Local (“Arena”)
Memory Allocators

John Lakos

Thursday, April 27, 2017

This version is for ACCU’17

Copyright Notice

Abstract

The runtime implications of the physical location of allocated memory are sometimes overlooked—even in the most performance-critical code. In this talk, we will examine how the performance of long-running systems can degrade when using just one global allocator (e.g., via new/delete). We will contrast the use of global allocators with various kinds of *local* allocators—allocators that allocate memory for a well-defined subset of objects in the system. We will also demonstrate how local allocators can reduce, if not entirely prevent, degradation seen in systems that rely solely on the global allocator. Six dimensions—*fragmentability*, *allocation density*, *variation*, *utilization*, *locality*, and *contention*—will be introduced to help characterize a given subsystem, assess the potential for accelerating its runtime performance, and—where appropriate—aid in determining the best local allocator to do so **Empirical evidence** will be presented to demonstrate that introducing an appropriate local allocator can often result in substantial reductions in run times (compared with a similar system relying solely on just a single, global allocator).

Important Recurring Questions

Are memory allocators really worth the trouble?

- ☐ What situations merit their use?
- ☐ How are they applied effectively?
- ☐ What’s the performance impact?

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

1. Introduction and Background

Important Questions

Why do we like the C++ language?

- ☐ It enables us to "fine-tune" at a low level when needed.
- ☐ It can deliver very high runtime performance.

Important Questions

Why do (should) we care about memory allocators?

- ☐ They enable us to “fine-tune” at a low level when needed.
- ☐ They can help to improve runtime performance.

Important Questions

What are the benefits?

- ☐ Not all memory is alike:
  - ✓ Fast, Shared, Protected, Mapped
- ☐ Other qualitative benefits:
  - ✓ Testing, Debugging, Measuring
- ☐ Enhanced runtime performance:
  - ✓ Better Locality, Less Contention

1. Introduction and Background

Important Questions

What are the benefits?

(anecdotal)

Bear Stearns `(c. 1997)`

System's (coalescing) allocator optimized for allocation, not deallocation.

Important Questions

What are the benefits?

(anecdotal)

Bloomberg (c. 2002)

Process (static) memory saved/restored via memory-mapped IO.

Important Questions

What are the benefits?

(anecdotal)

Bloomberg `(c. 2006)`

User interfaces observed to be "zippier" when using local allocator.

Important Questions

What are the common arguments against?

- Requires more up-front design effort
- Complicates user interfaces
- May actually degrade performance:
    - No special allocator needed
    - Poorly chosen allocator supplied

Addressing Allocator Concerns

These are valid concerns!

- Requires more up-front design effort
- Complicates user interfaces
- May actually degrade performance:
  - No special allocator needed
  - Poorly chosen allocator supplied

They can be addressed only with:

