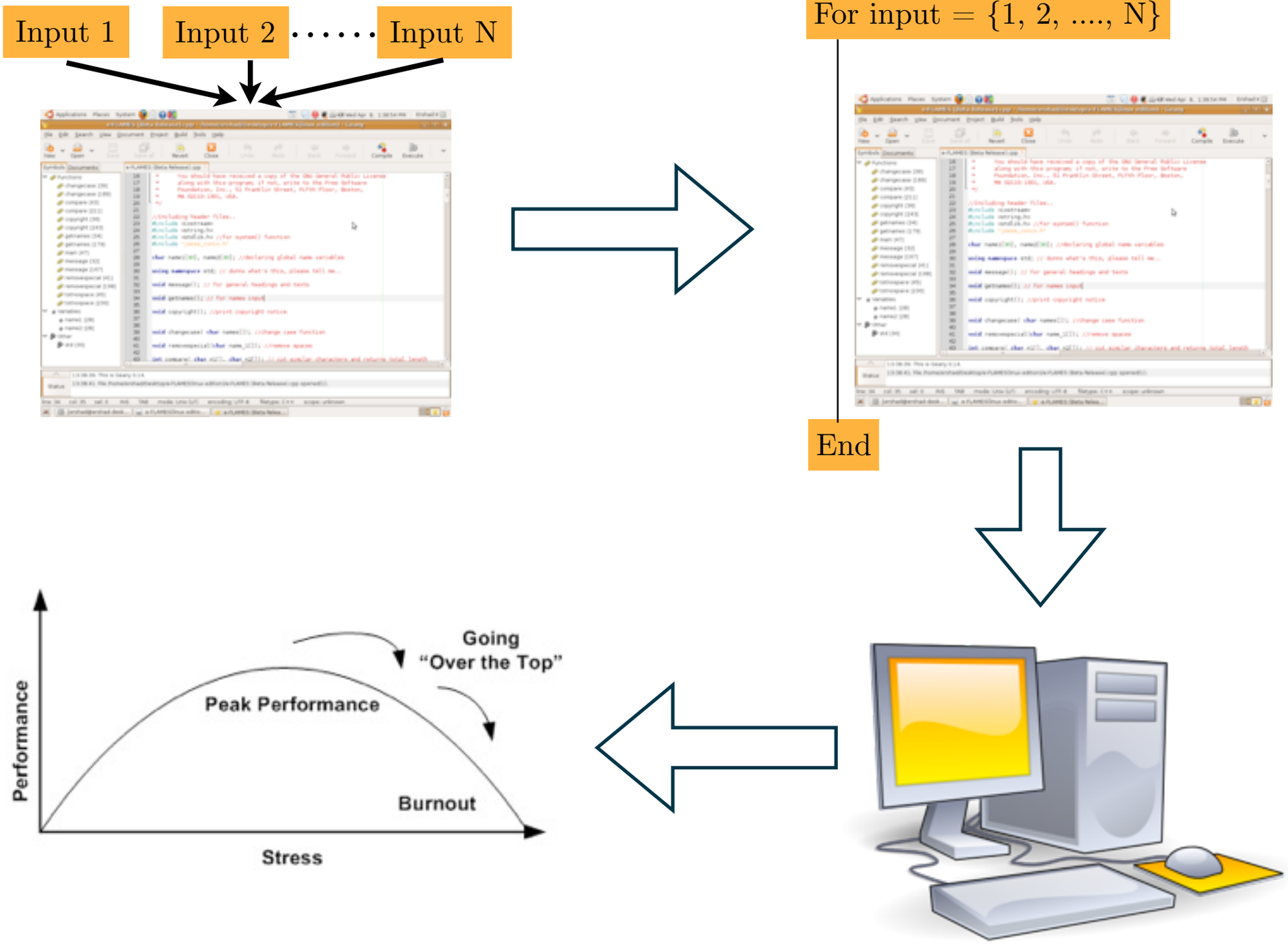
Computations using UCL's grid engine

Prasad Sudhakar
ICTEAM/ELEN, UCL
prasad.sudhakar@uclouvain.be

ICTEAM Signal Processing Seminar
29 May, 2012

Context

- Researchers perform thousands of simulations



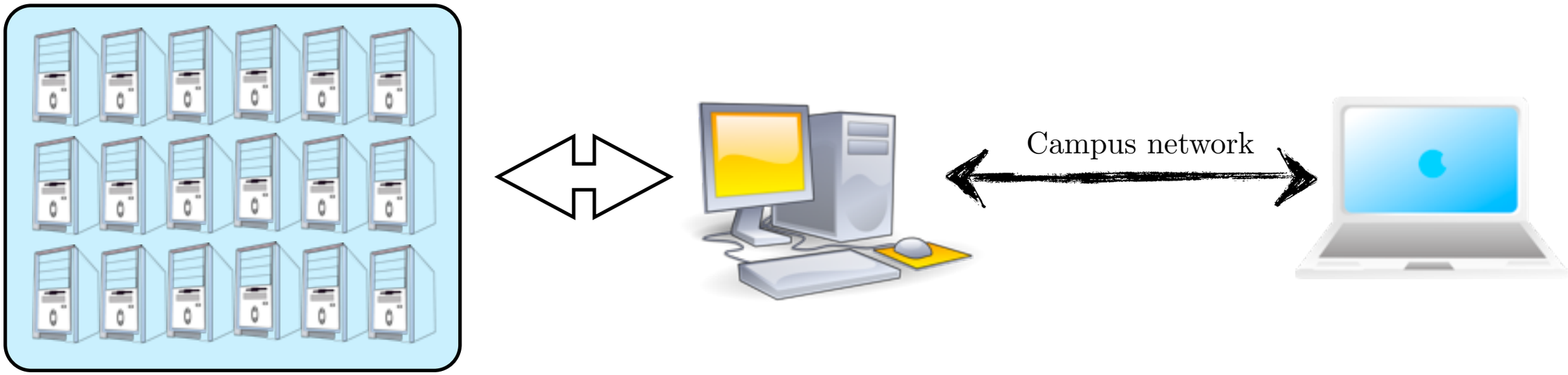
Drawbacks of standalone computing

- Lengthy time requirement
- Resource blocking
  - Hardware, software licenses, etc.
- Suboptimal
  - Insufficient resources
  - Non-native computation
- Manual intervention
  - Polling to check completion, etc.

Do we have any alternative?

Alternative - Computing on a grid

- Computational grid



Backend cluster | Interface | User computer

- Several independent CPUs
- Filesystem

- Submit and manage programs
- Access file system

