tests were performed on the Tintin cluster at the UPPMAX computer center using gcc 4.8.2 and OpenMPI 1.6.5. The number of workers and threads were specified as input parameters. General `mpirun` command is:

```
mpirun -np 1 -bynode ./exe_file input data
```

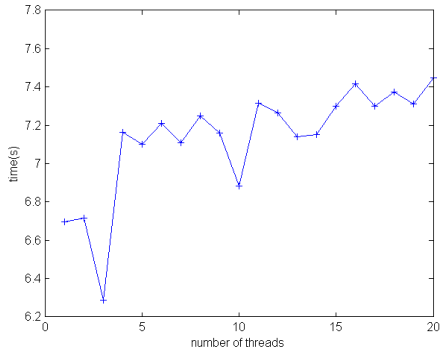
The code for every presented algorithm can be found in the following github repository: `https://github.com/Tourmaline/PPSC`.

3.1 Operations with vectors

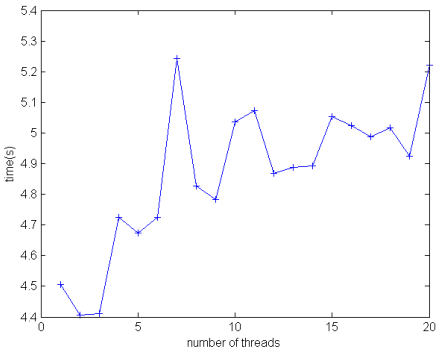
When we wish to perform **vector update**, the input and output vector-trees will have the same structure. So we need to go down the hierarchy of both input vectors and perform addition of the leaf-chunks. The **dot product** implementation is following the same idea as vector update. Dot product is performed on leaf-chunks and resulted values of two siblings are summed up.

In figure 4 the dependence on the number of threads allocated to each worker is plotted.

Also in the figure 5 results for different number of workers on 2 nodes are presented.

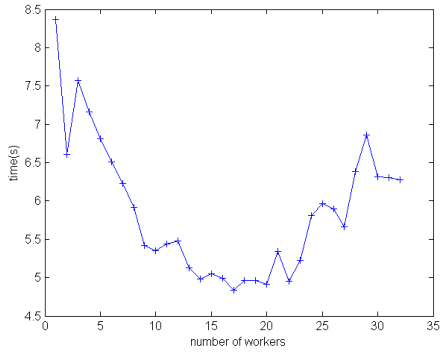


(a) Vector update

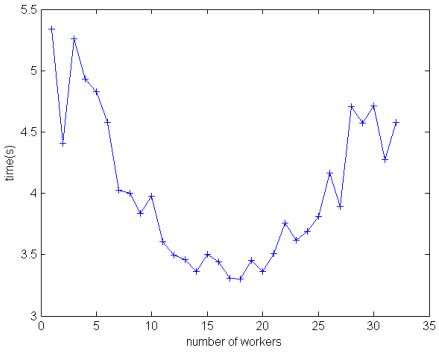


(b) Dot product

Figure 4: Vector operations for vectors of size 10000000 with 2 workers and different number of threads on each worker. Block size is 10000.



(a) Vector update

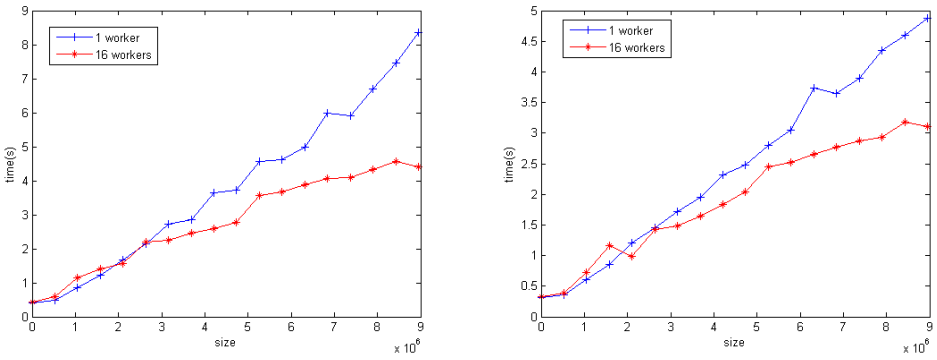


(b) Dot product

Figure 5: Vector operations for vectors of size 10000000 with different number of workers on 2 nodes and 1 thread.

The results are becoming better with increasing number of workers, but increasing number of threads does not help. It is because vector operations are so well optimized for one core, that the main part of time is spent on waiting for memory access. Therefore, increasing the number of threads will not improve results, because all these threads will finish their work fast and wait for the memory. Instead when we are increasing number of workers, memory is divided between cores. But one will not get a 2 times speed-up due to the communication overhead. So to gain something from increasing the number of threads one should increase the amount of work per thread.

In figure 6 the computation time for different sizes of vectors is plotted for both vector update and dot product operations. Results for 1 worker with 1 thread and 16 workers with 1 thread on each are compared. For both vector operations one can see speed up near just 2 times instead of possible 16.



(a) Vector update (b) Dot product

Figure 6: Comparison of results for vector operations with 1 and 16 workers with 1 thread on each. Block size is 10000.

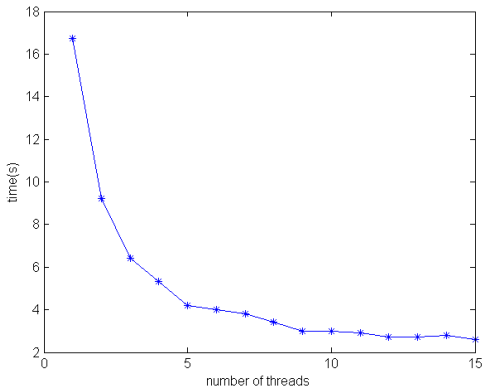


Figure 7: Matrix matrix multiplication for 6000×6000 matrix. With 2 workers and block size 1000.

3.2 Operations with matrices

With matrix-matrix and matrix-vector multiplication one should look on the number of levels in the tree and according to that decide which chunks to multiply. Here one can take advantage of the `CHUNK_ID_NULL`. When the `chunkID` is equal to `CHUNK_ID_NULL`, we know that the result will be a null-matrix or a null-vector. Matrix matrix multiplication task has already been implemented in the library.

