Chair for Computer Science 12 - High Performance Computing ***www.hpc.rwth-aachen.de***
Schwitanski, Terboven, Müller ***contact@hpc.rwth-aachen.de***

Introduction to High-Performance Computing, Winter 2019

Exercise 3: Parallel Computers and Networks

Problem 1. Scaling behavior
Weather forecast simulation is a typical use case for parallel processing. The area of interest (*space*) is typically divided into discrete parts, that map to a 2D or 3D grid. The simulation time is also divided into discrete time intervals, called *time steps.* For each time step the change of parameters for each grid element is calculated. Higher resolution in time and in space give better results. A simple pseudocode framework is given in Listing 3.1. For further considerations assume that the compute time for each time step is equal and each time step has equal sequential calculation time (*update_grid*).

```
for t in TIMESTEPS:
    for x in XDIM:
        for y in YDIM:
            calculate_change_in(x,y)
    update_grid
```

Listing 3.1: Pseudocode of a weather simulation

Problem 1.1. Strong scaling
Given is a parallel weather code, that is able to calculate a forecast in 12 hours with four processes. It is further known that 10% of the parallel runtime with four processes is not parallelizable.

a) Calculate the compute time with 16 parallel processes according to Amdahl.

b) How long would a serial execution of the computation take? What is the sequential portion of the code?

c) How many processes are needed to get a calculation time of 2 hours? What is the lower bound of parallel calculation time?

d) Plot the speedup of this code for up to 100 processes (e.g., with gnuplot or Excel)

- with serial execution time as baseline,
- with the initial configuration of four processes as baseline.

Problem 1.2. Weak scaling
Which parameter of a weather simulation could you adjust to achieve a constant parallel computation time ($T(n) = \text{const}$) with increasing parallel computation capability n?

a) Time resolution

b) Spatial resolution

c) Size of the region

Problem 2. Cache coherence
Cache coherence is an important property when working with multiple processors (or cores): Since every processor has typically local caches, local modifications of cache lines have to be reflected in main memory and also in the caches of the other processors (in case the same cache line is also loaded on other processors).
Cache coherence protocols ensure that the view on cache lines is consistent between processors. They are implemented in hardware, so they are invisible to the user. A basic protocol is the *MSI protocol* with the state diagram depicted in Figure 3.1.
According to the MSI protocol, a cache line can be in three different states:

- **M**odified: Cache line has been modified and resides inside the cache. It resides in no other cache. When it is evicted, it has to be written to memory to ensure consistency.
- **S**hared: Cache line has recently been read from memory but not yet modified. There might exist other copies of this cache line in one or more of the other caches.
- **I**nvalid: This cache line was invalidated recently and is unused. This state may occur if the cache line was in shared state and one processor requested exclusive ownership.

In Figure 3.1, there are two kinds of transactions: Processor initiated transactions are solid black, bus initiated transactions are dashed red. The notation A / B means that event A (on processor- or bus-side) leads to generation of action B.

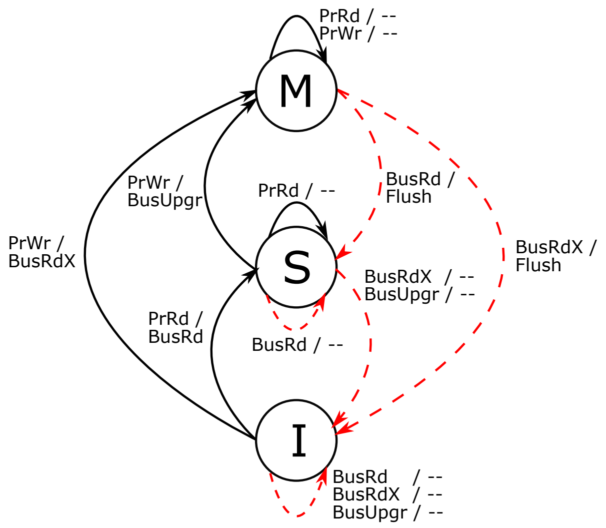


Figure 3.1: MSI protocol state machine

The following *processor requests* are possible:

- PrRd: Request to read a cache line
- PrWr: Request to write a cache line

The following *bus requests* are possible:

- BusRd: Request copy of cache line that is **not** intended to be modified (PrRd).
- BusRdX: Request **exclusive** copy of cache line that is intended to be modified (PrWr). This invalidates other copies of the cache line.

- BusUpgr: Request that invalidates the cache lines of the other processors. If a processors makes a PrWr request on a cache line that is shared, then the cache line has to be invalidated on the other processors (same as for BusRdX).
- Flush: Entire cache line flushed onto the bus (e.g. to serve cache line to other processor)

The following table shows an example of how the state of a *single* cache line changes when three processors (P1, P2, P3) access it using the MSI protocol state machine.
Notation: “R1” means “P1 requests to read the cache line (= PrRd)”, “W1” means “P1 requests to write the cache line (= PrWr)”. The columns P1, P2, P3 represent the state of the cache line on the given processor in the given time step.

t	Local Request	P1	P2	P3	Gen. Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Memory
2	W1	M	–	–	BusUpgr	–
3	R3	S	–	S	BusRd	P1’s cache (flush)
4	W3	I	–	M	BusUpgr	–
5	R3	I	–	M	–	–
6	R2	I	S	S	BusRd	P3’s cache (flush)