Why and how do we use it?

Why should we use a cluster?

- Several independent computing units
  - Intelligent automatic job scheduler
- Each program is a complete executable
  - No need to hog licenses
  - Faster
- Minimal intervention during execution; small, one time preparation overhead

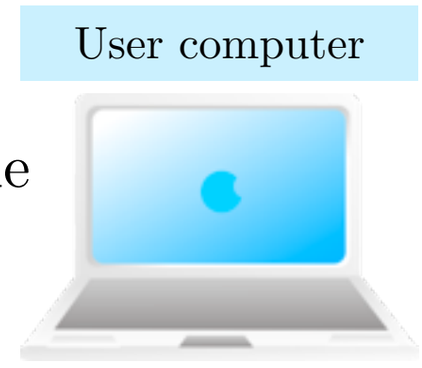
CAUTION:

It is not parallel or multithreaded programming.!

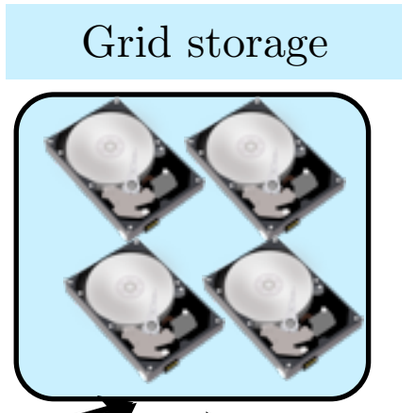
Using UCL's grid: an overview

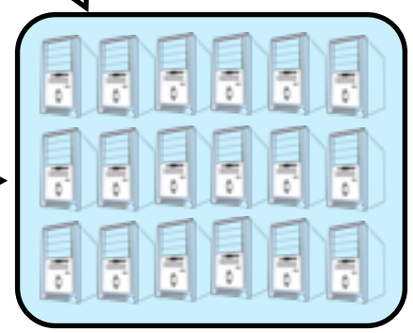
UCL's Institut de calcul intensif et de stockage de masse (CISM)

- Prepare your code



- Put your code on grid storage
- Transfer the results





- Compile the code:
  Contains most compilers
- Prepare the script
- Submit and manage jobs