For dense matrices leaf multiplication is done using the OpenBLAS library. For sparse matrix vector multiplication we used the *CSC (Compressed sparse column)* format for leaf matrices and the previously described vector-chunk class for the vector. In figure 7 one can see the decreasing of computational time with the increasing number of threads for the dense matrix matrix multiplication. Here threads have enough work to do therefore computational time is decreasing with increasing number of threads.

Timing and performance in plotted in the figure 8. The results for 1 worker with 1 thread and 2 workers with 15 threads on each are compared. One can see a huge speed up when 2

workers are used. The theoretical peak performance per node is 96 GFlop/s [1] and is plotted with the red line.

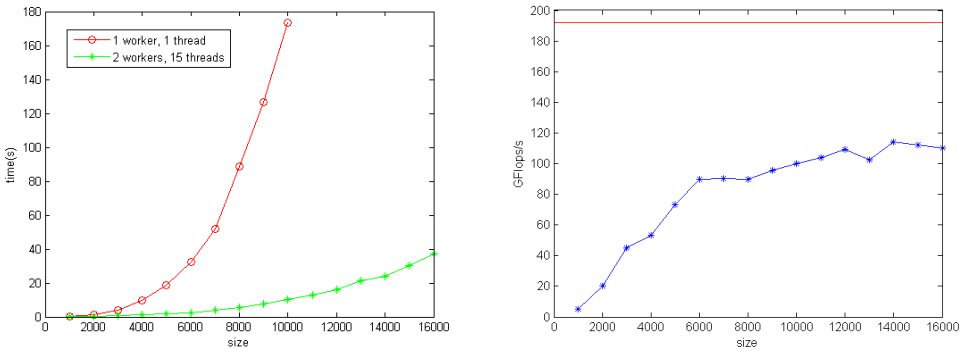


Figure 8: Matrix matrix multiplication. Left: results for different size of matrix. Block size is 1000. Right: performance for 2 workers with 15 threads on each.

Results for the sparse matrix vector multiplication are presented in the figure 9. The situation is similar like for vector operations, therefore to gain something one should increase number of workers instead number of threads.

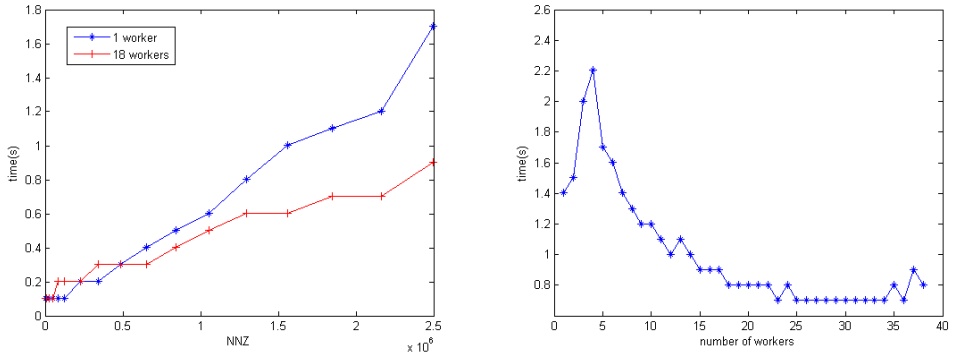


Figure 9: Matrix vector multiplication. Left: time for 1 worker and 18 workers with 1 thread. Rigth: time for different number of workers on 2 nodes and 1 threads per worker. Block size is 1000.

Matrix vector multiplication has very low operational intensity, therefore computational performance (see Figure 10) is very small (just a few MFlops/s).

4 General remarks about Chunks and Tasks

- Nice for hierarchical and recursive algorithms.
- Scalability to any number of processes. Possibility to use GPU.
- Easy understandable user interface.
- Communication and distribution of the *data and work* to the physical resources managed by the library.

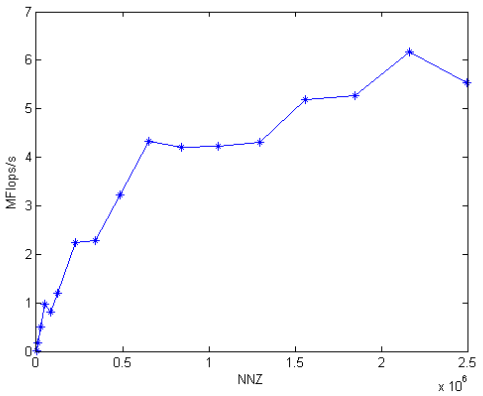


Figure 10: Matrix vector multiplication performance with 18 workers and 1 thread. Block size is 1000.

- Library takes care about load balancing. For example, so called speculative task execution is implemented. The idea is that library can decide if it is advantageous that task would be stolen by another worker or if it is better if this task will be executed on the current worker (if there is not much work to do).

Things that the user should remember:

- As almost any parallel program, it is hard to debug.
- Chunks are read only, so ones it is registered it is not possible to change it.
- Tasks can operate just with `chunkID`s or chunks which are given as input to the task. For example one cannot fetch a chunk using the `chunkID` inside a task, but instead should create another task which gets this `chunkID` as input. Because of that for small operations such as addition of two numbers it is needed to put numbers to the chunks and specify the task class representing addition. This restriction is needed to avoid blocking of the thread during execution of the task.
- The user can pass chunks or `chunkID`s as input parameters to the `execute` function. But it can be either just chunks or just `chunkID`s, one cannot mix them. Usually this does not cause problems. In case that the user must mix chunks and `chunkID`s as input, the problem can sometimes be solved the following way. The user can define the `execute` function with all the inputs as chunkIDs and inside the function register a new task which takes as input `chunkID`s of the needed parameter and operate with this chunk inside ther registered task.

Part II
Matlab's Parallel Computing Toolbox

5 Overview of Parallelism in Matlab

5.1 Built-in Functions

The well known rule of thumb for writing high-performance code in Matlab is to utilize the built-in pre-compiled functions as much as possible. This stems from the fact that those functions are not written in Matlab itself, but using lower-level languages like C/C++ and are based on performance libraries like BLAS or LAPACK. This rule is in a way both easy and challenging to follow. Easy, because currently the built-in functions cover a great deal of those tasks that one commonly encounters in numerical simulation. Hard, because to utilise the functions to there full potential one has to embrace the fact that the input and output arguments of the functions are meant to be N-dimensional arrays.

