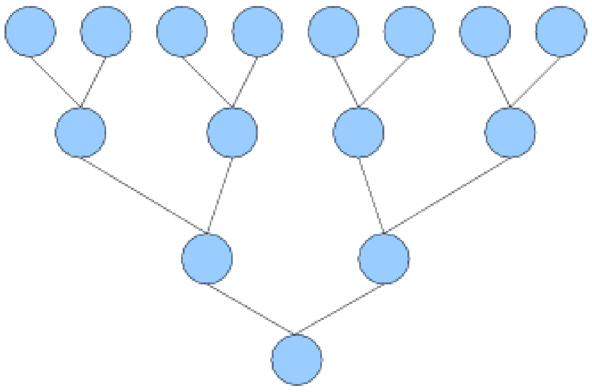
Technische Universität München
Institut für Informatik
Prof. Dr. Thomas Huckle
Michael Lieb, M. Sc.

WT 2013/2014

Parallel Numerics

Exercise 3: Vector–Vector Operations & P2P Communication II

1) Data Dependency Graphs & DAGs



"A *directed acyclic graph* (DAG) is a directed graph that has no positive cycles, that is, no cycles consisting exclusively of forwards arcs. A DAG can be used to represent a parallel algorithm, as we proceed to show. [...] Each node represents an operation performed by an algorithm, and the arcs are used to represent data dependencies. In particular, an arc $(i, j) \in A$ indicates that the operation corresponding to node j uses the results of the operation corresponding to node i. An operation could be elementary (e.g., an arithmetic or a binary Boolean operation [...]), or it could be a high–level operation like the execution of a subroutine."

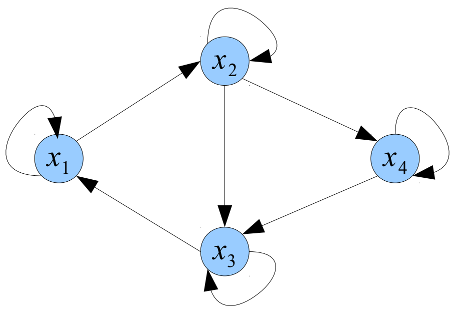
Bertsekas, Tsitsiklis: Parallel and Distributed Computation, Prentice–Hall International Editions, 1989

i) What is the difference between the data dependency graph and the direct acyclic graphs used to illustrate the program execution? How can one identify an iterative algorithm if only the data dependency graph is given?
DDG: Vertices represent data/results. Edge represents data dependency. Identifies parallel computations. Does not allow a parallel execution directly. Might contain cycles. Often does not take into account computer architecture (See ii))

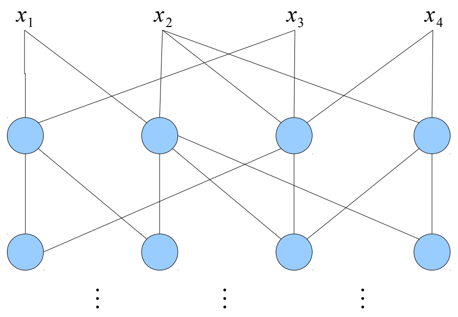
DAG: Represents actual execution. Vertices and edges have same semantics. No cycles. Infinite depth to represent iterative algorithms. Used in compiler construction for SIMD and VLIW.

Example 1:

$$\begin{aligned} x_1^{t+1} &= f_1(x_1^t, x_3^t) \\ x_2^{t+1} &= f_2(x_1^t, x_2^t) \\ x_3^{t+1} &= f_3(x_2^t, x_3^t, x_4^t) \\ x_4^{t+1} &= f_4(x_2^t, x_4^t) \end{aligned}$$



(a) *DDG for Example 1. Contains cycles for iterative algorithms.*



(b) *DAG for Example 1. No cycles* $\rightarrow$ ∞ *depth for iterative algorithms.*

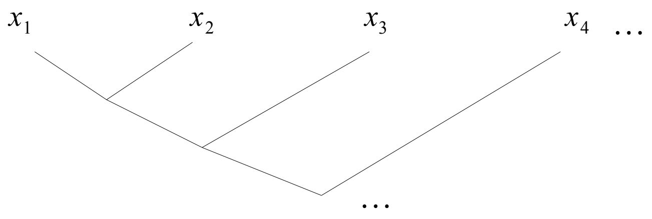
If the DDG contains any cycle the underlying algorithm is iterative. It holds:

$$\textit{Iterative Algorithm} \Leftrightarrow \textit{cycles in DDG} \Leftrightarrow \textit{depth}(DAG) = \infty.$$

ii) We want to sum up 2^k values. Give a scetch/evaluation scheme for this problem and a sequential algorithm (pseudocode). What is the minimal number of execution steps to be done? Assume that the cpu supports only unary and binary operations.

