

# Exercise 2

## 1 Amdahl's Law

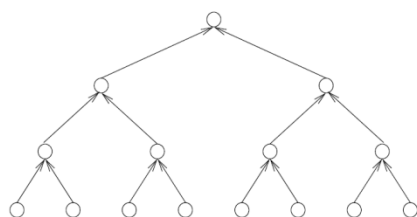
Assume a problem of size  $W$ , which has a non-parallelizable (serial) component of size  $W_{seq} = f \cdot W$ , where  $f$  is a fraction between 0 and 1.

- Express the speedup  $S$  in terms of  $f$  and  $p$ , where  $p$  is the number of processing elements.
- Prove that  $W/W_{seq}$  is an upper bound on its speedup, no matter how many processing elements are used.

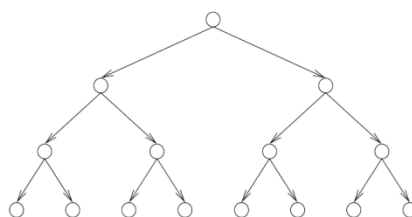
## 2 The DAG model of parallel computation

Parallel algorithms can often be represented by dependency graphs (see Figure): nodes represent tasks and directed edges represent the order in which the tasks must be performed. A node of the graph can be scheduled for execution as soon as the tasks at all the nodes that have incoming edges to that node have finished execution. Any deadlock-free dependency graph must be a directed, acyclic graph (DAG).

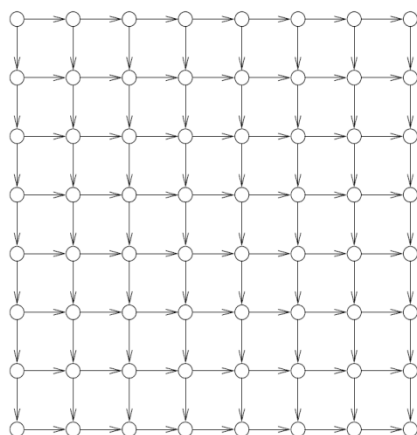
Let  $n$  be the number of nodes in each dependency graph.



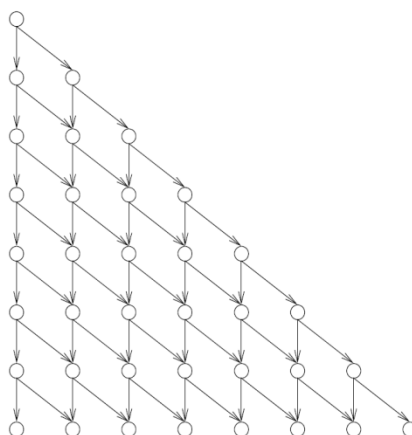
(a)



(b)



(c)



(d)

Compute for all four dependency graphs:

- the critical path length  $h$  (measured in nodes);
- the degree of concurrency  $C(W)$  (expressed in  $h$ );
- the maximum possible speedup if an unlimited number of processing elements is available;
- the values of speedup, efficiency, and the overhead if the number of processing elements  $p$  is equal to the half of  $C(W)$  (you can assume  $2p = C(W)$ ).

### 3 Adding Numbers

Let  $A$  be the efficient algorithm for adding  $n$  numbers on  $p$  processing elements introduced in “(3) Performance Metrics for Parallel Systems”.

- Assume that it takes 10 time units to communicate a number between two processing elements, and that it takes 1 unit of time to add two numbers. Express the parallel runtime of  $A$  in  $n$  and  $p$ .
- Express the standard speedup in  $n$  and  $p$  and plot the speedup curve for the problem size  $n = 5'000$  and  $p \in 2^i$ ,  $0 \leq i \leq 10$ .

**Scaled speedup** is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements; that is, if  $W$  is chosen as a base problem size for a single processing element, then the *Scaled speedup*  $= \frac{T_S(pW)}{T_P(pW, p)}$ .

- Now plot the scaled speedup curve for the same set of values of  $p$  and compare it with the standard speedup curve.

### 4 Image Processing on GPU

In Exercise 1 you optimized and parallelized an image processing routine for edge detection. Now, the goal is to implement the same routine for a GPU and to see what speedup is possible.

Before you start, please make sure that OpenCL version 1.1 or higher is supported on your GPU, and the necessary SDKs are installed. NVIDIA, AMD, and Intel support OpenCL on their GPUs. Depending on the GPU manufacturer you need to download and install a specific SDK:

- NVIDIA                      CUDA Toolkit                      <http://developer.nvidia.com/cuda-downloads>  
<http://developer.nvidia.com/opencv>
- Intel                              SDK for OpenCL                      <http://software.intel.com/en-us/vcsources/tools/opencv-sdk>

OpenCL device code (file extension \*.cl) can be built at host runtime or prebuilt during host code compilation time and read from file during host runtime. The former approach makes your heterogeneous code device independent, while the latter approach simplifies device code programming and syntax checking. For the latter approach you also need to install an OpenCL runtime for your CPU:

- Intel                              Runtime for the CPU                      [OpenCL™ Runtimes for Intel® Processors](http://software.intel.com/en-us/vcsources/tools/opencv-sdk)

Following OpenCL related source code files are given:

cl.hpp	OpenCL 1.2 C++ wrapper
main.h	header file imported in ocl.cpp
main.cpp	main routine and tests
ocl.cpp	OpenCL host code
edges.cl	OpenCL device code
Reame.txt	OpenCL installation guide

- Study the given source code and the installation guide carefully. Make sure that you can build and run it on your computer.
- Complete the edge detection procedure `edges(...)` in OpenCL C-syntax in “edges.cl” and check the output. Why does it not produce the identical result as the parallel CPU implementation? What are the differences? What is the speedup compared to the parallel CPU implementation?  
The OpenCL SDK contains an OpenCL syntax checker, pre-compiler, and debugger. To invoke the debugger you have to build the kernel with additional parameters and then to start the debugger on the OpenCL kernel on your CPU. The additional parameters are:  
`program.build(devices, “-g -s \”full-path\kernel-file-name\””)`