We can illustrate with a simple example. Say that given a matrix A you want to constrain the values of its elements by threshold values `max_value` and `min_value`. The conventional way to do this would be something like this:

```
1  [a, b] = size(A);
2  for i = 1:a
3     for j = 1:b
4           c = A(i,j);
5           if ( c > max_value )
6              A(i,j) = max_value;
7           elseif ( c < min_value )
8              A(i,j) = min_value;
9           end
10    end
11 end
```

While the "Matlab"-style code for this is simply

```
1  A = min( max( A, min_value ), max_value );
```

and gives a speed-up of 17.5 times for a 1000 by 1000 matrix, on a Lenovo x201i, core-i5 equipped laptop.

An additional reason for built-in functions performing so well is that many of them take advantage of multicore CPU architectures and can perform computations on several threads simultaneously . It seems that a comprehensive list of parallelised functions is not present in the documentation, but the Matlab Support Team provided a list on Matlab Central [2].

The fact that parallelism is becoming a common feature of the core functionality already available in Matlab is important to consider in order to avoid trying to reinvent the wheel. That said, the available function-level parallelism does not in any way put a cross on parallel programming in Matlab as a whole. As we will see, Matlab supports various parallelisation paradigms which can and should be used by anyone who wishes to accelerate the performance of their code.

5.2 Getting the Extra Functionality

As with many other Matlab features in order to be able to do parallel computing one is required to purchase a special toolbox. There are in fact two options to choose from. One is called the Parallel Computing Toolbox (PCT) and the other is the Distributed Computing Server (DCS). The major difference is that PCT allows you to spawn only up to 12 workers. The DCS is the designated solution for those who really want to massively parallelise their Matlab application, have access to a cluster, require a job scheduler etc. However the PCT is not there exclusively for testing out the functionality and prototyping algorithms before scaling up the parallelism. As we will see PCT gives full support for the GPU computing functionality available in MATLAB which gives the toolbox a substantial value on its own. Besides, even with 12 workers is it possible to get a significant performance gain, although as will be shown below somewhat less then one would expect if one is used to working with, say, OpenMP.

In this project all the testing was limited to the PCT. We believe, however, that this is also what a huge majority of Matlab users will have at their disposal and therefore the relevance of our investigation does not greatly suffer from this limitation .

5.3 What is Offered

Currently it seems like Mathworks has decided that the best strategy is to have multiple parallelisation paradigms under development at the same time. The result is that we have quite a few options when it comes to choosing the way we want to accelerate code, however as our analysis will show the functionality and flexibility of each of those tools is somewhat below expectations. Here is a bird-eye overview of the provided toolset:

- Parellelize a perfectly parallel for-loop using `parfor` instead of `for`.
- Employ data parallelism using `spmd` (single process, multiple data).
- Use a GPU to accelerate your code. This includes running built-in functions on the GPU, running user-created functions which are restricted to be element-wise operations and running CUDA kernels.
- Employ task-based parallelism by sending independent tasks to a scheduler that can execute them in parallel.

In the following sections of the report we will consider each of these options in greater detail.

6 Using `parfor`

Before discussing `parfor` let us briefly describe how one prepares the Matlab environment for parallel computations. The first thing to do is to spawn a pool of workers, this is done using the `parpool('clusterName', N)` command, where `N` is the number of workers. If one is running locally the name of the cluster should be `local`, which is also the default. After this is done one can begin with the parallel computations. In the workers can be killed by writing `delete(gcp)`, where `gcp` stands for "get current parallel pool". It is important to observe that even when running locally each worker is a separate process with its own memory-space.

As mentioned earlier `parfor` can be used to parallelise a perfectly parallel for-loop. For instance for a *daxpy* operation one could write the following

```
z = zeros(length(x),1);

parfor i=1:length(x)
    z(i) = a*x(i)+y(i);
end
```

Variables `x`, `y` and `z` are called sliced. That means that each worker gets its own slice of the variable to work on. Variable `a` is said to be broadcasted, since it is defined before the loop and is never assigned a value. Finally `i` is called the loop variable. Other variable types which are not used in *daxpy* include the following: temporary – a variable declared inside the loop that will seize to exist once the loop is ended; reduction – a variable which accumulates its values across the iterations, regardless of the iterations order. It might not seem that way from this short description but figuring out which type a variable gets assigned can be tricky business. A thorough description of different variable types can be found in the documentation [3].

As noted earlier all the data each worker needs is sent to it from the client. This communication is expensive and the task of the programmer is to some-how limit it. The possibilities to do that are scarce, since the communication is handled automatically.

7 Using `spmd`

If one wishes to build the parallelism around data, using `spmd` is the way to go.

The statements inside a `spmd` block are executed in parallel by the worker pool. Large datasets can be distributed between the workers using the `codistributed` statement. There is substantial freedom for the programmer to choose how the array will be partitioned. Functionality is provided for indexing and retrieving and looping over the part of the distributed array assigned to each worker. A large number of built-in functions support distributed arrays. More detailed information can be found in the documentations [4].

The workers can communicate with each other using MPI-like communication calls. However the communication time grows radically with the size of the message, so care should be taken when designing the communication patterns. The following communication calls are available:

- labSend — send data to another worker;
- labRecive — receive data form another worker;
- labSendRecieve — simultaneously send data to and receive data from another worker;
- labBarrier — block execution until all workers reach this call;
- labBroadcast — send data to all workers or receive data sent to all workers;
- labProbe — test to see if messages are ready to be received from other worker.

The set is limited, but somewhat covers the necessities. The code bellow illustrates using `spmd` for dense matrix multiplication. The result-matrix `C` is devided in columns among the workers. Since the size of the matrices is not large we replicate both `A` and `B` to all the workers.

```
nra = size(A,1);
nca = size(A,2);
ncb = size(B,2);

spmd (n)
    %devide the columns among the workers
```

```
7       start = floor(ncb/n)*(labindex-1)+1;
8
9       if(index < n)
10          endd = floor(ncb/n)*labindex;
11      else
12          endd = ncb;
13      end
14
15      %create a private matrix to hold the local result
16      Cd = zeros(nra,ncb-start+1);
17      %perform the multiplication
18      for i = 1:nra
19          for j = start:ncb
20              d = 0;
21              for k = 1:nca
22                  d = d + A(i,k)*B(k,j);
23              end
24              Cd(i,j-start+1) = d;
25          end
26      end
27
28  end
29  %gather all the coulmns into one matrix
30  C = Cd{1};
31  for i = 2:n
32      C = cat(2,C, Cd{i});
33  end
```

On rows 30-31 we concatenate the private matrices `Cd` into the full matrix `C`. The `{i}` is the notation used for accessing the `Cd` of worker `i`. This is not to be confused with Matlab's cell arrays which use a similar notation for accessing a cell.