☐ **Well-supported facts**

☐ **Careful measurement**

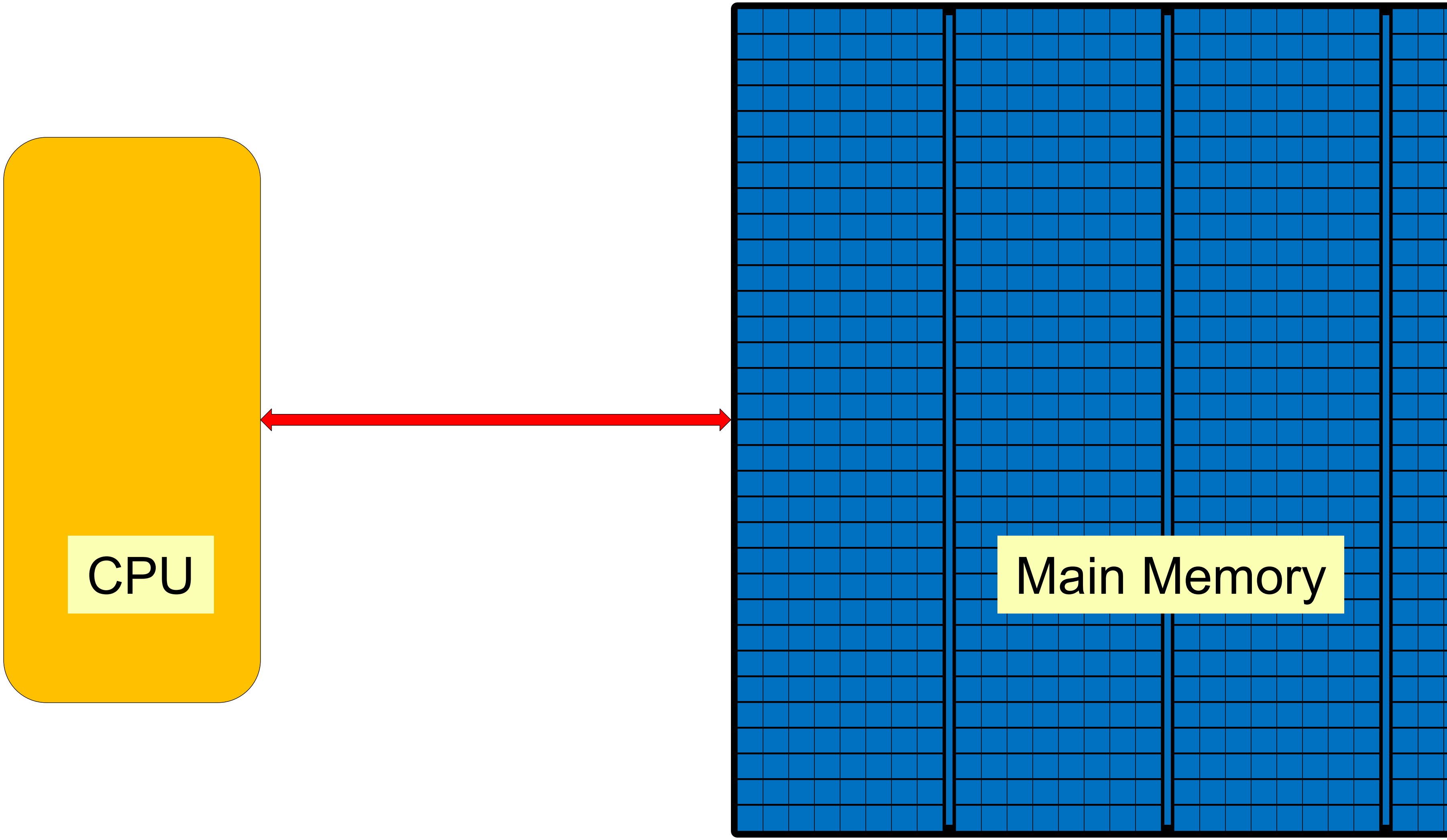
1. Introduction and Background

Review of Computer Memory

Main Memory

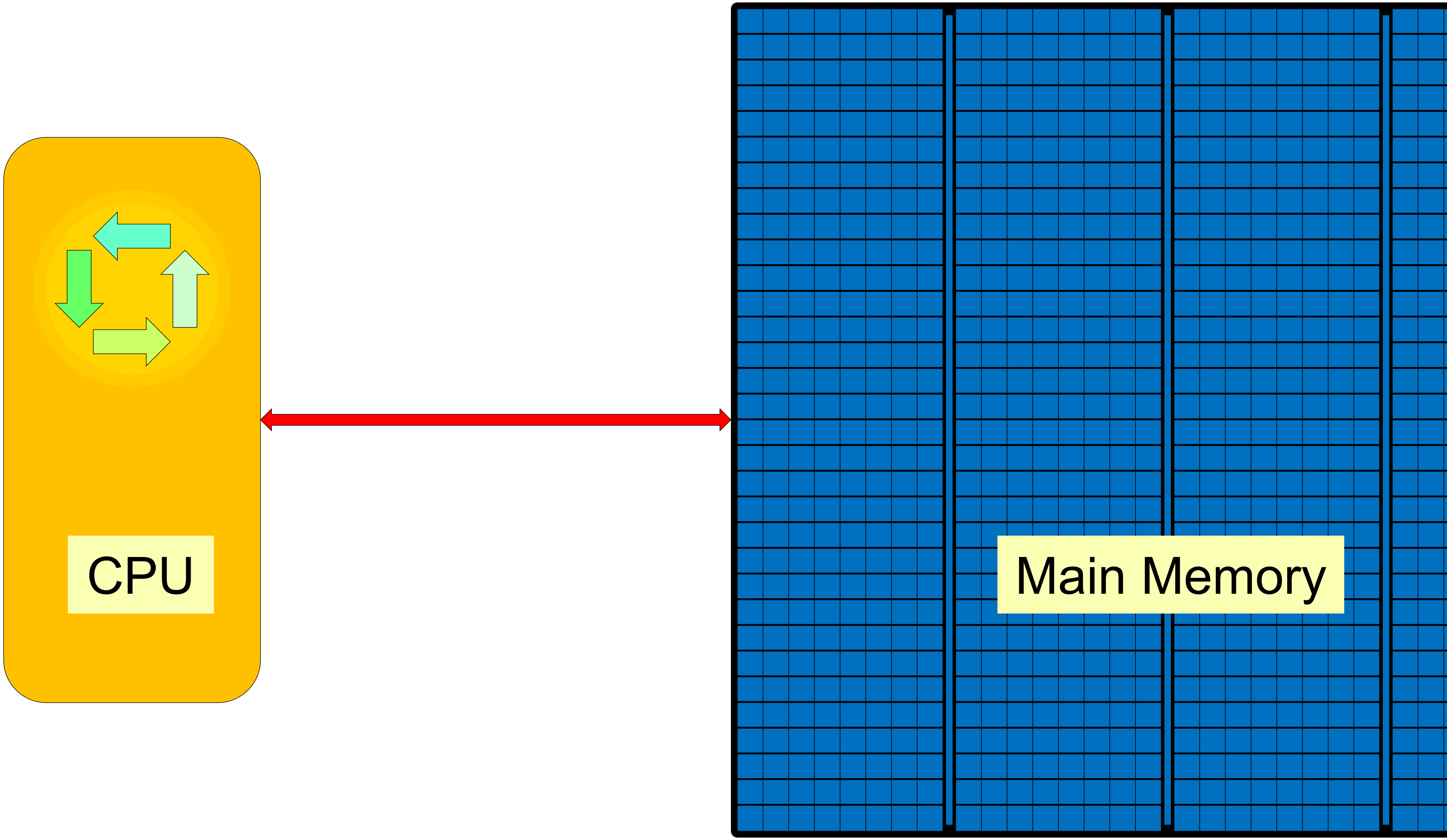
1. Introduction and Background

Review of Computer Memory

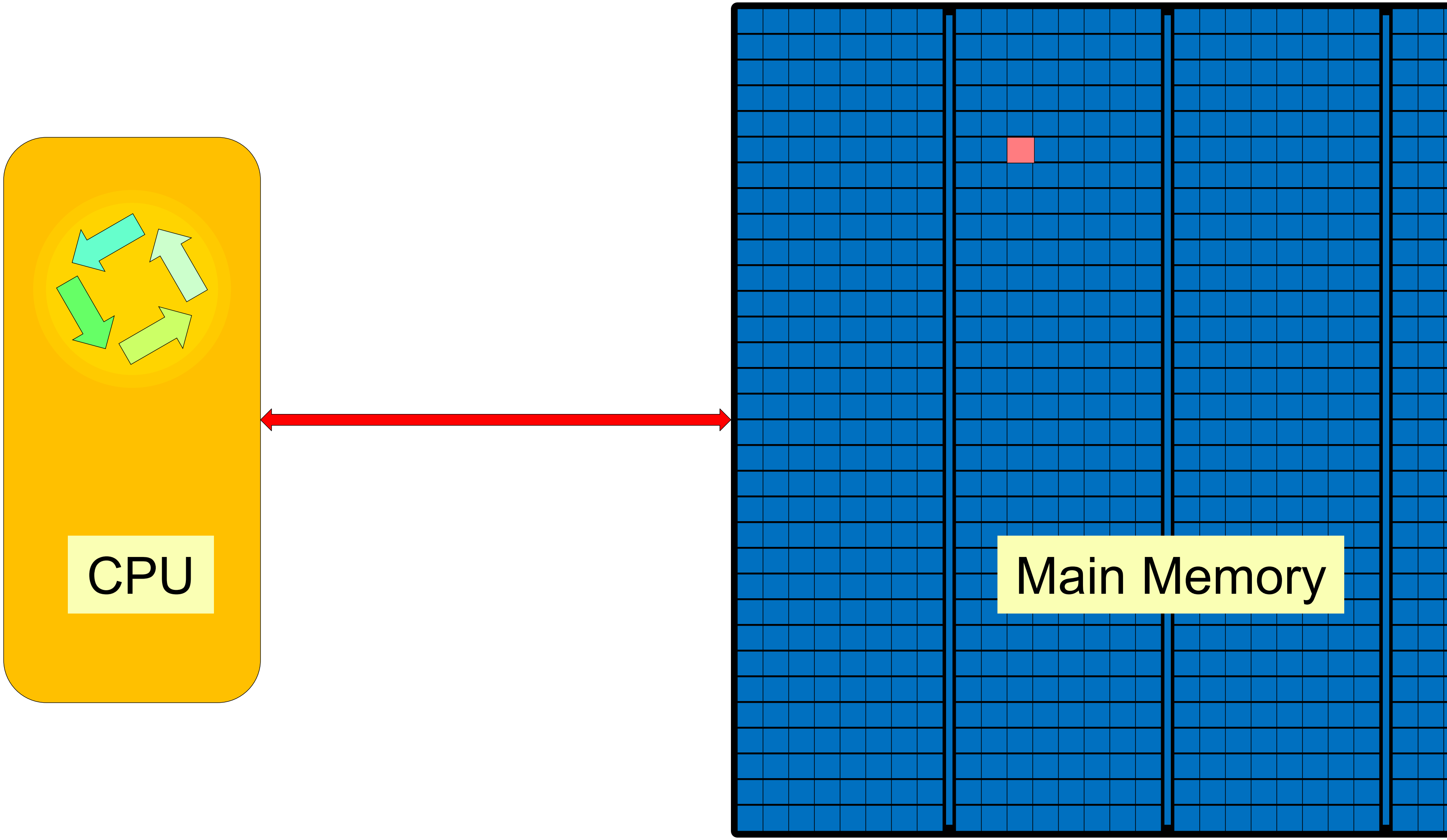


1. Introduction and Background

Review of Computer Memory



Review of Computer Memory

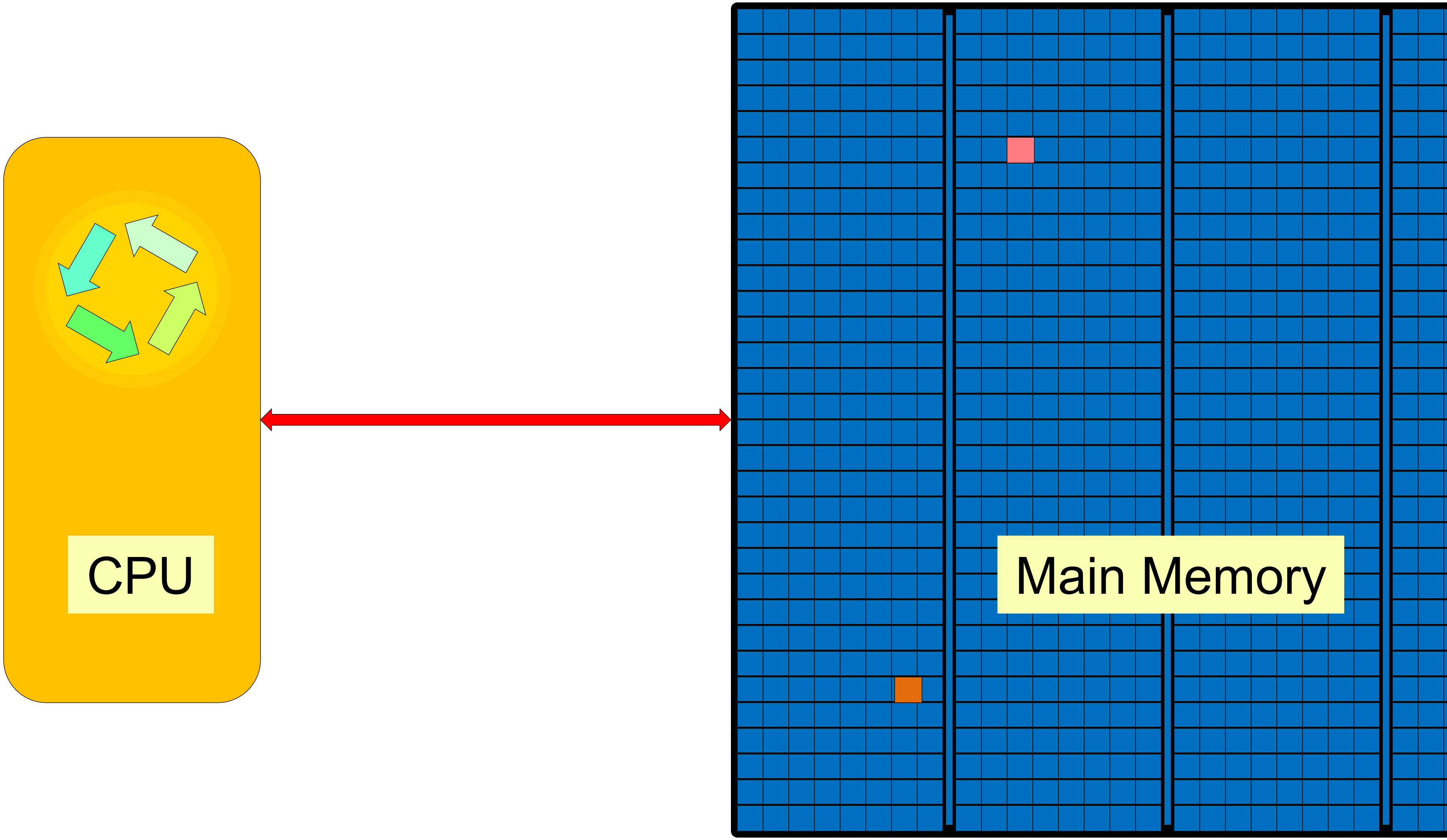


CPU

Main Memory

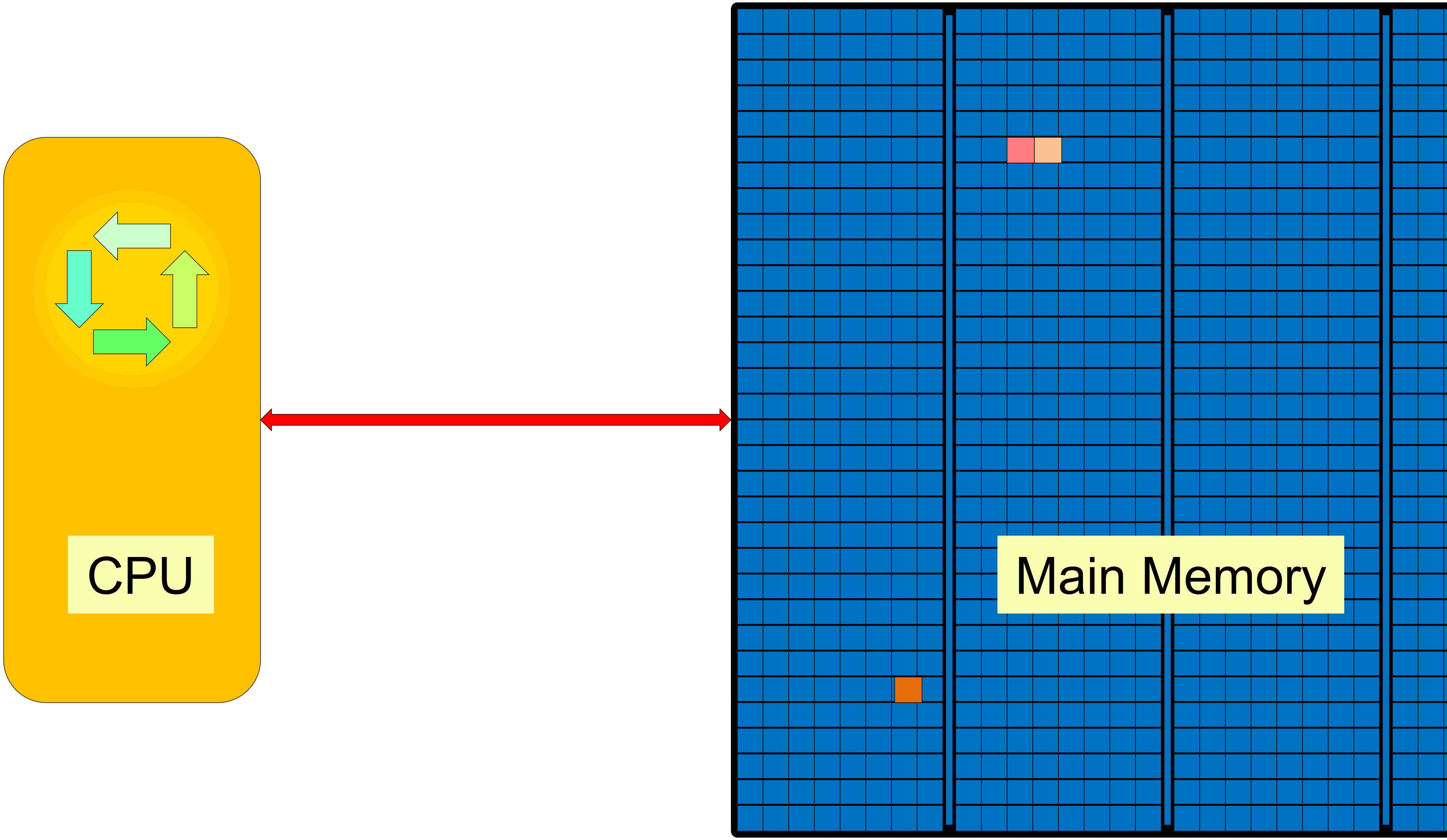
1. Introduction and Background

Review of Computer Memory



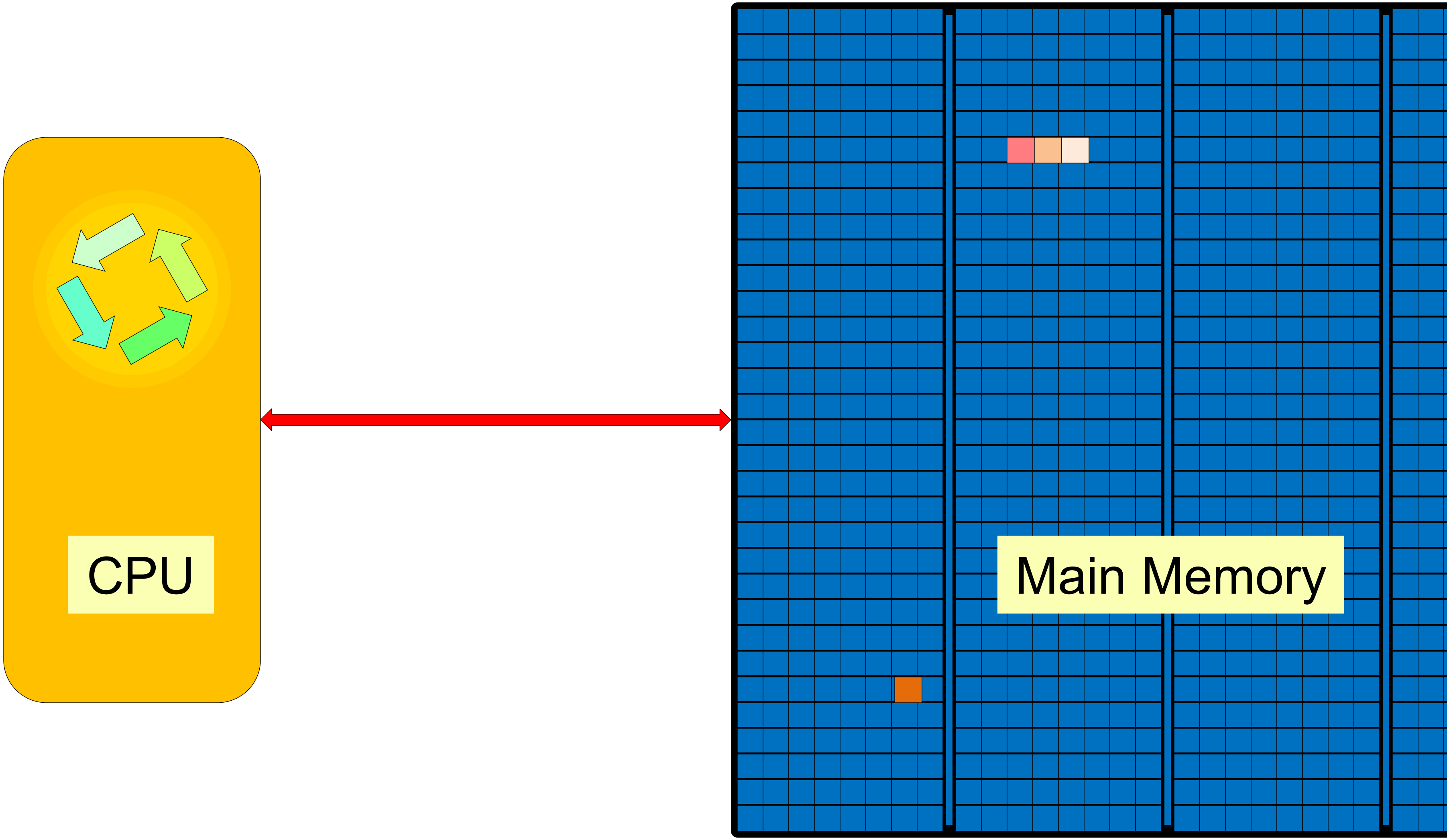
1. Introduction and Background

Review of Computer Memory



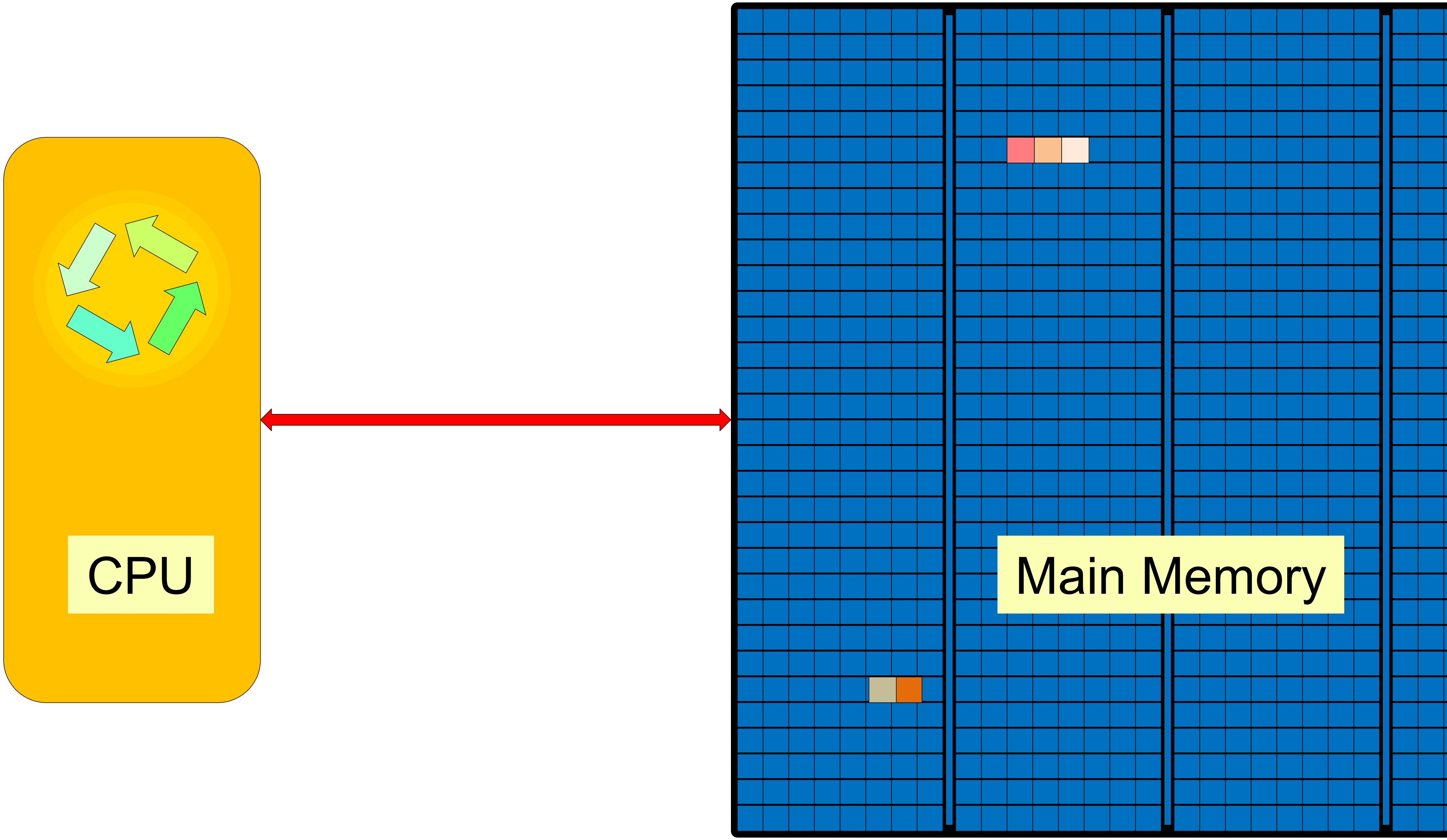
1. Introduction and Background

Review of Computer Memory



1. Introduction and Background

Review of Computer Memory



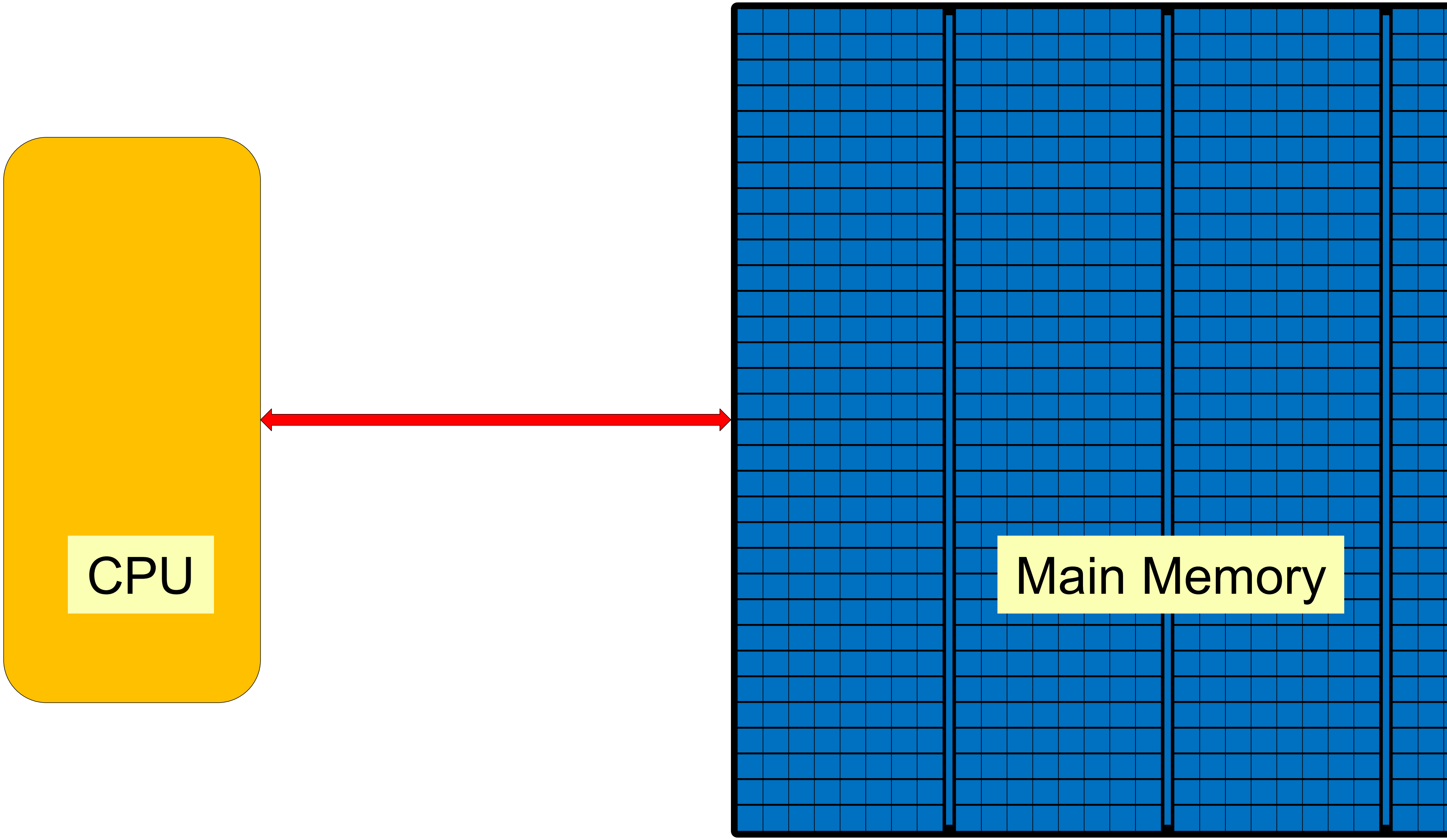
1. Introduction and Background

Review of Computer Memory

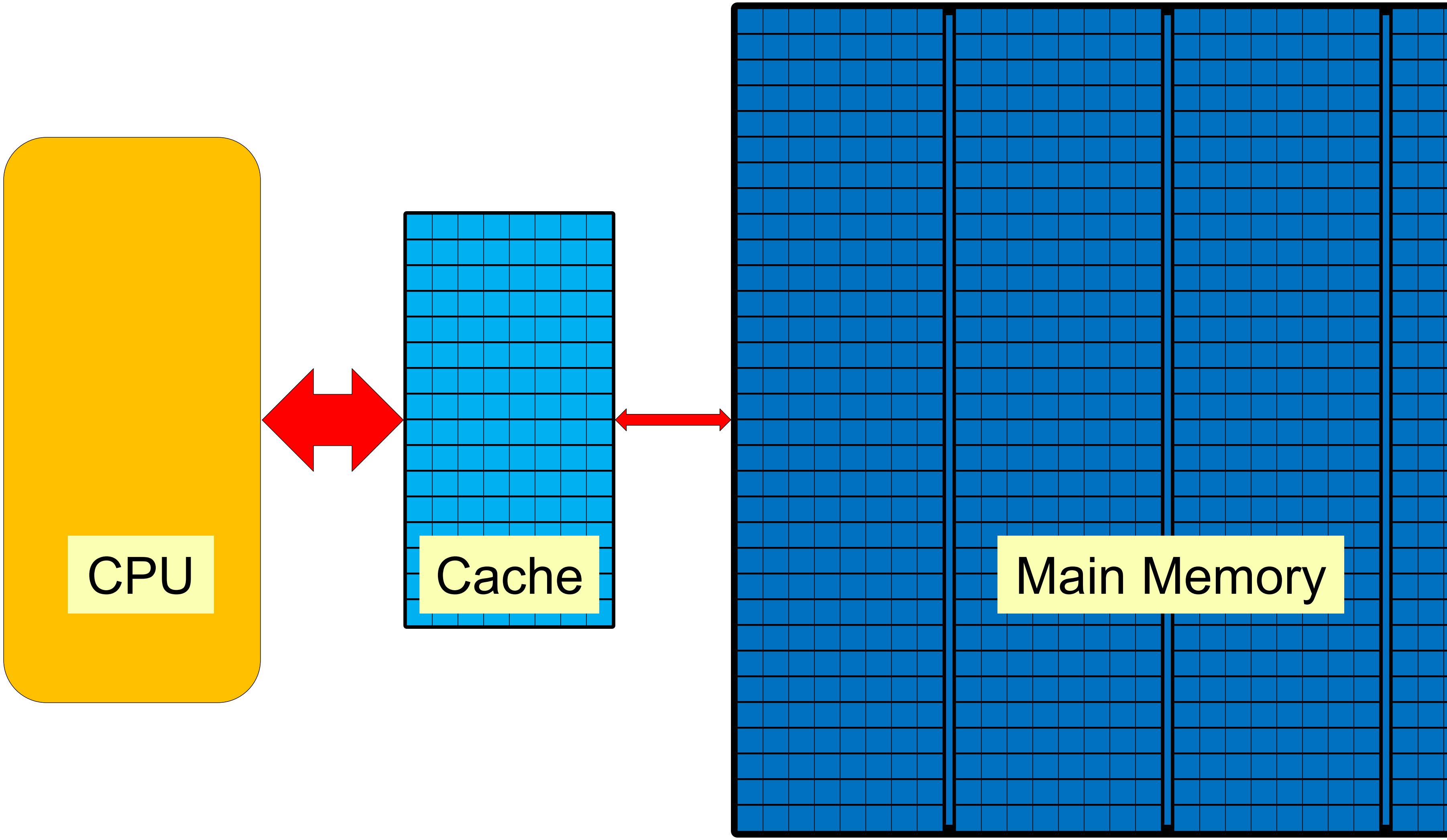
Cache Memory

1. Introduction and Background

Review of Computer Memory

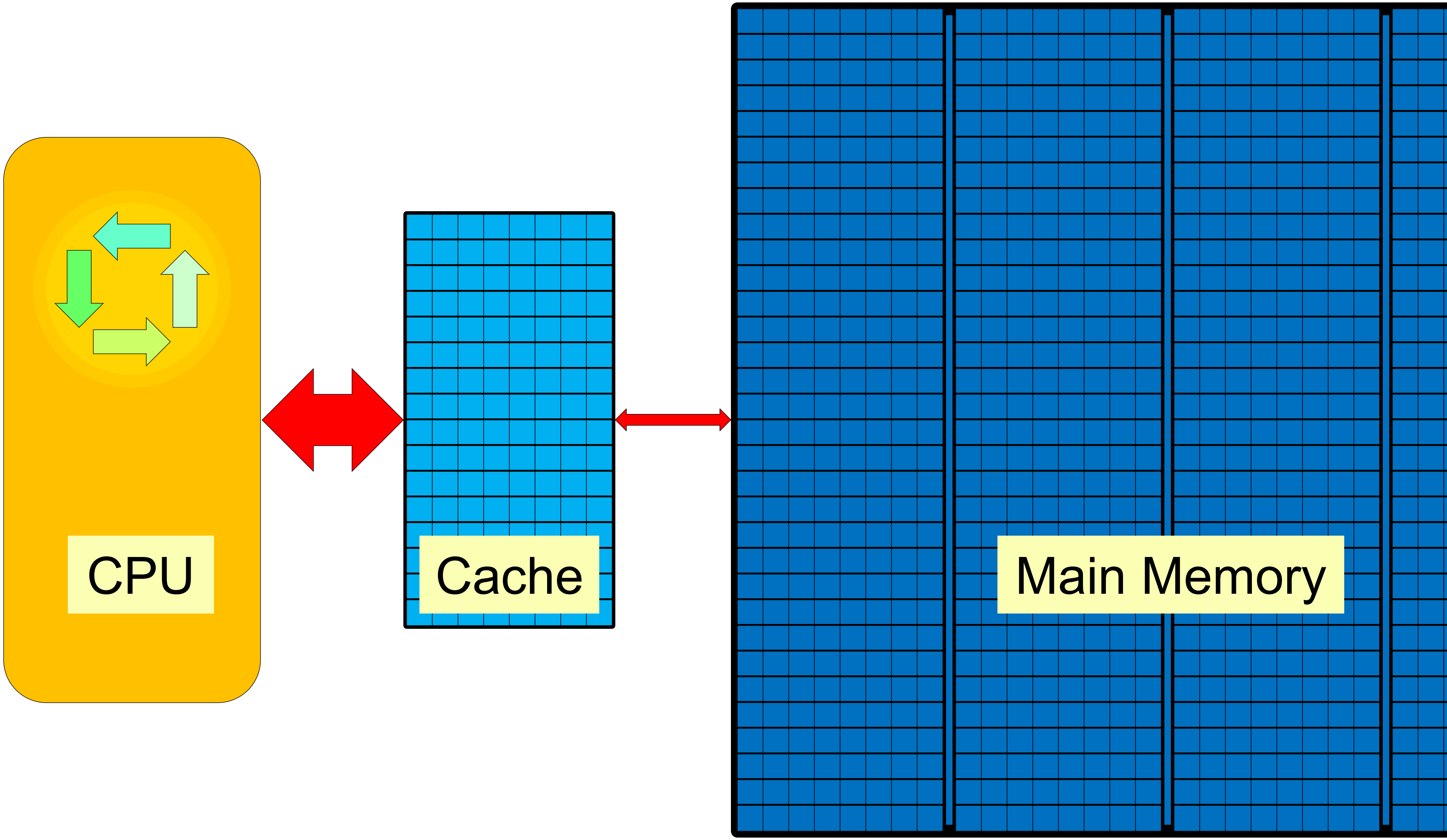


Review of Computer Memory



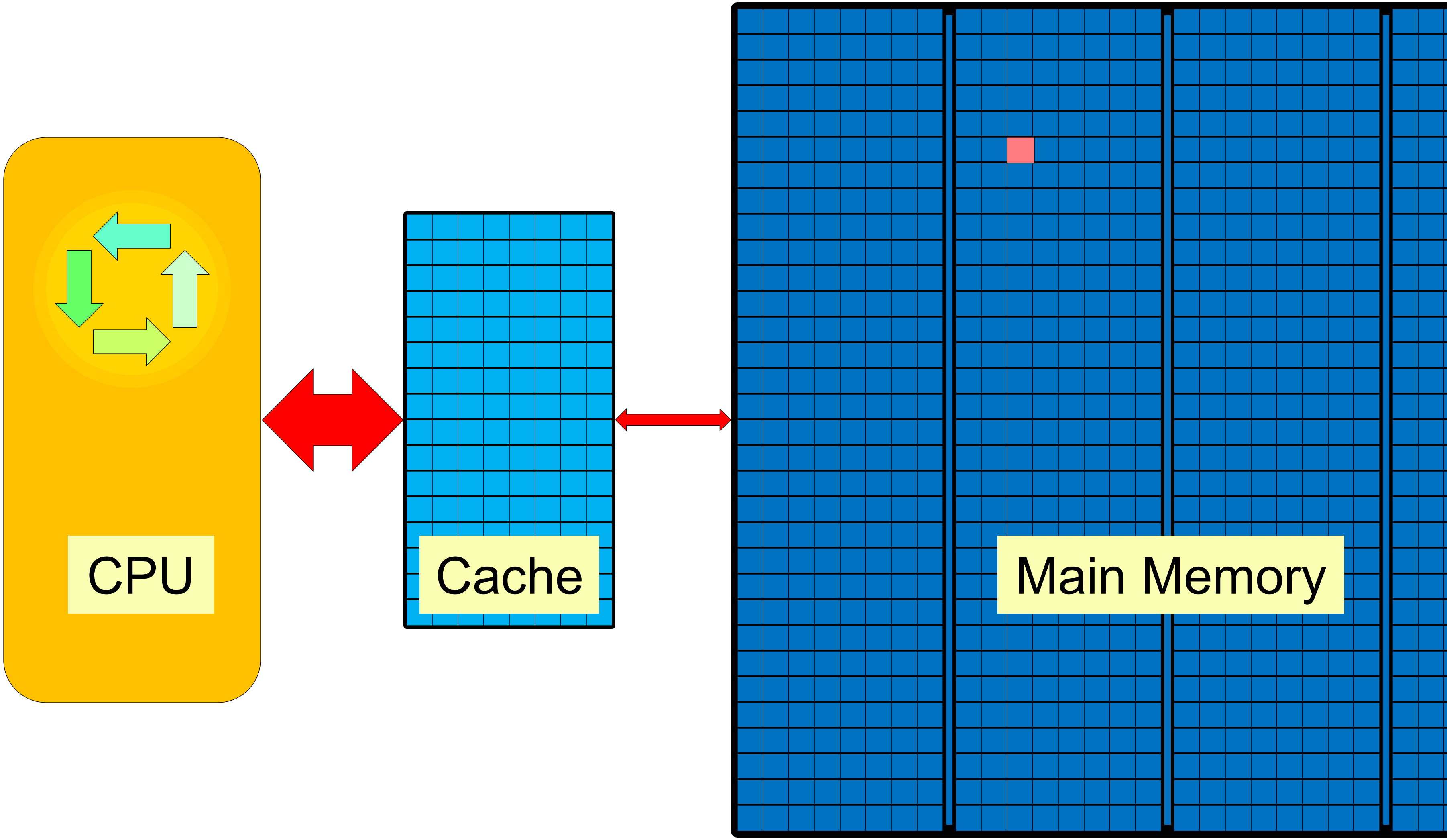
1. Introduction and Background

