The horrors of parallel programming

Max Neunhöffer

HPCGAP workshop 19–23 August 2013
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{c|c|c|c}
3 & -1 & -2 & -3 \\
\end{array} \]
Imagine a **plain list**, first entry is the **length**:

\[
L := \begin{array}{cccc}
3 & -1 & -2 & -3 \\
\end{array}
\]

How do you implement appending an element \(-4\) at the end?
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{cccc}
3 & -1 & -2 & -3
\end{array} \]

How do you implement appending an element $-4$ at the end?

1. First increase the length, then store the new element:
   - \[ L[0] := L[0] + 1 \]
   - \[ L[L[0]] := -4 \]
Imagine a plain list, first entry is the length:

\[
L := \begin{array}{cccc}
3 & -1 & -2 & -3
\end{array}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{c|c|c|c}
3 & -1 & -2 & -3 \\
\end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \( L[0] := L[0] + 1 \)
   - \( L[L[0]] := -4 \)

2. First store the new element, then increase the length:
   - \( L[L[0] + 1] := -4 \)
   - \( L[0] := L[0] + 1 \)

Both look good —
Imagine a plain list, first entry is the length:

\[
L := \begin{array}{cccc}
3 & -1 & -2 & -3 \\
\end{array}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!
Imagine a **plain list**, first entry is the **length**:

\[ L := \begin{array}{cccc} 3 & -1 & -2 & -3 \end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \( L[0] := L[0] + 1 \)
   - \( L[L[0]] := -4 \)

2. First store the new element, then increase the length:
   - \( L[L[0] + 1] := -4 \)
   - \( L[0] := L[0] + 1 \)

Both **look good** — but do not work in a multi-threaded world!
Imagine a plain list, first entry is the length:

\[
L := \begin{bmatrix}
3 & -1 & -2 & -3 & \\
\end{bmatrix}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!

\[
L = \begin{bmatrix}
4 & -1 & -2 & -3 & \\
\end{bmatrix}
\]
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{llll}
3 & -1 & -2 & -3
\end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!

\[ L = \begin{array}{llll}
5 & -1 & -2 & -3
\end{array} \]
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{c|c|c|c|c} 3 & -1 & -2 & -3 & \_ \end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \( L[0] := L[0] + 1 \)
   - \( L[L[0]] := -4 \)

2. First store the new element, then increase the length:
   - \( L[L[0] + 1] := -4 \)
   - \( L[0] := L[0] + 1 \)

Both look good — but do not work in a multi-threaded world!

\[ L = \begin{array}{c|c|c|c|c} 5 & -1 & -2 & -3 & -4 \end{array} \]
Imagine a **plain list**, first entry is the **length**:

\[
L := \begin{array}{c}
3 \\
-1 \\
-2 \\
-3 \\
0 \\
\end{array}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both **look good** — but do not work in a multi-threaded world!

\[
L = \begin{array}{c}
5 \\
-1 \\
-2 \\
-3 \\
-5 \\
\end{array}
\]
Imagine a plain list, first entry is the length:

\[
L := \begin{array}{cccc}
  3 & -1 & -2 & -3 \\
\end{array}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!

\[
L = \begin{array}{cccc}
  3 & -1 & -2 & -3 \\
\end{array}
\]
Imagine a plain list, first entry is the length:

\[
L := \begin{array}{cccc}
3 & -1 & -2 & -3
\end{array}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!
Imagine a plain list, first entry is the length:

\[
L := \begin{bmatrix}
3 & \text{-1} & \text{-2} & \text{-3} & \\
\end{bmatrix}
\]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \(L[0] := L[0] + 1\)
   - \(L[L[0]] := -4\)

2. First store the new element, then increase the length:
   - \(L[L[0] + 1] := -4\)
   - \(L[0] := L[0] + 1\)

Both look good — but do not work in a multi-threaded world!

\[
L = \begin{bmatrix}
3 & \text{-1} & \text{-2} & \text{-3} & \text{-5} \\
\end{bmatrix}
\]
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{c|c|c|c}
3 & -1 & -2 & -3 \\
\end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \( L[0] := L[0] + 1 \)
   - \( L[L[0]] := -4 \)

2. First store the new element, then increase the length:
   - \( L[L[0] + 1] := -4 \)
   - \( L[0] := L[0] + 1 \)

Both look good — but do not work in a multi-threaded world!

\[ L = \begin{array}{c|c|c|c|c}
4 & -1 & -2 & -3 & -5 \\
\end{array} \]
Imagine a plain list, first entry is the length:

\[ L := \begin{array}{cccc}
3 & -1 & -2 & -3 \\
\end{array} \]

How do you implement appending an element \(-4\) at the end?

