Chapter 4 "Performance Analysis of Parallel Programs"

according to the book
"Parallel Programming"
by T. Rauber and G. Rünger
Springer 2010

19. Februar 2013
1. Performance Evaluation of Computer Systems

2. Performance for Parallel Programs

3. Implementation of global communication operations
The response time of a program \( A \) can be split into

- the **user CPU time** of \( A \), capturing the time that the CPU spends for executing \( A \);
- the **system CPU time** of \( A \), capturing the time that the CPU spends for the execution of routines of the operating system issued by \( A \);
- the **waiting time** of \( A \), caused by waiting for the completion of I/O operations and by the execution of other programs because of time sharing.

In the following, we concentrate on the **user CPU time**.
MIPS as performance measure

A performance measure often used in practice to evaluate the performance of a computer system is the MIPS rate for a program $A$:

$$MIPS(A) = \frac{n_{\text{instr}}(A)}{T_{U \cdot CPU}(A) \cdot 10^6}. \quad (1)$$

$n_{\text{instr}}(A)$: number of instructions of program $A$

$T_{U \cdot CPU}(A)$: user CPU time of program $A$

**Modification:**

$$MIPS(A) = \frac{r_{\text{cycle}}}{CPI(A) \cdot 10^6},$$

where $r_{\text{cycle}} = 1/t_{\text{cycle}}$ is the clock rate of the processor.

$CPI(A)$: Clock cycles Per Instruction: average number of CPU cycles used for instructions of program $A$

Faster processors lead to larger MIPS rates than slower processors.
For program with scientific computations, the MFLOPS rate (Million Floating-point Operations Per Second) is sometimes used. The MFLOPS rate of a program $A$ is defined by

$$MFLOPS(A) = \frac{n_{flp\_op}(A)}{T_{U\_CPU}(A) \cdot 10^6} \text{ [1/s]},$$  (2)

$n_{flp\_op}(A)$: number of floating-point operations executed by $A$.
$T_{U\_CPU}(A)$: user CPU time of program $A$

The effective number of operations performed is used for MFLOPS: the MFLOPS rate provides a fair comparison of different program versions performing the same operations.
Different benchmark programs have been proposed for the evaluation of computer systems:

- **Synthetic benchmarks**
- **Kernel benchmarks**: small but relevant parts of real applications
- **Real application benchmarks** comprise several entire programs which reflect a workload of a standard user.
- popular benchmark suite: SPEC benchmarks (System Performance Evaluation Cooperation), see www.spec.org
- SPEC06 is the current version for desktop computers: 12 integer programs (9 written in C, 3 in C++) and 17 floating-point programs (6 written in Fortran, 3 in C, 4 in C++, and 4 in mixed C and Fortran).
Lecture outline

1. Performance Evaluation of Computer Systems

2. Performance for Parallel Programs

3. Implementation of global communication operations
The parallel runtime $T_p(n)$ of a parallel program $P$ with input size $n$ on $p$ processors is the time between the start of program $P$ and the termination of the computations of $P$ on all processors.

For computers with physically distributed memory $T_p(n)$ consists of:

- Time for local computations
- Time for data exchange with communication operations;
- Waiting time of processors e.g. because of load imbalance
- Time for synchronization of the executing processors or a subset of the executing processors

For computers with shared memory the time for data exchange is replaced by the time for the access to global data.
Costs of Parallel Programs

- The costs $C_p(n)$ of a parallel program $P$ with input size $n$ on $p$ processors is the total time that the participating processors require for the execution of $P$:

\[ C_p(n) = T_p(n) \cdot p \]

- The cost of a parallel program is a measure for all the computations performed.

- A parallel program is cost optimal if $C_p(n) = T^*(n)$ holds, where $T^*(n)$ is the runtime of the fastest sequential program; A cost optimal program requires as many computations as the fastest sequential program.

**Difficulty:** The fastest sequential program or method is possibly not known or can only be determined with a high efforts.

- The costs is often called work or Processor-Time-Product;
Speedup of a Parallel Program

The **Speedup** \( S_p(n) \) of a parallel program \( P \) with input size \( n \) on \( p \) processors is defined as:

\[
S_p(n) = \frac{T^*(n)}{T_p(n)}
\]

- \( T_p(n) = \) Parallel runtime of a parallel program \( P \) on \( p \) processors;
- \( T^*(n) = \) Runtime of an optimal sequential implementation for the solution of the problem;

The speedup is a **measure of the relative speed increase compared to the best sequential implementation**

- Typically at most **linear speedup** can be reached: \( S_p(n) \leq p \) (theoretical upper bound)
- In practice, due to cache effects a **super linear speedup** can occur.
Efficiency of a Parallel Program

- The **Efficiency** $E_p(n)$ of a parallel program $P$ with input size $n$ on $p$ processors is defined by

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

- $C_p(n) =$ Parallel program cost
- $T_p(n) =$ Parallel runtime of a parallel program $P$
- $T^*(n) =$ Runtime of the best sequential implementation

- The efficiency is a **measure** for the portion of the runtime, that is required for computations that are also present in the sequential program.