8 Using the GPU

Using GPUs for computations is becoming more and more popular, not surprisingly Mathworks does not want to lag behind and has packed Matlab with a big amount of features that make GPU computing easy. The only thing the user needs is a CUDA-enabled GPU.

In order to perform any computation on a GPU one has to transfer all the necessary data from the RAM to the GPUs global memory. In Matlab this is a zero-effort process, all one has to do is use the function `gpuArray`, which takes a normal array as argument and returns an array allocated on the GPU. Copying the data back is done with the function `gather`. Anyone with an experience of performing the same tasks using the CUDA library directly will appreciate how Matlab makes things really easy for the user here.

The first and quite natural expansion to GPU grounds by Matlab is GPU-enabling some of the built-in functions. The functions are simply overloaded to run on on the GPU if one of the input arguments is a gpuArray. Consider the following code snippet

```
1  Agpu = gpuArray(A);
2  Bgpu = gpuArray(B);
3
4  Cgpu = mtimes(Agpu,Bgpu);
5  C = gather(Cgpu);
```

Two GPU-arrays are created and the multiplication function is called to to multiply on the GPU. The result is then copied back using `gather`.

Besides for running built-in functions the user can also run their own, but with severe limitations. The biggest of them is that the function should boil down to some kind of element wise operation on arrays. More interestingly Matlab allows to run arbitrary CUDA kernel right from the Matlab environment. One loads the kernel into a `CUDAKernel` object using the .cu and .ptx files for the kernel, easily set the execution properties like the grid parameters and shared memory size, and then runs the kernel sending `gpuArray`s as arguments. The following snippet performing a *daxpy* gives a good illustration:

```
%establish gpu-arrays
xGpu = gpuArray(x);
yGpu = gpuArray(y);

n = length(x);
%load the kernel and set the properties
k = parallel.gpu.CUDAKernel('daxpy.ptx','daxpy.cu');

k.ThreadBlockSize = [256 1 1];
k.GridSize = [floor(n/256)+1, 1, 1];

%run the kernel
zGpu = feval(k, xGpu, yGpu, a, int32(n));

%copy back the result
z = gather(zGpu);
```

By default the output of `feval` is the first kernel argument which is a pointer not pointing to a `const`.

9 Using a Job Scheduler

When connected to a cluster (in our case the 'local' one) it is possible to submit jobs that contain multiple independent tasks. These tasks will then be executed in parallel. A task is essentially a function evaluation.

To submit a job one must first define the cluster that the job will be submitted to. The is done with the command `parcluster`. Jobs are created with the `createJob` function which takes a cluster object as argument.

To add tasks to a job one uses the `createTask` function which takes the following arguments: the job that the task object is created in, a function handle to the function which represent the actual work done by the task, the number of output arguments returned by the task, a row cell array of input arguments to be passed to the task.

To submit the job one calls the `submit` command, to `wait` function makes sure all the tasks have finished execution, the function `fetchOutputs` can be used to gather the results from all the tasks.

Consider the following code for calculating the dot product which demonstrates the discussed concepts.

```
l = length(x);
%numer of tasks
n = 12;
```

```
4
5  %divide the vectors among the tasks
6  for i =1:n
7          start(i) = floor(l/n)*(i-1)+1;
8          endd(i) = floor(l/n)*i;
9  end
10
11 %define the cluster and create the job
12 sched = parcluster;
13 joblist = createJob(sched);
14
15 %add 12 tasks to the job, each computing a part of the dot product
16 for i = 1:n
17          createTask(joblist,
               @dot_product_serial,1,{x(start(i):endd(i)),y(start(i):endd(i))});
18 end
19
20 %submit the job
21 submit(joblist);
22
23 %wait for the job to end
24 wait(joblist)
25
26 fetch the output
27 out = fetchOutputs(joblist);
28
29 %perform reduction on the output
30 z2=0;
31 for i = 1:n
32         z2 = z2 + out{i};
33 end
```

In this example all the tasks execute the same function, namely `dot_product_serial` which computes a dot product with a for-loop, but it is important to stress that the tasks can run completely different functions.

10 Benchmarking

10.1 Hardware, Software, Timing

All the computations were performed using a single node of UPPMAX Tintin HPC cluster. Such a node has two 8-core Opteron 6220 processors running at 3 Ghz and 64 Gb of RAM. The node we used was also equipped with a Tesla M2050 GPU. Basically this is comparable to a configuration of a modern high-performance workstation.

Matlab version 8.1.0.604 (R2013a) was used in all of the benchmarks.

Matlab's builtin `tic` and `toc` routines were used for timing the code. All of the implemented algorithms take the form of a Matlab function. It is the runtime of that function which is being timed.

The maximum number of threads allowed by PCT (12) will be used to compare the performance.

All of the codes used for the benchmarks are available here `https://www.dropbox.com/sh/xl3f4cv73121579/7S3wugE0n9`

10.2 Naming Convention

For simplicity the following notation will be used.

- `builtin` — this will refer to using Matlab's internal routines like `mult` or `dot`. If not stated otherwise the routines will run using only one thread. This is achieved by starting Matlab with a -singleCompThread parameter.
- `builtinGPU` — will refer to using Matlab's internal routines on GPU arrays.
- `serial` — shall refer to a naive serial implementation, typically using multiple for-loops.
- `parfor` — will refer to a parallel implementation using `parfor`.
- `spmd` — will refer to a parallel implementation using `spmd`.
- `scheduler` — will refer to using the job scheduler and submitting several tasks
- `CUDA` — will refer to using a CUDA core for doing the computations. Clearly the performance is then dependent on the kernel used, so this will have to be discussed.
- `gpuElement` — will refer to using a custom element-wise function to perform the computations on the GPU.

Not all of the above-mentioned will be implemented for every benchmark, but all will be benchmarked at least once. We will not show the code for all the implementations in the main body of the report either, instead the reader can find all the relevant code in the appendix.

