High Performance and Parallel Computing (MA 6252)

Lecture 9

Analysis of Parallel Programs

Ulrich Rüde

Lehrstuhl für Simulation
Universität Erlangen-Nürnberg

visiting Professor at
Department of Mathematics
National University of Singapore

Contents of High Performance and Parallel Computing

- Introduction
  - Computational Science and Engineering
  - High Performance Computing
- Computer Architecture
  - memory hierarchy
  - pipeline
  - instruction level parallelism
  - multi-core systems
  - parallel clusters
- Efficient Programming
  - computational complexity
  - efficient coding
  - architecture aware programming
  - shared memory parallel programming (OpenMP)
  - distributed memory parallel programming (MPI)
  - parallel programming with Graphics Cards

Homework 4 (50%): parallel computing project

- Due by end of teaching period
- Submit by e-mail to ulrich.ruede@fau.de

1. Write a simple program to determine the value of π. The algorithm suggested here is chosen for its simplicity: The method evaluates the integral of 4/(1+x*x) between 0 and 1.

The integral is approximated by a sum of n intervals which is defined in the beginning of the program; the approximation to the integral in each interval is (1/n)*4/(1+x*x). Each process then adds up every n'th interval (x = rank/n, rank/n+size/n,...). Finally, the sums computed by each process are added together using a reduction.

Homework 4 (cont'd)

2. Implement a MPI parallel Jacobi algorithm for an one-dimensional domain. The mapping of the domain to the MPI processes is depict in figure 1. Assume that the domain is not periodic; that is, the top process (rank = size - 1) only sends and receives data from the one under it (rank = size - 2) and the bottom process (rank = 0) only sends and receives data from the one above it (rank = 1).

Figure 1: Decomposition of the domain to the ranks.

For the computation that we're going to perform on an array data structure, we'll need the adjacent values. That is, to compute a new x[i], we will need x[i+1] and x[i-1].

The outermost two of these could be a problem if they are not in the local x but are instead on the adjacent processors. To handle this difficulty, we define ghost points that we will contain the values of these adjacent points. These ghost points are updated at the boundaries of the partitions after each iteration (see figure 2).

Calculate and print out the residual after each Jacobi iteration. Verify the correctness of the solvers by comparing them to the serial case.

Homework 4 (cont'd)

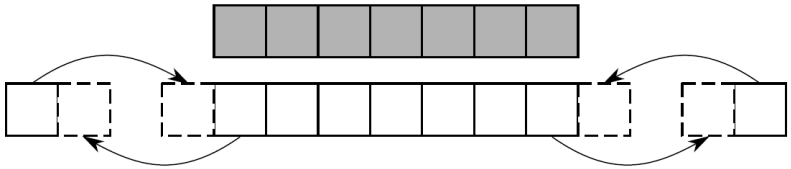


Figure 2: Ghostlayer exchange between the processes.

To avoid deadlocks, you can use the asynchronous MPI_Isend and MPI_Irecv in combination with the MPI_Waitall call. The following code snippet is a possible way to determine the required numbers of requests:

```
MPI_Request request[4];
MPI_Status status[4];
int reqcount = 0;

// check if neighbor exists then:
  MPI_Irecv(..., MPI_COMM_WORLD, &request[reqcount]);
  reqcount++;
...
// check if neighbor exists then:
  MPI_Isend(..., MPI_COMM_WORLD, &request[reqcount]);
  reqcount++;
...
MPI_Waitall(reqcount, request, status);
```

Homework 4 (cont'd)

3. Run the program with 2 and 4 processes and different number of grid points 10^k, for k = 2, 3, 4, 5, and 6. Plot the ratio of your parallel run times to the serial code run time as a function of k.

4. Describe how you would implement a parallel Gauss-Seidel method in two dimensions (i.e. which ordering would you choose and why?).

