Networking and Scalability

Parallel Programming

2023-12-07

Prof. Dr. Michael Kuhn
michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de
Networking and Scalability

Review

Introduction

Basics

Technologies

Scalability

Summary
Which MPI thread mode is the default?

1. MPI_THREAD_SINGLE
2. MPI_THREAD_FUNNELED
3. MPI_THREAD_SERIALIZED
4. MPI_THREAD_MULTIPLE
Which behavior does MPI_Ssend have?

1. Blocking local
2. Non-blocking local
3. Blocking non-local
4. Non-blocking non-local
Which function buffers data while sending?

1. MPI_Send
2. MPI_Bsend
3. MPI_Isend
4. MPI_Rsend
What is the difference between MPI_Reduce and MPI_Allreduce?

1. MPI_Reduce performs a local operation, MPI_Allreduce across all ranks
2. MPI_Reduce collects the value at the root rank, MPI_Allreduce at every rank
3. MPI_Allreduce performs a barrier before, MPI_Reduce does not
4. MPI_Allreduce performs a barrier afterwards, MPI_Reduce does not
• Shared memory systems have limited scalability
  • Machines usually have two to four processors with a few dozen cores
  • OpenMP is a convenient and high-level programming concept
• Complex problems require more resources than available on a single node
  • Simulations require more computational power and main memory
  • Multiple nodes are connected via a so-called interconnect
• Distributed memory can be scaled almost arbitrarily
  • These typically consist of a cluster of shared memory systems
  • The largest machines have up to 10,000,000 cores in several thousand nodes
Network connections are required to connect multiple nodes
  - Compute nodes have to communicate with each other
  - Storage nodes offer services via the network

Necessary for inter-process communication across nodes
  - Shared memory objects enable communication on one node
  - Message passing is a programming concept for distributed memory

Network can be designed using a variety of topologies
  - Bus, ring, star, fully connected mesh, fat tree etc.
Motivation...

Introduction

- Processors require data fast
  - 3 GHz equals three operations per nanosecond
  - Even accessing the main memory is too slow
  - Cache levels hide main memory latency
- Network and I/O extremely slow in comparison
  - Waiting for an HDD ruins performance
  - SSDs have alleviated the problem a bit
  - Network adds additional latency

<table>
<thead>
<tr>
<th>Level</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>≈ 5 ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>≈ 10 ns</td>
</tr>
<tr>
<td>RAM</td>
<td>≈ 100 ns</td>
</tr>
<tr>
<td>InfiniBand</td>
<td>≈ 500 ns</td>
</tr>
<tr>
<td>Ethernet</td>
<td>≈ 100,000 ns</td>
</tr>
<tr>
<td>SSD</td>
<td>≈ 100,000 ns</td>
</tr>
<tr>
<td>HDD</td>
<td>≈ 10,000,000 ns</td>
</tr>
</tbody>
</table>

[Bonér, 2012] [Huang et al., 2014]
Motivation...

• Computation is only one part of parallel applications
  • Store data in main memory and persist it to storage
  • Main memory and storage per node is also limited
• Storage nodes are usually separate
  • Exclude influence on each other
  • Nodes can be tuned for their respective workloads
• Data has to be transferred for each I/O operation
  • I/O typically also includes network latency
  • Node-local buffers can be used as a workaround
Networking and Scalability

Review

Introduction

Basics

Technologies

Scalability

Summary
Several components are necessary to build a network

- Network interface cards (NIC)
  - Each node is equipped with one or more of them
- Network cables
  - Supercomputers need hundreds to thousands of kilometers of cables
  - Fiber cables offer high frequencies and less loss than copper cables
- Network switches
  - Multiple switches can be required depending on the network topology

Network is often split into multiple sub-networks

- Separate communication, storage and management networks
• Have to consider both the hardware and software perspective
• Network technology should be adaptable to different environments
  • Allow using different network topologies depending on requirements
• Different network technologies typically have different interfaces
  • For convenience reasons, a high level of abstraction is preferred
  • High performance might require breaking the high level of abstraction
• Data should be transferred as efficiently as possible
  • High numbers of system calls can have a negative performance impact
  • Some network technologies use kernel bypass to improve performance
Performance characteristics are especially important in HPC
  • Network should introduce as little additional overhead as possible

Bandwidth (in GBit/s or GB/s)
  • Actual throughput might be less due to protocol overhead etc.

