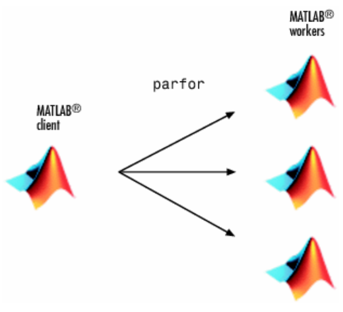
Parallel Computing Toolbox MATLAB

Jarmo Rantakokko

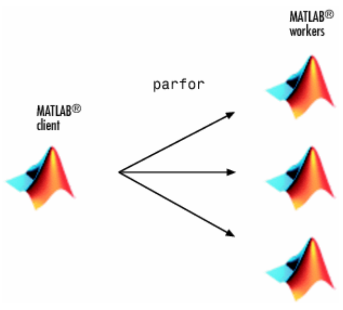


Topics:

1. Parallel for-loops
   Loop-level, ”shared” data
2. SPMD parallelization
   Same task, multiple data
3. Task parallelization
   Different task, multiple data
4. GPU acceleration

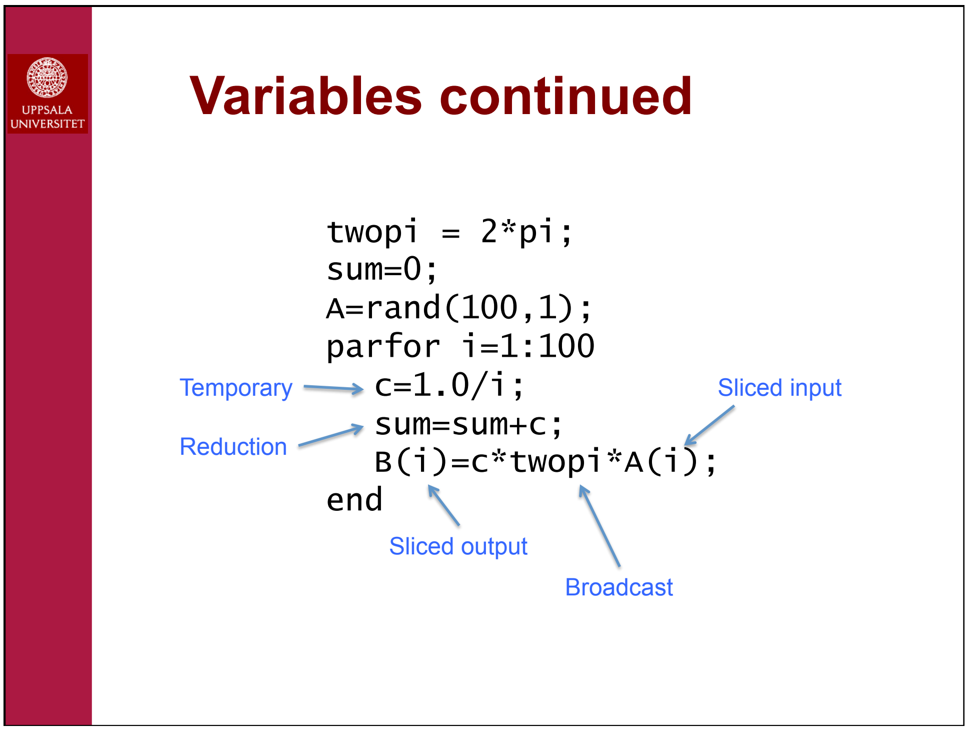
Parallel for

```
% Request 3 workers (max 12)
>> matlabpool 3

% Run in parallel
parfor i=1:n
   A(i)=func(B(i));
end

% Return workers
>> matlabpool close
```