- The ideal speedup $S_p(n) = p$ is equivalent to $E_p(n) = 1$. 

Amdahl’s Law

- When the parallel implementation requires a (constant) fraction $f$, $0 \leq f \leq 1$, to be computed \textbf{sequentially}, the runtime of the parallel implementation is composed of:
  - The runtime $f \cdot T^*(n)$ of the \textbf{sequential part} and
  - The runtime of the \textbf{parallel part}, which is at least $(1 - f)/p \cdot T^*(n)$

- \textbf{Attainable speedup}

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}.$$  

- \textbf{Example: } $f = 0.2 \rightarrow S_p(n) \leq 5$ independent from the number $p$ of processors;
  → The sequential part has a big influence on the attainable speedup; for an efficient utilization of a \textbf{high number of processors, a reduction of the sequential parts} is required.
The **scalability** of a parallel program on a parallel computer is a measure for the property **to get a performance increase proportional to the number** $p$ **of processors**.

**Common observations:**

- For a fixed problem size $n$ and an increasing number of processors $p$ a **saturation of the speedup** occurs;
- For a fixed number of processors $p$ and an increasing problem size $n$, an **increase of the speedup** occurs.

**Concretization:** Scalability means, that the efficiency of a parallel program is constant when the number of processors $p$ and the problems size $n$ is increased.
Lecture outline

1. Performance Evaluation of Computer Systems

2. Performance for Parallel Programs

3. Implementation of global communication operations
   - Communication operations on static networks
   - Communication operations on a hypercube network
We consider global communication operations and their implementation on static interconnection networks (array, ring, mesh, hypercube)

How can these communication operations be efficiently implemented on these networks?
Assumptions for the analysis

The following analysis makes the following assumptions:

(a) The links of the network are **bidirectional**, i.e., messages can be sent simultaneously in both directions

(b) Each node can **simultaneously send** out messages on all outgoing links;

**organization:** use of output buffers with a separate controller for each link

(c) Each node can **simultaneously receive** messages on all incoming links;

**organization:** input buffer for each incoming link

(d) Each message consists of several bytes which are transmitted without any interruption

(e) The time for transmitting a message consists of

- the **startup time** $t_s$ (independent of the message size)
- the **transfer time** $U = n \cdot t_c$ (proportional to the length $n$ of the message)
- transmitting a single message of $n$ bytes takes time

$$T(n) = t_s + n \cdot t_c$$
The **startup time** contains

- the time to **construct** the message (inserting header with address and control information)
- the **waiting time** required if the selected output link is currently busy
- the **propagation time**, expressing the time between sending the first bit by the sender and receiving the first bit by the receiver

For most networks, the startup time is significantly larger than the time required for an arithmetic operation.

In the following, we investigate the execution of communication operations on different networks.

**Goal:** derivation of **asymptotic running times**

~~> no exact timing formulas are derived
Specializations and Generalizations

The following facts can be used:

- If a communication operation can be executed in time $x$, then a specialization of the communication operation can also be executed in time $x$.
- If a communication operation can be executed in time $x$, then a generalization of the communication operation takes at least time $x$. 
Section outline

3. Implementation of global communication operations

  Communication operations on static networks

  Communication operations on a hypercube network
Complete Graph Network

- **Single Gather** requires time $O(1)$:
  each node sends its message to the root node $i$
  node $i$ can simultaneously receive all messages
  analogously: **Scatter**

- A **total exchange** can also be performed in time $O(1)$:
  each node sends out all its messages at the same time
  $\leadsto$ two messages are exchanged between two arbitrary nodes
  the corresponding link is bidirectional $\leadsto$ all exchanges require $O(1)$
Linear array Network

- \( G = (V, E), V = \{1, \ldots, p\}, E = \{(i, i + 1); 1 \leq i < p\} \)

- **Single-broadcast**: the root process sends the message to its left and right neighbors; the neighbors forward the message step by step
  - **worst case**: root process at the end of the linear array
    \( \sim p - 1 \) steps required
  - **best case**: root process in the middle of the linear array
    \( \sim \lceil p/2 \rceil \) steps required