Latency (in ns)
  • Highly dependent on software overheads
  • Depending on distance, physics also becomes important (≈ 1 ms per 100 km)

Robustness and error rate
  • Network should handle faults in single cables or switches
  • Other factors might cause errors that should be detected and corrected

TCP/IP support
  • TCP usage is almost ubiquitous and some applications support nothing else
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
  2. Execute system call to send data
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
  2. Execute system call to send data
  3. Kernel copies data to kernel space and passes it to NIC
Use case: Sending data to another process

1. Data is produced by application and resides in its address space
2. Execute system call to send data
3. Kernel copies data to kernel space and passes it to NIC
4. NIC transfers data to target node’s NIC
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
  2. Execute system call to send data
  3. Kernel copies data to kernel space and passes it to NIC
  4. NIC transfers data to target node’s NIC
  5. Target NIC copies received data to kernel space
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
  2. Execute system call to send data
  3. Kernel copies data to kernel space and passes it to NIC
  4. NIC transfers data to target node’s NIC
  5. Target NIC copies received data to kernel space
  6. Target process executes system call and copies data
• Use case: Sending data to another process
  1. Data is produced by application and resides in its address space
  2. Execute system call to send data
  3. Kernel copies data to kernel space and passes it to NIC
  4. NIC transfers data to target node’s NIC
  5. Target NIC copies received data to kernel space
  6. Target process executes system call and copies data
• Different terminology depending on network technology
  • Ethernet uses a network interface card (NIC)
  • InfiniBand uses a host channel adapter (HCA)
Socket Based

[Chenfan, 2016]
RDMA and Zero Copy Basics

Socket Based

RDMA Based

[Chenfan, 2016]
Remote direct memory access (RDMA)
- Target’s memory can be accessed directly without interruption
- Memory might have to be registered and/or locked

Zero copy
- Avoid copies between user space and kernel space
- Potential copies within the kernel (kernel buffer and driver buffer)
- Additional copies between kernel space and device

Copying is expensive from performance and energy perspectives
- Copying data once reduces maximum throughput by a factor of two etc.
Network stacks have been designed for different requirements
  - High latencies, low throughputs and potentially high error rates
  - TCP/IP includes support for retransmissions etc.

Packets are typically small
  - Ethernet normally uses 1,500 bytes frames, TCP up to 64 KiB
  - Worst case: One interrupt per packet

Operating systems implement their own network stacks
  - Operations have to be performed in software
  - Software overheads can be problematic for high-speed interconnects
• Interrupts can quickly accumulate for high packet rates
  • Interrupts prevent applications from performing computations
• Polling requires processor time to check for new packets
  • Can be more efficient if many packets can be retrieved at once
• Parts of network protocols can be provided in hardware
  • TCP Offload Engine is widely used to improve TCP performance
• DMA allows data to be copied without involving processor
  • Otherwise, processor would have to copy data actively
Kernel Bypass

Basics

- Traditionally, talking to the network card requires the kernel
  - Kernel manages and talks to the NIC via a driver
  - Applications talk to kernel via system calls
- Context switches and interrupts cause high overhead
  - Kernel bypass allows applications to talk to the NIC directly
- Different approaches exist already [Majkowski, 2015]
  - Many require special hardware support or dedicated NICs
  - For instance, specialized network API that manages queues on NIC
• How many additional memory buffers does zero copy require?
  1. One memory buffer in kernel space
  2. One memory buffer in user space
  3. No additional memory buffers
Networking and Scalability

Review
Introduction
Basics
Technologies
Scalability
Summary
• Hardware improves exponentially, but at different rates
  • Storage capacity and throughput are lagging behind computation
• Network and memory throughput are even further behind
  • Transferring data has become a very costly operation
Performance Development...