Classification of variables

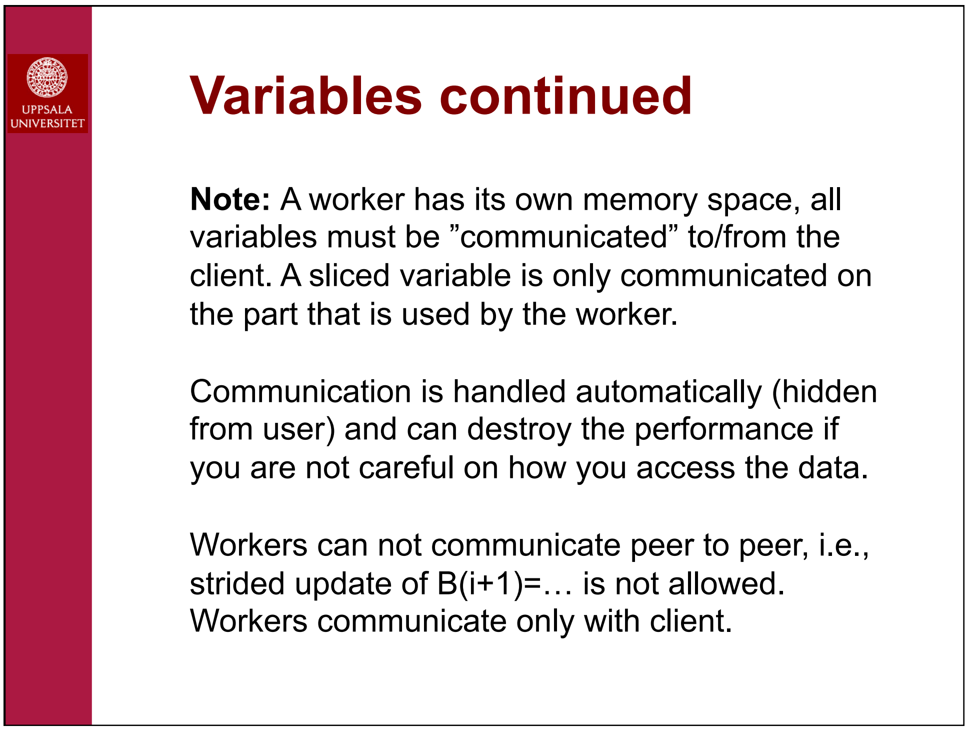
Classification	Description
Loop	Loop index for arrays
Sliced input	Array whos elements are read in parallel by different workers
Sliced output	Array whos elements are written in parallel by different workers
Broadcast	A variable defined before parallel and used inside parallel, but never assigned
Reduction	A variable that is accumulated in parallel
Temporary	A variable that is created inside parallel, not available outside parallel



Variables continued

```
twopi = 2*pi;
sum=0;
A=rand(100,1);
parfor i=1:100
    c=1.0/i;
    sum=sum+c;
    B(i)=c*twopi*A(i);
end
```

Temporary
Sliced input
Reduction
Sliced output
Broadcast



Variables continued

Note: A worker has its own memory space, all variables must be ”communicated” to/from the client. A sliced variable is only communicated on the part that is used by the worker.

Communication is handled automatically (hidden from user) and can destroy the performance if you are not careful on how you access the data.

Workers can not communicate peer to peer, i.e., strided update of B(i+1)=… is not allowed. Workers communicate only with client.

UPPSALA UNIVERSITET

Parfor example MxM

```
parfor i=1:n
   for j=1:n
      for k=1:n
         C(i,j)=C(i,j)+A(i,k)*B(k,j);
      end
   end
end
```