Review of Computer Memory



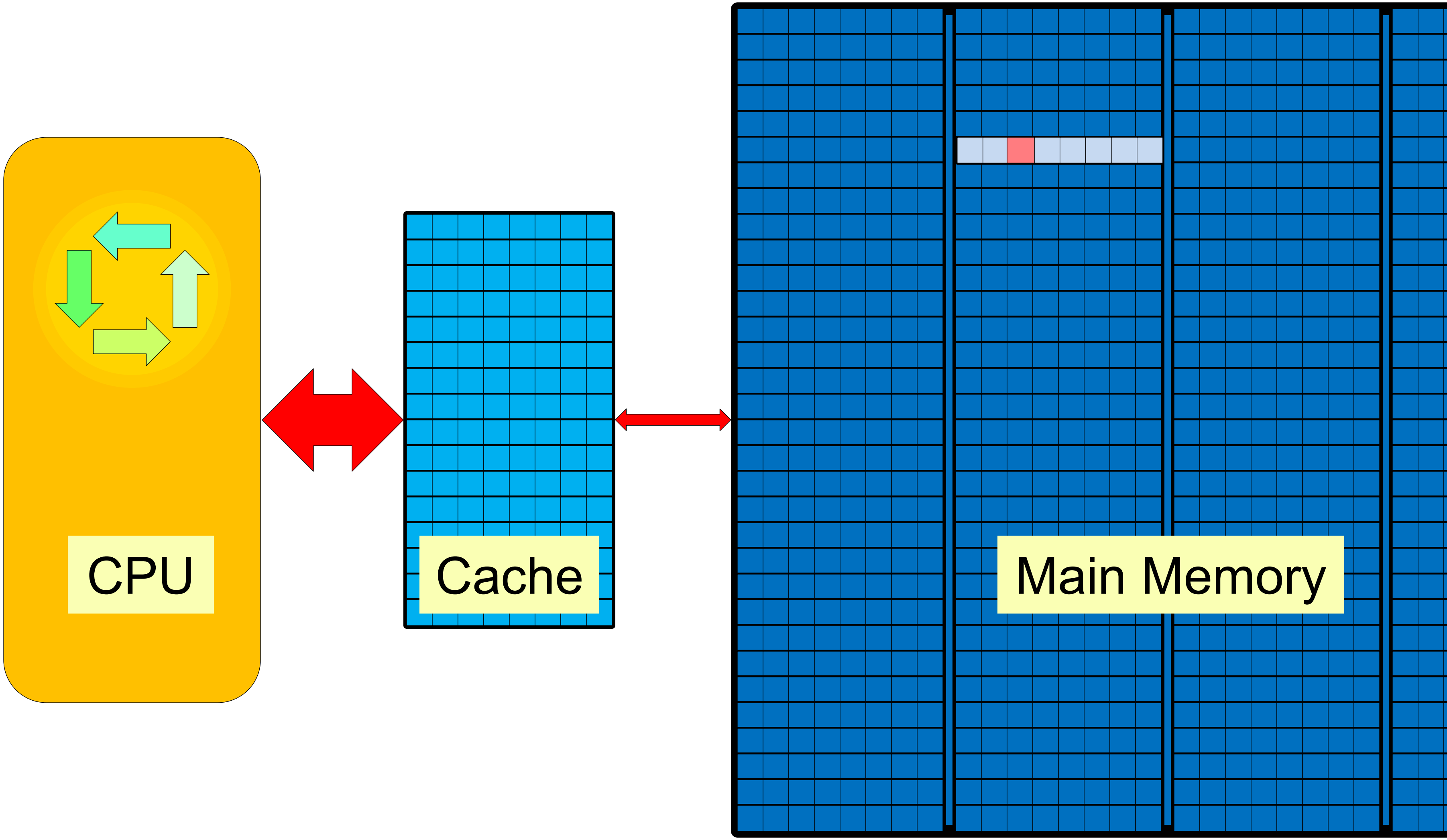
1. Introduction and Background

Review of Computer Memory



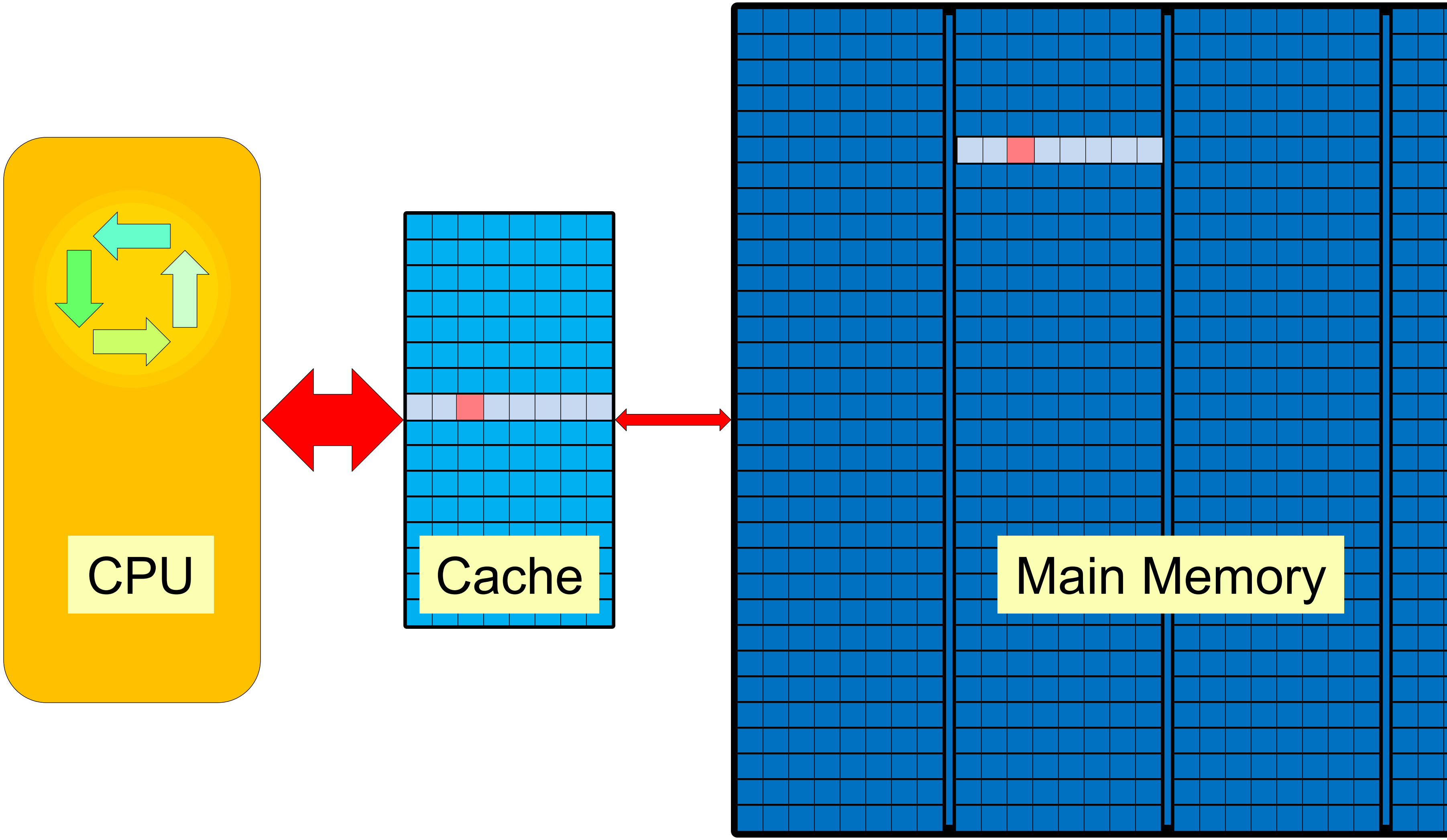
1. Introduction and Background

Review of Computer Memory



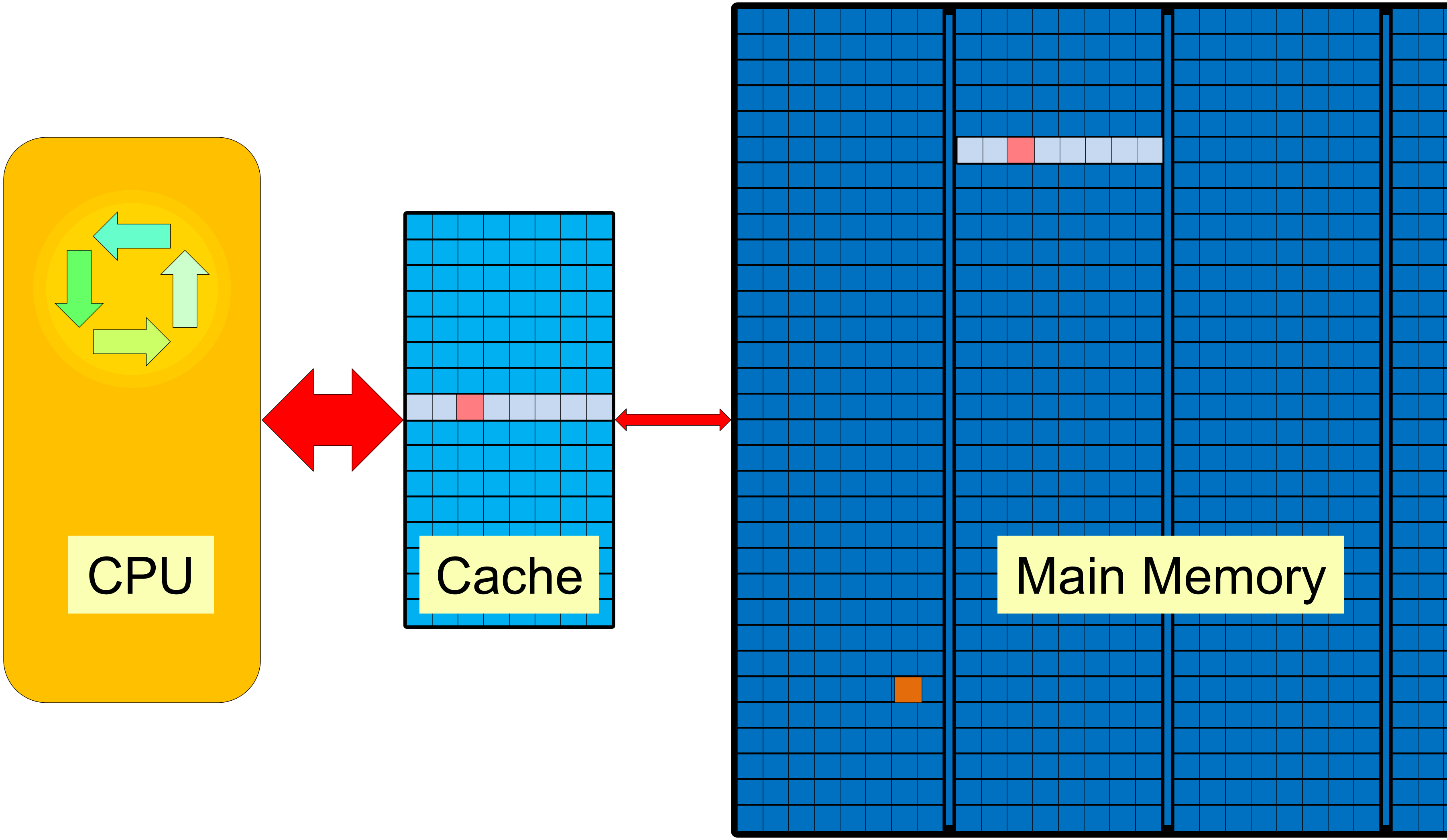
1. Introduction and Background

Review of Computer Memory



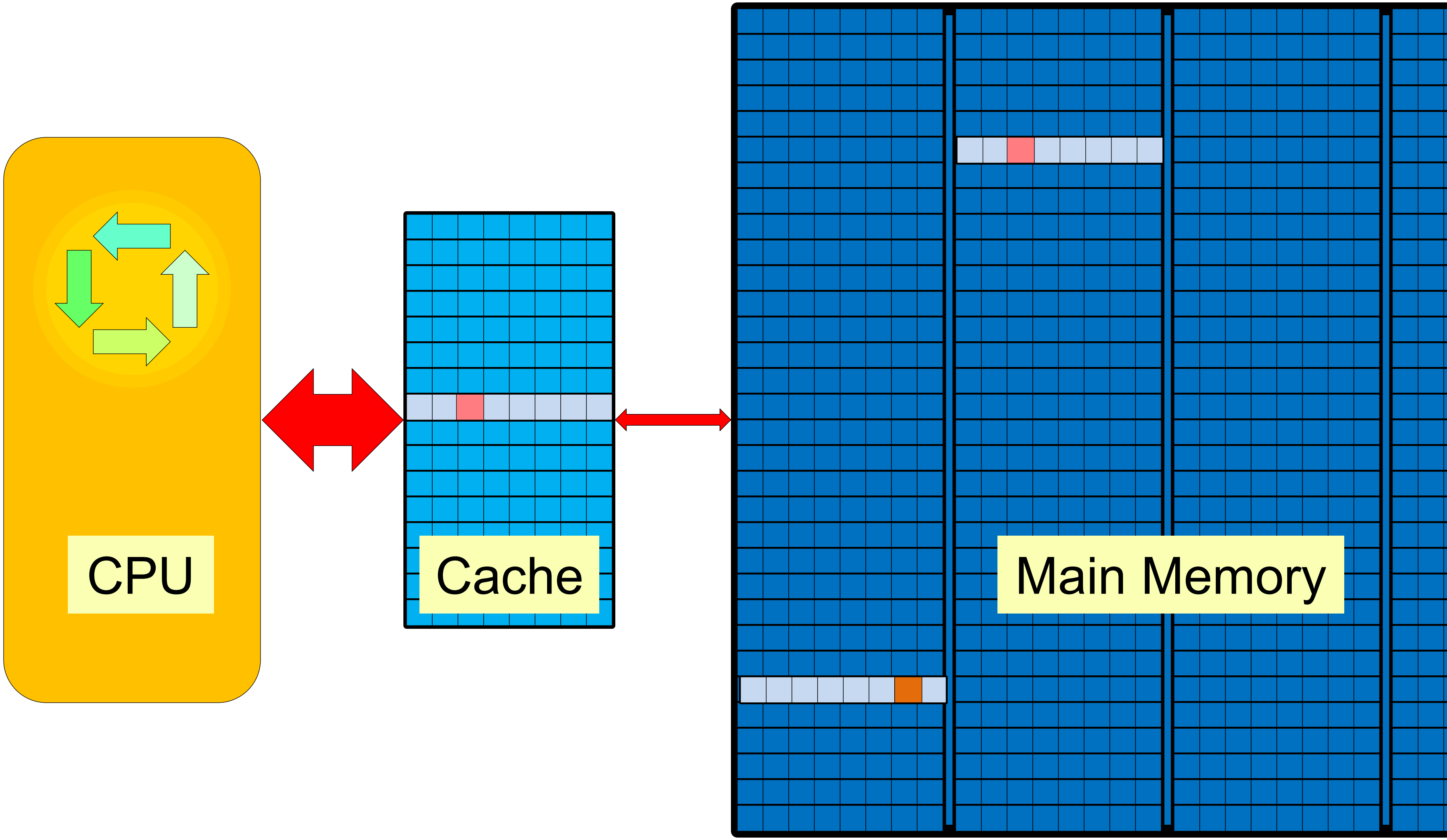
1. Introduction and Background

Review of Computer Memory



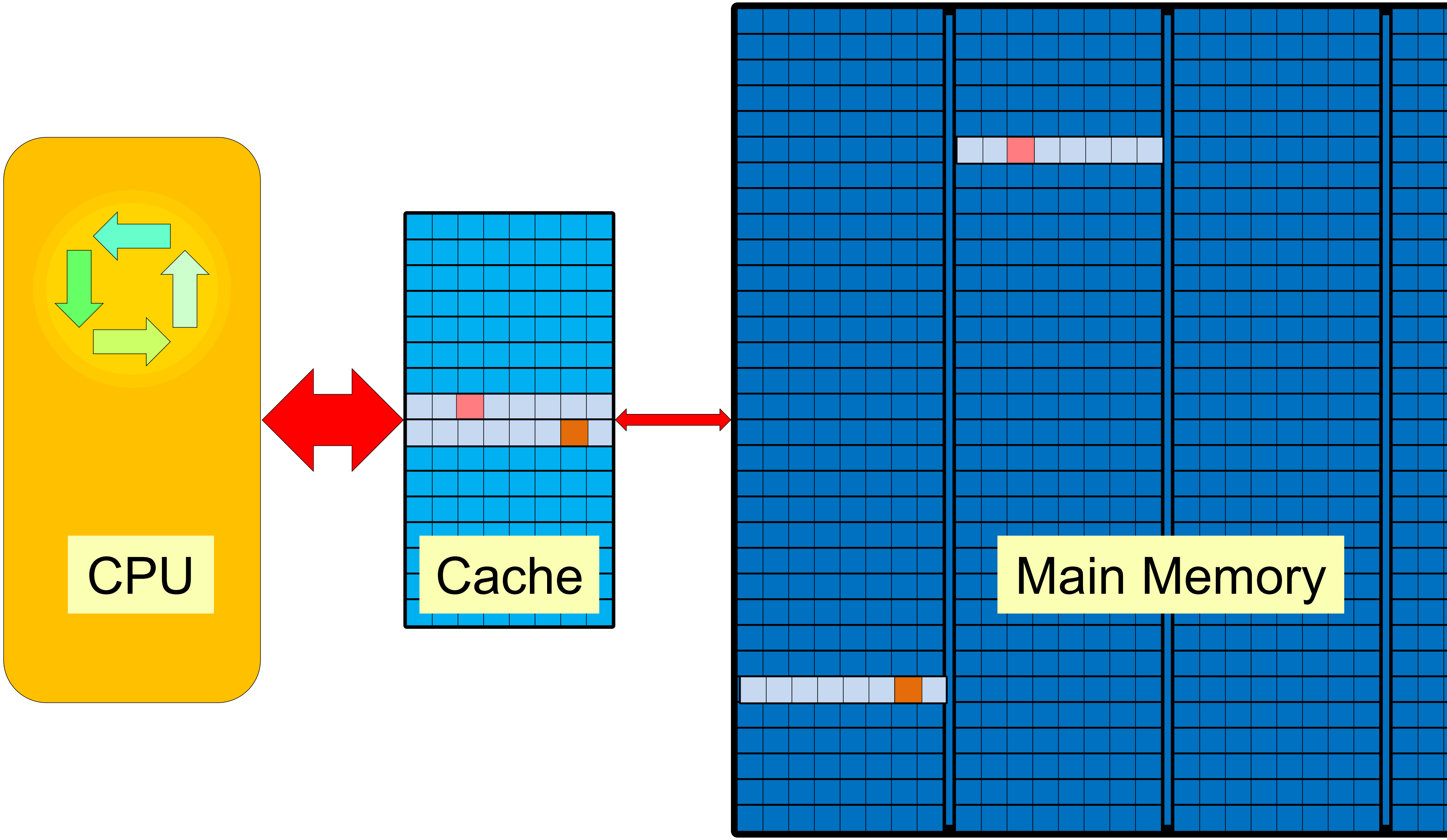
1. Introduction and Background

Review of Computer Memory



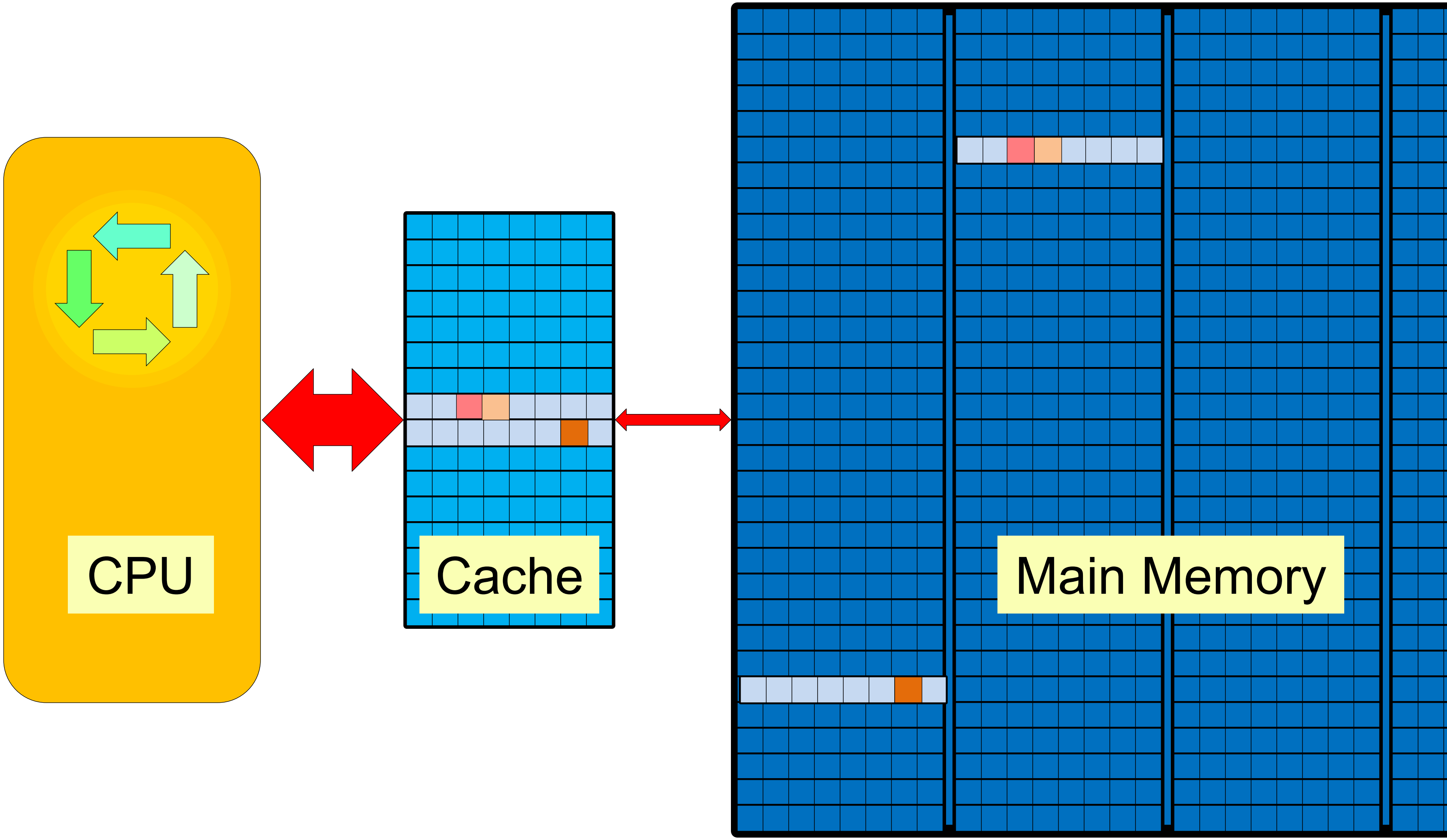
1. Introduction and Background

Review of Computer Memory



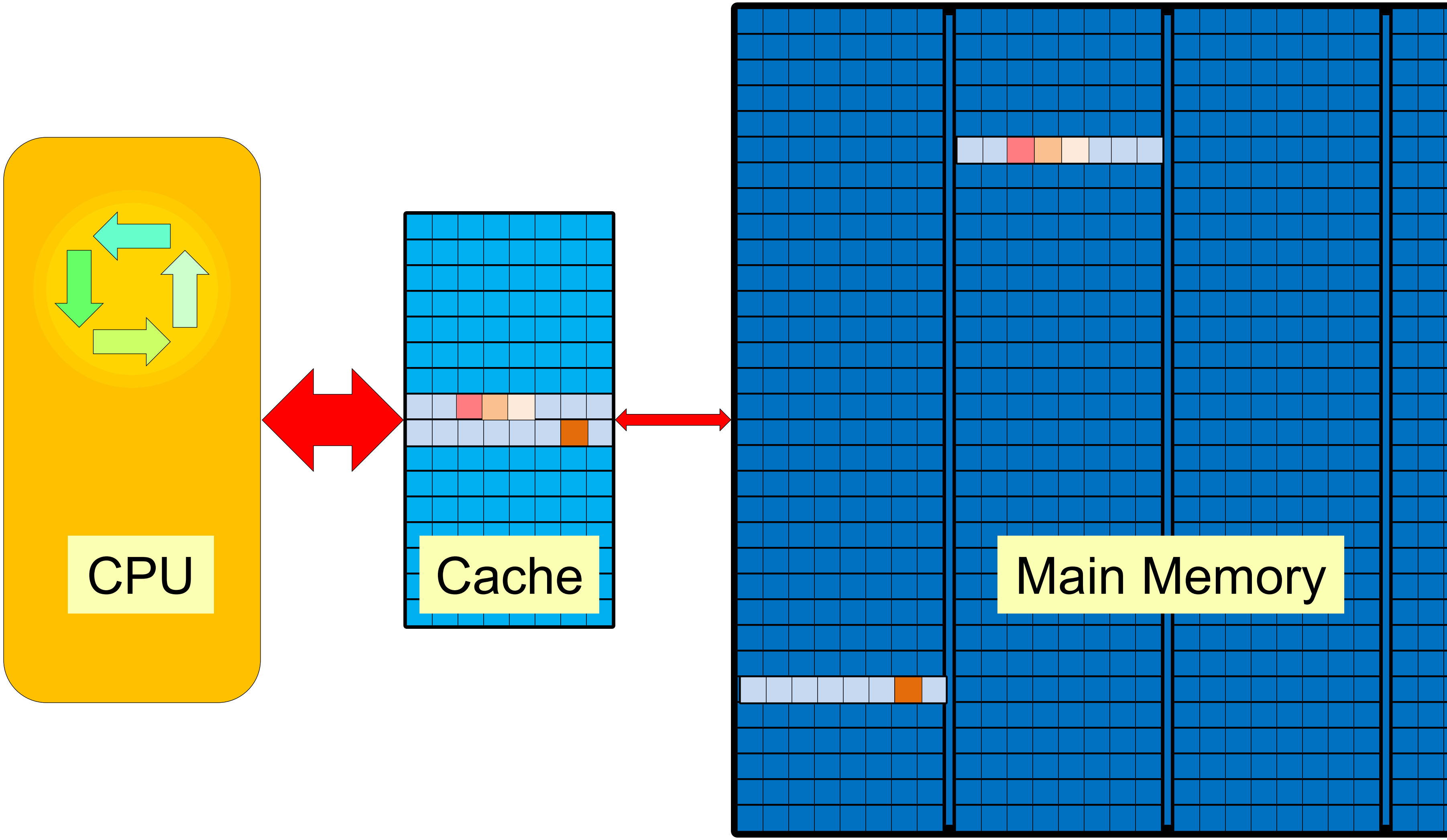
1. Introduction and Background

Review of Computer Memory



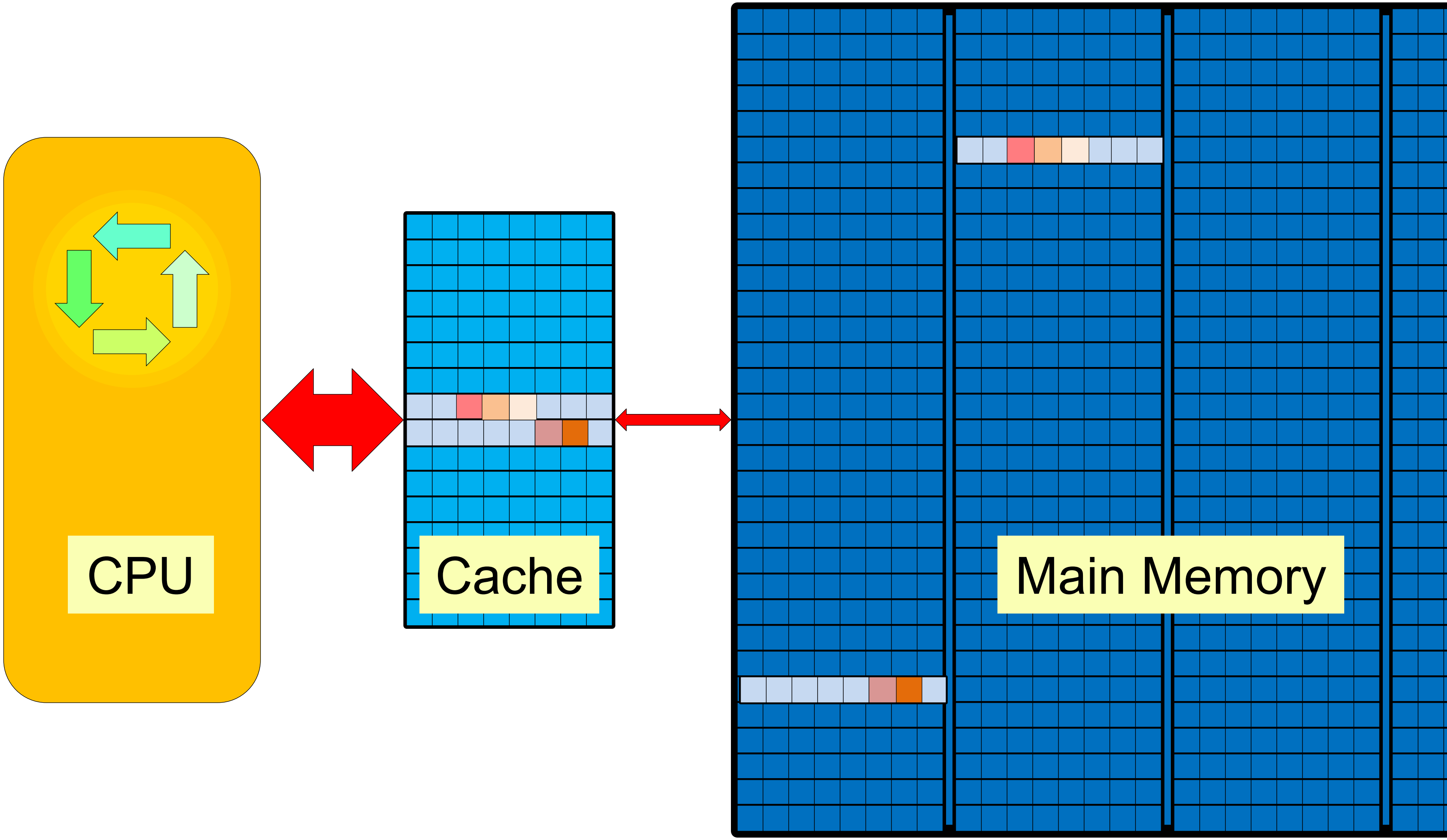
1. Introduction and Background

Review of Computer Memory



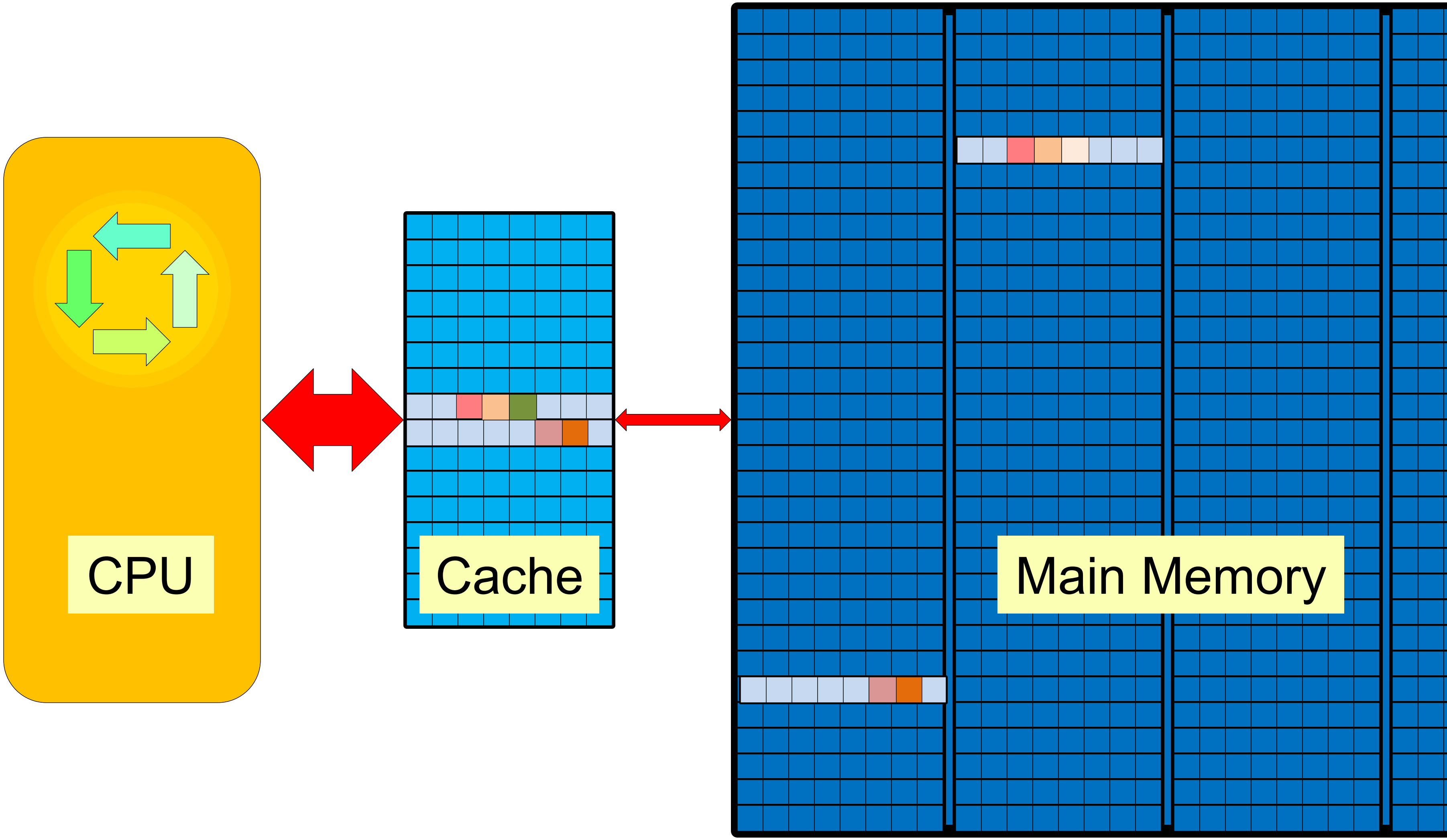
1. Introduction and Background

Review of Computer Memory



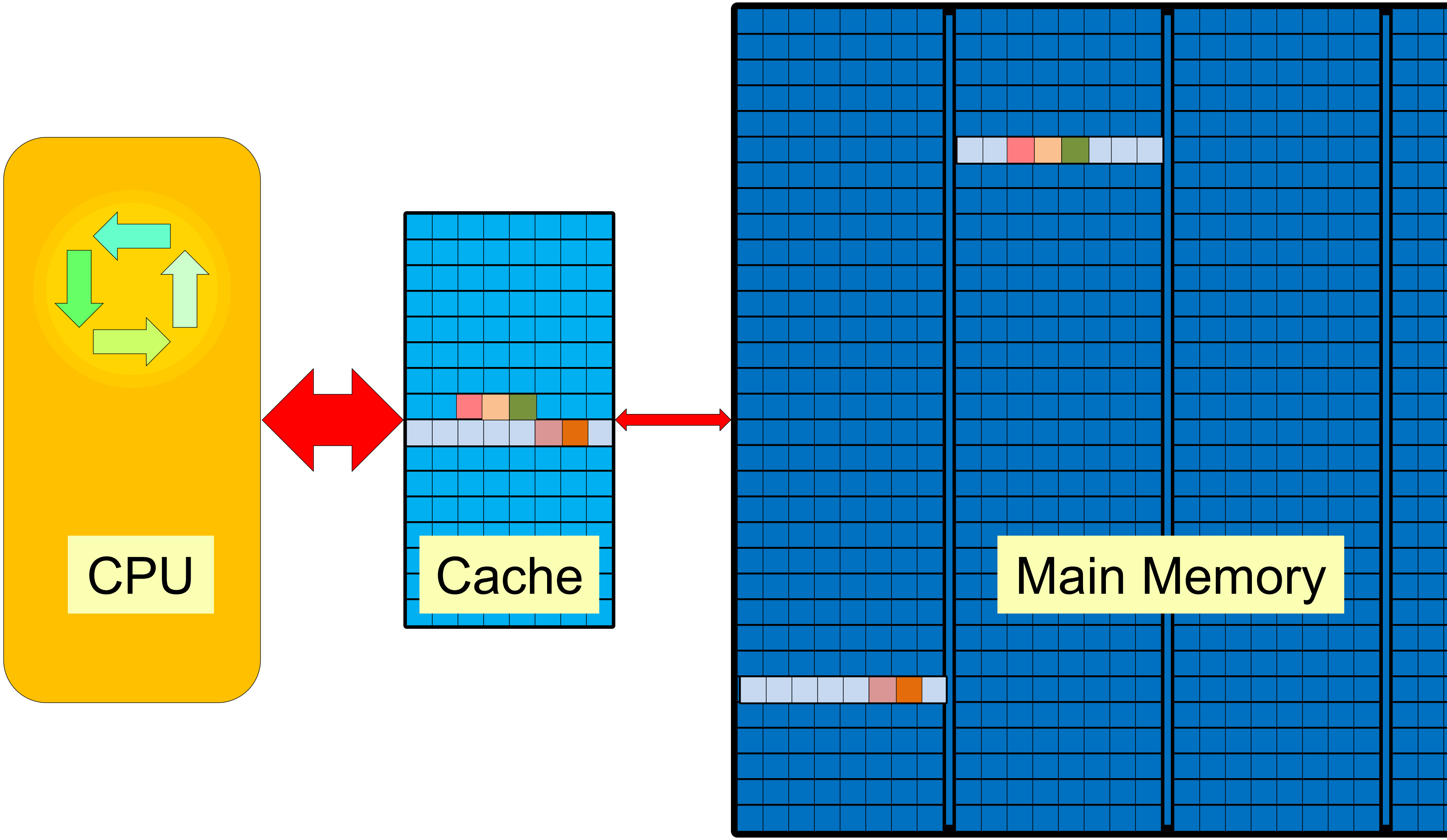
1. Introduction and Background

Review of Computer Memory



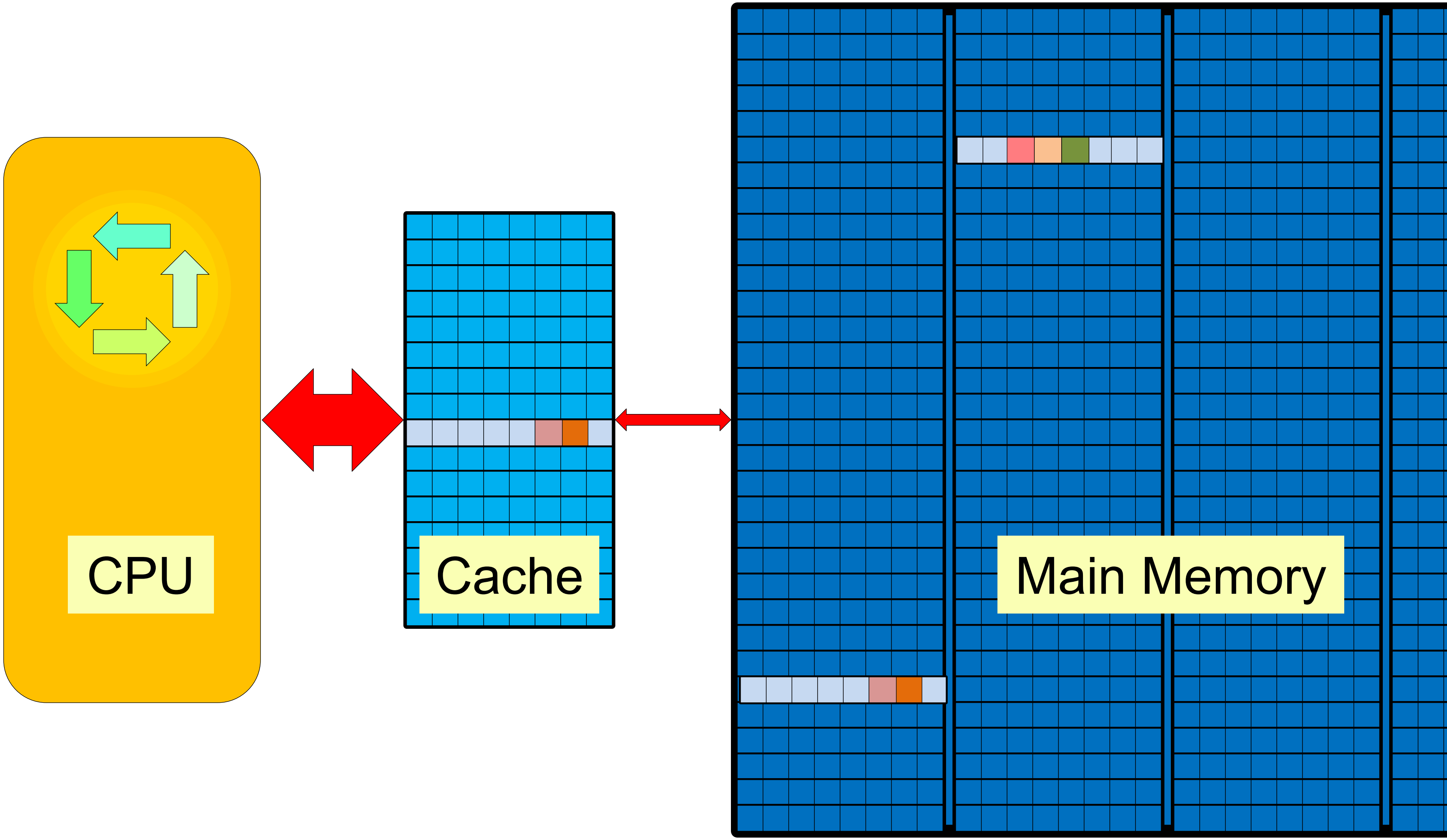
1. Introduction and Background

Review of Computer Memory



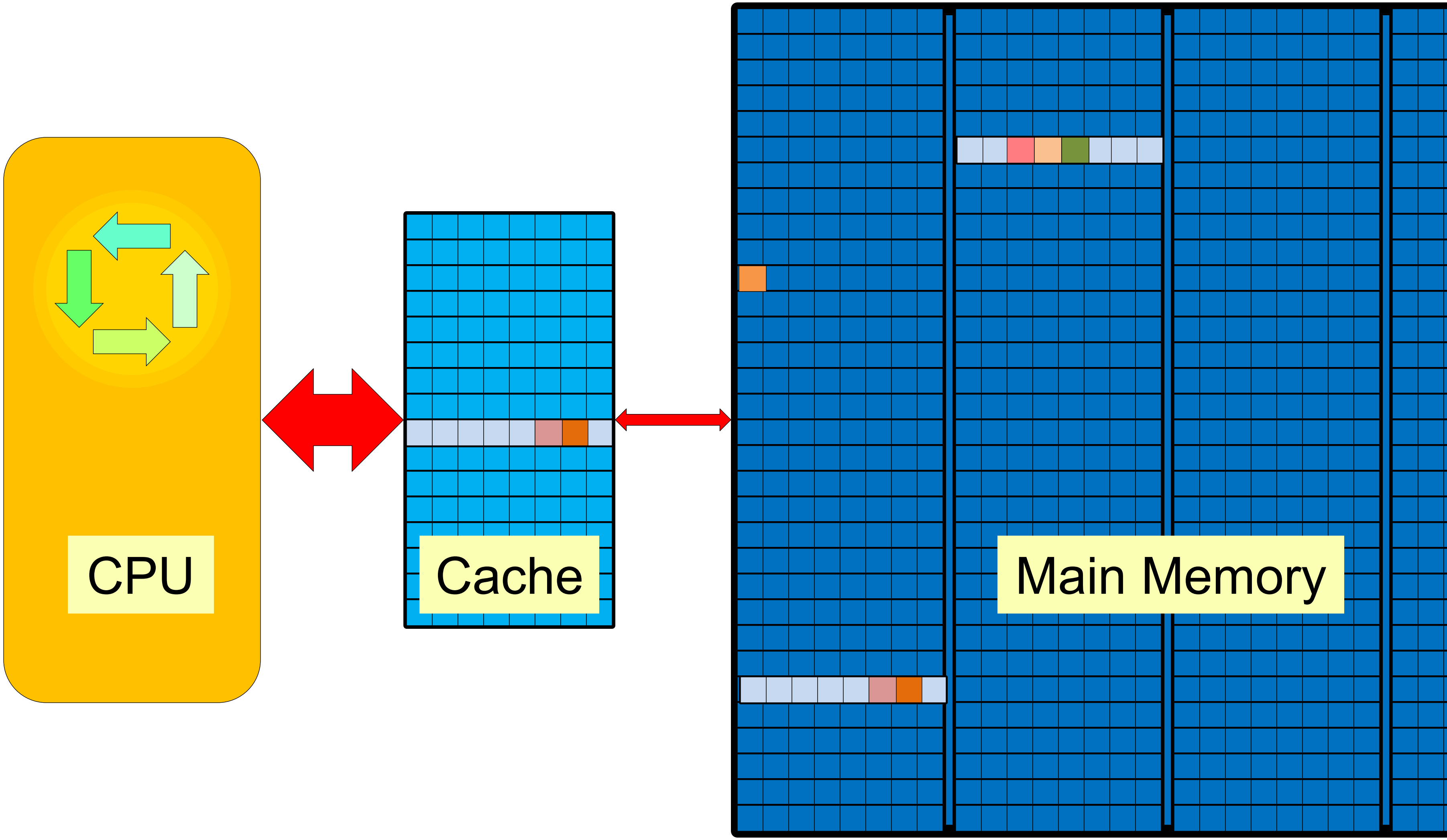