10.3 Vector update: *daxpy*

The first benchmark is performing a vector update, following the BLAS naming conventions called *daxpy*. A trivial analysis shows that a function performing a *daxpy* for a vector of size N has to do $2N$ loads and N writes to memory while performing only N flops. This shows that accessing the memory is the clear bottle-neck here.

There isn't much to say about the different implementations when it comes to *daxpy* due to the simplicity of the operation. Here is the summary:

- `builtin` — simply `z = a*x+y`.
- `serial` — a for-loop running over the vectors' elements, the body of the loop performs a *daxpy* on an element.
- `parfor` — the same but the for-loop is exchanged to a `parfor`-loop.
- `cuda` — each thread calculates a *daxpy* for one element of the vector.
- `gpuElement` — the element-wise function performs an element-wise *daxpy*.

Figure 11 shows the performance of all the considered implementations. We see that the result for `parfor` is strikingly poor. This has to do with the communication overhead of transferring the data from the Matlab client to the workers. We see that in order to get a speed-up out of `parfor` the computational intensity of the algorithm has to be substantial.

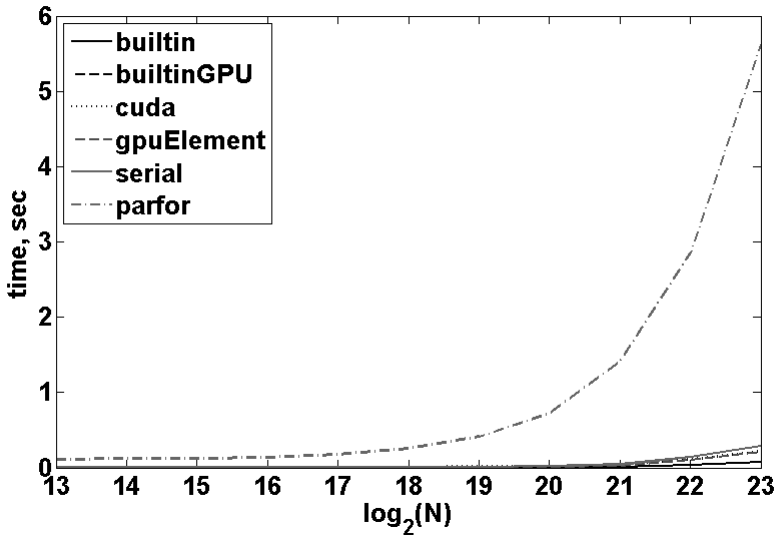


Figure 11: Performance of different *daxpy* implementations for vectors of different size N.

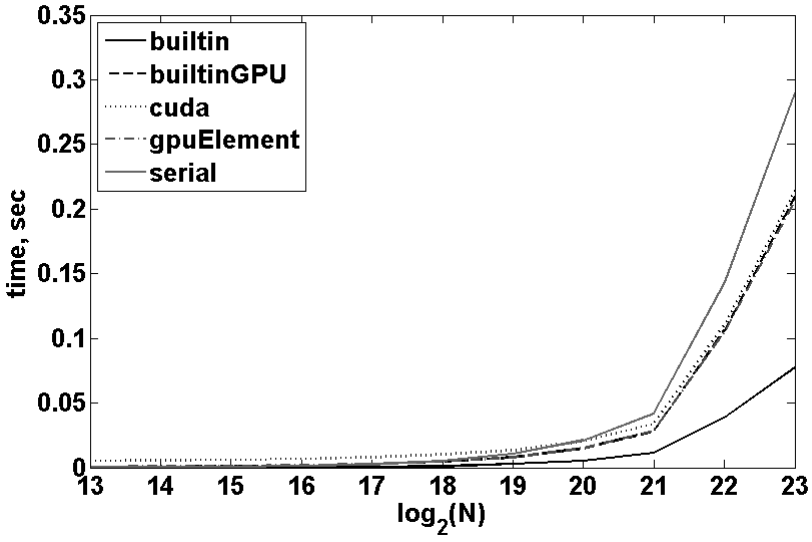


Figure 12: Performance of different *daxpy* implementations for vectors of different size N, `parfor` excluded.

The results for the other implementations are hard to analyse on figure 11 because of the `parfor`-curve standing out so much. Figure 12 shows the same data but excluding the `parfor`. We see that the fastest computation was performed by Matlab's built in routine. The built-in routine does not suffer from additional memory transfer overheads as all the GPU implementations do and is pre-compiled which makes it much faster then naive `serial` loop.

All of the GPU implementations showed very similar performance. All of them beat the

serial implementation. The CUDA kernel seems to be slightly slower used for small vectors. The `gpuElement` and `builtinGPU` perform nearly identically. The reason for this similarity in performance is probably the fact that there is not much space for optimisations when it comes to such a simple operation. Besides, Matlab's own routine should treat a general N-d array, while our kernel is written for working with vectors specifically.

10.4 Dot product

We continue with the *dot product* operation. The implementations are as follows.

- `builtin` — `z = dot(x,y)`.
- `serial` — a for-loop over the vectors, the body of the loop performs element-wise multiplication and adds the result to `z`.
- `parfor` — the same but the for-loop is exchanged to a `parfor`-loop.
- `spmd` — each worker performs the *dot product* on a part of the vectors, the resulting 12 partial sums are then added up in a for-loop to get the final answer. All of the workers get a full copy of `x` and `y` i.e. array-distribution is not used.
- `scheduler` — the same concept as with `spmd` but using tasks instead of workers.
- `cuda` — each thread-block performs a *dot product* on a part of the vectors. The output is then summed up on the CPU.

Before looking at other implementations we note that the result for `scheduler` was completely unsatisfactory, the time for a vector of size 2^7 was 10 seconds. This shows that the overhead form using the scheduling system and building up the job from different tasks is very large.

The results the other implementations for different sizes of the vectors N are shown on figure 13. Two results stand out: the `daxpy` and the `spmd` both being slow. From the curve for the `serial` implementation we know that the actual computing time done by the `daxpy` and the `spmd` are small. Therefore what we see is basically the communication time between the client and the workers which grows non-linearly with the size of the data.