Serial evaluation scheme: We concider $f = \sum_{i=1}^{2^k} x_i = (\ldots((x_1 + x_2) + x_3) + x_4 + \ldots$

DDG:



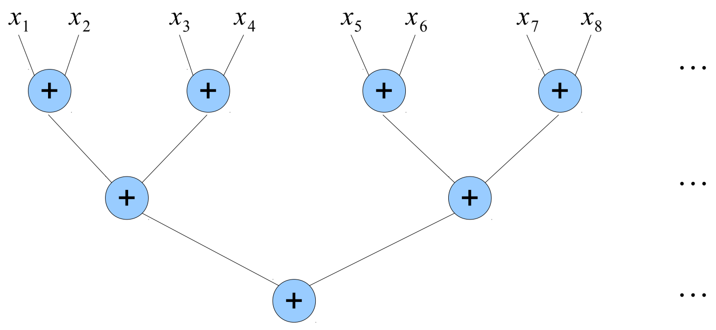
Sequential algorithm:

```
for i=1..$2^k$
    sum += $x_i$;
```

There have to be done $2^k - 1$ *execution steps. Note that there are* 2^k *different schemes (different algorithmic realizations) possible.*

iii) Give an optimal parallel algorithm (not in pseudocode, just thoughts) for the problem from exercise *ii)* using a butterfly scheme (see the illustration above). Hereby, in every step node 0 sums up the first two entries, node 1 sums up the values three and four and so on. What is the minimal number of time steps required? What is the maximal number of processors one can use for such a problem? What happens if more processors are used?

Butterfly scheme:



The tree depth is $\log_2(2^k) = k$. *The minimal number of time steps is* k. *The maximal number of processors is* 2^{k-1}. *More processors remain idle.*

2) BLAS — Basic Linear Algebra Subroutines

Make yourself familiar with the different BLAS levels. Set up a table having the columns blas level, description, complexity and example. Why are BLAS routines preinstalled on every computer and, usually, not linked directly (static) into the programs using them?

blas level	description	complexity	example	
1	...	...	$\alpha \cdot a$	$\alpha \in \mathbb{R}, a \in \mathbb{R}^d$
	...	...	$\alpha \cdot \beta$	$\beta \in \mathbb{R}$
2	...	...	...	
...				

blas level	description	complexity	example
1	Vector-vector operations	$\mathcal{O}(n)$	$\alpha x + y,\ \alpha \in \mathbb{R},\ x, y \in \mathbb{R}^n$
	Scalar-vector operations	$\mathcal{O}(n)$	
2	Matrix-vector operations	$\mathcal{O}(n^2)$	$\alpha Ax + \beta y,\ A \in \mathbb{R}^{n\times n},\ \beta \in \mathbb{R}$
3	Matrix-matrix operations	$\mathcal{O}(n^3)$	$\alpha AB + \beta C,\ B, C \in \mathbb{R}^{n\times n}$

BLAS routines are preinstalled and, usually, not linked directly because similar idea as MPI: specify syntax and semantics and the vendors provide optimized implementations.

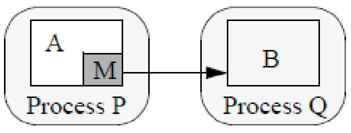
3) P2P Communication II: Blocking vs Non-blocking

i) In MPI many descriptions use the term *buffer*. Make yourself familiar with this term and explain the difference between the following types of buffers: Program variables,

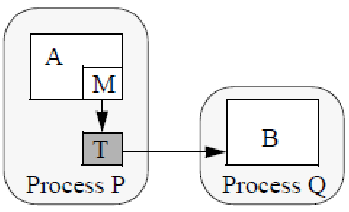
user-level buffer (assigned to MPI) and hardware/system buffer.
Program variables: Used by MPI calls, e.g.

```
int a[10]; MPI_Send(&a,...)
```

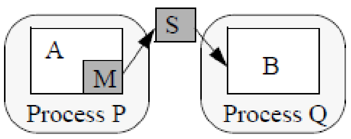
User-level buffer assigned to MPI: Allocated for MPI via application, e.g. if buffered send (`Bsend`) is used. User knows size of temporary buffer and is able to split message into pieces.
Hardware/system buffer: Message is copied to a temporary system buffer. Problems may occur, e.g. system storage not accessible by user, message too large for buffer, additional overhead.