- Network bandwidth has increased steadily over the years
- Two main competitors in HPC
  1. Ethernet
  2. InfiniBand
- InfiniBand supports multiple links
  - x1 is base performance
  - x4, x8 and x12 are faster

<table>
<thead>
<tr>
<th>Technology</th>
<th>Bandwidth</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet</td>
<td>10 Mbit/s</td>
<td>1980</td>
</tr>
<tr>
<td>Fast Ethernet</td>
<td>100 Mbit/s</td>
<td>1995</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>1 Gbit/s</td>
<td>1998</td>
</tr>
<tr>
<td>InfiniBand SDR x1</td>
<td>2 Gbit/s</td>
<td>2001</td>
</tr>
<tr>
<td>InfiniBand SDR x12</td>
<td>24 Gbit/s</td>
<td>2001</td>
</tr>
<tr>
<td>10 Gigabit Ethernet</td>
<td>10 Gbit/s</td>
<td>2002</td>
</tr>
<tr>
<td>InfiniBand DDR x12</td>
<td>48 Gbit/s</td>
<td>2005</td>
</tr>
<tr>
<td>InfiniBand QDR x12</td>
<td>96 Gbit/s</td>
<td>2007</td>
</tr>
<tr>
<td>100 Gigabit Ethernet</td>
<td>100 Gbit/s</td>
<td>2010</td>
</tr>
<tr>
<td>InfiniBand FDR x12</td>
<td>163.64 Gbit/s</td>
<td>2011</td>
</tr>
<tr>
<td>InfiniBand EDR x12</td>
<td>300 Gbit/s</td>
<td>2014</td>
</tr>
<tr>
<td>Omni-Path</td>
<td>100 Gbit/s</td>
<td>2015</td>
</tr>
<tr>
<td>400 Gigabit Ethernet</td>
<td>400 Gbit/s</td>
<td>2017</td>
</tr>
<tr>
<td>InfiniBand HDR x12</td>
<td>600 Gbit/s</td>
<td>2017</td>
</tr>
</tbody>
</table>

[Wikipedia, 2020]
InfiniBand is a networking standard
- Promoted by the InfiniBand Trade Association
- Mellanox is the major vendor for InfiniBand (now part of Nvidia)

Mostly used in HPC due to high throughput and low latency
- Throughputs up to 600 GBit/s
- Latencies of less than 500 ns

InfiniBand provides support for RDMA
- Used by MPI’s own RDMA support
InfiniBand...

Technologies

• No standard API
  • Standard only has a list of verbs such as `ibv_open_device`
  • De-facto standard software stack by OpenFabrics Alliance
  • `libibverbs` for Linux, kernel support since 2005

• Packets of up to 4 KB for messages
  • RDMA read or write
  • Send or receive
  • Transaction operation
  • Multicast operation
  • Atomic operation
• How much throughput can we typically expect from Gigabit Ethernet?

1. 10 MB/s
2. 12 MB/s
3. 100 MB/s
4. 120 MB/s
5. 1 GB/s
6. 1.2 GB/s
Networking and Scalability

Review
Introduction
Basics
Technologies
Scalability
Summary
Reminder: Scalability is ambiguous and can apply to different components

- We have taken a look at the scalability of hardware architectures before
- How big we can scale something while keeping the benefits
  - How easy it is to increasing a network’s size
  - How well invested money correlates with improved performance
  - How well an application can run on more cores/nodes

- We will now take a look at the scalability of parallel applications
• When writing parallel applications, we must consider scalability
  • Scalability describes how an application behaves with increasing parallelism
• HPC systems are usually very expensive and should be used accordingly
  • Procurement costs can reach up to €250,000,000
• To determine scalability, we have to analyze performance
  • HPC systems are complex, performance yield is often not optimal
  • Many different components interact with each other
    • Processors, caches, main memory, network, storage system etc.
Motivation...