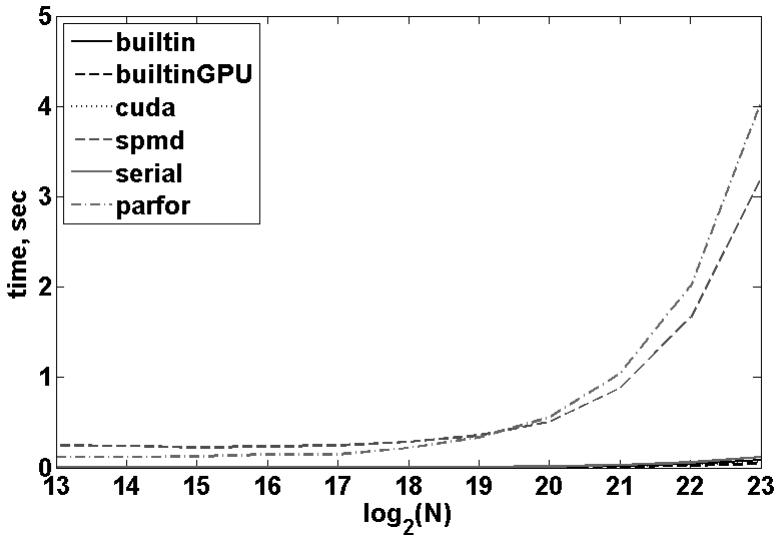


Figure 13: Performance of different *dot product* implementations for vectors of different size N.

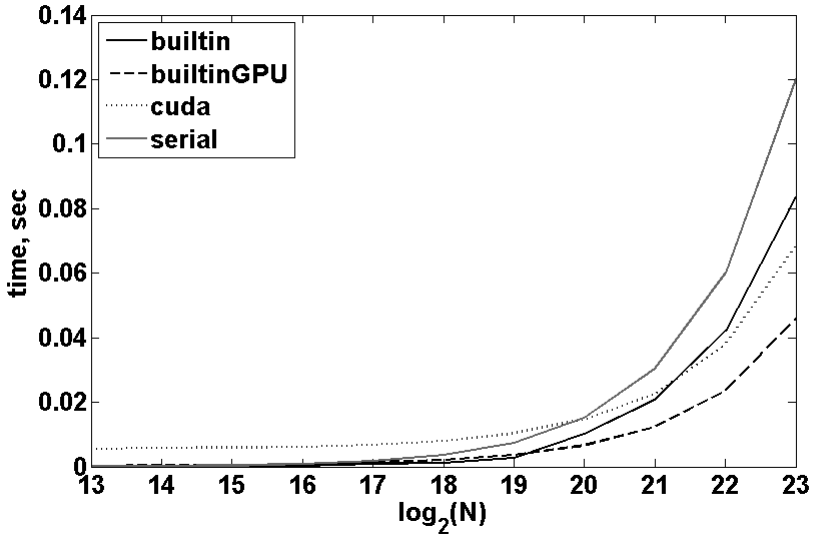


Figure 14: Performance of different *spmd* implementations for vectors of different size N, `parfor` and `spmd` excluded.

Figure 14 shows the same result but excluding `parfor` and `spmd`. We see that the fastest routine is the `builtinGPU`. For large N `cuda` comes second, but for smaller ones `builtin` is better. It is nice too see that Matlab's built-in GPU routines do give a speed-up even for a *dot product* which is not computationally intensive.

10.5 Dense Matrix-Matrix Multiplication

The next benchmark is the multiplication of two dense matrices: $C = A \cdot B$. This task is significantly more computationally expensive, so the hope is to see sum good speed-ups.

Here is how the *matrix multiplication* is implemented using the different parallelisation techniques.

- `builtin` — simply `C=A*B`. For this benchmark we will consider both the single- and multi-threaded versions of the built-in routine, the nick-name for the multi-threaded will be `builtinMulti`.
- `serial` — naive implementation with three for-loops: one loops over rows of `C`, the second over columns, and the third performs the appropriate dot product.
- `parfor` — the same as `serial` but the outer loop is a `parfor` loop.
- `spmd` — The matrix C is divided column-wise among the workers. The resulting columns are then serially concatenated.
- `cuda` — A kernel with so called "tiled" matrix multiplication is implemented, a thorough description can be found in [6] and on numerous web-sites.

Square matrices of size $N \times N$ will be considered. Since the complexity of *matrix multiplication* grows rapidly with N for the slower algorithms the largest N considered is 2^{11}, while for the faster algorithms the size is increased up to 2^{13}.

As with the other benchmarks we begin by looking at the performance of all the algorithms for various N, see figure 15. We observe that all parallel and `builtin` implementations beat the `serial` one. For large N both `parfor` and `spmd` now give a substantial speed-up. However, their performance is still far below that of the `builtin` and `cuda` implementations.

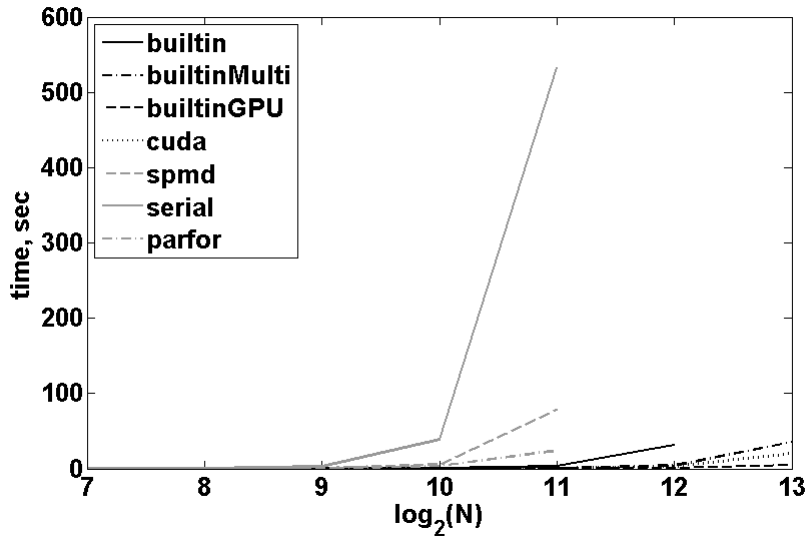


Figure 15: Performance of different *matrix multiplication* implementations for matrices of different size $N \times N$.