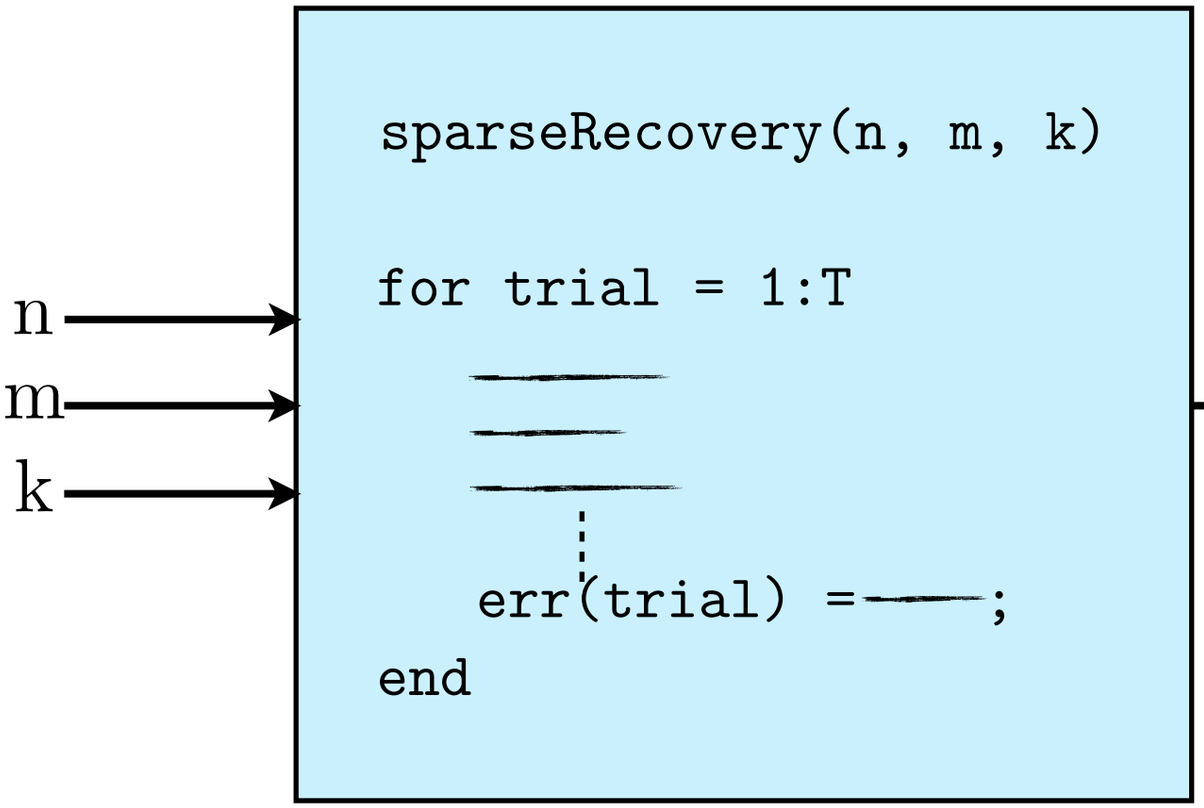
Obtaining a user account

- Details at http://www.cism.ucl.ac.be/login/
  - http://www.cism.ucl.ac.be/cilog
- /home/[group name]/[chosen login]
  - e.g. /home/elen/psudhaka
- **lemaitre, lm9**: hostname of the frontend machines
- Use SSH/SCP for communication
  - e.g. `ssh psudhaka@lemaitre.cism.ucl.ac.be`
- Requirements
  - You have to be affiliated to UCL
  - Approval of your supervisor

Acknowledge the use of CISM facility in your papers, reports, theses, etc.

Example

- Problem: Sparse vector recovery




```
function sparseRecovery(n, m, k)

TOTAL_TRIALS = 5;
sigma = 0;

logfileName = sprintf('../logs/log_n%d_m%d_k%d.txt', n, m, k);
logfile = fopen(logfileName, 'w');

fprintf(logfile, 'Executing k = %d, m = %d\n', k, m);

error = zeros(TOTAL_TRIALS, 1);

for trial = 1:TOTAL_TRIALS
    fprintf(logfile, 'Trial = %d\n', trial);


    A = randn(m, n);

    for ind = 1:n
        A(:, ind) = A(:, ind)./norm(A(:, ind));
    end

    fname = sprintf('../data/x_n%d_k%d_trial%d.mat', n, k, trial
    load(fname);

    % generating observation vector

    b = A*x;% + randn(m, 1);

    % BPDN recovery using spgl1

    options.verbosity = 0;
    [xhat] = spgl1(A, b, [], sigma, [], options);

    error(trial) = norm(x-xhat, 2)/norm(x, 2);

end

outFile = sprintf('../results/error_n%d_m%d_k%d.mat', n, m, k);
save(outFile, 'error');

fprintf(logfile, 'Finished');

fclose(logfile);
```

- n = 1024
- m = {16, 32, 64, 128, 256, 512}
- k = {2, 4, 8, 16, 32, 64, 128, 256}

Preparing your code: 1/2

- **DON'T**
  - Add explicit paths: e.g. no `addpath()` in Matlab code
    - Include path during compilation
  - Ask for user input
    - Use file input and output

```
fname = sprintf('../data/x_n%d_k%d_trial%d.mat', n, k, trial);
load(fname);
```

```
outFile = sprintf('../results/error_n%d_m%d_k%d.mat', n, m, k);
save(outFile, 'error');
```