1. Introduction and Background

Review of Computer Memory



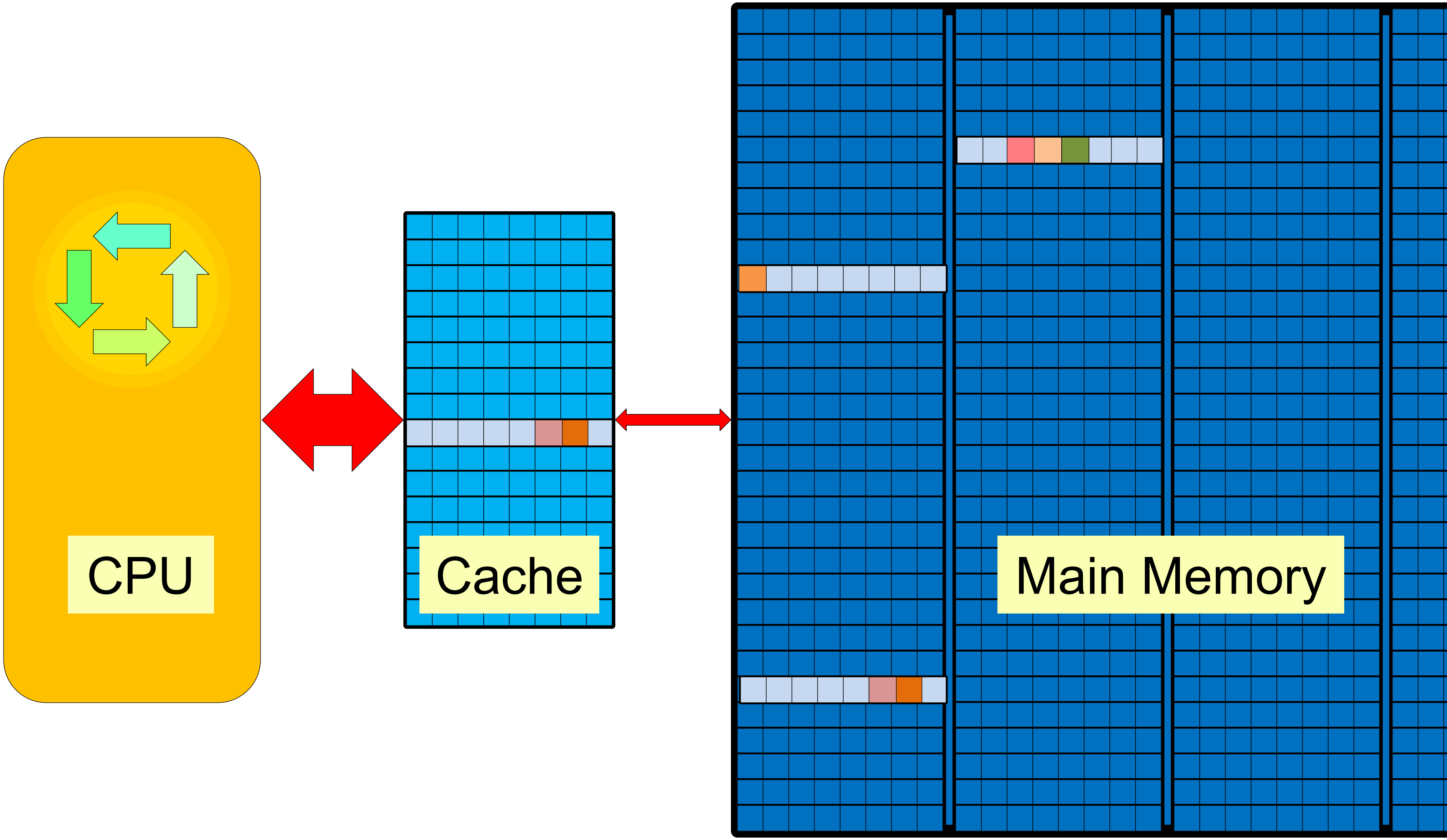
1. Introduction and Background

Review of Computer Memory



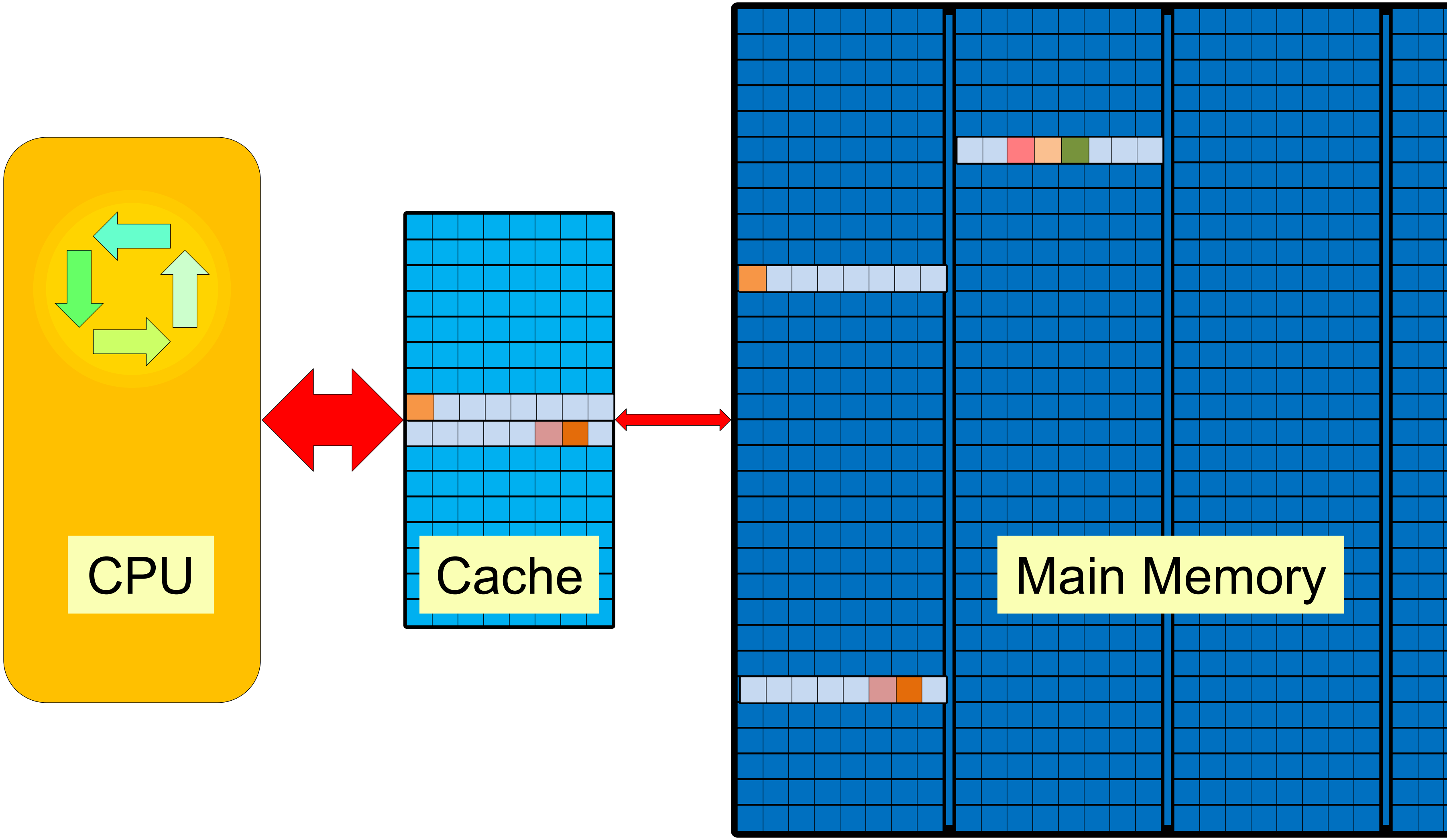
1. Introduction and Background

Review of Computer Memory



1. Introduction and Background

Review of Computer Memory



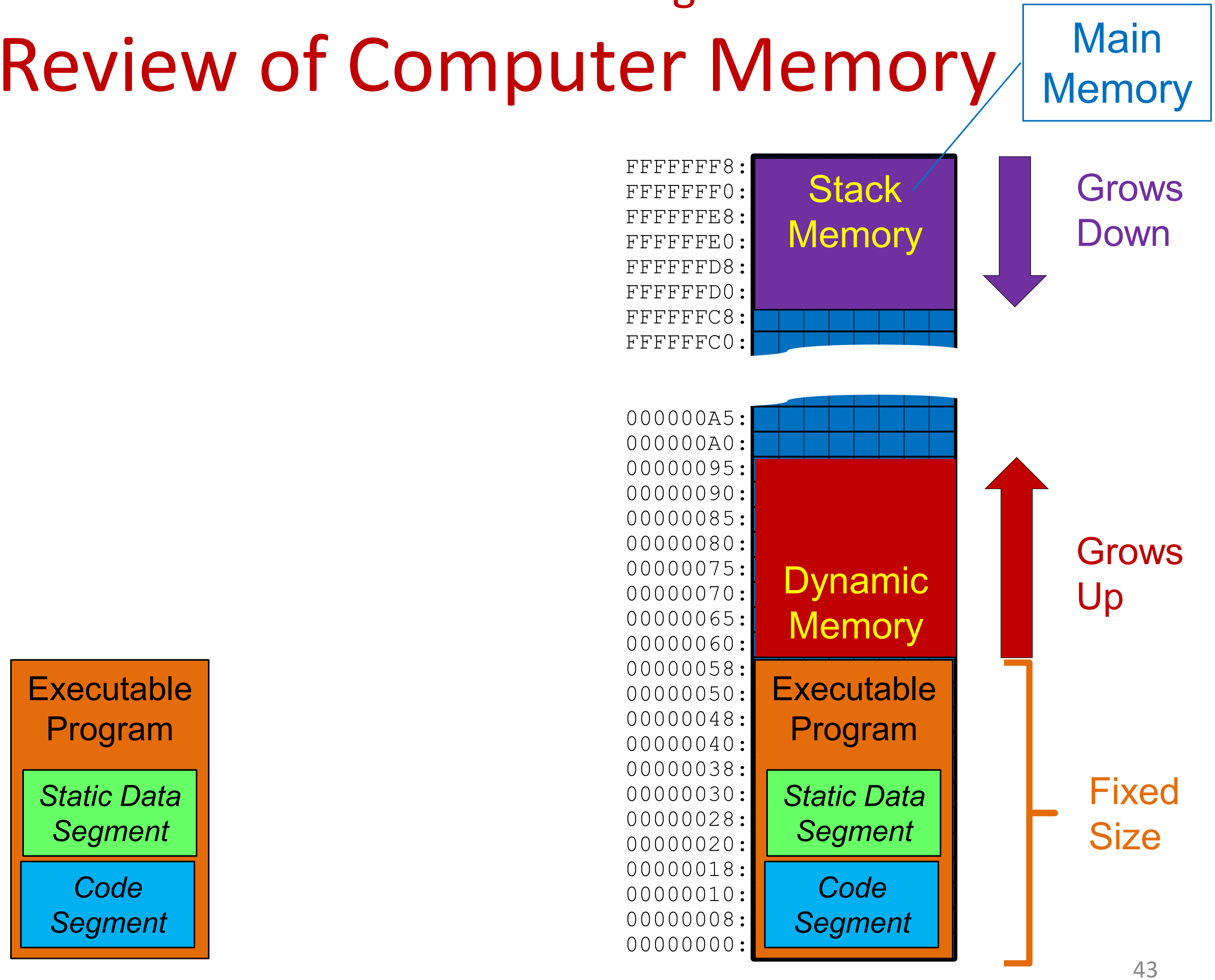
1. Introduction and Background

Review of Computer Memory

(Main) Memory Segments

1. Introduction and Background

Review of Computer Memory



1. Introduction and Background

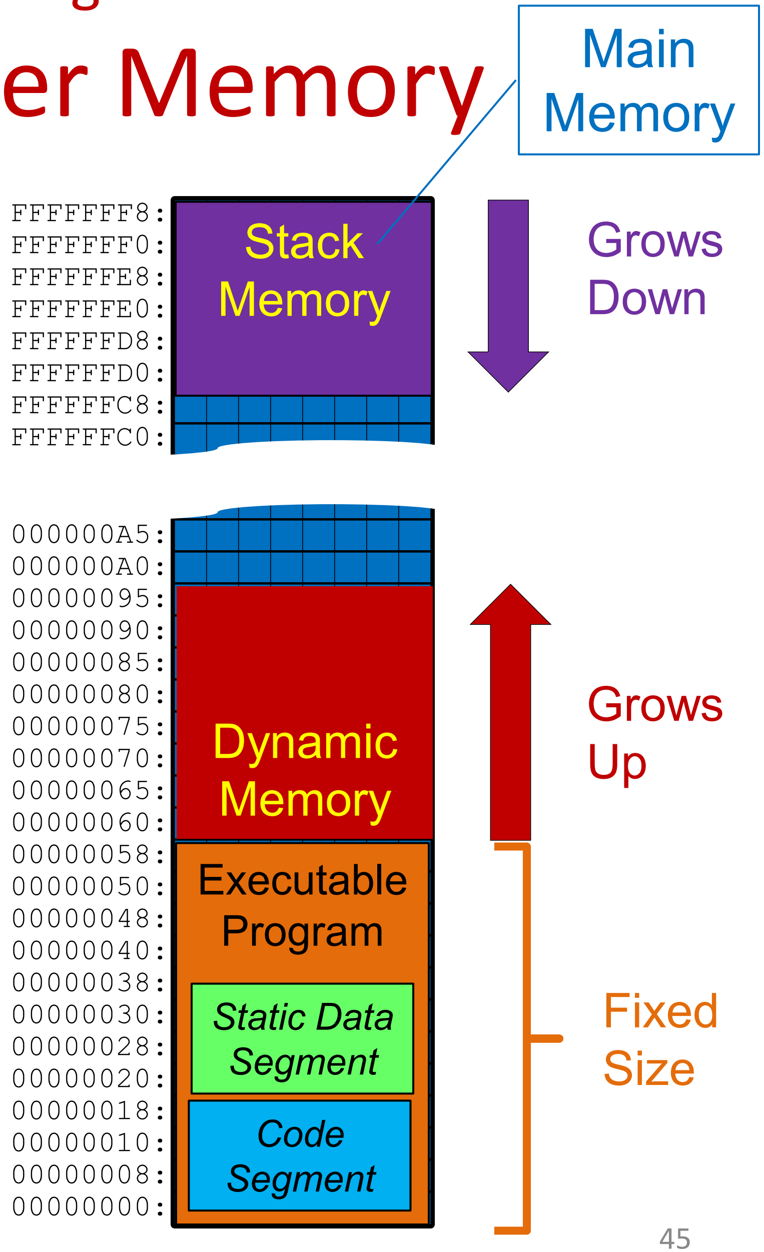
Important Questions

What is a

Memory Allocator?

1. Introduction and Background

Review of Computer Memory



1. Introduction and Background

Review of Computer Memory

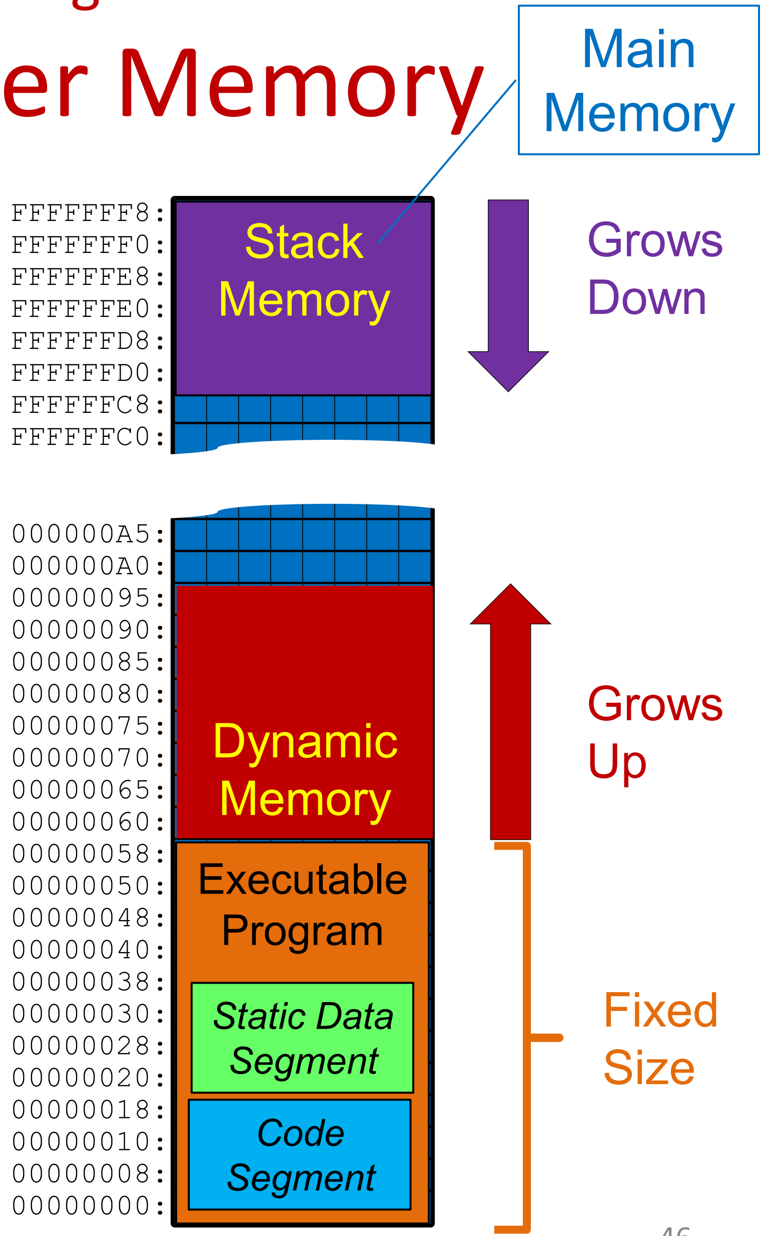
Special-Purpose Local Allocator:

```
// alloca.h
void *alloca(size_t nBytes);
```

General-Purpose Global Allocator:

```
// malloc.h
void *malloc(size_t nBytes);
void free(void *address);
```

C-Language Memory-Allocation Utilities



Memory Allocator Definition (take 1)

A *memory allocator* organizes a* region of computer memory, dispensing and reclaiming authorized access to suitable sub-regions on demand.