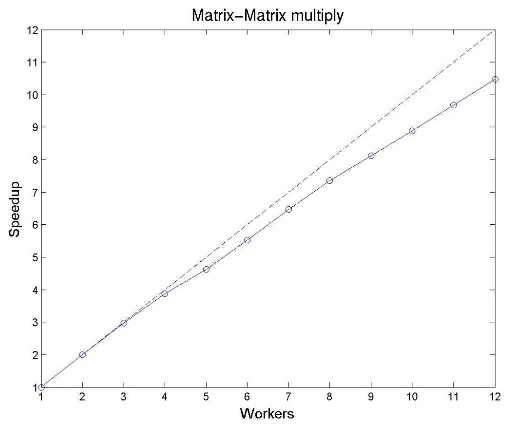
Sliced input

Sliced output

Broadcast

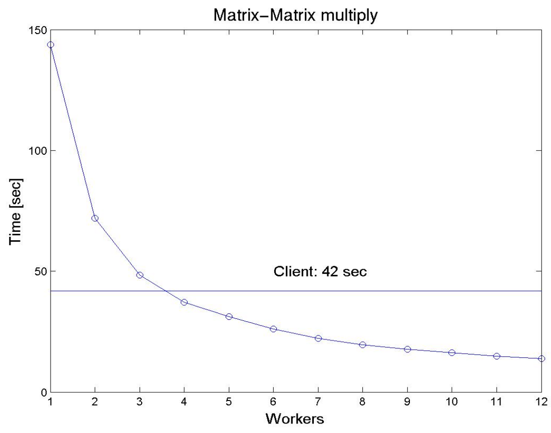
UPPSALA UNIVERSITET

Parallel Performance



Speedup: $S=T_1/T_P$ where T_1 is the run-time using one worker and T_P using P workers.

Parallel vs Serial



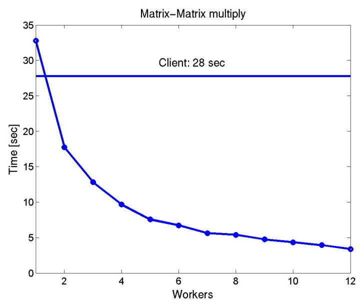
Note: C matrix is both sliced input and output.
Lot of communication to/from client-worker (both ways)!

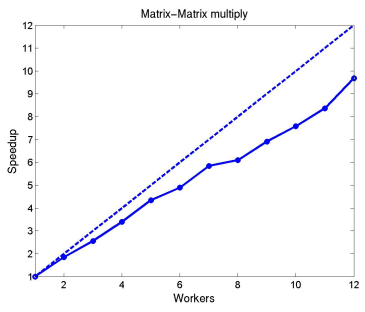
Improved version MxM

```
parfor i=1:n
   for j=1:n
      d=0;
      for k=1:n
         d=d+A(i,k)*B(k,j);
      end
      C(i,j)=d;
   end
end
```

Note: Now C is only *sliced output*, d is a temporary.

Parallel Performance





Small overhead in reading A and B, writing C

Example Enumeration Sort

```
for i=1:n
   rank=1;
   for j=1:n
        if indata(i)>indata(j)
          rank=rank+1;
        end
   end
   outdata(rank)=indata(i);
end
```

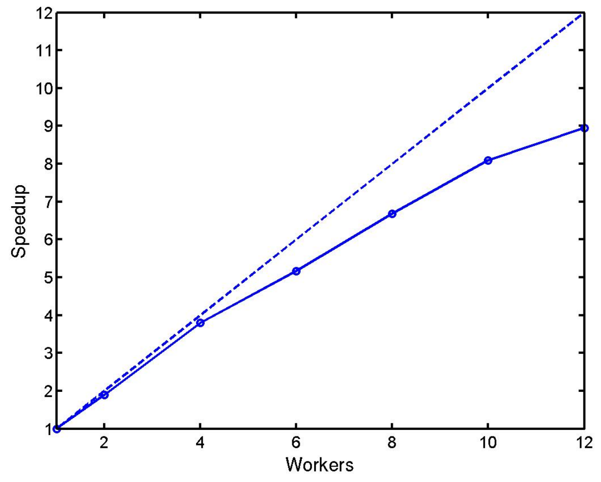
Note: i-loop is perfectly parallel, all indata is needed by all workers (broadcast) but outdata is written in parallel irregularly (prohibits parfor).

Example Enumeration Sort

```
parfor i=1:n
   rank=1;
   for j=1:n
        if indata(i)>indata(j)
          rank=rank+1;
        end
   end
   rankarray(i)=rank;
end
```

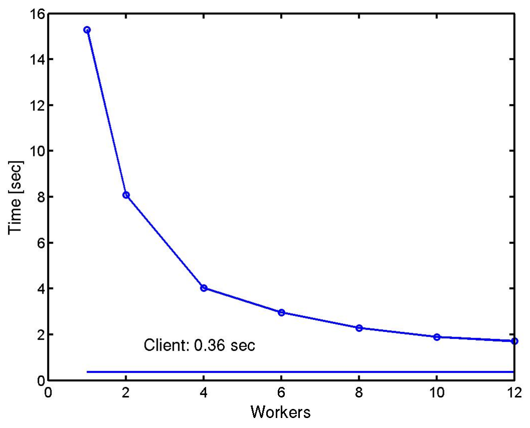
=> rankarray *sliced output* and we can use **parfor**