Scalability

• In addition to procurement costs, operating costs are also quite high
  • 1. Frontier (USA): 1.2 EFLOPS at 22.7 MW ≈ € 52,700,000/a (in Germany)
  • 5. LUMI (Finland): 380 PFLOPS at 7.1 MW ≈ € 16,500,000/a (in Germany)
  • 74. Levante (Germany): 10 PFLOPS at 2 MW ≈ € 4,600,000/a

• Communication and I/O are often responsible for performance problems
  • High latency, which causes excessive waiting times for processors
  • Communication and I/O typically happen synchronously
• The performance improvement we get is called speedup
  • In the best case, the speedup is equal to the number of tasks
  • In reality, the speedup is usually lower due to overhead (communication, I/O etc.)
• Speedup can sometimes be higher than the number of tasks
  • This is called a superlinear speedup and usually points at a problem
  • For example, each task’s data suddenly fits into the cache
    • This means that the measured problem became too small
    • Larger problems will not fit and therefore have a lower speedup
• Speedup: \( S(n) = \frac{T(1)}{T(n)} \)
  • \( T(1) \): Runtime of one task
  • \( T(n) \): Runtime of \( n \) tasks
• Speedup: $S(n) = \frac{T(1)}{T(n)}$
  - $T(1)$: Runtime of one task
  - $T(n)$: Runtime of $n$ tasks

• Requirement: Choose fastest algorithm
  - $T(1)$ is not necessarily the parallel version executed with one task
  - Sometimes a serial algorithm might be the fastest choice
• Speedup: \( S(n) = \frac{T(1)}{T(n)} \)
  - \( T(1) \): Runtime of one task
  - \( T(n) \): Runtime of \( n \) tasks

• Requirement: Choose fastest algorithm
  - \( T(1) \) is not necessarily the parallel version executed with one task
  - Sometimes a serial algorithm might be the fastest choice

• Efficiency: \( E(n) = \frac{S(n)}{n} \)
  - Normalizes the speedup to the \([0, 1]\) range
Amdahl’s Law

- Amdahl’s law describes an upper limit for the speedup
  - Every application contains a serial portion that limits the speedup
- $f$ is the serial portion ($\in [0, 1]$)

\[
S(n) = \frac{1}{f + \frac{1-f}{n}} \Rightarrow S_{\text{max}} = \frac{1}{f}
\]

- Even seemingly small serial portions have a large impact
  - $f = 0.01 \Rightarrow S_{\text{max}} = 100$
  - Try to keep serial portion as small as possible
- Problem: Only applies if problem size is fixed
  - It usually makes sense to increase the problem size if more nodes are available
Amdahl’s Law…

- Examples
  - 5% serial portion
    - $S_{max} = 20$
  - 50% serial portion
    - $S_{max} = 2$
- Parallelization might sometimes not be worth it
  - Weigh up required effort against potential speedup

[Daniels220, 2008]
Gustafson’s Law

- Gustafson’s law also describes an upper limit for the speedup
  - In contrast to Amdahl’s law, problem size can be increased
  - However, time has to be fixed (problem size has to be chosen appropriately)
  - Every application contains a serial portion

- \( f \) is the serial portion (\( \in [0, 1] \))

\[
S(n) = n + f(1 - n) = n + f - fn = n - fn + f = n - f(n - 1)
\]

- Also does not apply to all kinds of applications
  - Problem sizes cannot always be scaled up arbitrarily
Gustafson’s Law

- Examples
  - 5% serial portion
    - $S(120) = 114$
  - 50% serial portion
    - $S(120) = 60$
- Much better than with Amdahl’s law
  - Increasing problem size compensates overhead

[Peahihawaii, 2011]
Strong and Weak Scaling

- Scaling behavior can be generalized based on problem size
  - Increasing speedup with constant problem size is harder
  - Algorithms can be judged and compared based on their scaling behavior
Strong and Weak Scaling

- Scaling behavior can be generalized based on problem size
  - Increasing speedup with constant problem size is harder
  - Algorithms can be judged and compared based on their scaling behavior
- Weak scaling
  - Increase problem size together with task count (related to Gustafson’s law)
Strong and Weak Scaling

• Scaling behavior can be generalized based on problem size
  • Increasing speedup with constant problem size is harder
  • Algorithms can be judged and compared based on their scaling behavior

• Weak scaling
  • Increase problem size together with task count (related to Gustafson’s law)