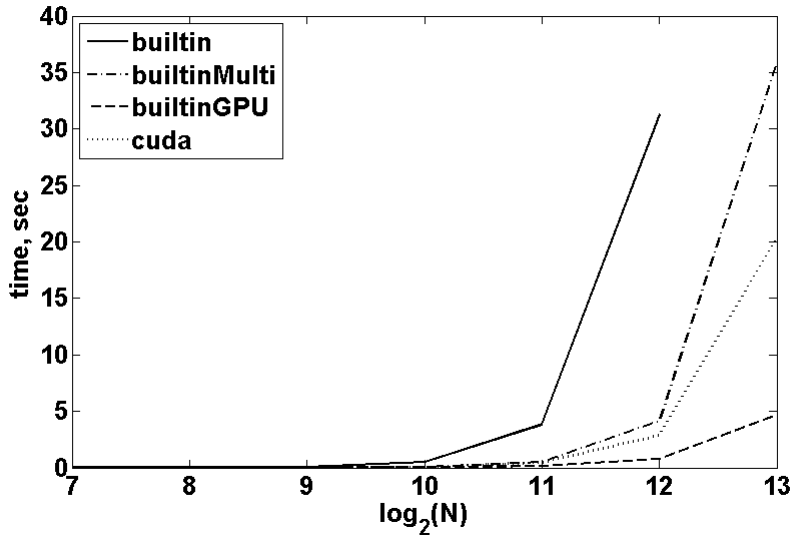


Figure 16: Performance of different *matrix multiplication* implementations for matrices of different size $N \times N$, `parfor` and `spmd` and `serial` excluded.

On figure 16 we have a clearer view of the performance of the faster implementations. As with *dot product* the `builtinGPU` secures a win. It runs significantly faster than the `builtin` and `builtinMulti`. This shows that for those who have a high-performance GPU testing Matlab's GPU routines is a very good idea for decreasing run-times. The fact that `cuda` manages to come second is a compliment to GPU computing in general: a simple CUDA kernel managed to beat Matlab's highly optimised multi-threaded CPU routine.

The same results can be view from another angle by looking at the speed-ups, see figures 17 and 18 below.

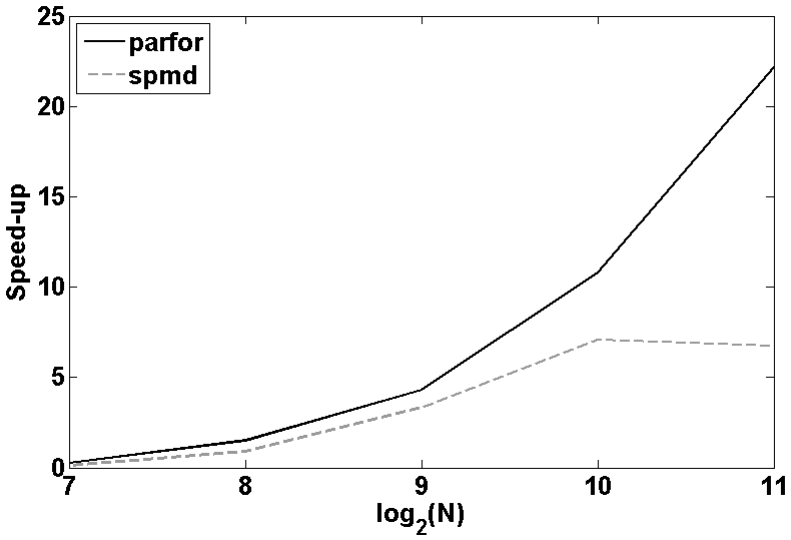


Figure 17: The speed-up of different *matrix multiplication* implementations against `serial` for matrices of different size $N \times N$.

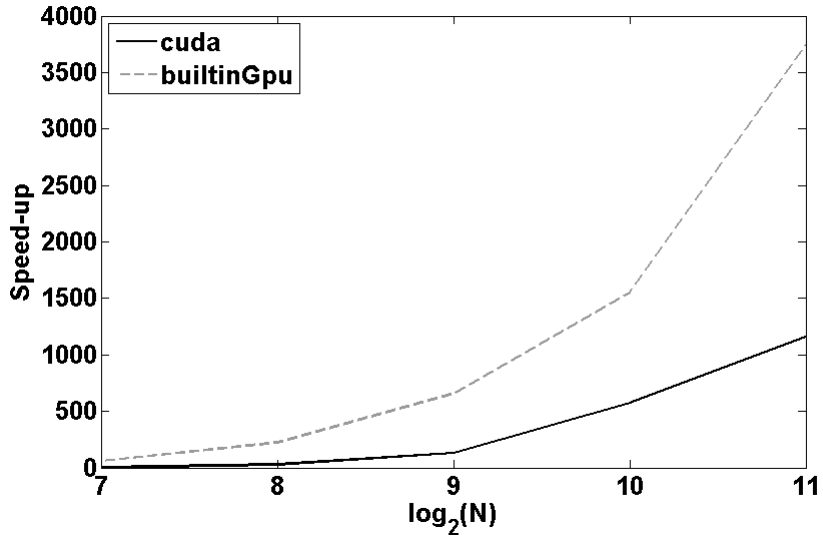


Figure 18: The speed-up of different *matrix multiplication* implementations against `serial` for matrices of different size $N \times N$.

10.6 Sparse Matrix-Vector Multiplication

Another interesting operation to test is sparse matrix-vector multiplication. There are several ways to store a sparse matrix in memory and the choice of the storage format to a big extent determines the algorithms to be use for the *spmv*. Here we use one of the common storage

formats called Compressed Sparse Row (CSR). In CSR one stores three arrays: an array holding all non-zeros, an array of the column-indices of the non-zeros and and array of row-pointers which point to the locations in the non-zeros array where a new row begins.

To generate the sparse matrices the built-in Matlab command `sprand(nRows, nCols, density)` was used. The generated matrix is not stored in CSR format however, but in the so called coordinate format which differs in the way that it stores the row-indices of the non-zeros explicitly. Therefore a routine was used to convert the generated matrix to CSR format. However, when using Matlab's own routine for *spmv* the conversion was not performed, so the `builtin` implementation does actually work with a different storage format.

Here is how the *spmv* was implemented with the available tools.

- `builtin` — simply `z = A*x`, where `A` is a sparse matrix generated by Matlab.
- `serial` — implemented as two for-loops. The first one loops through the elements of the resulting vector, the second one performs the dot product of a matrix row with the vector `x`.
- `parfor` — the same as `serial` but the first for-loop is a `parfor` loop.
- `cuda` — the same idea as in the `parfor` each thread takes care of one element of the resulting vector.