Example Enumeration Sort



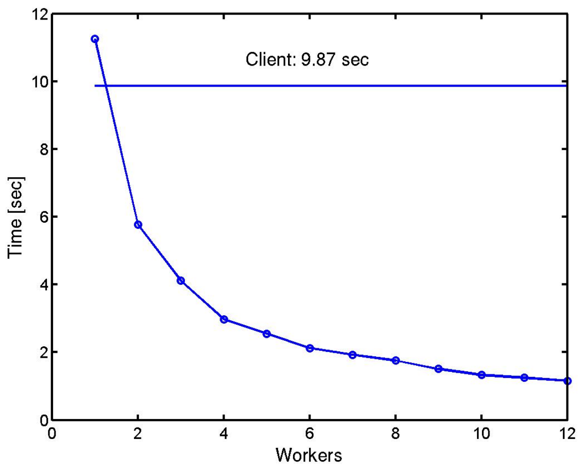
UPPSALA UNIVERSITET

Example Enumeration Sort



WTF! Serial code on Client runs 5 times faster than parallel code on 12 workers! Why, large overhead in starting up workers, communicating data and code poorly optimized on workers?!

UPPSALA UNIVERSITET



Improvement: Implement enumsort as a function (functions are better optimized than scripts) and increase the problem size (overhead in starting up workers and communicating data is diminished).

UPPSALA UNIVERSITET

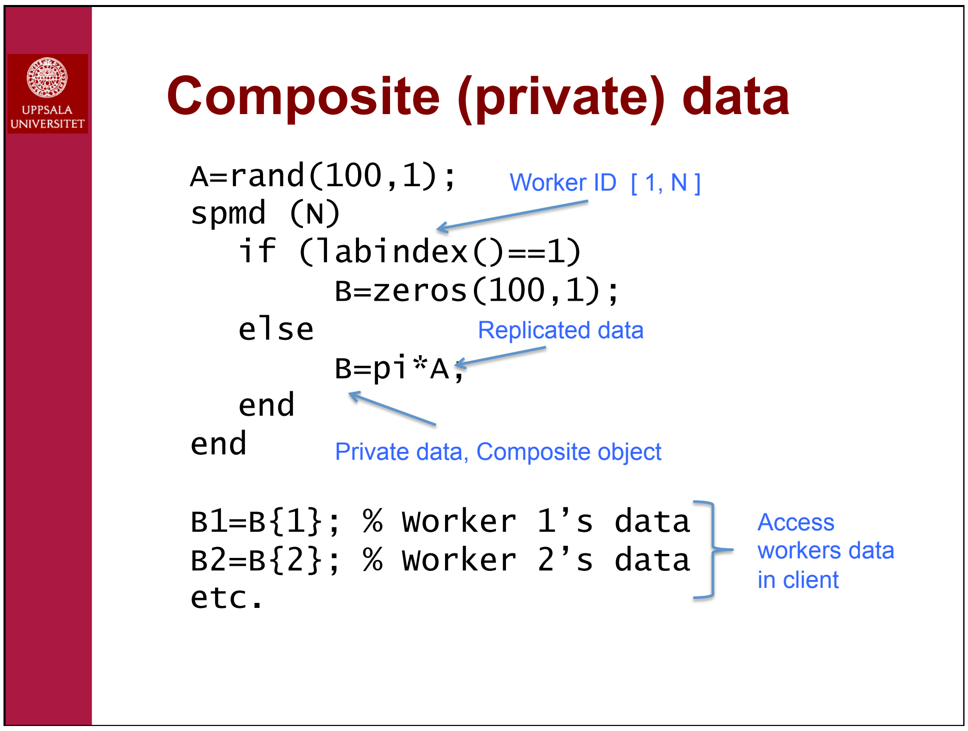
Single Program Multiple Data (SPMD)

```
% Request 4 workers (max 12)
>> matlabpool 4

% Run in parallel on 4 workers
spmd (4)
    < statements >
end

% Return workers
>> matlabpool close
```

UPPSALA UNIVERSITET

Variables in SPMD

- Each worker can read data from client defined outside spmd (replicated data).
- All data assigned inside spmd are composite (private) but can be accessed from client. Can also communicate worker-worker with explicit send-recv calls.
- Large data sets can be distributed and divided over workers (distributed array).