*possibly non-contiguous

1. Introduction and Background

General versus Special Allocator

A *General-Purpose Allocator*

- Is designed to work reasonably well for *all* use cases.
- Satisfies *all* requirements for memory allocators.

A *Special-Purpose Allocator*

- (Typically) works especially well for *some* use cases.
- Need not satisfy *all* requirements for allocators - E.g.:
    - May not be safe to use in a multi-threaded program.
    - May not reuse individually freed memory.
- Requires specific knowledge of the context of use.

Global versus Local Allocator

A *Global Allocator*

- Operates on a single ubiquitous region of memory.
- Exists throughout the lifetime of a program.
- Is inherently accessible from all parts of a program.

A *Local Allocator*

- Operates on a local sub-region (“arena”) of memory.
- May exist for less than the lifetime of a program.
- Is (typically) supplied for client use via a “reference”.
- Can (typically) be used to free memory unilaterally.

Global, General Allocator Utility

C:

```
// <malloc.h>
void *malloc(size_t nbytes);
void free(void *address);
```

C++:

```
// <new>
namespace std {
void *operator new(size_t nbytes);
void operator delete(void *address);
…
```

1. Introduction and Background

General/Special × Global/Local

	Global	Local

General	`malloc/free`
	`new/delete`
	tcmalloc
	jemalloc
		`multipool_allocator`
		Any general algorithm applied to a physically (and temporally) local region of memory.

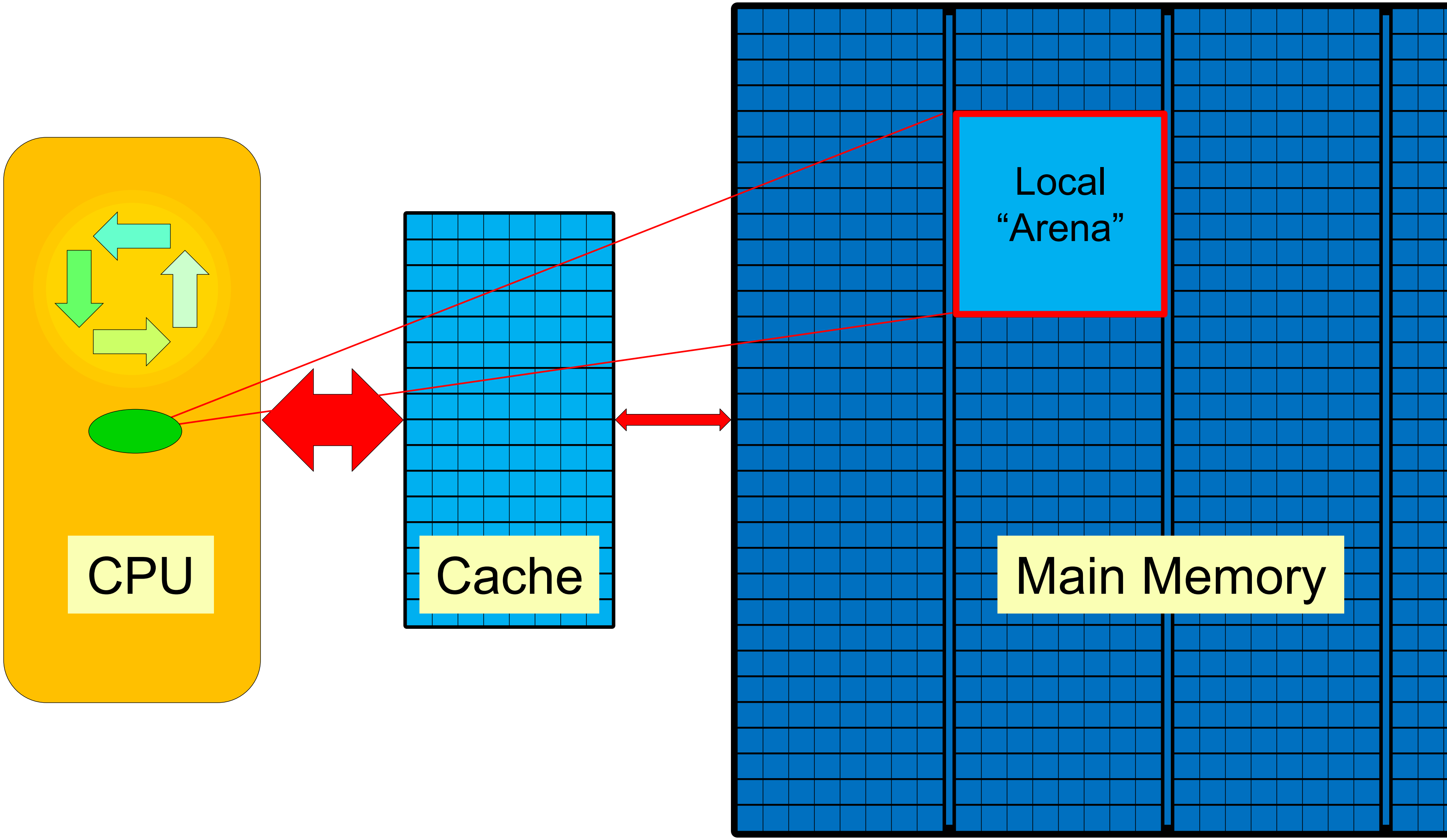
Special	An unsynchronized *tcmalloc* allocator “plugged into” (i.e., used to implement) `malloc/free`	`alloca`
		`monotonic_allocator`
		An unsynchronized version of a `multipool_allocator`

Memory Allocator Definition (take 2)

A *memory allocator* is a stateful ***utility*** or ***mechanism*** that organizes a* region of computer memory, dispensing and reclaiming authorized access to suitable sub-regions on demand.

*possibly non-contiguous

What is a local memory allocator?



1. Introduction and Background

Local Allocator Mechanism

```
class LocalAllocator {
    // internal data structure
  public:
    LocalAllocator(const LocalAllocator&) = delete;
    LocalAllocator& operator=(const LocalAllocator&)
                                              = delete;
    // CREATORS
    LocalAllocator(/* … */);
    // MANIPULATORS
    void *allocate(std::size_t nBytes);
    void deallocate(void *address);
};
```

1. Introduction and Background

Local Allocator Mechanism

```
class LocalAllocator {
    // internal data structure
  public:
    LocalAllocator(const LocalAllocator&) = delete;
    LocalAllocator& operator=(const LocalAllocator&)
                                          = delete;
    // CREATORS
    LocalAllocator(/* … */);
    // MANIPULATORS
    void *allocate(std::size_t nBytes);
    void deallocate(void *address);
};
```

1. Introduction and Background

Local Allocator Mechanism

```
class LocalAllocator {
    // internal data structure
  public:
    LocalAllocator(const LocalAllocator&) = delete;
    LocalAllocator& operator=(const LocalAllocator&)
                                          = delete;
    // CREATORS
    LocalAllocator(void *begin, void *end);
    // MANIPULATORS
    void *allocate(std::size_t nBytes);
    void deallocate(void *address);
};
```

Local Allocator Mechanism

```
class LocalAllocator {
    // internal data structure
  public:
    LocalAllocator(const LocalAllocator&) = delete;
    LocalAllocator& operator=(const LocalAllocator&)
                                           = delete;
    // CREATORS
    LocalAllocator(void *begin, void *end);
    // MANIPULATORS
    void *allocate(std::size_t nBytes);
    void deallocate(void *address);
};
```

1. Introduction and Background

Local Allocator Mechanism

```
class LocalAllocator {
    // internal data structure
  public:
    LocalAllocator(const LocalAllocator&) = delete;
    LocalAllocator& operator=(const LocalAllocator&)
                                          = delete;
    // CREATORS
    LocalAllocator(void *begin, void *end);
    // MANIPULATORS
    void *allocate(std::size_t nBytes);
    void deallocate(void *address);
    void release(); // local allocators only
};
```

Memory Allocator Definition (take 3)

A *memory allocator* is (the client-facing interface for) a stateful *utility* or *mechanism* that organizes a* region of computer memory, dispensing and reclaiming authorized access to suitable sub-regions on demand.

*possibly non-contiguous

Memory Allocator Interfaces

Allocators can be supplied for use in multiple ways:

1. As (stateful) utility functions.
    - ❖ Doesn't support allocator objects.
2. As a "reference wrapper" template parameter.
    - ✓ Concrete allocator type is available for use by client's compiler.
    - ❖ Forces a client to be a template in order to hold the allocator reference.
    - ❖ Allocator type affects the C++ type of the client object.
3. As the address of a pure abstract base class.
    - ✓ Allocator can be held via a base-class reference by a non-template class.
    - ✓ The choice of allocator does not affect the C++ type of the client object.
    - ❖ Allocator must be accessed via its virtual-function interface.
    - ❖ Object must somehow hold an extra address – even for the default case.

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

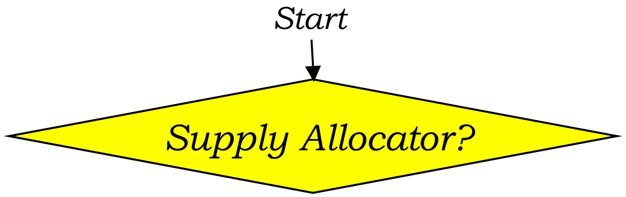
   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

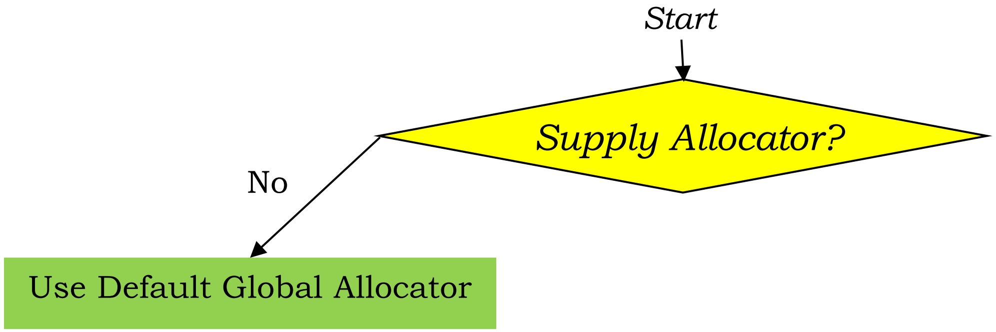
2. Understanding the Problem

Supply a local allocator? Which one?



2. Understanding the Problem

Supply a local allocator? Which one?



Start

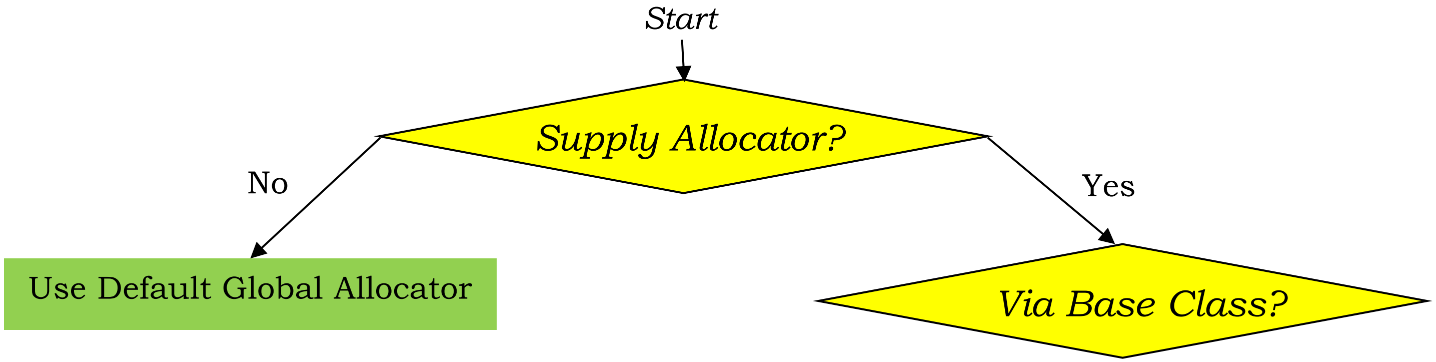
Supply Allocator?

No

Use Default Global Allocator

2. Understanding the Problem

Supply a local allocator? Which one?



Start

Supply Allocator?

No

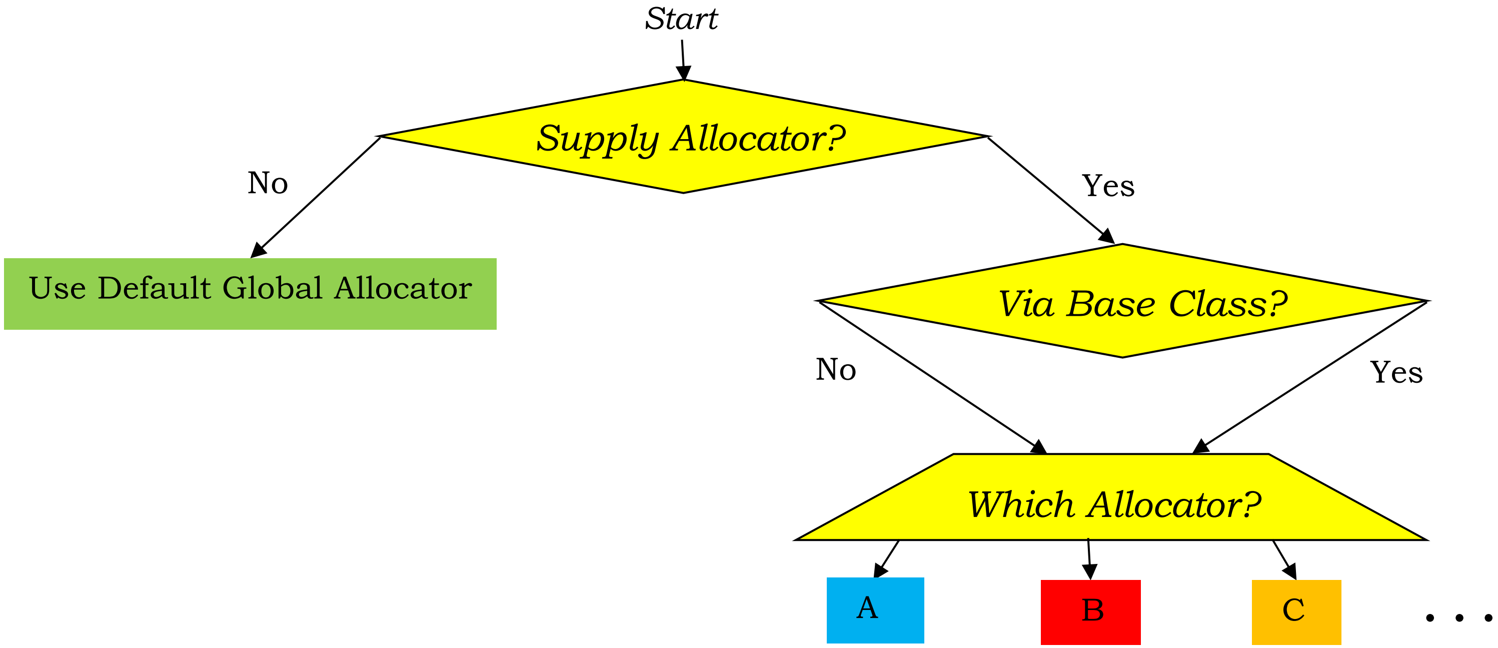
Yes

Use Default Global Allocator

Via Base Class?

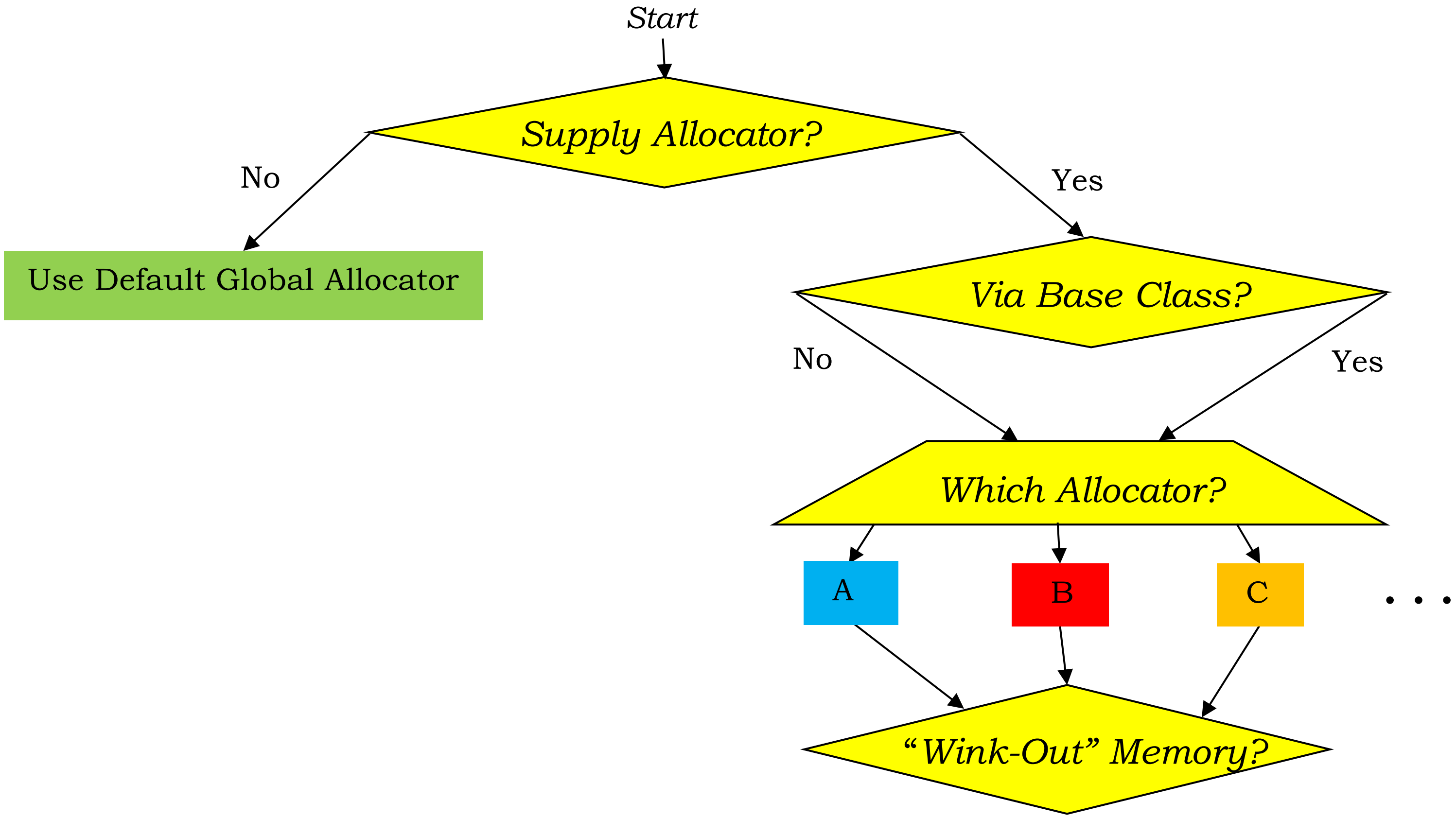
2. Understanding the Problem

Supply a local allocator? Which one?



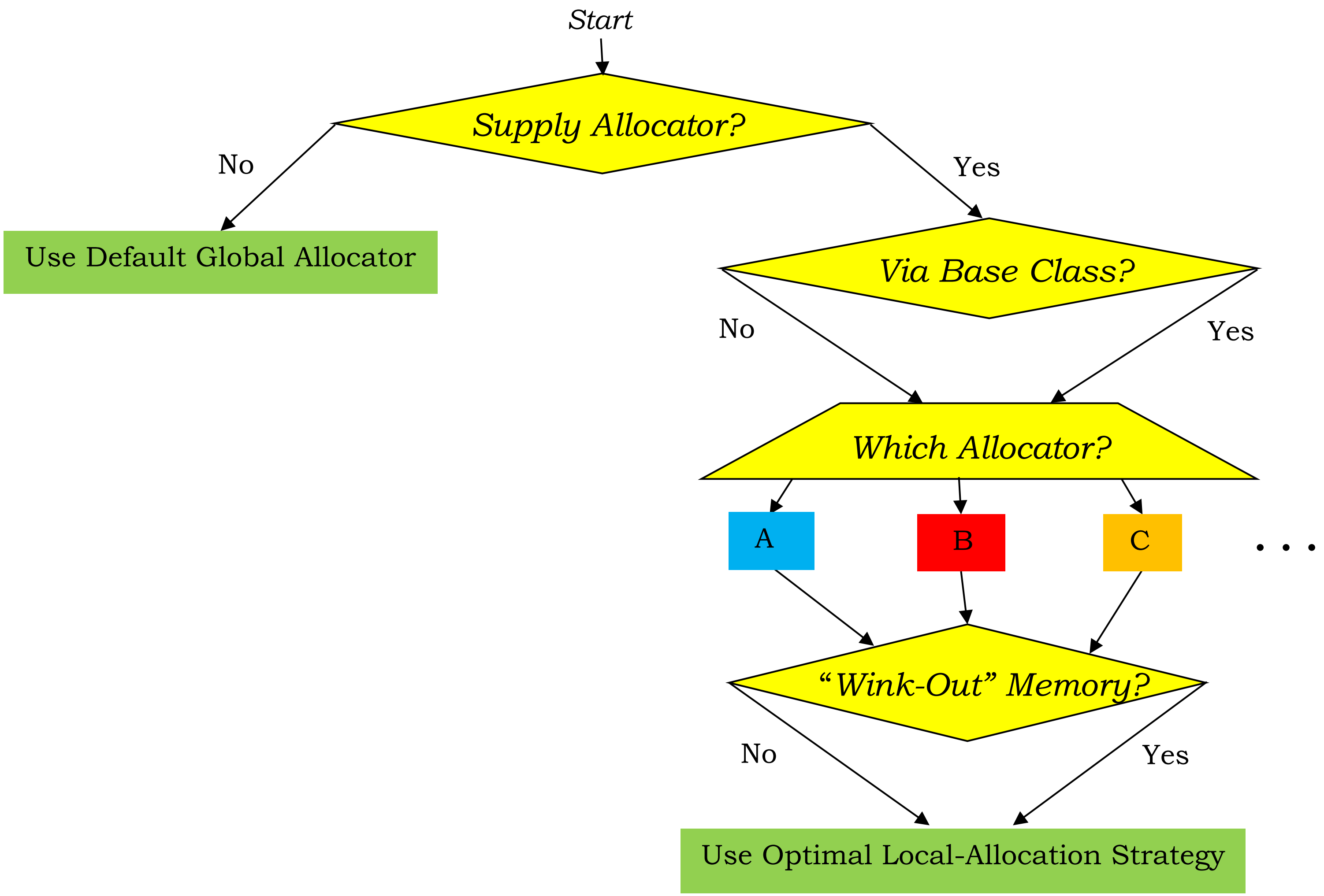
2. Understanding the Problem

Supply a local allocator? Which one?



2. Understanding the Problem

Supply a local allocator? Which one?