1. First increase the length, then store the new element:
   - \( L[0] := L[0] + 1 \)
   - \( L[L[0]] := -4 \)

2. First store the new element, then increase the length:
   - \( L[L[0] + 1] := -4 \)
   - \( L[0] := L[0] + 1 \)

Both look good — but do not work in a multi-threaded world!

\[ L = \begin{array}{cccc}
5 & -1 & -2 & -3 & -5 \\
\end{array} \]
Thus: No order of events actually works.

Also: Another thread reading can see a corrupt list.

It is even worse: It is not even clear that thread 2 sees the changes thread 1 has made!

This is because of modern cache architectures. Even statements like

$L[0] := L[0] + 1$

might have problems!

Solutions:

- Use read-only data as much as possible!
- Organise exclusive access by program logic.
- Use locking — be it explicit or implicit.

HPCGAP offers: regions, read-only objects, private data, the atomic statement and atomic objects.
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

Solutions:
Use read-only data as much as possible!
Organise exclusive access by program logic.
Use locking — be it explicit or implicit.
HPCGAP offers: regions, read-only objects, private data, the atomic statement and atomic objects.
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
Thus: No order of events actually works. Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
- Even statements like $L[0] := L[0] + 1$ might have problems!
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
- Even statements like $L[0] := L[0] + 1$ might have problems!

Solutions:
- Use read-only data as much as possible!
Thus: No order of events actually works.

Also: Another thread reading can see a corrupt list.

It is even worse:

- It is *not even clear* that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
- Even statements like $L[0] := L[0] + 1$ might have problems!

Solutions:

- Use *read-only data* as much as possible!
- Organise *exclusive access* by program logic.
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
- Even statements like $L[0] := L[0] + 1$ might have problems!

Solutions:

- Use read-only data as much as possible!
- Organise exclusive access by program logic.
- Use locking — be it explicit or implicit.
Thus: No order of events actually works.
Also: Another thread reading can see a corrupt list.

It is even worse:

- It is not even clear that thread 2 sees the changes thread 1 has made!
- This is because of modern cache architectures.
- Even statements like $L[0] := L[0] + 1$ might have problems!

Solutions:

- Use read-only data as much as possible!
- Organise exclusive access by program logic.
- Use locking — be it explicit or implicit.

**HPCGAP offers:** regions, read-only objects, private data, the atomic statement and atomic objects.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.

**Reasons:**

- All threads see the same value.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.

**Reasons:**

- All threads see the same value.
- It could be changed in one thread whilst another is running.
In a single-threaded system a global variable for configuration or other purposes might be a good idea.

In a multi-threaded system it isn’t.

Reasons:

- All threads see the same value.
- It could be changed in one thread whilst another is running.
- Even read access will need some kind of locking.
In a single-threaded system a global variable for configuration or other purposes might be a good idea.

In a multi-threaded system it isn’t.

Reasons:
- All threads see the same value.
- It could be changed in one thread whilst another is running.
- Even read access will need some kind of locking.

Solution:
- Avoid global variables or global state if at all possible.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.

**Reasons:**
- All threads see the same value.
- It could be changed in one thread whilst another is running.
- Even read access will need some kind of locking.

**Solution:**
- Avoid global variables or global state if at all possible.
- Use additional arguments for configuration.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.

**Reasons:**
- All threads see the same value.
- It could be changed in one thread whilst another is running.
- Even read access will need some kind of locking.

**Solution:**
- Avoid global variables or global state **if at all possible**.
- Use **additional arguments** for configuration.
- For **global caching**, use proper **locking**.
In a single-threaded system a **global variable** for configuration or other purposes might be a good idea.

In a **multi-threaded system** it isn’t.

**Reasons:**
- All threads see the same value.
- It could be changed in one thread whilst another is running.
- Even read access will need some kind of locking.

**Solution:**
- Avoid global variables or global state if at all possible.
- Use additional arguments for configuration.
- For global caching, use proper locking.