Composite (private) data

```
A=rand(100,1);
spmd (N)
   if (labindex()==1)
         B=zeros(100,1);
   else
         B=pi*A;
   end
end
```

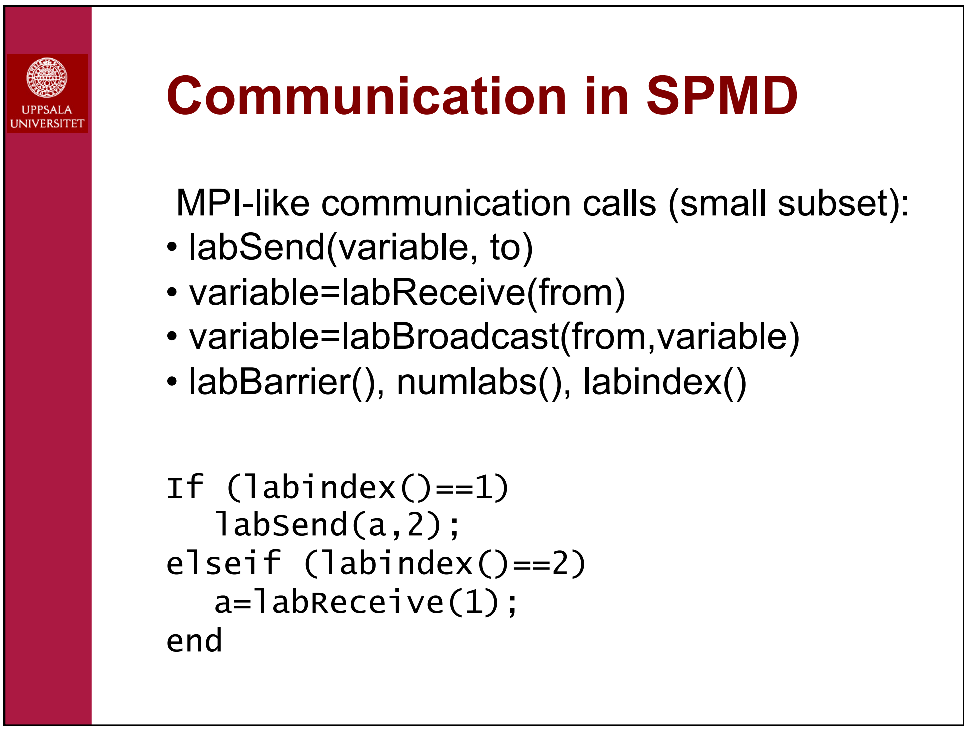
Worker ID [1, N]

Replicated data

Private data, Composite object

```
B1=B{1}; % Worker 1's data
B2=B{2}; % Worker 2's data
etc.
```

Access workers data in client



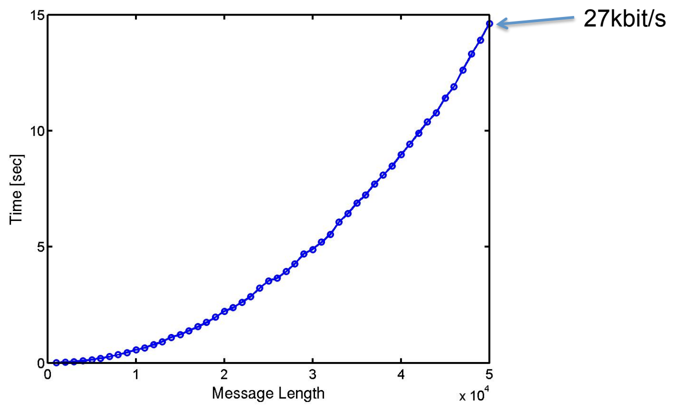
Communication in SPMD

MPI-like communication calls (small subset):

- labSend(variable, to)
- variable=labReceive(from)
- variable=labBroadcast(from,variable)
- labBarrier(), numlabs(), labindex()

```
If (labindex()==1)
   labSend(a,2);
elseif (labindex()==2)
   a=labReceive(1);
end
```

But the communication is extremely slow and grows non-linearly with message length!