Parallelism Granularity

This corresponds to the hierarchical structure of parallel supercomputers

- Machine instructions
  - superscalar
  - pipelining
  - vector (SIMD)
- Loops
- Subroutines
- Programs

Parallel Execution Paradigms

- Corresponds to „parallel languages“
- Control driven parallelism

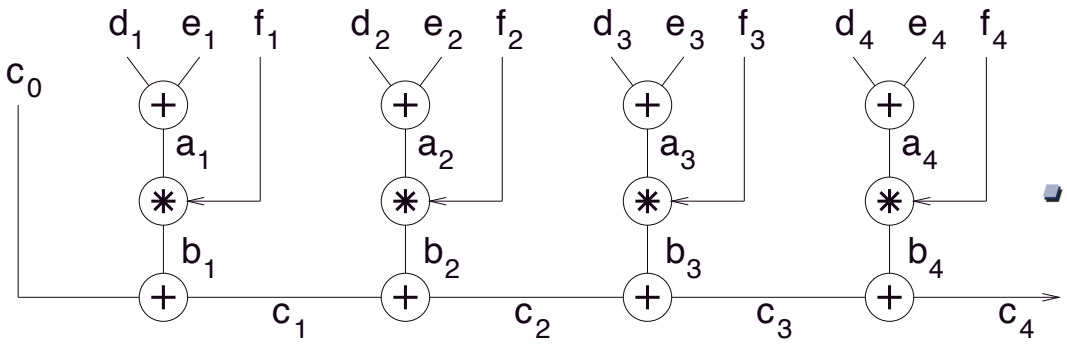
This programming model is based on conven- tional procedural/ imperative programming paradigm, which is augmented by explicit parallel constructs in the form of message passing or synchronization primitives. The basis of this model is a machine architecture with several conventional CPUs, each having its own program counter. The state of processing is characterized by the collective value of all the program counters.

Communicating Sequential Processes (CSP)

- This corresponds to MPI programming
- Special languages for CSP: e.g. Occam

Data Flow Parallelism

```
input d,e,f
c[0]=0
for (i=1; i<=4; i++) {
    a[i] = d[i] + e[i];
    b[i] = a[i] * f[i];
    c[i] = b[i] + c[i-1];
}
```

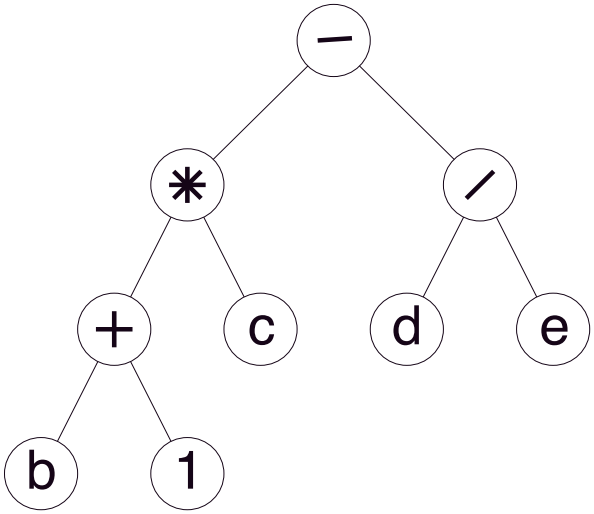


- Data is pointing to executable code which may be executed as soon as all inputs are available. This corresponds to an eager, bottom up evaluation of parallel code fragments.
- Requires an efficient technique to check whether data is available for computation. Data that is required as input for several operations must be copied.
- Disadvantage of data flow parallelism: Operations may be executed whose result is later found to be irrelevant. The typical example is the parallel execution of an if–then–else construct. Here both branches may be executed in parallel, even though the evaluation of the condition must make one of the branches irrelevant.

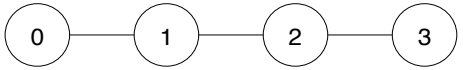
Reduction driven parallelism

Different from the data flow parallelism, he- re the operations point to their operands, which must be evaluated. The parallel execution of a simple expression leads to a tree. This programming paradigm corresponds to a functional program.

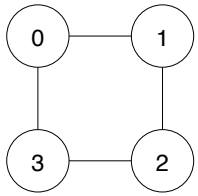
$$a := (b + 1) * c - d/e$$



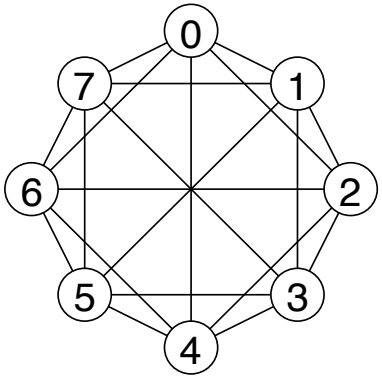
Network Topologies (1)