  The results for a sparse matrix with density 0.05 are shown on figure 19. None of the parallel implementations managed to beat Matlab's serial built-in routine. The naive CUDA kernel that we used did outperform the serial implementation however. The `parfor` offered no advantage over `serial` being several times slower.

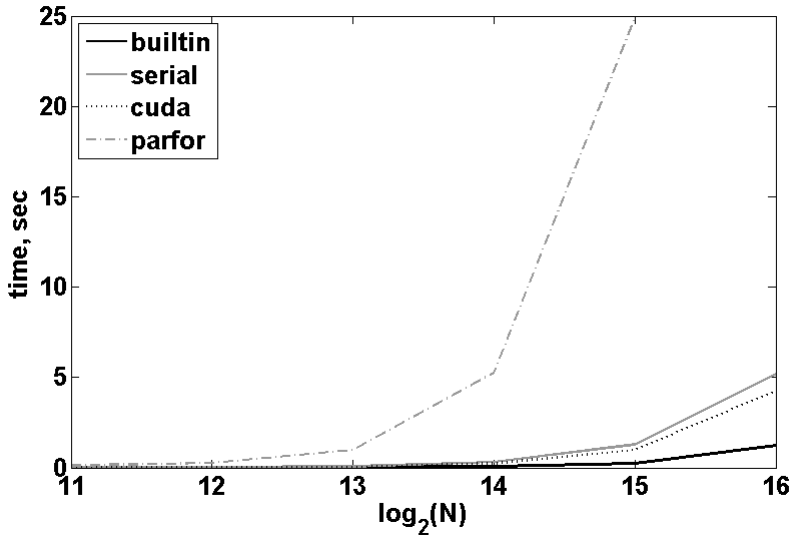


Figure 19: Performance of different *spmv* implementations for matrices of different size $N \times N$.

Experimenting with reasonably higher (up to 25%) densities did not affect the results significantly.

The poor results of the parallel implementations are due to the low computational intensity of the *spmv* operation. As with *daxpy* and *dot product* we see that `parfor` suffers severely from the communication overhead.

11 Conclusions

In this report we have tested the various parallel computing capabilities available in Matlab's PCT. Here we summarize our experiences with each of the available tools and draw some conclusions regarding their usability for different categories of Matlab users.

11.1 GPU computing

Matlab's implementation of GPU computing capabilities is excellent. Allocating arrays on the GPU and transferring data to and from the device is extremely easy. The tested built-in GPU routines provide a very substantial speed-up, and since they have exactly the same syntax as their CPU counterparts it is very easy for users to take advantage of them in their code.

Running CUDA kernels is also extremely easy. For those starting out with CUDA-programming Matlab is a great platform for checking out different kernels without having to deal manually with data transfer and other things.

Matlab also provides a great facility to test kernels. The chances that Matlab has already implemented the functionality of your kernel as one of it's internal routines is pretty high so it is often very easy to check if the kernel works correctly.

The *spmv* kernel is good example. In order to test the kernel in C/C++ one would have to create a bunch of support-routines: to generate a random sparse matrix of desired density, to generate a random vector, to check the result of the multiplication on the CPU etc. Using an external library is of course also an option but in practice using Matlab will usually turn out to be easier and faster.

That said, there is no straight-forward way to debug a kernel in Matlab, so completely getting away for C/C++ is probably impossible.

Executing custom Matlab functions on the GPU is not really usable in its current form, when the function has to boil down to some element-wise operation. The reason is that most of those operations already exist as GPU-enabled Matlab internal routines. It will be interesting to see if Mathworks peruses this line of development further and comes up with a way of to allow users to write more general functions.

In conclusion we would like to point out that the performance gain from using the GPU is very hardware dependent. In our benchmarks we obtained significant speedup but we did use a high-end GPU which most users will not have access to. Therefore one should always start by testing out the hardware one has at hand before relying on the GPU routines to accelerate the performance.

11.2 The `parfor` and `spmd`

The biggest plus of `parfor` is that it is very easy to change a serial code to a parallel one, when the algorithm allows for perfect parallelisation. In all the benchmarks used in this report changing a single `for` to a `parfor` was the only thing done to get a parallelised version of the algorithm.

The biggest downside is that using `parfor` (and `spmd`) is only beneficial if the computational intensity of the algorithm is high. The overhead from communicating with the workers is very significant.

The best possible development of the current implementation would be support for multi-threading on a shared-memory machine. This would remove the communication overhead and make the `parfor` much more efficient for a regular PCT user. Since the restrictions on what can be inside the `parfor` loop are already quite sever this would not require change in programming practice.

The `spmd` method gives much more freedom in terms of what each worker is allowed to do and makes simple communication between the workers available. This potentially makes `spmd` useful for algorithm prototyping.

The bottom line here is that Mathworks has to work on the communication-times /patterns before the regular user will be able to take advantage of the provided functionality.

11.3 Task Parallelism

The ability to pack several tasks withing one job and running them in parallel can be handy for such things as Monte-Carlo simulations, ODE parameter sweeps and other simulations which require executing a multiple of independent, not necessarily identical, tasks.

The functionality for handling input-output to the tasks is quite easy to use as well as the process of setting up a job and packing several tasks into it.

As with `parfor` and `spmd` the functionality is built with big computational clusters in mind, so even on a local machine one has to connect to a "cluster", submit jobs to a "queue" and so on. It would seem logical to have a more single-workstation friendly implementation to be found in PCT and a cluster oriented on DCS.

References

[1] Emanuel H. Rubensson and Elias Rudberg, *Chunks and Tasks: A programming model for parallelization of dynamic algorithms* , Parallel Computing, 2013.

[2] List of built in Matlab functions with support for multithreading, http://www.mathworks.com/matlabcentral/answers/95958

[3] Documentation on variable classification in a `parfor` loop, http://www.mathworks.se/help/distcomp/advanced-topics.html

[4] Documentation on working with codistributed arrays in Matlab, http://www.mathworks.se/help/distcomp/working-with-codistributed-arrays.html

[5] Optimizing Parallel Reduction in Cuda, http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[6] Kirk, B. D., Hwu, W. W., *Programming Massively Parallel Processors,* Eslsevier, 2010