Pingpong-test between two workers.

Distributed data

```
len=1e7;
Adist=distributed.rand(1.len);
B=distributed.zeros(1,len);
spmd
   for i=drange(1:len)
         B(i)=pi*Adist(i);
   end
end
Bglob=gather(B); Blocal=gather(B,lab)
```

Partitions of A and B, private and local data

Full size, all elements collected to client

The distributed arrays are split into different partitions (private data) and assigned to the different workers. The function *drange* picks out each partitions iteration indexes.

Distributed data

User can also define partitions by using distribution objects, **codistributor1d** and **codistributor2dbc**.

Ex: Distribute A into 4 partions of sizes 10, 10, 15 and 15 in the first dimension (index).

```
A=zeros(50,100);
dim=1; part=[10 10 15 15];
spmd 4
   dist1D=codistributor1d(dim,part);
   Adist=codistributed(A,dist1D);
   for i=drange(1:50)
      Adist(i,:)=…
   end
end
```

Performance in SPMD

Use composite data (private) and replicated arrays. Reading and writing distributed arrays takes very long time! (But, allows to solve larger problem that would not fit into one processors memory if run on a cluster.)

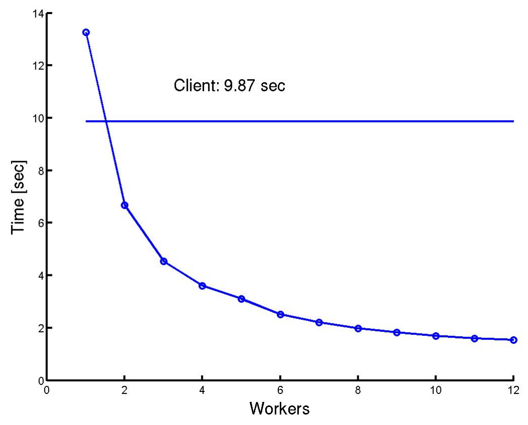
Note, workers can not access neighbour data!

```
for i=drange(1:100)
      B(i)=pi*Adist(i+1);
end
```

Not allowed => Communicate explicitly with labSend and labReceive.

Also, restrict the communication to small data sets, the communication time grows quickly with message length. Use functions for parallel code!!!

```
function [resultarr]=enumspmd(indata,nsize,workers)

spmd (workers)
   outdata=zeros(nsize,1);
   slice=nsize/numlabs();
   i1=(labindex()-1)*slice+1;
   i2=i1-1+slice;
   for i=i1:i2
      rank=1;
       for j=1:nsize
         if (indata(j)<indata(i))
            rank=rank+1;
          end
       end
       outdata(rank)=indata(i);
    end
end

resultarr=outdata{1};
for i=2:workers
   resultarr=max(resultarr,outdata{i});
end
```



Enumeration sort using SPMD, some extra overhead in communication client-worker and reduction of distributed outdata-array.

Task parallelism

Can create independent tasks (defined as Matlab functions) and schedule them to available workers (cores).

Can define arbitrary number of tasks (not limited to 12) and let the system schedule and load balance the work.

Note, we use functions for tasks. Then all data are local and private in the workers.

```
% Create scheduler
sched=findResource('scheduler','type','local');
joblist=createJob(sched); % Create a job queue

% Insert tasks to the queue
task1=createTask(joblist,@matmul,1,{A B});
task2=createTask(joblist,@matmul,1,{A2 B2});
task3=createTask(joblist,@matinv,1,{A});
submit(joblist); % Submit the job

% Wait for task2
waitForState(task2);
Res=get(task2,'OutputArguments');
C2=Res{1};

% Wait for all tasks
waitForState(joblist);
results=getAllOutputArguments(joblist);
C1=results{1}; C3=results{3};

destroy(joblist); % Destroy the job queue
```

Performance with Tasks

Starting workers and scheduling tasks to workers is **EXTREMLY** slow, taking several minutes.