2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

1

Global Allocator

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

1 * 2

Global Allocator **x** Type Parameter

Abstract Base

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator **x** Type Parameter

Abstract Base

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator

x

Type Parameter

Abstract Base

3

Monotonic

Multipool

Multipool<Monotonic>

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator x Type Parameter

Abstract Base

3 * 2

Monotonic

Multipool

Multipool<Monotonic>

x

Type Parameter

Abstract Base

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator

x

Type Parameter
Abstract Base

3 * 2 * 2

Monotonic
Multipool
Multipool<Monotonic>

x

Type Parameter
Abstract Base

x

Normal Destruction
(Magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator

x

Type Parameter
Abstract Base

12 = 3 * 2 * 2

Monotonic
Multipool
Multipool<Monotonic>

x

Type Parameter
Abstract Base

x

Normal Destruction
(Magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

2 = 1 * 2

Global Allocator x Type Parameter / Abstract Base

Global Allocator	x	Type Parameter Abstract Base

12 = 3 * 2 * 2

Monotonic Multipool Multipool<Monotonic>	x	Type Parameter Abstract Base	x	Normal Destruction (Magically) “Winked Out”

---

14 Allocation Strategies (AS)

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

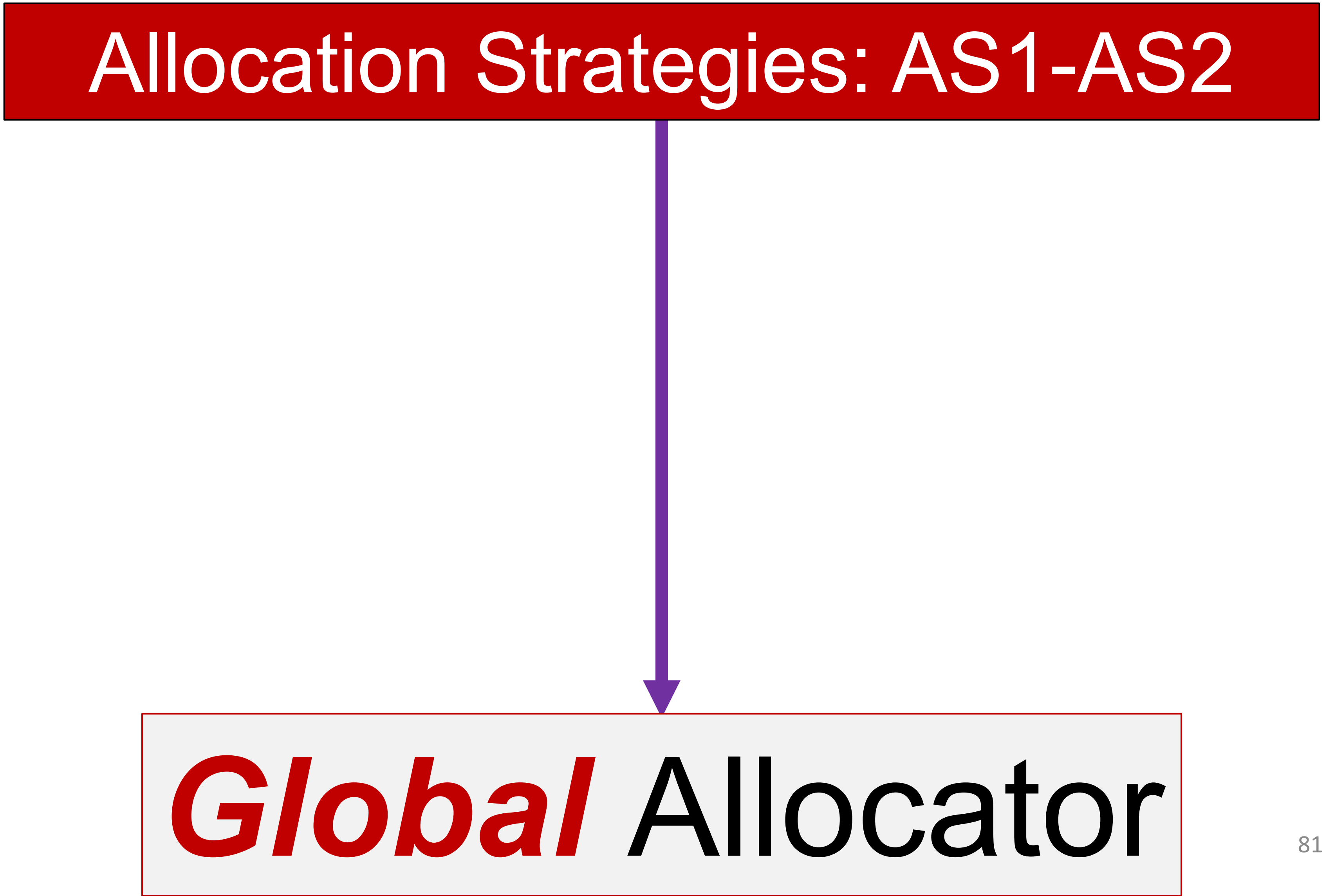
Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies



Allocation Strategies: AS1-AS2

Global Allocator

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Standard Allocator

```
class allocator {
    // no data members
  public:
    // CREATORS
    allocator() { }
    allocator(const allocator& ) { }
    ~allocator() { }
    // MANIPULATORS
    allocator operator=() = delete;
    void *allocate(std::size_t nBytes) {
                        return ::operator new(nBytes); }
    void deallocate(void *address) {
                           ::operator delete(address); }
};

// FREE OPERATORS
bool operator==(const allocator&, const allocator&) {
                                          return true; }
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Standard Allocator

```
class allocator {
    // no data members
  public:
    // CREATORS
    allocator() { }
    allocator(const allocator& ) { }
    ~allocator() { }
    // MANIPULATORS
    allocator operator=() = delete;
    void *allocate(std::size_t nBytes) {
                        return ::operator new(nBytes); }
    void deallocate(void *address) {
                           ::operator delete(address); }
};

// FREE OPERATORS
bool operator==(const allocator&, const allocator&) {
                                        return true; }
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

AS1: Standard Allocator
Default Global Allocator

```
class allocator {
    // no data members
  public:
    // CREATORS
    allocator() { }
    allocator(const allocator& ) { }
    ~allocator() { }

    // MANIPULATORS
    allocator operator=() = delete;
    void *allocate(std::size_t nBytes) {
                        return ::operator new(nBytes); }
    void deallocate(void *address) {
                           ::operator delete(address); }
};

// FREE OPERATORS
bool operator==(const allocator&, const allocator&) {
                                             return true; }
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myFunction()
{
   std::vector<int> v;
}
```

Same object code generated as

```
myFunction()
{
   std::vector<int, allocator> v;
}
```

Our Tool Chest of Allocation Strategies

```
myFunction()
{
   std::vector<int> v;
}
```

Same object code generated as

Type Parameter

```
myFunction()
{
   std::vector<int, allocator> v;
}
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::list<int> *> system(N);










}  // 'system' goes out of scope (and is destroyed).
```

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::list<int> *> system(N);









}  // 'system' goes out of scope (and is destroyed).
```

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;
    std::vector<std::list<int> *> system(N);
    for (int i = 0; i < N; ++i) {
        system[i] = new std::list<int>;
        // build up list of elements
    }




}  // ‘system’ goes out of scope (and is destroyed).
```

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::list<int> *> system(N);

    for (int i = 0; i < N; ++i) {
        system[i] = new std::list<int>;

        // build up list of elements
    }

    // Do benchmark (e.g., access links).



}  // ‘system’ goes out of scope (and is destroyed).
```

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::list<int> *> system(N);

    for (int i = 0; i < N; ++i) {
        system[i] = new std::list<int>;

        // build up list of elements
    }

    // Do benchmark (e.g., access links).

    for (int i = 0; i < N; ++i) {
        delete system[i];
    }
}  // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Protocol:
Pure Abstract Base Class

```
class Allocator {
    // no data members
  public:
    // CREATORS
    virtual ~allocator();  // Defined empty in ‘.cpp’ file.

    // MANIPULATORS
    virtual void *allocate(std::size_t nBytes) = 0;
I   virtual void deallocate(void *address) = 0;
};
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Protocol:
Pure Abstract Base Class

```
class Allocator {
    // no data members
  public:
    // CREATORS
    virtual ~allocator();  // Defined empty in ‘.cpp’ file.

    // MANIPULATORS
    virtual void *allocate(std::size_t nBytes) = 0;
I   virtual void deallocate(void *address) = 0;
};
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
class NewDeleteAllocator : public Allocator {
    // no data members
  public:
    // CREATORS
    NewDeleteAllocator() = default;
    ~NewDeleteAllocator() = default;
    NewDeleteAllocator(const NewDeleteAllocator&) = delete;

    // MANIPULATORS
    NewDeleteAllocator& operator=(const NewDeleteAllocator&)

    inline void *allocate(std::size_t nBytes) override {
                            return ::operator new(nBytes); }

    inline void deallocate(void *address) override {
                               ::operator delete(address); }
};
```

Concrete Derived Class

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

AS2

Concrete Derived Class

```
class NewDeleteAllocator : public Allocator {
    // no data members
  public:
    // CREATORS
    NewDeleteAllocator() = default;
    ~NewDeleteAllocator() = default;
    NewDeleteAllocator(const NewDeleteAllocator&) = delete;

    // MANIPULATORS
    NewDeleteAllocator& operator=(const NewDeleteAllocator&)

    inline void *allocate(std::size_t nBytes) override {
                                return ::operator new(nBytes); }

    inline void deallocate(void *address) override {
                                   ::operator delete(address); }
};
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

AS2

Concrete Derived Class

```
class NewDeleteAllocator : public Allocator {
    // no data members
  public:
    // CREATORS
    NewDeleteAllocator() = default;
    ~NewDeleteAllocator() = default;
    NewDeleteAllocator(const NewDeleteAllocator&) = delete;

    // MANIPULATORS
    NewDeleteAllocator& operator=(const NewDeleteAllocator&)

    inline void *allocate(std::size_t nBytes) override {
                            return ::operator new(nBytes); }

    inline void deallocate(void *address) override {
                               ::operator delete(address); }
};
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Via Protocol

```
myFunction()
{
    NewDeleteAllocator a;
    std::pmr::vector<int> v(&a);

    // …
}
```

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);

        // build up list of elements
    }

    // Do benchmark (e.g., access links).

    for (int i = 0; i < N; ++i) {
        delete system[i];
    }

}  // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;
    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        delete system[i];
    }
}  // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);

        // build up list of elements
    }

    // Do benchmark (e.g., access links).

    for (int i = 0; i < N; ++i) {
        delete system[i];
    }

}  // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).

    for (int i = 0; i < N; ++i) {
        delete system[i];
    }
}  // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);

        // build up list of elements
    }

    // Do benchmark (e.g., access links).

    for (int i = 0; i < N; ++i) {
        delete system[i];
    }

}  // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark()
{
    const int N = 1000;

    std::vector<std::pmr::list<int> *> system(N);

    NewDeleteAllocator a;

    for (int i = 0; i < N; ++i) {
        system[i] = new std::pmr::list<int>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        delete system[i];
    }
}  // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

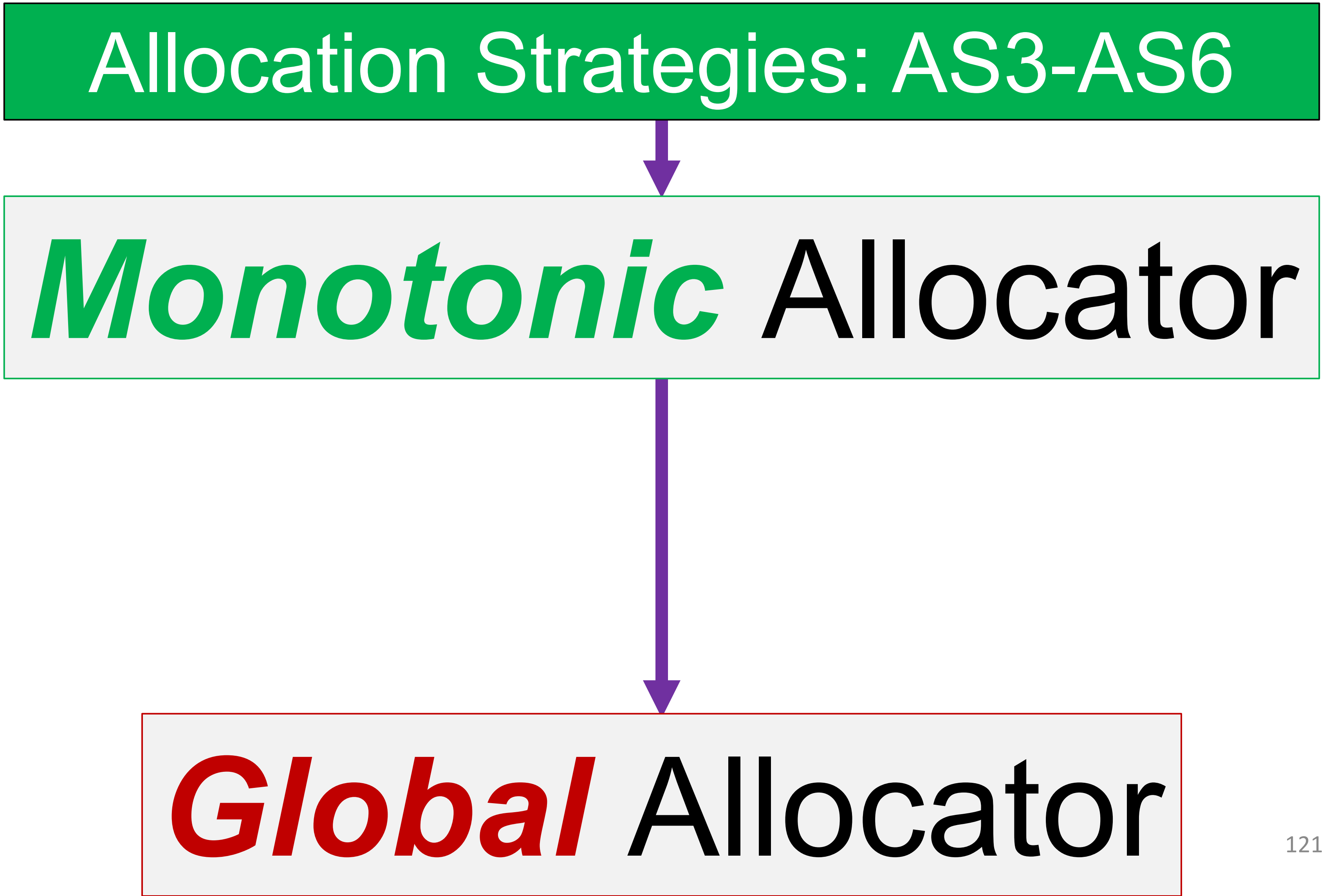
Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies



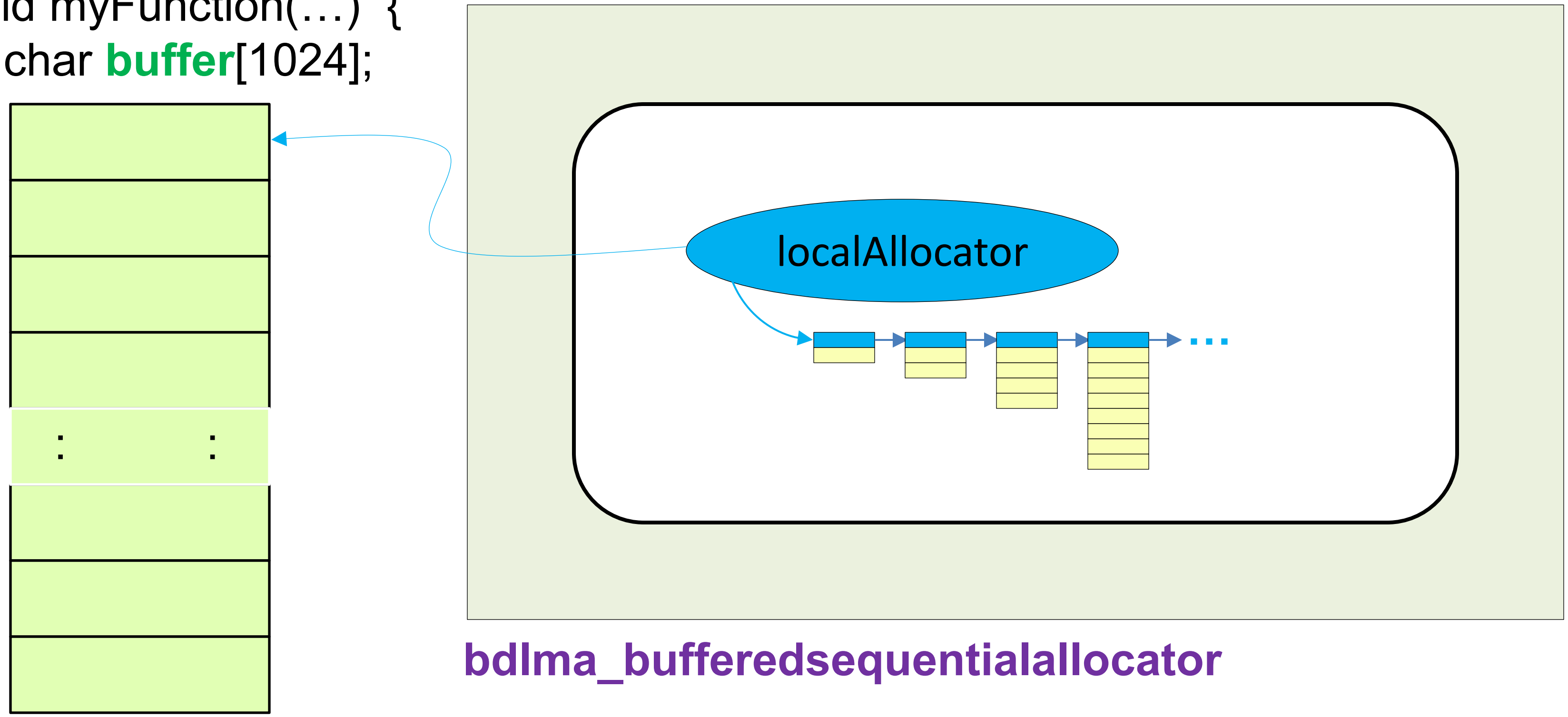
Allocation Strategies: AS3-AS6

Monotonic Allocator

Global Allocator

Our Tool Chest of Allocation Strategies

void myFunction(…) {
char **buffer**[1024];



bdlma_bufferedsequentialallocator

bdlma::BufferedSequentialAllocator **local Allocator**(**buffer**, sizeof **buffer**);
bsl::vector(&**local Allocator**);
// …
}

Note that deallocate is a No-Op!

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) "Winked Out"
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) "Winked Out"
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) "Winked Out"
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) "Winked Out"
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) "Winked Out"
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) "Winked Out"

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wrapper>();
        a.deallocate(system[i]);
    }
}     // ‘a’ goes out of scope (and is destroyed).
      // ‘system’ goes out of scope (and is destroyed).
```

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}    // ‘a’ goes out of scope (and is destroyed).
     // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}    // 'a' goes out of scope (and is destroyed).
     // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}    // 'a' goes out of scope (and is destroyed).
     // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}     // 'a' goes out of scope (and is destroyed).
      // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}    // ‘a’ goes out of scope (and is destroyed).
     // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}     // 'a' goes out of scope (and is destroyed).
      // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}    // 'a' goes out of scope (and is destroyed).
     // 'system' goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}     // ‘a’ goes out of scope (and is destroyed).
      // ‘system’ goes out of scope (and is destroyed).
```

Normal Destruction

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
~~    for (int i = 0; i < N; ++i) {~~
~~        system[i]->~std::list<int, wra~~
~~        a.deallocate(system[i]);~~
~~    }~~
}     // 'a' goes out of scope (and is destroyed).
      // 'system' goes out of scope (and is destroyed).
```

(magically)
"Winked Out"

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