- **multi-broadcast**:
  1. 1st step: each node sends its own message to its two neighbors
  2. 2nd step: each node \( i \in \{2, \ldots, p - 1\} \) receives the messages from node \( i - 1 \) and \( i + 1 \) and forwards them to nodes \( i + 1 \) and \( i - 1 \)
  3. step \( k \): each node \( i \) with \( k \leq i \leq p - 1 \) receives the message from node \( i - k + 1 \) and forwards it to \( i + 1 \)
     each node \( i \) with \( 2 \leq i \leq p - k + 1 \) receives the message from node \( i + k - 1 \) and forwards it to \( i - 1 \)
The messages sent to the right (left) proceed in each step **one position** to the right (left)

→ after $p - 1$ steps, all messages have arrived at their destination

→ running time $\Theta(p)$

**Scatter** and **Gather** are **specializations** of multi-broadcast

→ implementation in $p - 1$ steps

→ running time $\Theta(p)$
**total exchange**: we consider an arbitrary edge \((k, k + 1)\)

- this edge splits the linear array in two subsets with \(k\) and \(p - k\) nodes
- total exchange: each node in one of the two subsets sends a message to each node of the other subset
  \(\Rightarrow k \cdot (p - k)\) messages must be transmitted over edge \((k, k + 1)\)
- for \(k = p/2\): \(p^2/4\) messages must be transmitted over a single link
  \(\Rightarrow\) running time \(\Theta(p^2)\)
Ring network

- **single-broadcast**: \( \lceil (p - 1)/2 \rceil \) steps
- **multi-broadcast**: similar to a linear array network
  - step 1: each node sends its message in both directions
  - step \( k \): (2 \( \leq k \leq \lceil p - 1/2 \rceil \))
    each node sends the messages received in the opposite direction
- diameter \( \lceil p/2 \rceil \) \( \rightsquigarrow \) running time \( \lceil p/2 \rceil \)

Illustration:

- **total exchange**: consider two nodes splitting the ring in two subsets with \( p/2 \) nodes each
  \( \rightsquigarrow \) \( p^2/4 \) messages must be transmitted over these two nodes (in each direction)
  \( \rightsquigarrow \) \( p^2/8 \) steps \( \rightsquigarrow \) running time \( \Theta(p^2) \)
3. Implementation of global communication operations
   Communication operations on static networks
   Communication operations on a hypercube network
We consider a hypercube network with $d$ dimensions and $p = 2^d$ nodes.

Construction of a spanning tree for the network.

Root at process $\alpha = 00 \cdots 0 = 0^d$.

The children in the tree are chosen by inverting one of the zero bits that are right to the rightmost unity bit.

Illustration for $d = 4$:

Note: all child nodes differ from their present node in exactly one bit position.

There is a corresponding edge in the hypercube network.
The resulting spanning tree has **depth** $d$:
consider an **arbitrary path** from a leaf to the root.
Let $(v_i, v_{i+1})$ be an arbitrary edge on the path
⇝ $v_{i+1}$ contains one 1 less than $v_i$
there are exactly $d$ bit positions ⇝ path has maximum length $d$

Using an **arbitrary node** $i$ as root, the spanning tree is constructed as follows:
let $\oplus$ be the **bitwise exor operation**
let $T_0$ be the spanning tree from above;
The spanning tree $T_i$ for root $i$ results from $T_0$ by using $x \oplus i$ for each
node $x$ of $T_0$.
Let $(v, w)$ be an edge in $T_0$ ⇝ $v$ and $w$ differ in one bit position
⇝ $v \oplus i$ and $w \oplus i$ also differ in one bit position
⇝ $(v \oplus i, w \oplus i)$ is an edge of the hypercube network

**summary**: Single-broadcast in $d = \log p$ steps
Multi-broadcast on a hypercube network

- Using the same spanning trees as for a single-broadcast leads to collisions: example: $d = 3$
  the spanning trees for root node 000 and 110 use the same edges (010, 011) and (100, 101), implying no simultaneous transmission possible.