Problem 2.1. Applying the MSI protocol
Complete the following table using the MSI protocol state machine:

t	Local Request	P1	P2	P3	Gen. Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	W2	–	M	–	BusRdX	Memory
2	R2					
3	R1					
4	W2					
5	R1					
6	R3					
7	W1					
8	R1					
9	W2					
10	R3					

Problem 2.2. Adaptation of the MSI protocol
As you can see in the table with the MSI protocol example, P1 sends a BusUpgr request in $t = 2$ when it performs a write access on the cache line. Since the cache line is at that point in time not present in any other processor, this bus request is actually not required. How could you adjust the state machine such that no BusUpgr request is sent in this case? Describe the adaptation in a few sentences.

Problem 3. Key figures of networks

Problem 3.1. Fully connected networks

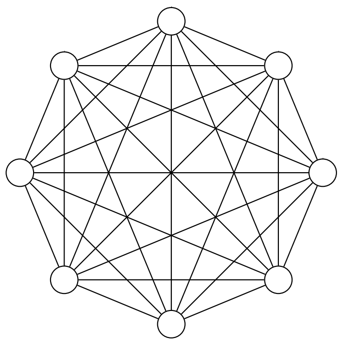


Figure 3.2: Example of a fully connected network with $N = 8$

a) Calculate the number of edges in a fully connected network with N nodes.

b) What is the edge connectivity in a fully connected network?

c) What is the diameter in a fully connected network?

d) Calculate the bisection bandwidth for a fully connected network. Assume that the bandwidth B of each link is 1.

   Hint: What kind of networks remain after bisecting the fully connected network? Bisection bandwidth is edge count of the total network minus edge count of both half-networks. Distinguish between N being even and odd in your calculations.

Problem 3.2. Modeling collective communication with LogP
A common global communication operation in parallel programs is the reduction-to-all. P processes have a vector of size n; they reduce this vector to a single common vector of size n with a given binary operation (e.g., $+(.,.)$ or $max(.,.)$).
Example: If there are four processes P_0, P_1, P_2, P_3 with vectors $v_{P_0} = [5,3]$, $v_{P_1} = [3,5]$, $v_{P_2} = [5,0]$, $v_{P_3} = [0,1]$ and a binary operation $+(.,.)$, then the reduced vector is $v_r = [13,9]$ and should be available at all processes afterwards (see also Figure 3.3).
There are several common communication patterns to implement the reduction-to-all functionality:

- The cyclic reduction; each process gets the current result from one neighbor and forwards the own result to the next. After P-1 steps each process has the reduced result.

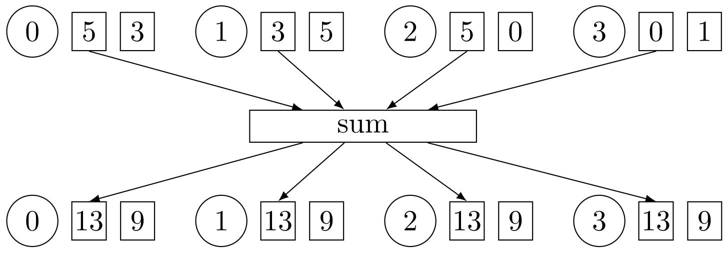


Figure 3.3: Example of the reduction-to-all operation: + with $P = 4$ and $n = 2$

- The reduce-and-broadcast; the vector is reduced to a single process and broadcasted afterwards. There are two different tree layouts for reduction and broadcast available, namely *binary tree* and *binomial tree.*

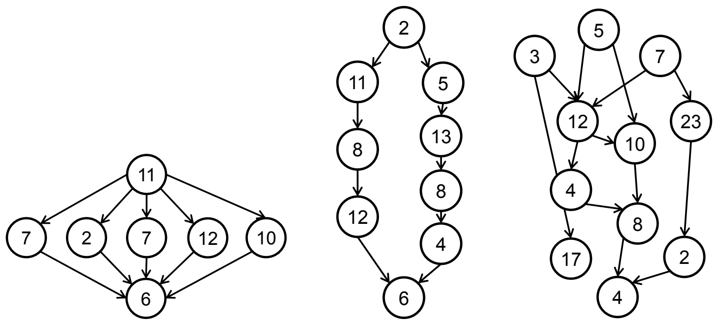
Given is the computation time $T_c(n) = 1$, the network transmission time $T_n(n) = 1$ for a vector of size n, and the number of processes $P = 8$.

a) Illustrate all three communication patterns in a graphical way.

b) Sketch a LogP model diagram for the *cyclic reduction* and the *reduction and broadcast* for the binary tree.
Hint: Do not explicitly draw arrows for the L, g, and o phases. Instead, merge these phases into a single arrow for each network transmission (T_n).

c) Compare the total runtime. What potential effect would have streaming (communicate calculated values in large without CPU being involved) for large vectors?

Problem 4. Task dependency graphs
Given are the following task dependency graphs:



Determine the total work, critical path length and average concurrency of all task dependency graphs.
Note: Have a look at the slides "Parallelization strategies" (slides 18-19) for these metrics.

Discussion of this exercise on November 29th, 2019.