• Strong scaling
  • Increase task count with constant problem size (related to Amdahl’s law)
Strong and Weak Scaling

- Scaling behavior can be generalized based on problem size
  - Increasing speedup with constant problem size is harder
  - Algorithms can be judged and compared based on their scaling behavior

- Weak scaling
  - Increase problem size together with task count (related to Gustafson’s law)

- Strong scaling
  - Increase task count with constant problem size (related to Amdahl’s law)

- Example: Matrix calculation
  - Matrix contains $1,000 \times 1,000$ elements
  - Calculation for one element requires elements from neighbors
Strong Scaling

• Parallelization with 5 tasks
  • Each task has a sub-matrix of $200 \times 1,000$ elements
  • Each task has to communicate $2 \times 1,000$ elements with others
  • Communication-to-computation ratio is $1:100$
• Parallelization with 5 tasks
  • Each task has a sub-matrix of $200 \times 1,000$ elements
  • Each task has to communicate $2 \times 1,000$ elements with others
  • Communication-to-computation ratio is 1:100

• Parallelization with 10 tasks
  • Each task has a sub-matrix of $100 \times 1,000$ elements
  • Each task has to communicate $2 \times 1,000$ elements with others
  • Communication-to-computation ratio is 1:50
Strong Scaling

- Parallelization with 5 tasks
  - Each task has a sub-matrix of 200 × 1,000 elements
  - Each task has to communicate 2 × 1,000 elements with others
  - Communication-to-computation ratio is 1:100

- Parallelization with 10 tasks
  - Each task has a sub-matrix of 100 × 1,000 elements
  - Each task has to communicate 2 × 1,000 elements with others
  - Communication-to-computation ratio is 1:50

- Parallelization with 100 tasks
  - Each task has a sub-matrix of 10 × 1,000 elements
  - Each task has to communicate 2 × 1,000 elements with others
  - Communication-to-computation ratio is 1:5
- Parallelization with 10 tasks and doubled matrix size
  - Each task has a sub-matrix of $141 \times 1,414$ elements
  - Each task has to communicate $2 \times 1,414$ elements with others
  - Communication-to-computation ratio is 1:70

- Parallelization with 10 tasks and tenfold matrix size
  - Each task has a sub-matrix of $316 \times 3,162$ elements
  - Each task has to communicate $2 \times 3,162$ elements with others
  - Communication-to-computation ratio is 1:158

- Parallelization with 100 tasks and hundredfold matrix size
  - Each task has a sub-matrix of $100 \times 10,000$ elements
  - Each task has to communicate $2 \times 10,000$ elements with others
  - Communication-to-computation ratio is 1:50
Weak Scaling

Scalability

- Parallelization with 10 tasks and doubled matrix size
  - Each task has a sub-matrix of $141 \times 1,414$ elements
  - Each task has to communicate $2 \times 1,414$ elements with others
  - Communication-to-computation ratio is 1:70

- Parallelization with 10 tasks and tenfold matrix size
  - Each task has a sub-matrix of $316 \times 3,162$ elements
  - Each task has to communicate $2 \times 3,162$ elements with others
  - Communication-to-computation ratio is 1:158
Weak Scaling

Scalability

- Parallelization with 10 tasks and doubled matrix size
  - Each task has a sub-matrix of $141 \times 1,414$ elements
  - Each task has to communicate $2 \times 1,414$ elements with others
  - Communication-to-computation ratio is 1:70

- Parallelization with 10 tasks and tenfold matrix size
  - Each task has a sub-matrix of $316 \times 3,162$ elements
  - Each task has to communicate $2 \times 3,162$ elements with others
  - Communication-to-computation ratio is 1:158

- Parallelization with 100 tasks and hundredfold matrix size
  - Each task has a sub-matrix of $100 \times 10,000$ elements
  - Each task has to communicate $2 \times 10,000$ elements with others
  - Communication-to-computation ratio is 1:50
• Speedup graphs visualize performance
- Speedup graphs visualize performance
- Optimal speedup
  - Perfect scaling, no overhead
• Speedup graphs visualize performance
• Optimal speedup
  • Perfect scaling, no overhead
• Linear speedup
  • Constant overhead