```
myBenchmark() {
    const int N = 1000;
    std::vector<std::list<int, wrapper> *> system(N);
    bdlma::BufferedSequentialAllocator a; // monotonic
    for (int i = 0; i < N; ++i) {
        void *p = a.allocate(sizeof std::list<int, wrapper>);
        system[i] = new(p) std::list<int, wrapper>(&a);
        // build up list of elements
    }
    // Do benchmark (e.g., access links).
    for (int i = 0; i < N; ++i) {
        system[i]->~std::list<int, wra
        a.deallocate(system[i]);
    }
}     // ‘a’ goes out of scope (and is destroyed).
      // ‘system’ goes out of scope (and is destroyed).
```

(magically)
“Winked Out”

Same object code generated as

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) "Winked Out"
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) "Winked Out"
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) "Winked Out"
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) "Winked Out"
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) "Winked Out"
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) "Winked Out"

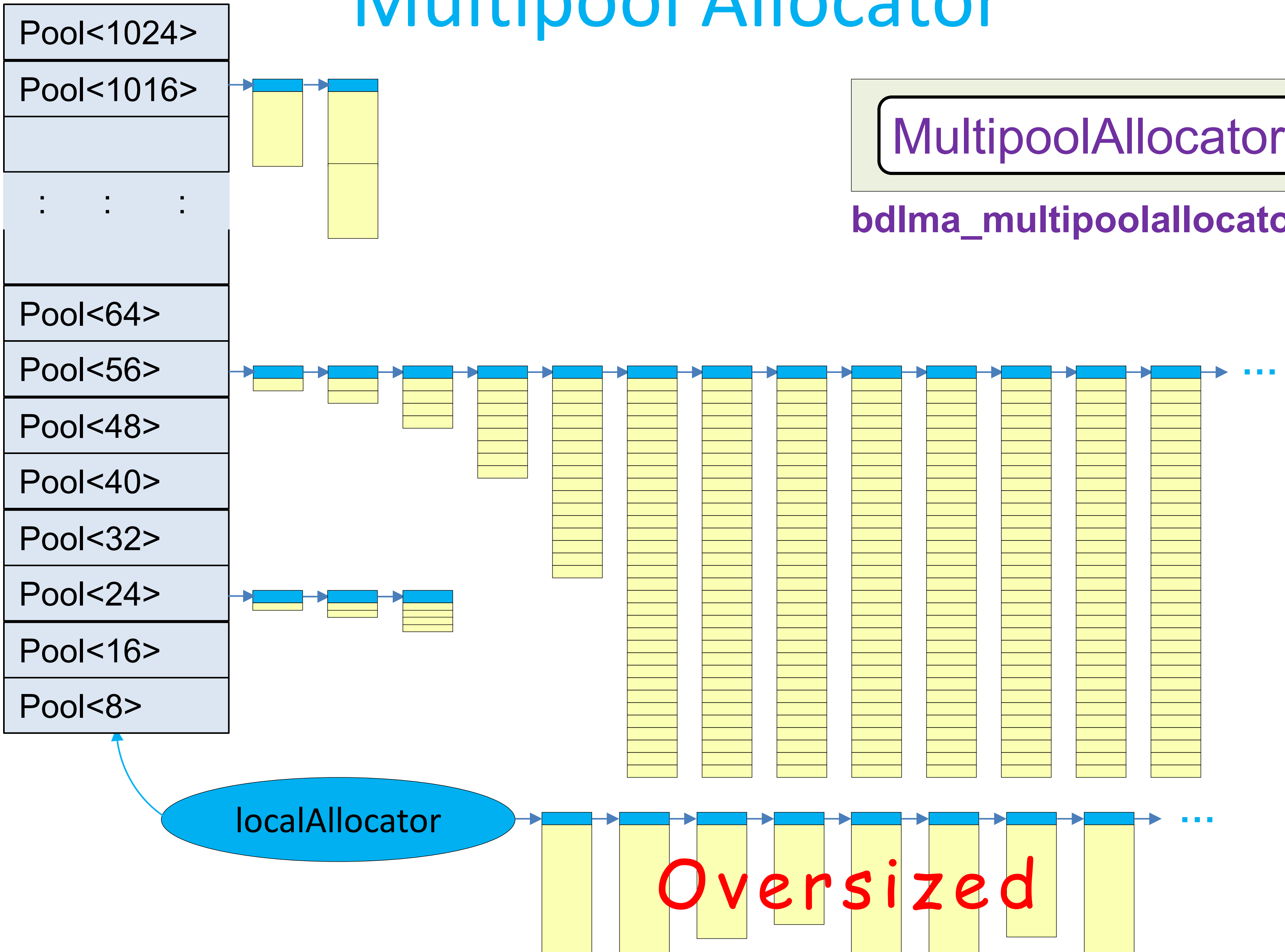
Our Tool Chest of Allocation Strategies

Allocation Strategies: AS7-AS10

Multipool Allocator

Global Allocator

Multipool Allocator



MultipoolAllocator

bdlma_multipoolallocator

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) "Winked Out"
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) "Winked Out"
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) "Winked Out"
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) "Winked Out"
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) "Winked Out"
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) "Winked Out"

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Allocation Strategies: AS11-AS14

Multipool Allocator

Monotonic Allocator

Global Allocator

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

Our Tool Chest of Allocation Strategies

Label	Allocator Type	Allocator Binding	Destruction of Allocated Objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	(magically) “Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	(magically) “Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	(magically) “Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	(magically) “Winked Out”
AS11	Multipool<Monotonic>	Type Parameter	Normal Destruction
AS12	Multipool<Monotonic>	Type Parameter	(magically) “Winked Out”
AS13	Multipool<Monotonic>	Abstract Base	Normal Destruction
AS14	Multipool<Monotonic>	Abstract Base	(magically) “Winked Out”

2. Understanding the Problem

Characterizing Usage Scenarios

What **basic** *“size” parameters* characterize *software usage*?

2. Understanding the Problem

Characterizing Usage Scenarios

Two fundamental (architectural) *"size" parameters* spring to mind:

2. Understanding the Problem

Characterizing Usage Scenarios

Two fundamental (architectural) *“size” parameters* spring to mind:

N: number of **instructions** executed

2. Understanding the Problem

Characterizing Usage Scenarios

Two fundamental (architectural) *"size" parameters* spring to mind:

N: number of **instructions** executed

W: number of active **threads**

2. Understanding the Problem

Characterizing Usage Scenarios

Two fundamental (architectural*) *“size” parameters* spring to mind:

N: number of **instructions** executed

W: number of active **threads**

*Note that these parameters are deliberately chosen to be platform independent.

2. Understanding the Problem

Characterizing Usage Scenarios

What *“aspects”* of software affect optimal *allocation strategy*?

2. Understanding the Problem

Characterizing Usage Scenarios

We initially proposed five *“dimensions”* to help us characterize *aspects* software usage:

D 1

V 2

L 3

U 4

C 5

2. Understanding the Problem

Characterizing Usage Scenarios

We initially proposed five *"dimensions"* to help us characterize *aspects* software usage:

- **D** DENSITY of *allocation operations*
- **V** VARATION of *allocated sizes*
- **L** LOCALITY of *accessed memory*
- **U** UTILIZATION of *allocated memory*
- **C** CONTENTION of *concurrent allocations*

2. Understanding the Problem

Characterizing Usage Scenarios

These “*dimensions*” are intended to be rough indications, not precise measures:

2. Understanding the Problem

Characterizing Usage Scenarios

These “*dimensions*” are intended to be rough indications, not precise measures:

- They are indented to be scaled (somehow) to fit the range [0 .. 1].

2. Understanding the Problem

Characterizing Usage Scenarios

These "*dimensions*" are intended to be rough indications, not precise measures:

- They are indented to be scaled (somehow) to fit the range [0 .. 1].
- 0 implies the *minimum* for the *aspect*, whereas 1 implies the *maximum*.

2. Understanding the Problem

Characterizing Usage Scenarios

These “*dimensions*” are intended to be rough indications, not precise measures:

- They are indented to be scaled (somehow) to fit the range [0 .. 1].
- 0 implies the *minimum* for the *aspect*, whereas 1 implies the *maximum*.
- Note that these *“dimensions”* are far from independent.

2. Understanding the Problem

Characterizing Usage Scenarios

DENSITY of *allocation operations*

$$\mathbf{D} = \frac{numAllocDeallocOps}{\mathbf{N}}$$

2. Understanding the Problem

Characterizing Usage Scenarios

DENSITY of *allocation operations*

$$\mathbf{D} = \frac{numAllocDeallocOps}{\mathbf{N}}$$

0.0: No allocation instructions occur.

1.0: Every instruction is either `allocate` or `deallocate`.

2. Understanding the Problem

Characterizing Usage Scenarios

DENSITY of *allocation operations*

$$\mathbf{D} = \frac{numAllocDeallocOps}{\mathbf{N}}$$

0.0: No allocation instructions occur.

1.0: Every instruction is either `allocate` or `deallocate`.

Consider: `std::vector<int>`

2. Understanding the Problem

Characterizing Usage Scenarios

VARIATION of *allocated sizes*

$$\mathbf{V} = \frac{numUniqueAllocSizes}{numAllocOps}$$

2. Understanding the Problem

Characterizing Usage Scenarios

VARIATION of *allocated sizes*

$$\mathbf{V} = \frac{numUniqueAllocSizes}{numAllocOps}$$

0.0: All allocated sizes are the same.

1.0: Every allocated size is different.

2. Understanding the Problem

Characterizing Usage Scenarios

VARIATION of *allocated sizes*

$$\mathbf{V} = \frac{numUniqueAllocSizes}{numAllocOps}$$

0.0: All allocated sizes are the same.

1.0: Every allocated size is different.

Consider: `std::set<int>`
`std::string`

2. Understanding the Problem

Characterizing Usage Scenarios

LOCALITY of *accessed memory*

I = num access instructions on subregion over duration

M = num bytes of memory in subregion

T = num transitions out of subregion over duration

2. Understanding the Problem

Characterizing Usage Scenarios

*Note that **LOCALITY** (**L**) can play a critical role in long-running programs even when the allocation **DENSITY** (**D**) is negligible!*

M

T = num

n over duration

$$\mathbf{L}_{physical} = \frac{\mathbf{I}}{\mathbf{M} * \mathbf{T}} \qquad \mathbf{L} = \frac{\mathbf{I}}{\mathbf{M} * \mathbf{T}} \qquad \mathbf{L}_{temporal} = \frac{\mathbf{I}}{\mathbf{M} * \mathbf{T}}$$

0.0: Subregion is large or not accessed repeatedly.

1.0: Subregion is small and accessed repeatedly.

2. Understanding the Problem

Characterizing Usage Scenarios

UTILIZATION of *allocated memory*

$$\mathbf{U} = \frac{maxMemoryInUse}{totalMemoryAllocated}$$

2. Understanding the Problem

Characterizing Usage Scenarios

UTILIZATION of *allocated memory*

$$\mathbf{U} = \frac{maxMemoryInUse}{totalMemoryAllocated}$$

0.0: Memory is repeatedly freed/reallocated.

1.0: <u>All</u> allocations precede deallocations.

2. Understanding the Problem

Characterizing Usage Scenarios

UTILIZATION of *allocated memory*

$$\mathbf{U} = \frac{maxMemoryInUse}{totalMemoryAllocated}$$

0.0: Memory is repeatedly freed/reallocated.

1.0: All allocations precede deallocations.

Consider: `vector<int>`

`vector<string>`

2. Understanding the Problem

Characterizing Usage Scenarios

CONTENTION of *concurrent allocations*

$$\mathbf{C} = \frac{expectedNumConcurrentAllocations}{\mathbf{W}}$$

2. Understanding the Problem

Characterizing Usage Scenarios

CONTENTION of *concurrent allocations*

$$\mathbf{C} = \frac{expectedNumConcurrentAllocations}{\mathbf{W}}$$

0.0: At most one thread has non-zero ($\mathbf{D}$).

1.0: The allocation **D**ENSITY ($\mathbf{D}$) per thread is 1.

2. Understanding the Problem

Characterizing Usage Scenarios

CONTENTION of *concurrent allocations*

$$\mathbf{C} = \frac{expectedNumConcurrentAllocations}{\mathbf{W}}$$

0.0: At most one thread has non-zero (**D**).

1.0: The allocation **D**ENSITY (**D**) per thread is 1.

*Note that thread **C**ONTENTION (**C**) is strongly correlated with allocation **D**ENSITY (**D**).*

Characterizing Usage Scenarios

Summary

- **D** DENSITY of *allocation operations*
- **V** VARATION of *allocated sizes*
- **L** LOCALITY of *accessed memory*
- **U** UTILIZATION of *allocated memory*
- **C** CONTENTION of *concurrent allocations*

2. Understanding the Problem

Characterizing Usage Scenarios

Mnemonic: ?

D **DENSITY** of *allocation operations*

V **VARATION** of *allocated sizes*

L **LOCALITY** of *accessed memory*

U **UTILIZATION** of *allocated memory*

C **CONTENTION** of *concurrent allocations*

2. Understanding the Problem

Characterizing Usage Scenarios

Mnemonic: **D.V.L.U.C.** **the DUCK**

- **D** **D**ENSITY of *allocation operations*
- **V** **V**ARATION of *allocated sizes*
- **L** **L**OCALITY of *accessed memory*
- **U** **U**TILIZATION of *allocated memory*
- **C** **C**ONTENTION of *concurrent allocations*

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What <u>must</u> we remember about memory allocators?

3. Analyzing the Benchmark Data

Roadmapping the Benchmarks

Considerations:

- We wanted to explore each dimension to observe its effects on optimal memory allocation.
- Our first thought was to create a single benchmark that spanned all five dimensions.
    - Find the centroid.
    - Vary the arguments along each dimension separately
    - Such a single benchmark is not at all easy.
- We finally settled on four separate benchmarks.
    - Benchmark I addresses the first two dimensions.

3. Analyzing the Benchmark Data

Tested Across Many Problem Sizes

Considerations:

- We tried not to assume the answers we expected.
  - Explored a wide range of problem sizes, **N**.
- Used successive *powers of two*.
  - We often show just the exponent in tables: 5 versus $2^{5.}$
- Contrasted results across disparate physical sizes.
  - E.g., by holding overall problem size **N** constant.
- Traded-off comparable parameters – E.g.,
  - *Subsystem size* versus *number of subsystems*
  - *Subsystem iterations* versus *experiment repetitions*

3. Analyzing the Benchmark Data

Platforms Used For These Benchmarks

All of results presented here are from a server having ***dual*** Intel Xeon E5-2620v2 processors.

Each processor:

- Ivy Bridge EP – "Sandy Bridge" architecture (c. 2013)
- 6 cores (for a total of 12 cores)
- 15 MB of L3 cache
- running at a fixed clock rate of 2.1 GHz
- 16GB of DDR3-1600 RAM (13G available to processes)

See: http://ark.intel.com/products/75789

Platforms Used For These Benchmarks

All benchmark programs were

- compiled using gcc-5.1,
- using optimizing “-O3 –march=native”,
- and run under Linux 3.18.

All experiments used only one core at a time

Except, that is, for Benchmark IV, which measures CONTENTION (**C**) and used more of the available cores.

3. Analyzing the Benchmark Data

Alternative Global Allocators

We investigate alternative global allocators:

- *tcmalloc*
- *jemalloc*

> “We determined that the **native allocators** (e.g., the one currently shipped with GCC on Linux) **performed as well or better**.”

3. Analyzing the Benchmark Data

Benchmark Road Map

I. **Short Running:** Build Up, Use, Tear Down

- Allocation **D**ENSITY and **V**ARIATION in Allocated Sizes

II. **Long Running:** Time-Multiplexed Subsystems

- Access **L**OCALITY – both *Physica*l and *Temporal*

III. **Short Running:** Varying Memory Reusability

- Memory **U**TILIZATION

IV. **Multithreaded:** Varying Numbers of Threads

- Allocator **C**ONTENTION

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

I. **Short Running:** Build Up, Use, Tear Down

- ➢ Allocation **D**ENSITY and **V**ARIATION in Allocated Sizes

II. **Long Running:** Time-Multiplexed Subsystems

- ➢ Access **L**OCALITY – both *Physica*l and *Temporal*

III. **Short Running:** Varying Memory Reusability

- ➢ Memory **U**TILIZATION

IV. **Multithreaded:** Varying Numbers of Threads

- ➢ Allocator **C**ONTENTION

3. Analyzing the Benchmark Data

Benchmark I: **D**ENSITY, **V**ARIATION

Considerations:

- Initially we wanted to investigate allocation **D**ENSITY.
  - Focused on allocation/deallocation costs themselves.
- Chose a variety of common data structures.
  - Used `int`, `string`, `vector`, and `unordered_set`.
- Didn't want access **L**OCALITY to dominate results.
  - Wrote to ***just*** the first byte of each newly allocated element.
- Later Incorporated **V**ARIATION into allocated memory.
  - `vector` objects' capacities were reserved up front.
  - `string` lengths were 33-1000 (uniformly distributed).

Benchmark I: DENSITY, VARIATION

Simple Data Structures

Label	Data Structure
DS1	`vector<int>`
DS2	`vector<string>`
DS3	`unordered_set<int>`
DS4	`unordered_set<string>`

Benchmark I: **D**ENSITY, **V**ARIATION

Plan:

3. Analyzing the Benchmark Data

Benchmark I: **D**ENSITY, **V**ARIATION

Plan:

- For each data structure in a thoughtfully chosen set:
  - Create the data structure.
  - Access it lightly.
  - Destroy it.
  - Repeat (until the problem size *N* is reached).

3. Analyzing the Benchmark Data

Benchmark I: **D**ENSITY, **V**ARIATION

Plan:

- For each data structure in a thoughtfully chosen set:
  - Create the data structure.
  - Access it lightly.
  - Destroy it.
  - Repeat (until the problem size *N* is reached).
- We chose the overall problem size to be $N = 2^{27}$.
  - The container *size (S)* varies from 2^8 to 2^{16}.
  - The number of experiment repetitions *(R) = N/S.*

3. Analyzing the Benchmark Data

Benchmark I: **DENSITY**, **VARIATION**

Contrasting access times across system sizes

Overall Problem Size = 2^{27}

$\log_2 \mathbf{N} = 27$

These
are all
exponents
of **2**.

Container Size (S)	Experiment Repititions (R)
8	19
9	18
10	17
11	16
12	15
13	14
14	13
15	12
16	11

Benchmark I: **D**ENSITY, **V**ARIATION

Plan:

- For each data structure in a thoughtfully chosen set:
  - Create the data structure.
  - Access it lightly.
  - Destroy it.
  - Repeat (until the problem size N is reached).
- We chose the overall problem size to be $N = 2^{27}$.
  - The container *size (S)* varies from 2^8 to 2^{16}.
  - The number of repetitions *(R) = N/S.*
- Each result entry is absolute **RUNTIME** (in seconds).

3. Analyzing the Benchmark Data

Benchmark I: **D**ENSITY, **V**ARIATION

DS1

Each result entry represents absolute runtime in seconds.

`vector<int>`

Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS1

`vector<int>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS1 `vector<int>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS1 `vector<int>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS1 `vector<int>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS1

`vector<int>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.20	1.90	0.30	0.40	0.40	0.40	0.80	1.00	0.90	1.10	0.60	0.70	0.80	0.70
2^{7}	0.90	1.60	0.30	0.40	0.40	0.40	0.50	0.70	0.60	0.70	0.50	0.50	0.60	0.50
2^{8}	0.80	1.00	0.20	0.40	0.40	0.30	0.40	0.60	0.50	0.60	0.30	0.50	0.50	0.50
2^{9}	0.80	1.00	0.20	0.40	0.40	0.40	0.30	0.50	0.50	0.50	0.30	0.40	0.40	0.40
2^{10}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{11}	0.70	0.90	0.20	0.30	0.40	0.30	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{12}	0.70	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{13}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{14}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{15}	0.80	0.90	0.20	0.30	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40