**HPCGAP offers:**
- regions, shared objects, locking, thread local variables
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [ [ 1, 319 ], [ 2, 681 ] ]
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

gap> while true do a := 1; a := 2 ; od;

This one is rather obvious.
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

```
gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [ [ 1, 319 ], [ 2, 681 ] ]
```

- This one is rather obvious.
- In general, these things can be very subtle.
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

```gap
gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [[1, 319], [2, 681]]
```

- This one is rather obvious.
- In general, these things can be very subtle.
- Some problem might occur with very small probability.
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [ [ 1, 319 ], [ 2, 681 ] ]

- This one is rather obvious.
- In general, these things can be very subtle.
- Some problem might occur with very small probability.
- Thus it is difficult to reproduce and difficult to fix.
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [ [ 1, 319 ], [ 2, 681 ] ]

- This one is rather obvious.
- In general, these things can be very subtle.
- Some problem might occur with very small probability.
- Thus it is difficult to reproduce and difficult to fix.

Solution: use synchronisation to avoid
In a parallel program, the behaviour can depend on some more or less random order, in which some events occur.

```
gap> while true do a := 1; a := 2 ; od;
!sh
--- Switching to thread 5
[5] gap> Collected(List([1..1000],i->a));
[5] [ [ 1, 319 ], [ 2, 681 ] ]
```

- This one is rather obvious.
- In general, these things can be very subtle.
- Some problem might occur with very small probability.
- Thus it is difficult to reproduce and difficult to fix.

Solution: use synchronisation to avoid

HPCGAP offers: semaphores, channels, synchronisation variables
Locking is useful, but what if two threads wait for each other’s lock:
Locking is useful, but what if two threads wait for each other’s lock:

gap> a := ShareSpecialObj([1,2,3]);;
gap> b := ShareSpecialObj([1,2,3]);;
gap> c := CreateSemaphore(0);
<semaphore 0xb557060: count = 0>
gap> while true do atomic a do atomic b do
> a[1] := b[1]; od; od;
> SignalSemaphore(c); od;

--- Switching to thread 5
[5] gap> while true do atomic b do atomic a do
[5] > SignalSemaphore(c); od;
Locking is useful, but what if two threads wait for each other’s lock:

gap> a := ShareSpecialObj([1,2,3]);;
gap> b := ShareSpecialObj([1,2,3]);;
gap> c := CreateSemaphore(0);
<semaphore 0xb557060: count = 0>
gap> while true do atomic a do atomic b do
>   a[1] := b[1]; od; od;
>   SignalSemaphore(c); od;

When only one loop runs, c will increase steadily.
Locking is useful, but what if two threads wait for each other’s lock:

```
gap> a := ShareSpecialObj([1,2,3]);;
gap> b := ShareSpecialObj([1,2,3]);;
gap> c := CreateSemaphore(0);<semaphore 0xb557060: count = 0>
gap> while true do atomic a do atomic b do
  > a[1] := b[1]; od; od;
> SignalSemaphore(c); od;
```

--- Switching to thread 5
```
[5] gap> while true do atomic b do atomic a do
[5]   > SignalSemaphore(c); od;
```

When only one loop runs, \(c\) will increase steadily.

When the second loop is started, everything will deadlock.
Solution:

- Use locking *only if necessary.*
Solution:

- Use locking *only if necessary*.
- Lock *briefly*, release *quickly*.
Solution:

- Use locking only if necessary.
- Lock briefly, release quickly.
- If you need to lock two things, use only one atomic statement.
Solution:

- Use locking *only if necessary*.
- Lock *briefly*, release *quickly*.
- If you need to lock *two things*, use *only one atomic statement*.
- If this is impossible, *always lock in the same order*. 
Solution:

- Use locking **only if necessary**.
- Lock **briefly**, release **quickly**.
- If you need to lock **two things**, use **only one** atomic statement.
- If this is impossible, always lock in the same order.
- GAP’s **region precedence** should protect you.
Solution:

- Use locking *only if necessary*.
- Lock *briefly*, release *quickly*.
- If you need to lock *two things*, use *only one atomic statement*.
- If this is impossible, *always lock in the same order*.
- GAP’s *region precedence* should protect you.
- *Do not use* ShareSpecialObj!
Solution:

- Use locking **only if necessary**.
- Lock **briefly**, release **quickly**.
- If you need to lock **two things**, use **only one atomic statement**.
- If this is impossible, **always lock in the same order**.
- GAP’s **region precedence** should protect you.
- **Do not use** ShareSpecialObj!

**HPCGAP offers**: deadlock protection, region precedence
Basic problem: Data is in one place but is needed in another.
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!
**Basic problem:** Data is in one place but is needed in another. This is mostly a *distributed memory problem*, but *not exclusively!*.

Moving data, or *communication* becomes necessary.
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.

Solution:

- reorganise algorithms to keep data local,
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.

Solution:

- reorganise algorithms to keep data local,
- keep your caches in mind,
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.