⇒ Each task needs to take at least 10's of minutes or hours to execute to get any parallel performance!

MathWorks answer: Use MATLAB Distributed Computing Server (MDCS), the local scheduler in parallel toolbox was at first developed to allow you to quickly locally test your code before running it in *(1) large quantities* with *(2) large amounts of data* on a *(3) MDCS cluster*.

GPU Acceleration

```
% Establish data on GPU
>> A=rand(100,100);
>> Agpu=gpuArray(A);
>> bgpu=gpuArray.ones(100,1);

% Compute on GPU, mldivide
>> xgpu=Agpu\bgpu;

% Gather data from GPU
>> x=gather(xgpu);
```

Built in functions on GPU

abs	complex	filter	ipermute	mldivide	sec
acos	cond	filter2	iscolumn	mod	sech
acosh	conj	find	isempty	mpower	shiftdim
acot	conv	fft	isequal	mrdivide	sign
acoth	conv2	fft2	isequaln	mtimes	sin
acsc	convn	fftn	isfinite	NaN	single
acsch	cos	fftshift	isfloat	ndgrid	sinh
all	cosh	fix	isinf	ndims	size
angle	cot	flip	isinteger	ne	sort
any	coth	fliplr	islogical	nnz	sprintf
arrayfun	cov	flipud	ismatrix	norm	sqrt
asec	cross	floor	ismember	normest	squeeze
asech	csc	fprintf	isnan	not	std
asin	csch	full	isnumeric	num2str	sub2ind
asinh	ctranspose	gamma	isreal	numel	subsasgn
atan	cumprod	gammaln	isrow	ones	subsindex
atan2	cumsum	gather	issorted	pagefun	subsref
atanh	det	ge	issparse	perms	sum
beta	diag	gt	isvector	permute	svd
betaln	diff	horzcat	kron	plot (and related)	tan
bitand	disp	hypot	ldivide	plus	tanh
bitcmp	display	ifft	le	pow2	times
bitget	dot	ifft2	length	power	trace
bitor	double	ifftn	log	prod	transpose
bitset	eig	ifftshift	log10	qr	tril
bitshift	eps	imag	log1p	rank	triu
bitxor	eq	ind2sub	log2	rdivide	true
blkdiag	erf	inf	logical	real	uint16
bsxfun	erfc	int16	lt	reallog	uint32
cast	erfcinv	int2str	lu	realpow	uint64
cat	erfcx	int32	mat2str	realsqrt	uint8
ceil	erfinv	int64	max	rem	uminus
chol	exp	int8	mean	repmat	uplus
circshift	expm1	interp1	meshgrid	reshape	var
classUnderlying	eye	interp2	min	rot90	vertcat
colon	false	inv	minus	round	zeros

User functions on GPU

% Apply function to each element of array on GPU

```
>> ygpu=arrayfun(myfun,xgpu);
```

(The first time you call arrayfun to run a particular function on the GPU, there is some overhead time to set up the function for GPU execution. Subsequent calls of arrayfun with the same function can run significantly faster.)

% Evaluate CUDA kernel on GPU

```
>> ygpu=feval(KERN,xgpu);
```

Performance using GPU

Hardware: Nvidia GeForce GT 650M, 384 cores, 1024MB

Results: Slow down, no improvement, not even for built in functions such matrix-matrix multiplication!

Summary

- Four constructs for parallelism
  - For-loops with parfor, similar to OpenMP
  - Single Program Multiple Data, SPMD, with MPI-like communication calls
  - Task parallelism with dynamic scheduling
  - GPU acceleration

- Private memory on the workers. Can distribute and replicate data but not access other workers data without explicit communication in SPMD.

- Performance is not comparable to MPI/Pthreads/ OpenMP/CUDA, the parallel overheads are high. Only for large scale problems a significant speedup can be achieved.