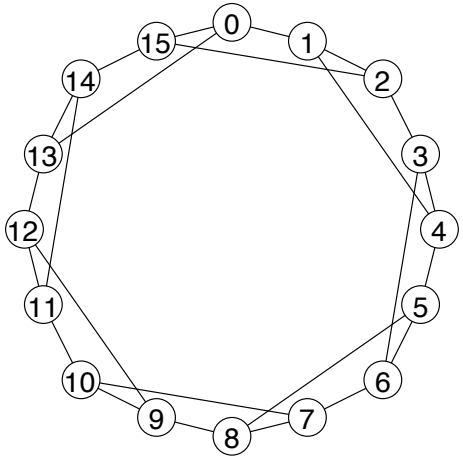
Chain



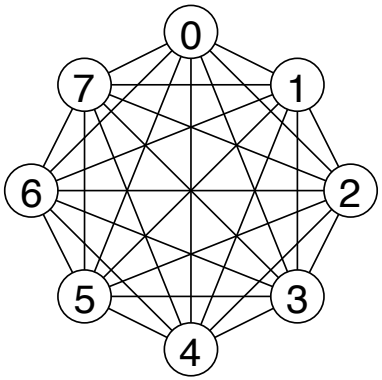
Ring



Barrel shifter

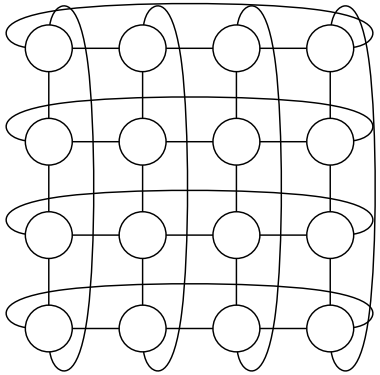


Chordal Ring
of degree 3

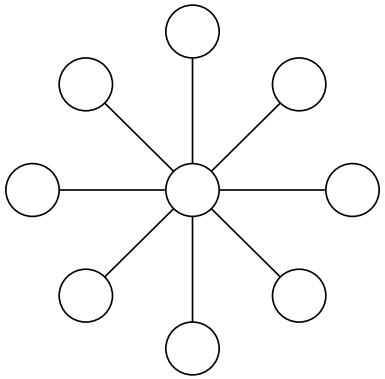


Completely
connected
graph

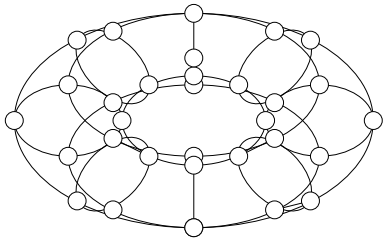
Network Topologies (2)



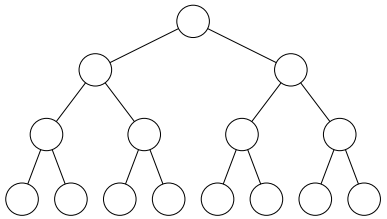
Mesh and Torus



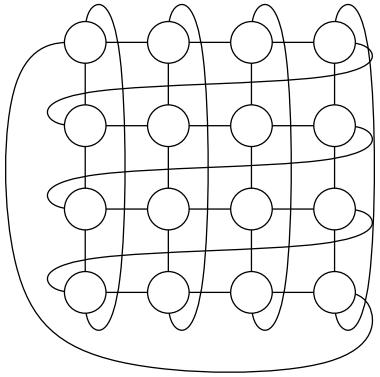
Star



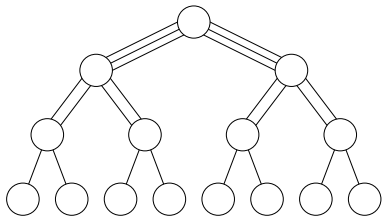
8x4 Torus



Tree

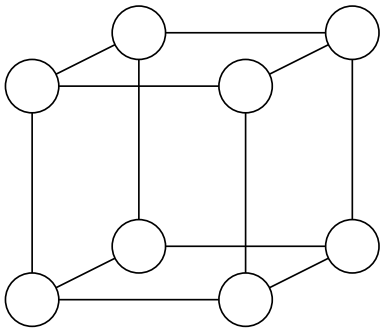


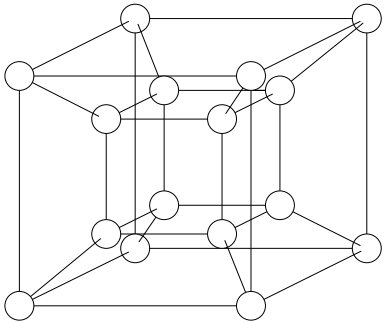
Illiac Mesh



Fat Tree

Network Topologies (3)