<Graph showing speedup against number of tasks with different line types representing Superlinear, Optimal, Typical (10%), Typical (20%), and Linear speedup.>
• Speedup graphs visualize performance
  • Optimal speedup
    • Perfect scaling, no overhead
  • Linear speedup
    • Constant overhead
  • Typical speedup
    • Overhead keeps growing
    • With 10 or 20% serial portion according to Amdahl’s law

```
  Speedup
  Number of Tasks

Superlinear
Optimal
Linear
Typical (10%)
Typical (20%)
```

```
  Speedup
  Number of Tasks

Superlinear
Optimal
Linear
Typical (10%)
Typical (20%)
```
• Speedup graphs visualize performance
• Optimal speedup
  • Perfect scaling, no overhead
• Linear speedup
  • Constant overhead
• Typical speedup
  • Overhead keeps growing
  • With 10 or 20% serial portion according to Amdahl’s law
• Superlinear speedup
  • Negative overhead?
Example: Comparing search algorithms

- We have $n$ search algorithms with runtimes $t_i$
- Want to find the fastest one
Superlinear Speedup

Example: Comparing search algorithms
- We have \( n \) search algorithms with runtimes \( t_i \)
- Want to find the fastest one

Serial version
1. Set \( t_{\text{min}} = \infty \)
2. Run each algorithm \( i \)
   2.1 If it runs longer than \( t_{\text{min}} \), terminate it
   2.2 Otherwise, set \( t_{\text{min}} = t_i \)

The serial version has a runtime of
\[ t_{\text{serial}} \geq t_{\text{min}} \times n \]
if we do not run the fastest algorithm first.
Example: Comparing search algorithms

- We have \( n \) search algorithms with runtimes \( t_i \)
- Want to find the fastest one

Serial version

1. Set \( t_{min} = \infty \)
2. Run each algorithm \( i \)
   2.1 If it runs longer than \( t_{min} \), terminate it
   2.2 Otherwise, set \( t_{min} = t_i \)

The serial version has a runtime of \( t_{serial} \geq t_{min} \times n \)

- \( t_{serial} > t_{min} \times n \) if we do not run the fastest algorithm first
• Parallel version
  1. Run each algorithm $i$ on its own core
     1.1 As soon as the first one finishes, set $t_{\text{min}} = t_i$ and terminate all other algorithms
• Parallel version
  1. Run each algorithm \( i \) on its own core
     1.1 As soon as the first one finishes, set \( t_{\text{min}} = t_i \) and terminate all other algorithms
• The parallel version has a runtime of \( t_{\text{parallel}} = t_{\text{min}} \)

\[
S = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \Rightarrow S \geq \frac{t_{\text{min}} \times n}{t_{\text{min}}} \Rightarrow S \geq n
\]
• What mistake did we make to achieve a superlinear speedup?
  1. We did not make a mistake
  2. We did not choose the fastest serial algorithm
  3. We cannot run each algorithm on its own core
• Improved serial version
  1. Run each algorithm $i$ for a time slice $t$
     1.1 As soon as the first one finishes, set $t_{min} = t_i$ and terminate all other algorithms
- Improved serial version
  1. Run each algorithm \( i \) for a time slice \( t \)
     1.1 As soon as the first one finishes, set \( t_{\text{min}} = t_i \) and terminate all other algorithms
- The improved version has a runtime of \( t_{\text{serial}} \approx t_{\text{min}} \times n \)
  - Overhead depends on length of the time slice
  - This gets rid of the superlinear speedup
Networking and Scalability

Review
Introduction
Basics
Technologies
Scalability
Summary
• Networking is necessary to build distributed memory systems
  • Shared memory systems have limited scalability
• Network technologies have different performance characteristics
  • The two major competitors are Ethernet and InfiniBand
• High-performance networking requires optimizations
  • RDMA, zero copy, offloading and kernel bypass help reduce overhead
• Scalability can be measured using speedup and efficiency
  • There are limits to scalability, demonstrated by Amdahl and Gustafson’s laws