  - Use graphics
    - Use `isdeployed()`

```
if(~isdeployed)
        figure, stem(x);
end
```

  - Use verbose for diagnostics
    - Use log files

```
logfileName = sprintf('../logs/log_n%d_m%d_k%d.txt', n, m, k);
logfile = fopen(logfileName, 'w');

fprintf(logfile, 'Executing k = %d, m = %d\n', k, m);
```

Preparing your code: 2/2

- Create a wrapper
  - that takes only one string parameter
  - calls your function with correct parameters

```
function wrapperSparseRecovery(param)

param = str2num(param);
```
Convert string to numerical value

```
n = 1024;
kArray  = 2.^(1:8);
mArray = [16, 32, 64, 128, 256, 512];
```
Store all possible input parameters in arrays

```
[kIndex, mIndex] = ind2sub([length(kArray) length(mArray)], param);
```
Split one parameter index into multiple indices

```
k = kArray(kIndex);
m = mArray(mIndex);
```
Fetch the parameters

```
sparseRecovery(n, m, k);
```
Call the function

- Move code to grid workspace
  - `scp -r [mycodedir] [login]@lm9.cism.ucl.ac.be:`

Compiling your code

- Requirements
  - Compilers: e.g. Matlab `mcc`
  - Run time libraries (RTL): e.g. Matlab MCR
- Popular compilers and RTLs are already installed on lm9
- Steps
  1. login to lm9: `ssh [login]@lm9.cism.ucl.ac.be`
  2. change working directory
  3. open matlab: `/usr/local/matlab_R2010A/bin/matlab -nojvm -nosplash -nodisplay`
  4. compile code: `mcc -m myfilename.m -R -singleCompThread -I [library_paths]`
- Produces standalone executable

Preparing the script: 1/2

- Script to submit the executable on the grid
  - loads libraries, sets up environment variables and path
- Passes the string variable to the executable
  - Task number is passed as the parameter
- Calls the executable

```
#!/bin/sh

#$ -N wrapperSparseRecovery

#$ -l h_rt=8:00:00
#$ -l h_rss=32G
#$ -l p=5520

#$ -M prasad.sudhakar@uclouvain.be
#$ -m bes

#$ -cwd

export LD_LIBRARY_PATH=/mnt/homezfs/elen/psudhaka/SPScode/src:/usr/local/matlab_R2010A/runtime/
glnxa64:/usr/local/matlab_R2010A/bin/glnxa64:/usr/local/matlab_R2010A/sys/os/glnxa64:/usr/local/
matlab_R2010A/sys/java/jre/glnxa64/jre/lib/amd64:/usr/local/matlab_R2010A/sys/java/jre/glnxa64/
jre/lib/amd64/native_threads:/usr/local/matlab_R2010A/sys/java/jre/glnxa64/jre/lib/amd64/server
export XAPPLRESDIR=/usr/local/matlab_R2010A/X11/app-defaults
export PATH=.:$PATH

wrapperSparseRecovery $SGE_TASK_ID
exit
```

Preparing the script: 2/2

- Use customised Matlab script to generate the job script
  - Available on SPS website
  - `start_script myfilename`
  - Generates myfilename.sh

- Customisation

  Matlab version used: to load appropriate libraries

  Required run time and memory

  Email id for communication

Submitting your programs

- Login to lemaitre
  - `ssh [login]@lemaitre.cism.ucl.ac.be`
- Change working directory
- Submit the jobs
  - `qsub -t 1:MAX_PARAM myprogram.sh`
    - starts jobs with unique *JOB_ID.TASK_ID;* in run state or queue state
    - creates its own log files
      - myprogram.sh.o*JOB_ID.TASK_ID*: standard output file
      - myprogram.sh.e*JOB_ID.TASK_ID*: standard error file
- `qstat`: Check the status of jobs
- `qdel [job_id]`: deletes a particular job

Then..

- Wait for results
  - Check for `qstat` to show no running jobs
- Transfer result files and plot.!

- Debugging
  - Check standard error output files

Extras

- To use third party libraries
  - if installed on lm9, include proper paths
  - if not, install in your home directory and include paths

- To access the grid outside UCL network
  - First ssh to the computer 'hall', then login to lemaitre/lm9
    - e.g. `ssh psudhaka@hall.cism.ucl.ac.be`
  - For file transfer, use SSH tunneling

Summary

1. Prepare code according to the guidelines
2. Create wrapper code
3. Transfer and compile the code on **lm9**
4. Create script for submitting jobs
5. Submit jobs on **lemaitre**
6. Wait and collect results

For more information, visit: http://www.uclouvain.be/cism

Mailing list: egs-cism@listes.uclouvain.be

Power of automation