Solution:
- reorganise algorithms to keep data local,
- keep your caches in mind,
- use buffers and queueing to avoid latency,
Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.

Solution:
- reorganise algorithms to keep data local,
- keep your caches in mind,
- use buffers and queueing to avoid latency,
- estimate communication needs, compare with computational throughput.
Communication

Basic problem: Data is in one place but is needed in another.

This is mostly a distributed memory problem, but not exclusively!

Moving data, or communication becomes necessary.

Communication takes time, bandwidth is limited as well as latency.

See talk about parallel orbit enumeration.

Solution:

- reorganise algorithms to keep data local,
- keep your caches in mind,
- use buffers and queueing to avoid latency,
- estimate communication needs, compare with computational throughput.

HPCGAP offers: shared memory model, fast object serialisation, access to fast networking using MPI and ZeroMQ
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, **RAM is not all the same.**
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, **RAM is not all the same.**

**But it gets worse ...**

When I was a child, microprocessors ran at speeds like **1 MHz.**
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz:
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years. Back then, reading one byte from memory took about 300 ns.
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.

This is in many cases only about 42 times faster in 30 years.
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

**But it gets worse ...**

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.

This is in many cases only about 42 times faster in 30 years.

As if this is not bad enough: in modern machines, multiple cores share this bandwidth!
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse . . .

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.

This is in many cases only about 42 times faster in 30 years.

As if this is not bad enough: in modern machines, multiple cores share this bandwidth!

In *lovelace* and *babbage*, 8 cores share 64 GB,
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

But it gets worse …

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.

This is in many cases only about 42 times faster in 30 years.

As if this is not bad enough: in modern machines, multiple cores share this bandwidth!

In *lovelace and babbage*, 8 cores share 64 GB, access to “remote memory” is considerably slower.
As Markus explained, on a NUMA (Non-Uniform Memory Access) machine, RAM is not all the same.

**But it gets worse ...**

When I was a child, microprocessors ran at speeds like 1 MHz. Today, a typical clock rate is 2 GHz: 2000 times faster in 30 years.

Back then, reading one byte from memory took about 300 ns. Today, reading the first word in a new place takes 7 ns and subsequent words take 0.5 ns each.

This is in many cases only about 42 times faster in 30 years.

As if this is not bad enough: in modern machines, multiple cores share this bandwidth!

In *lovelace* and *babbage*, 8 cores share 64 GB, access to “remote memory” is considerably slower.

This is called the *memory wall*.
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
- be aware or beware of NUMA,
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
- be aware or beware of NUMA,
- maybe use more explicit data movement rather than global shared memory assumptions — the distributed model is back!
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
- be aware or beware of NUMA,
- maybe use more explicit data movement rather than global shared memory assumptions — the distributed model is back!
- This will also be relevant for GPU computing.
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
- be aware or beware of NUMA,
- maybe use more explicit data movement rather than global shared memory assumptions — the distributed model is back!
- This will also be relevant for GPU computing.
- Sometimes, explicit copying provides you with locality.
Solution:

- view memory access bandwidth and latency as additional scarce resources to manage,
- use your caches on all levels,
- be aware or beware of NUMA,
- maybe use more explicit data movement rather than global shared memory assumptions — the distributed model is back!
- This will also be relevant for GPU computing.
- Sometimes, explicit copying provides you with locality.

HPCGAP offers: thread local allocation, parallel garbage collection, MPI and ZeroMQ for explicit communication.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
Debugging parallel programs

Debugging parallel programs can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
Debugging a parallel program can be a *nightmare*:
- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
- Time measurements and profiling are more difficult.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
- Time measurements and profiling are more difficult.

Solution:

- There is no really good solution.
Debugging parallel programs

Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
- Time measurements and profiling are more difficult.

Solution:

- There is no really good solution.
- Explicit synchronisation can help.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
- Time measurements and profiling are more difficult.

Solution:

- There is no really good solution.
- Explicit synchronisation can help.
- Waiting for communication or locks is often the problem.
Debugging a parallel program can be a nightmare:

- Many things happen at the same time.
- Usually one does not even have a terminal for each thread.
- Correct behaviour sometimes depends on more or less random order of some events (race conditions).
- Runs are often not reproducible.
- In a parallel program there is no notion of single stepping.
- Heisenbugs are common.
- Often one has worse than expected performance and has to find the reason.
- Time measurements and profiling are more difficult.

Solution:

- There is no really good solution.
- Explicit synchronisation can help.
- Waiting for communication or locks is often the problem.

HPCGAP offers: nice UI and break loops for individual threads.