User buffer only



Usage of a user-level buffer T



Usage of a system buffer S

ii) MPI offers a blocking and a non-blocking send and receive command. What does this mean with respect to a variable used as buffer. Make yourself familiar with the semantics of the blocking and non–blocking P2P operations.
Blocking send is performed immediately without waiting for the corresponding receive. Does not return until the message has been savely stored away, i.e. the sender can reuse the buffer. It may or may not block.
A non-blocking send returns immediately after system notifies that sending has been started. Therefore, the buffer maybe still filled with the message → requires send-complete test (`MPI_Wait()`, `MPI_Test()`). Less possibility of deadlocking code. You can be sure it will not block.
Blocking receive does not return until the message has been stored in receive buffer.

iii) What is the advantage of combining blocking and non-blocking operations?
Advantage: Overlapping communication and computation. For instance: send buffer shall be reused immediately, receive buffer not.

4) Non-blocking Operations in Action

The following question is taken from the exam of WT 2005/06. Given is the following source code running on three processors:

```
...
double     a,b;
int        myRank,noOfProcessors;
MPI_Status status;

MPI_Init(...);
MPI_Comm_size( MPI_COMM_WORLD, &noOfProcessors );
MPI_Comm_rank( MPI_COMM_WORLD, &myRank );

a = myRank+1;

if (myRank==0) {
  for (int i=1; i<noOfProcessors; i++) {
    MPI_Recv(&b,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&status);
    a+=b;
  }
}
else {
  MPI_Send(&a,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
...
```

i) What value is stored within the variable `a` on every processor after the source code extract has finished?

	Rank 0		Rank 1		Rank 2	
	a	b	a	b	a	b
`Header`	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
`a=myRank+1;`	1	$\perp$	2	$\perp$	3	$\perp$
`MPI_Send();`		—	2	$\perp$	3	$\perp$
`MPI_Recv(i=1);`	1	2		—		—
`a+=b;`	3	2		—		—
`MPI_Recv(i=2);`	3	3 (∗)		—		—
`a+=b;`	6	3		—		—

ii) Give a technical term for the type of communication implemented by this source code fragment.
Collective operation.

iii) What may happen if one uses non–blocking communication within the source code not adding additional statements?

`MPI_Irecv()` *might terminate although message (value) is not stored in* **`b`***, e.g. in* $(*)$ *old value is added, i.e. 2 instead of 3.*

5) Vector–Vector Operations

In Scientific Computing vector-vector operations are frequently occurring operations. The dimension n of the vectors is often very large. Hence, parallelisation of the basic linear algebra operations can be advantageous.

In this exercise we will parallelise the adding and the dot-product of two vectors $\vec{a}$ and $\vec{b}$. The result is a vector $\vec{r}$ or a number z, respectively:

$$\vec{r} = \vec{a} + \vec{b}, \qquad z = \vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$$

Write a parallel program for adding two vectors as follows:

- process 0 allocates three arrays `a`, `b`, `r` with dimension `LENGTH` using `(float*)malloc(LENGTH*sizeof(float))`
- process 0 stores random numbers in the arrays `a`, `b` using the function `rand()`
- partition these vectors in p equal parts (p is the number of started processes), and calculate the adding of the parts by the p processes separately (process 0 has to send the parts to the other processes respectively)
- process 0 collects the parts and stores the result in array `r`

What is the speedup for calculating the sum of two vectors with `LENGTH=10.000.000` with 2,4,8 processes? Use the function `MPI_Wtime()` to determine the time spent in the loop for calculating the sum.

Write a similar program to calculate the dot-product. What ist the speedup? In this program process 0 has to add p numbers in the end. How can you parallelize this addition? Is it advantageous?
See sourcecode to corresponding tutorial on webpage.