Illustration:

solution: use of alternative spanning trees
Spanning trees for multi-broadcast for hypercube
Construction of the spanning trees

- $N_k =$ set of all nodes with bit representations containing $k$ unity bits

$$\{00 \cdots 0\} N_1 \ N_2 \cdots N_{(d-2)} \ N_{d-1}\{11 \cdots 1\}$$

00⋯0 has position $n(00\cdots 0)=0$, 11⋯1 has position $n(11\cdots 1)=2^d-1$

- $N_k$ is split into disjoint subsets $R_{k1}, \ldots, R_{kn_k}$
  $R_{ki}$ contains all nodes from $N_k$ that can be transformed to each other by a bit rotation to the left
  The subsets $R_{ki}$ form equivalence classes of $N_k$
  The subsets $R_{ki}$ are ordered as follows:

$$\{00 \cdots 0\} R_{11} R_{21} \cdots R_{2n_2} \cdots R_{k1} \cdots R_{kn_k} \cdots R_{(d-2)1} \cdots R_{(d-2)n_{d-2}} R_{(d-1)1}\{11 \cdots 1\}$$

- Each node $t \in \{0, 1\}^d$ gets a number $m(t)$ with $m(00\cdots 0) = 0$ and $m(t) = 1 + [(n(t) - 1) \mod d]$, i.e. the nodes are numbered in a round-robin fashion by 1⋯$d$. 
Representation of the sets $N_k$

example: $d = 4$:

\[
\begin{align*}
N_0 & \quad (0000) \\
N_1 & \quad \begin{array}{cccc}
0 & 1 & 2 & 3 \\
(0001) & (0010) & (0100) & (1000) \\
\end{array} \\
N_2 & \quad \begin{array}{cccc}
1 & 2 & 3 & 4 \\
(0011) & (0110) & (1100) & (1001) \\
\end{array} \\
N_3 & \quad \begin{array}{cccc}
3 & 4 & 1 & 2 \\
(1101) & (1011) & (0111) & (1110) \\
\end{array} \\
N_4 & \quad (1111)
\end{align*}
\]

each node gets the number written above it
Definition of the node sets $E_i$

- We define $m + 1$ node sets $E_0, E_1, \ldots, E_m$
- $E_i$ is the set of all end nodes of edge sets $A_i$

$$E_0 = \{(00 \cdots 0)\}$$

$$E_i = \{t \in \{0,1\}^d \mid (i - 1)d + 1 \leq n(t) \leq i \cdot d\} \quad \text{for } 1 \leq i < m$$

$$E_m = \{t \in \{0,1\}^d \mid (m - 1)d + 1 \leq n(t) \leq 2^d - 1\} \quad \text{with } m = \left\lceil \frac{2^d - 1}{d} \right\rceil.$$ 

- For each set $E_i$ the following holds:
  - $E_i$ contains $d$ contiguous nodes
  - $\forall$ all nodes in $E_i$ have a different number $m(t) \in \{1, \ldots, d\}$
Definition of the edges sets $A_i$

- Each node $t \in E_i$ is connected with a node $t'$ which results from $t$ by inverting the bit at position $m(t)$ from the right; this bit is always a unity bit by construction.

- **Special case:** if $m(11 \cdots 1) = d$, not the $d$th bit but the $(d-1)$th bit from the right is inverted, i.e., $((11 \cdots 1), (1011 \cdots 1)) \in A_m$, but $((11 \cdots 1), (011 \cdots 1)) \not\in A_m$
we consider the case \( d = 4 \); the following spanning tree results

spanning tree with root 00 \( \cdots \) 0;
the edge sets \( A_i, i = 1, \cdots, 4 \), of the different stages are indicated by dotted arrows
Total exchange on a hypercube network

- construction of a recursive algorithm with $2^d - 1$ steps
- $d = 1$: two nodes send a message to each other
- $d \rightarrow \phi + 1$: the hypercube of dimension $(d + 1)$ is split into two sub-hypercubes $C_1$ and $C_2$ with dimension $(d)$
- **Phase 1**: a total exchange is performed simultaneously in $C_1$ and $C_2$; each node of $C_1$ or $C_2$ exchanges messages with each other node of $C_1$ or $C_2$, respectively; this takes $2^d - 1$ steps
- **Phase 2**: each node in $C_1$ or $C_2$ sends its messages for all nodes in the other sub-hypercube to the corresponding node in this sub-hypercube. Each node sends $2^d$ messages $\leadsto 2^d$ steps for this phase
- **Phase 3**: The messages received in phase 2 are distributed within $C_1$ or $C_2$ by a recursive use of the algorithm; this is similar to phase 1 $\leadsto 2^d - 1$ steps for phase 3
- Phase 1 and 2 can be performed simultaneously, since different links of the hypercube network are used $\leadsto$ Phase 1 and 2 together require $2^d$ steps
- Phase 3 must be performed after phase 2
  $\leadsto$ total number of steps in $2^d + 2^d - 1 = 2^{d+1} - 1$
  $\leadsto$ overall running time $\Theta(p) = \Theta(2^d)$