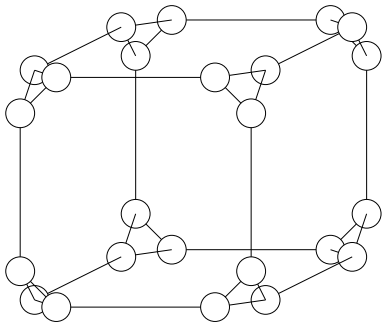
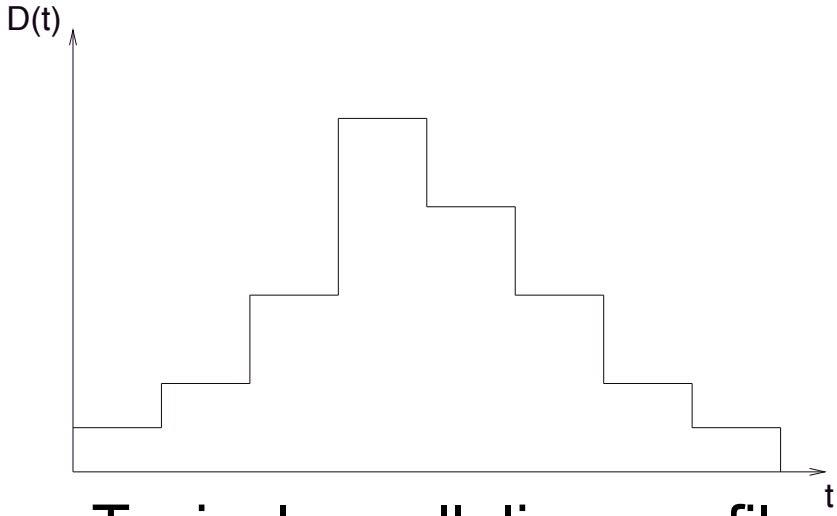


Tabelle 4.1: Network properties

Type	Degree	Diameter	Connections	Bis. width	Symmetry
lin. Kette	2	$N-1$	$N-1$	1	no
Ring	2	$\lceil N/2 \rceil$	N	2	yes
Voll. Vern.	$N-1$	1	$N(N-1)/2$	$(N/2)^2$	yes
Bin. Baum	3	$2((\log_2 N)-1)$	$N-1$	1	no
2D-Gitter	4	$2(\sqrt{N}-1)$	$2N-2\sqrt{N}$	$2\sqrt{N}$	no
2D-Torus	4	$2\lfloor \sqrt{N}/2 \rfloor$	$2N$	$2\sqrt{N}$	yes
Hypercube	$\log_2 N$	$\log_2 N$	$N \log_2 N$	$N/2$	yes

Analysis of Parallel Performance

- Assuming elementary operations that all take 1 unit of time
- A parallel program that executes $D(t)$ parallel instructions at time step t
- $D(t)$ is called the parallelism profile of a program



Typical parallelism profile for divide-and-conquer algorithm

Program run from the start time t_s to t_e needs a total of W operations

$$W = \Delta \int_{t_s}^{t_e} D(t)dt = \Delta \sum_{i=1}^{m} i \cdot t_i.$$

Δ is the computing capacity of a single processor and t_i is the time spent using i processors

Analysis of parallelism (2)

- define average parallelism as $$A = \frac{1}{t_e - t_s} \int_{t_s}^{t_e} D(t)dt.$$

Execution time $T(n)$ with n available processors for the two extreme cases of $n = 1$ and $n = \infty$ by

$$T(1) = \frac{W}{\Delta} = \sum_{i=1}^{n} \frac{W_i}{\Delta}, \quad W_i = i \cdot t_i,$$

$$T(\infty) = \sum_{i=1}^{n} \frac{W_i}{i\Delta} = \sum_{i=1}^{n} t_i,$$

Asymtotic speedup S_∞ is defined as

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{n} W_i}{\sum_{i=1}^{n} \frac{W_i}{i}}.$$

Average execution times

R_i denotes performance (say in MFlops) in the ith of m program parts. Then the *harmonic mean*, defined as

$$R_h = \frac{m}{\sum_{i=1}^{m} R_i^{-1}}$$

represents the total number of operations divided by the total execution time and is thus an appropriate measure for the average performance.

The harmonic average is the basis to define an
harmomic mean speedup for a parallel program.

Assume n processors are given
program may execute with $1, \ldots, n$ processors

$T_1 = \frac{1}{R_1} = 1$ sequential execution time on uniprocessor.

$R_i = i$ idealized performance with i processors

Resulting execution time: $T_i = \frac{1}{R_i} = \frac{1}{i}$

Example 1

Assume on an n-processor machine any number of processors is used to execute $1/n$th of the total problem. Thus $f_1 = f_2 = \ldots f_n = \frac{1}{n}$ and thus

$$S_n = \frac{1}{\sum_{i=1}^{n} \frac{1}{in}} = \frac{n}{\psi(n+1) + \gamma}$$

Sum can be represented in closed form with

$$\psi(x) := \ln(\Gamma(x)), \quad \gamma = 0.5772156\ldots$$

where $\Gamma(x)$ is the Gamma function and γ is Euler's constant. A few typical values for the speedup are:

$$S_2 = 1.33... \quad S_{32} = 7.88... \quad S_{1024} = 136.$$

These figures are not very impressive and the efficiency deteriorates quite significantly for larger processor numbers.

Example 2

Next we consider a more optimistic case. We assume that each number i of processors $i = 1, \ldots, n$ is used for a portion of the overall work which is proportional to i. Thus the time when the program operates in highly parallel mode is longer than in example 1. Thus

$$f_1 = \frac{1}{\sigma}, f_2 = \frac{2}{\sigma}, \ldots f_n = \frac{n}{\sigma}, \text{ where } \sigma = \sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

and

$$S_n = \frac{1}{\sum_{i=1}^{n} \frac{i}{i\sigma}} = \frac{\sigma}{n} = \frac{n+1}{2}.$$

This leads to a parallel efficiency tending asymptotically towards 50% for large n.

Example 3

Finally, we consider the trivial but famous case

$$f_1 = \alpha, f_2 = \ldots = f_{n-1} = 0, f_n = 1 - \alpha.$$

Here the program is executed in only two modes. A purely sequential mode is needed for a fraction of α of the total computation. In a second phase, the problem can be optimally parallelized with any number n of processors. This phase covers the remaining part of $1 - \alpha$ of the computation. This leads to a speedup

$$S_n = \frac{1}{\alpha/1 + (1-\alpha)/n} = \frac{n}{n\alpha + 1 - \alpha} = \frac{n}{\alpha(n-1) + 1}$$

Here, even if the number of processors tends to ∞, the speedup is limited by

$$\lim_{n\to\infty} S_n = \frac{1}{\alpha},$$

and the efficiency tends to 0 when the processor number is increased infinitely. This effect is known as *Amdahl's law*.

Speedup, Efficiency

Parallel program execution with the following characteristics
$T(n)$ execution time in unit time steps on an n processor system

The the *speedup* is now defined as

$$S(n) = \frac{T(1)}{T(n)} \quad (1 \leq S(n) \leq n)$$

Typically, the speedup satisfies

$$S(n) \leq n.$$

Counterexamples?

The *efficiency* is defined as

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

Strong Scaling/ Weak Sacling

- Consider efficiency depending on number of available processors
- strong scaling: when problem size fixed
  - Amdahls law will limit efficiency
  - with growing number of processors efficiency will deteriorate
  - often what one wants in practice
- weak scaling: problem size grows with number of processors
  - if the algorithm is scalable (means O(N)
  - then good efficiency for all processor numbers possible
  - not always useful in practice

A critique of efficiency and speedup

- Speedup is a relative measure
  - it tends to be better when the original (sequential program is slow) because then the parallel overhead is (relatively small)
  - looking only at speedup and efficiency favors „bad algorithms“ and „bad implementations“.
  - often good and more complex algorithms give better times to solution though their speedup/efficiency is only mediocre.
  - Parallel computing research that only quantifies speedup/efficiency is scientifically unsound.
  - All high quality parallel computing research must also quantify absolute performance metrics (such as „time to solution“).

End of Lecture

Thank you for your interest and your participation!