2^{16}	0.80	0.90	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40	0.20	0.40	0.40	0.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS2 `vector<string>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	68.90	67.30	12.90	12.80	13.30	12.90	18.10	17.80	18.20	17.70	15.50	14.80	15.60	14.80
2^{7}	68.80	68.20	12.80	12.90	13.20	12.90	20.60	20.20	20.60	20.40	15.10	14.30	15.00	14.40
2^{8}	70.80	68.90	13.20	12.80	13.60	12.90	30.80	30.40	30.70	30.30	15.30	14.60	15.40	14.70
2^{9}	73.10	71.20	13.50	13.50	13.90	13.50	38.20	37.60	38.00	37.30	15.90	15.10	15.90	15.10
2^{10}	75.40	74.30	13.60	13.50	14.00	13.70	41.10	40.30	41.60	40.90	16.00	15.10	15.90	15.00
2^{11}	76.90	74.50	13.60	13.50	14.10	13.60	43.90	43.20	43.70	42.60	16.00	15.00	16.00	15.10
2^{12}	76.10	74.80	13.70	13.50	14.00	13.60	41.20	38.80	40.60	39.40	15.90	14.90	15.80	15.00
2^{13}	76.10	74.80	13.60	13.60	14.00	13.60	41.40	39.20	41.30	39.90	15.90	15.00	15.80	14.90
2^{14}	78.30	76.50	13.60	13.60	14.00	13.60	45.80	42.30	44.80	44.00	16.10	15.20	16.20	15.40
2^{15}	90.40	91.00	20.20	20.10	20.50	20.10	62.20	58.70	62.20	58.20	26.00	25.00	26.00	24.90
2^{16}	103.0	103.0	21.50	21.30	21.80	21.30	66.50	59.20	65.10	59.90	27.00	25.30	27.10	25.20

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS3 `unordered_set<int>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	10.20	11.00	5.08	4.88	5.62	5.34	7.16	7.12	7.50	7.20	6.19	5.73	6.40	5.81
2^{7}	12.50	13.30	5.04	4.81	5.68	5.24	6.37	6.22	6.71	6.31	5.80	5.46	6.08	5.50
2^{8}	15.80	16.40	4.99	4.79	5.54	5.22	5.95	5.81	6.21	5.92	5.65	5.32	5.82	5.40
2^{9}	18.30	19.00	5.01	4.80	5.53	5.18	5.78	5.56	6.01	5.70	5.56	5.20	5.76	5.21
2^{10}	21.40	22.30	4.99	4.83	5.55	5.20	5.72	5.46	5.95	5.55	5.52	5.27	5.68	5.24
2^{11}	25.50	26.10	4.98	4.81	5.56	5.16	5.67	5.44	5.86	5.65	5.53	5.23	5.69	5.26
2^{12}	27.10	28.00	5.02	4.81	5.55	5.20	6.42	6.10	6.57	6.25	5.51	5.12	5.68	5.27
2^{13}	27.90	28.80	5.03	4.81	5.59	5.21	7.34	6.91	7.46	7.03	5.61	5.16	5.71	5.24
2^{14}	28.50	29.00	5.03	4.80	5.58	5.26	7.03	6.59	7.18	6.68	5.64	5.19	5.80	5.34
2^{15}	28.30	29.20	5.03	4.78	5.56	5.28	7.11	6.65	7.20	6.83	5.68	5.17	5.78	5.24
2^{16}	31.60	31.80	5.02	4.76	5.60	5.22	6.79	6.37	6.93	6.46	5.68	5.17	5.79	5.24

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS4 `unordered_set<string>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	103.0	120.0	52.20	51.90	52.40	51.20	58.40	57.60	59.70	58.90	55.10	54.10	56.90	55.30
2^{7}	103.0	122.0	52.50	52.10	52.90	51.80	63.30	61.90	64.40	63.80	55.30	54.00	56.80	55.70
2^{8}	109.0	128.0	53.60	53.00	53.70	52.60	76.30	74.70	77.40	75.90	56.50	54.90	57.90	56.70
2^{9}	113.0	134.0	54.50	53.40	54.90	53.00	83.10	81.70	82.80	81.40	57.30	56.70	58.00	56.40
2^{10}	119.0	143.0	56.60	54.90	56.90	54.60	87.60	85.90	88.10	86.50	58.80	56.90	59.20	57.30
2^{11}	122.0	144.0	57.00	55.30	57.70	54.90	90.70	89.20	90.70	88.40	59.40	57.60	60.00	57.80
2^{12}	122.0	146.0	57.90	55.90	58.40	55.70	93.20	90.70	93.20	90.70	60.50	58.30	60.70	58.40
2^{13}	124.0	148.0	58.20	56.30	58.50	55.90	95.10	91.50	94.30	92.00	60.50	58.20	60.70	58.70
2^{14}	139.0	166.0	59.10	57.30	59.60	56.80	98.50	94.10	97.80	95.80	61.80	59.60	62.20	60.00
2^{15}	176.0	211.0	66.00	62.70	66.20	62.40	121.0	115.0	122.0	115.0	76.50	73.30	76.80	74.00
2^{16}	196.0	232.0	78.50	72.00	79.10	71.00	137.0	127.0	136.0	127.0	87.10	82.40	87.80	82.90

3. Analyzing the Benchmark Data

Benchmark I: **D**ENSITY, **V**ARIATION

Questions

and/or

Discussion?

Benchmark I: DENSITY, VARIATION

Composite Data Structures

Label	Data Structure
DS5	`vector<vector<int>>`
DS6	`vector<vector<string>>`
DS7	`vector<unordered_set<int>>`
DS8	`vector<unordered_set<string>>`
DS9	`unordered_set<vector<int>>`
DS10	`unordered_set<vector<string>>`
DS11	`unordered_set<unordered_set<int>>`
DS12	`unordered_set<unordered_set<string>>`

Benchmark I: DENSITY, VARIATION

Composite Data Structures:

- The composite data elements were much larger.
- We wanted runtimes to be roughly comparable.
- We kept the overall problem size $N = 2^{27}$.
  - The outer container *size (S) still* varies from 2^8 to 2^{16}.
  - The inner container size *(K)* was fixed at 2^7.
  - Now, the number of repetitions $(R) = N/(K \cdot S)$.

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

Contrasting access times across system sizes

Overall Problem Size = 2^{27}

These are all *exponents* of **2**.

$\log_2 \mathbf{N} = 27$

Outer Container Size (S)	Inner Container Size (fixed)	Experiment Repetitions (R)
8	7	12
9	7	11
10	7	10
11	7	9
12	7	8
13	7	7
14	7	6
15	7	5
16	7	4

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

Composite Data Structures:

- The composite data elements were much larger.
- We wanted runtimes to be roughly comparable.
- We kept the overall problem size $N = 2^{27}$.
  - The outer container *size (S) still* varies from 2^8 to 2^{16}.
  - The inner container size *(K)* was fixed at 2^7.
  - Now, the number of repetitions $(R) = N/(K \cdot S)$.
- These adjustment kept runtimes manageable.
  - The number of leaf elements remained comparable.

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS5

`vector<vector<int>>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	0.97	1.00	0.19	0.13	0.20	0.17	0.24	0.20	0.20	0.21	0.21	0.19	0.20	0.21
2^{7}	0.96	0.96	0.22	0.16	0.18	0.14	0.21	0.20	0.19	0.20	0.16	0.20	0.21	0.19
2^{8}	0.99	1.00	0.19	0.13	0.18	0.17	0.27	0.30	0.27	0.29	0.19	0.19	0.20	0.21
2^{9}	0.99	1.02	0.19	0.13	0.18	0.14	0.36	0.33	0.33	0.36	0.19	0.15	0.20	0.20
2^{10}	1.01	1.04	0.19	0.18	0.19	0.14	0.37	0.36	0.36	0.38	0.22	0.19	0.20	0.22
2^{11}	1.02	1.05	0.19	0.13	0.19	0.14	0.36	0.35	0.36	0.36	0.20	0.15	0.20	0.22
2^{12}	1.03	1.05	0.19	0.19	0.22	0.18	0.33	0.36	0.32	0.32	0.20	0.21	0.20	0.19
2^{13}	1.02	1.05	0.19	0.13	0.22	0.19	0.35	0.35	0.34	0.33	0.20	0.21	0.22	0.19
2^{14}	1.05	1.10	0.19	0.17	0.19	0.16	0.38	0.36	0.38	0.37	0.17	0.19	0.20	0.19
2^{15}	1.13	1.18	0.22	0.19	0.19	0.16	0.50	0.45	0.47	0.45	0.21	0.21	0.17	0.18
2^{16}	1.29	1.32	0.22	0.19	0.20	0.17	0.54	0.47	0.52	0.50	0.22	0.21	0.22	0.21

3. Analyzing the Benchmark Data

Benchmark I: **Density, Variation**

DS6

`vector<vector<string>>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	72.60	72.70	9.06	9.06	9.36	8.98	41.70	40.00	41.20	39.20	11.20	10.30	11.20	10.30
2^{7}	74.90	76.00	8.92	8.98	9.29	8.89	46.50	44.80	46.00	43.00	11.40	11.00	12.70	10.30
2^{8}	85.50	85.20	17.10	17.40	17.30	16.90	62.90	58.40	61.30	58.40	22.80	22.50	23.30	22.00
2^{9}	96.40	96.30	18.40	18.70	19.00	18.40	66.20	59.00	64.70	59.30	24.20	22.70	24.50	22.30
2^{10}	102.0	102.0	18.70	18.60	19.10	18.60	67.00	59.60	65.90	59.00	24.80	22.50	24.80	22.50
2^{11}	102.0	101.0	18.40	18.70	19.20	18.20	62.40	55.00	61.30	54.20	24.80	22.60	25.10	22.30
2^{12}	104.0	103.0	18.50	18.70	19.40	18.30	61.60	54.20	60.50	53.40	24.90	22.70	25.10	22.30
2^{13}	103.0	104.0	18.80	18.40	19.00	18.60	61.80	53.40	59.90	53.50	25.30	22.60	25.10	22.60
2^{14}	97.10	96.30	19.20	19.60	20.10	19.20	60.60	53.70	60.20	52.90	29.00	26.70	29.20	26.30
2^{15}	88.10	88.70	23.40	23.20	23.70	23.40	62.60	54.40	60.90	53.90	33.40	30.60	33.20	30.70
2^{16}	76.70	76.70	25.00	25.30	25.80	25.00	63.40	54.80	62.90	54.30	35.00	32.80	35.50	32.40

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS7 `vector<unordered_set<int>>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	28.80	28.70	2.97	2.69	3.43	2.98	4.89	4.37	5.33	4.73	3.21	2.65	3.64	3.05
2^{7}	28.30	28.50	2.97	2.66	3.36	2.95	4.99	4.44	5.43	4.91	3.20	2.62	3.61	2.97
2^{8}	28.20	28.10	2.94	2.62	3.33	2.92	5.02	4.53	5.53	4.97	3.23	2.60	3.60	3.01
2^{9}	31.80	31.70	2.92	2.61	3.33	2.93	5.08	4.54	5.52	4.92	3.16	2.58	3.58	2.96
2^{10}	46.60	47.20	2.92	2.61	3.33	2.89	5.07	4.49	5.48	4.93	3.15	2.58	3.57	2.98
2^{11}	54.30	54.10	2.92	2.61	3.33	2.89	5.63	4.75	5.88	5.37	3.16	2.60	3.61	2.98
2^{12}	54.70	54.80	2.96	2.66	3.34	2.91	6.90	5.79	7.28	6.23	4.15	3.05	4.58	3.40
2^{13}	55.10	56.00	3.51	2.95	3.77	3.21	7.01	6.03	7.47	6.35	4.27	3.08	4.65	3.48
2^{14}	51.00	50.90	3.53	2.99	3.81	3.25	7.08	6.00	7.47	6.46	4.29	3.14	4.71	3.47
2^{15}	44.80	45.40	3.58	3.01	3.83	3.26	7.07	6.04	7.55	6.52	4.35	3.14	4.75	3.53
2^{16}	38.20	38.20	3.58	3.06	3.86	3.30	7.14	6.11	7.58	6.47	4.37	3.18	4.80	3.54

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS8

`vector<unordered_set<string>>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	114.0	116.0	26.00	23.80	26.30	24.00	56.20	54.70	56.90	54.60	27.50	25.80	27.90	26.00
2^{7}	123.0	130.0	26.50	24.40	25.70	23.50	62.70	60.10	62.70	60.50	27.50	26.30	28.20	26.10
2^{8}	162.0	171.0	31.70	27.30	32.20	27.80	78.00	74.20	79.20	73.90	35.00	32.00	35.50	32.50
2^{9}	175.0	181.0	36.80	28.00	38.10	28.00	81.70	74.10	81.20	74.90	36.30	32.10	37.20	32.10
2^{10}	176.0	183.0	40.00	28.90	37.40	28.20	82.10	74.50	82.10	74.70	36.90	32.00	37.40	32.20
2^{11}	176.0	183.0	39.30	28.00	37.30	28.00	81.40	74.40	82.00	74.30	36.90	32.10	37.80	32.10
2^{12}	179.0	185.0	39.40	28.00	37.10	28.00	81.80	74.10	81.60	74.40	37.00	32.00	37.80	32.20
2^{13}	173.0	178.0	39.60	27.90	36.90	28.20	81.80	73.60	81.50	74.30	37.20	32.00	37.80	32.40
2^{14}	157.0	160.0	41.00	29.90	38.80	29.90	81.50	74.10	82.20	74.00	44.00	39.30	45.10	39.20
2^{15}	122.0	131.0	47.60	35.80	44.80	36.20	85.20	75.50	83.70	76.10	50.50	45.20	51.00	45.50
2^{16}	95.40	106.0	51.40	40.50	48.10	38.90	84.80	76.20	88.70	75.90	53.10	48.50	54.80	48.20

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS9

`unordered_set<vector<int>>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	**AS1**	**AS2**	**AS3**	**AS4**	**AS5**	**AS6**	**AS7**	**AS8**	**AS9**	**AS10**	**AS11**	**AS12**	**AS13**	**AS14**
$_2$6	0.97	0.94	0.23	0.19	0.24	0.21	0.26	0.27	0.30	0.26	0.25	0.26	0.25	0.24
$_2$7	1.40	1.43	0.22	0.21	0.22	0.19	0.24	0.26	0.25	0.27	0.24	0.26	0.24	0.24
$_2$8	1.35	1.39	0.25	0.22	0.24	0.23	0.30	0.35	0.34	0.33	0.24	0.23	0.25	0.24
$_2$9	1.29	1.32	0.22	0.18	0.22	0.17	0.37	0.38	0.37	0.36	0.23	0.22	0.19	0.22
$_2$10	1.32	1.38	0.24	0.22	0.22	0.19	0.41	0.39	0.42	0.39	0.23	0.24	0.23	0.22
$_2$11	1.34	1.36	0.23	0.21	0.22	0.17	0.44	0.42	0.43	0.41	0.23	0.23	0.25	0.22
$_2$12	1.34	1.41	0.22	0.20	0.22	0.16	0.46	0.42	0.45	0.43	0.23	0.17	0.27	0.22
$_2$13	1.46	1.54	0.22	0.18	0.22	0.16	0.48	0.49	0.49	0.48	0.23	0.21	0.25	0.21
$_2$14	1.53	1.61	0.22	0.17	0.22	0.18	0.43	0.42	0.45	0.41	0.24	0.22	0.24	0.22
$_2$15	1.61	1.76	0.25	0.21	0.24	0.19	0.50	0.49	0.50	0.49	0.24	0.18	0.23	0.21
$_2$16	1.79	1.92	0.28	0.25	0.29	0.24	0.55	0.51	0.56	0.55	0.30	0.23	0.32	0.24

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS10

`unordered_set<vector<string>>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	73.00	73.20	9.41	9.39	9.34	8.97	41.70	39.70	41.10	39.30	11.20	10.40	11.20	10.30
2^7	74.70	75.30	9.32	9.34	9.24	8.87	46.20	43.70	45.30	44.20	12.70	10.60	11.40	10.80
2^8	83.10	85.40	18.00	17.30	16.90	17.20	62.20	58.90	61.90	57.60	23.20	22.30	23.10	22.40
2^9	91.40	94.90	19.00	19.00	18.80	18.60	65.00	59.90	64.40	58.90	24.30	22.60	24.10	22.60
2^{10}	98.20	101.0	19.20	18.90	19.10	18.60	66.50	59.70	65.40	59.10	24.80	22.60	24.60	22.70
2^{11}	99.50	101.0	19.00	19.10	19.30	18.40	66.90	59.50	66.10	58.70	24.90	22.70	25.10	22.50
2^{12}	102.0	105.0	19.40	19.00	19.20	18.80	67.00	58.90	65.80	59.40	25.30	22.60	25.10	22.70
2^{13}	103.0	104.0	19.00	19.20	19.40	18.40	66.70	59.20	66.20	58.20	25.30	22.90	25.50	22.60
2^{14}	95.80	97.20	19.80	20.00	20.30	19.30	62.80	55.60	61.90	54.30	29.20	26.80	29.60	26.50
2^{15}	87.10	89.80	24.00	23.70	24.00	23.50	64.30	55.00	61.90	54.90	33.60	30.80	33.50	31.00
2^{16}	77.10	78.20	25.60	25.70	26.00	25.10	63.90	55.50	63.30	54.50	35.30	33.00	35.70	32.60

3. Analyzing the Benchmark Data

Benchmark I: DENSITY, VARIATION

DS11 `unordered_set<unordered_set<int>>`

	Global		**Monotonic**				**Multipool**				**Multipool<Mono>**			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	28.70	29.10	3.06	2.75	3.55	3.14	4.96	4.40	5.41	4.84	3.24	2.73	3.73	3.15
2^7	29.10	29.00	3.02	2.71	3.47	3.06	5.03	4.52	5.49	4.89	3.23	2.66	3.68	3.08
2^8	28.80	29.10	3.00	2.68	3.45	3.04	5.18	4.55	5.57	4.98	3.24	2.66	3.65	3.06
2^9	31.80	32.30	2.99	2.64	3.43	2.98	5.12	4.54	5.55	4.95	3.22	2.60	3.65	2.99
2^{10}	46.50	47.10	2.95	2.65	3.40	2.99	5.13	4.57	5.62	4.96	3.21	2.58	3.62	2.97
2^{11}	53.30	53.50	2.94	2.64	3.43	2.96	5.58	4.84	5.75	5.39	3.20	2.63	3.67	3.01
2^{12}	54.60	55.00	3.02	2.66	3.43	2.98	6.47	5.94	6.99	6.28	3.83	3.00	4.21	3.38
2^{13}	56.50	56.50	3.38	2.98	3.72	3.26	7.04	6.04	7.48	6.45	4.15	3.03	4.58	3.39
2^{14}	52.10	52.20	3.50	2.99	3.88	3.25	7.35	6.07	7.83	6.59	4.33	3.05	4.76	3.38
2^{15}	45.70	46.20	3.62	2.99	3.95	3.27	7.70	6.39	8.11	6.83	4.43	3.06	4.81	3.44
2^{16}	39.30	39.30	3.72	3.05	4.03	3.31	7.57	6.30	8.09	6.61	4.52	3.10	4.92	3.45

3. Analyzing the Benchmark Data

Benchmark I: **DENSITY, VARIATION**

DS12 `unordered_set<unordered_set<string>>`

	Global		Monotonic				Multipool				Multipool<Mono>			
		Virtual			Virtual				Virtual				Virtual	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
Size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	121.0	125.0	25.90	23.70	26.10	23.90	56.30	54.50	56.70	54.70	27.40	25.80	27.80	26.00
2^7	141.0	145.0	26.40	24.30	25.60	23.40	62.10	59.60	62.50	60.00	27.90	25.80	28.30	25.80
2^8	165.0	173.0	31.50	27.30	32.20	27.70	77.40	73.70	77.80	74.20	34.80	31.90	35.60	32.20
2^9	171.0	178.0	35.90	27.60	34.40	27.80	80.00	73.70	79.70	74.60	35.70	32.00	36.50	31.90
2^{10}	177.0	182.0	38.70	28.60	35.60	27.90	81.10	74.30	81.30	74.30	36.70	31.80	37.10	32.00
2^{11}	177.0	183.0	38.20	27.60	36.20	27.70	81.30	74.30	82.20	74.10	37.00	32.00	37.80	31.90
2^{12}	179.0	186.0	39.10	27.70	36.50	28.00	81.60	73.50	81.50	74.10	37.30	31.80	37.90	32.10
2^{13}	165.0	169.0	39.00	27.80	36.70	27.80	81.30	73.90	82.80	73.50	37.30	32.10	38.30	32.10
2^{14}	153.0	156.0	40.90	29.60	38.70	29.60	81.50	74.10	82.40	73.70	44.40	39.20	45.40	39.10
2^{15}	122.0	131.0	47.60	35.70	44.80	36.10	85.70	75.20	83.90	75.40	51.00	45.10	51.40	45.50
2^{16}	100.0	111.0	51.40	40.40	48.00	38.80	85.10	75.50	86.20	75.60	53.60	48.40	54.60	48.20

Benchmark I: **D**ENSITY, **V**ARIATION

Questions

and/or

Discussion?

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I. **Short Running:** Build Up, Use, Tear Down
   - Allocation **D**ENSITY and **V**ARIATION in Allocated Sizes

II. **Long Running:** Time-Multiplexed Subsystems
   - Access **L**OCALITY – both *Physica*l and *Temporal*

III. **Short Running:** Varying Memory Reusability
   - Memory **U**TILIZATION

IV. **Multithreaded:** Varying Numbers of Threads
   - Allocator **C**ONTENTION

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Considerations:

- Investigate access LOCALITY.
  - Observe its effects both *physically* and *temporally*.
- Simulate concurrent subsystems.
  - Vary both their *sizes* and *time slices* independently.
- Access LOCALITY ***should*** dominate results.
  - Time accessing data should be substantial (hours).
  - Time to set-up/tear-down should be negligible (seconds).
- Simulate use of local allocator using global allocator.
  - Measure runtime affects of ***memory diffusion*** using *AS0*.

Benchmark II: LOCALITY

Creation Plan:

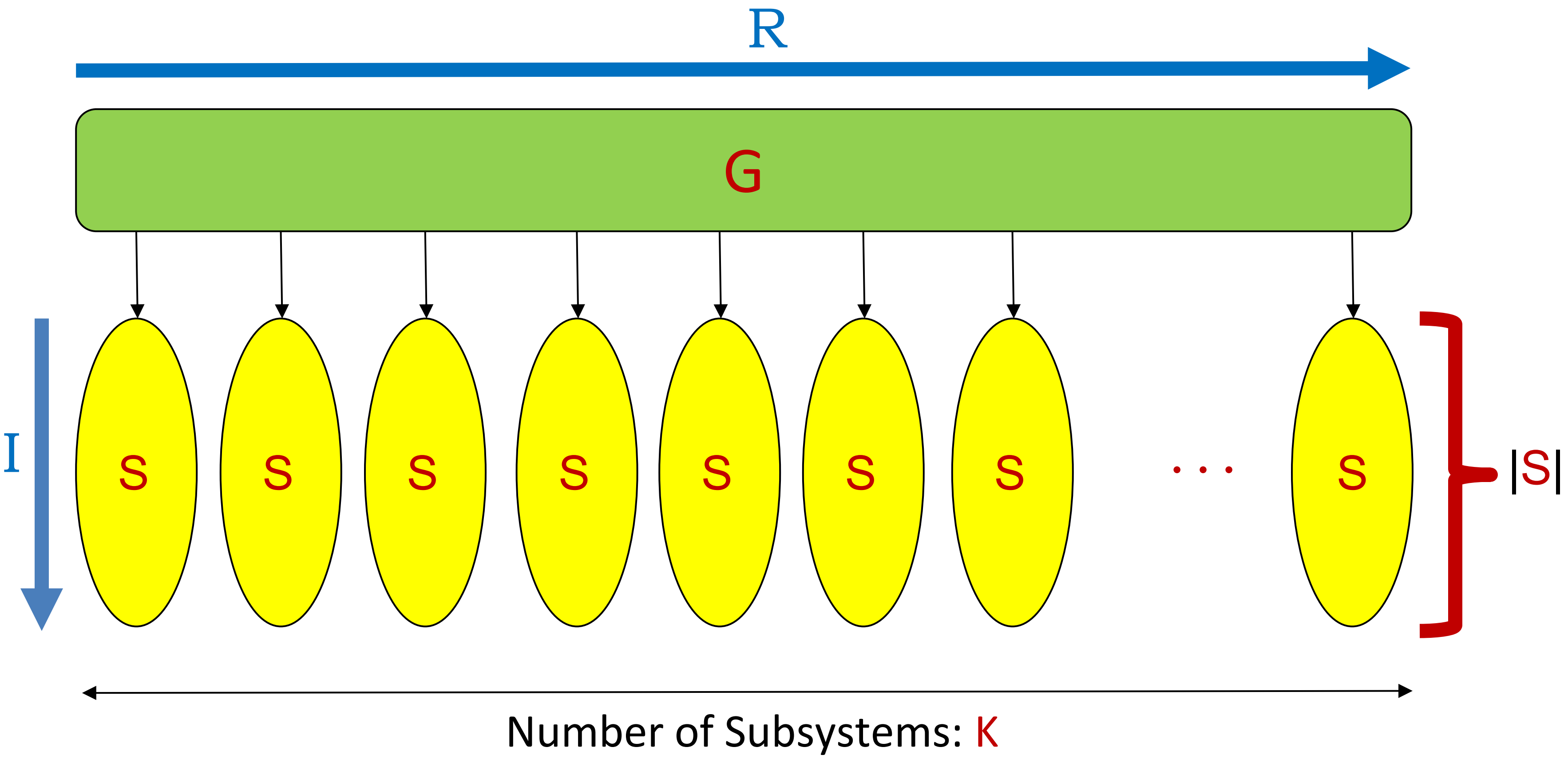
- Build up a data structure **G**: `vector<list<int>>`
  - of K subsystems: **G**.`size()`
  - each of size |**S**|: **S**.`length()`
- Tear down data structure **G**.
  - Occurs automatically at the end of the program.
- The result is an initialized data structure **G**.
  - Also used to measure the Build-up + Tear-down runtime.

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

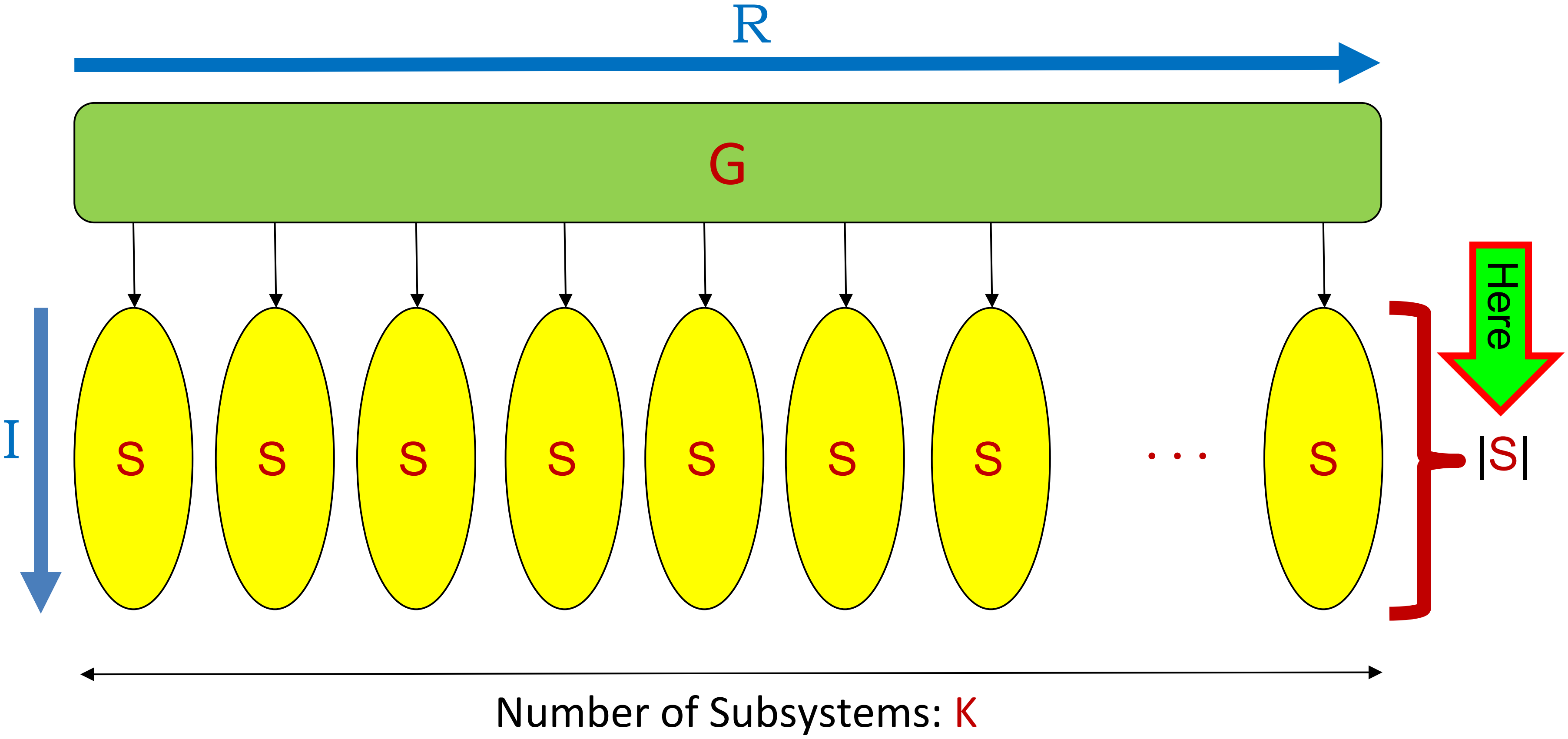


Number of Subsystems: K

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot \mathrm{I} \cdot \mathrm{R}$

Physical System Size $|G| = K \cdot |S|$



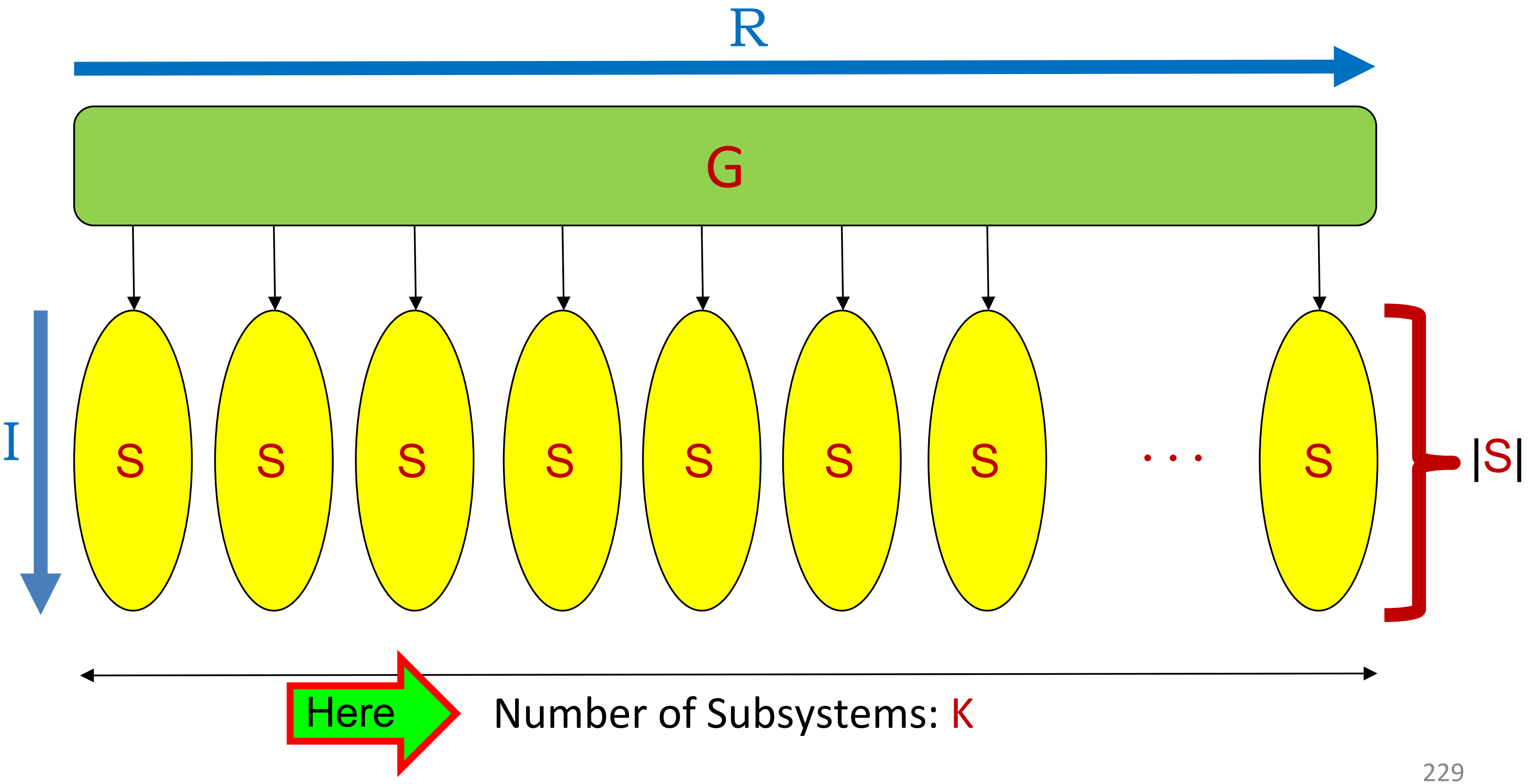
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|



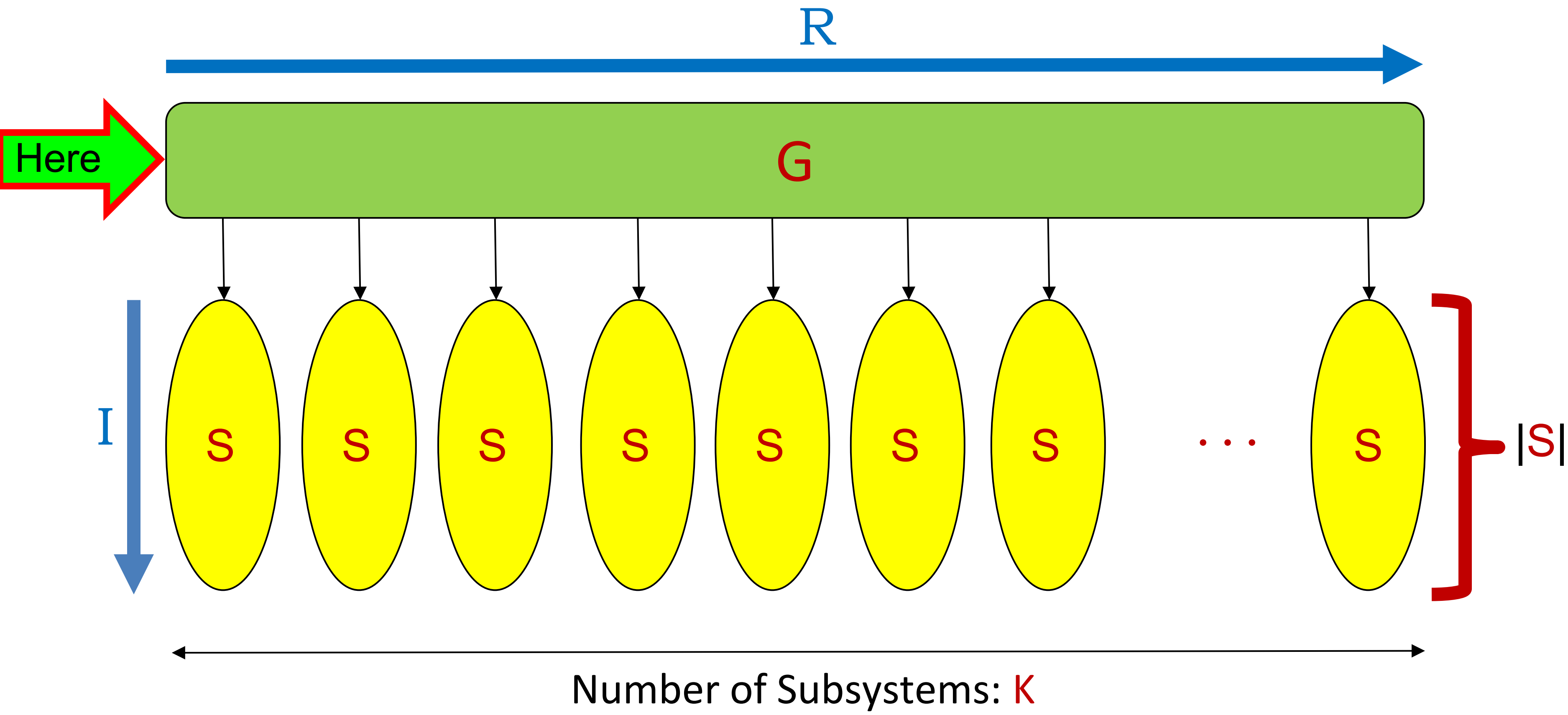
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Here

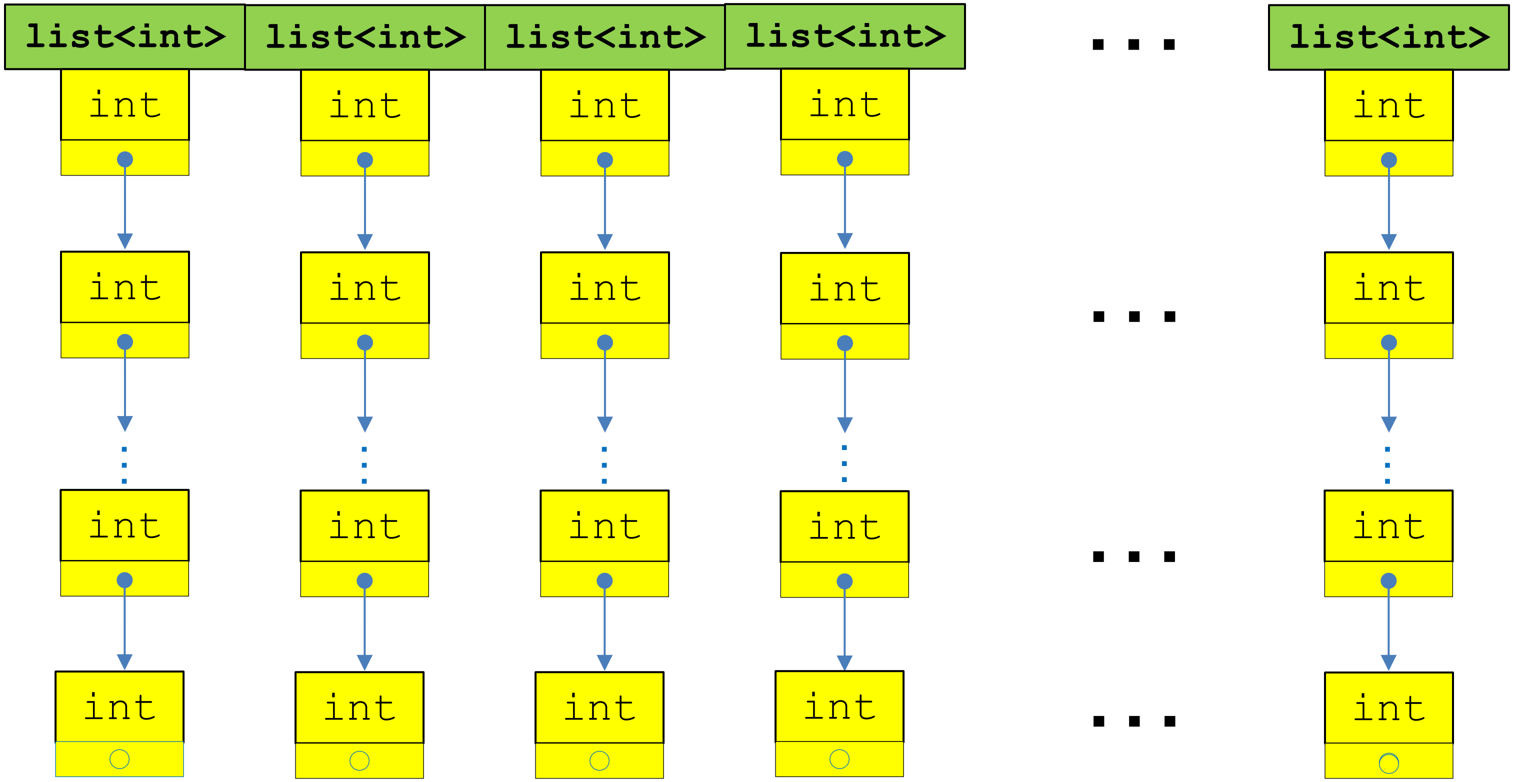
Physical System Size $|G| = K \cdot |S|$



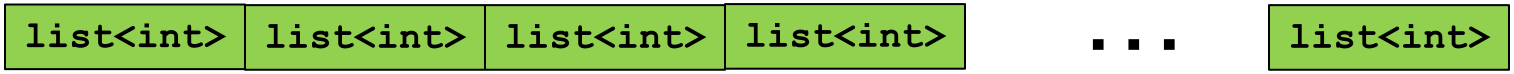
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

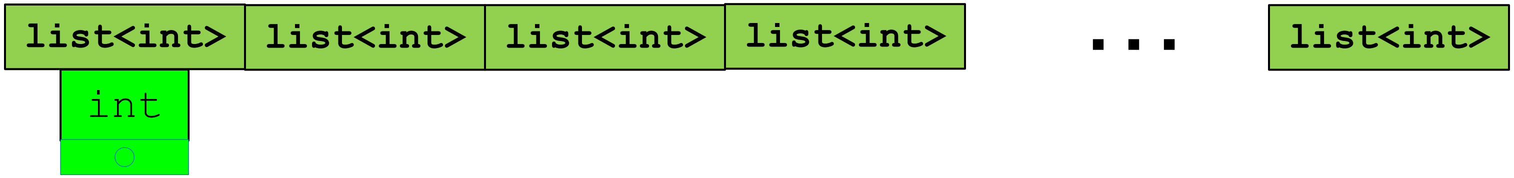


Benchmark II: LOCALITY



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



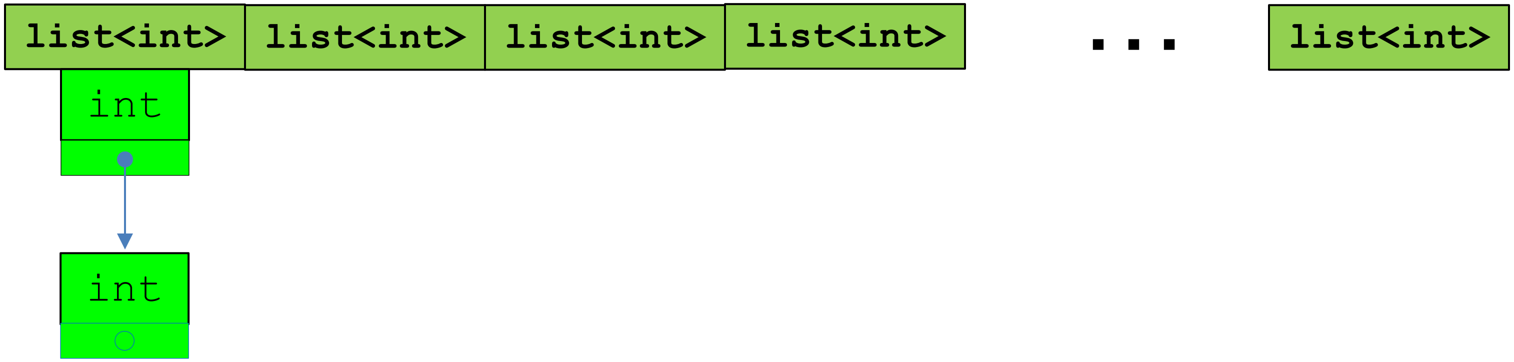
`list<int>` `list<int>` `list<int>` `list<int>` . . . `list<int>`

`int`

○

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



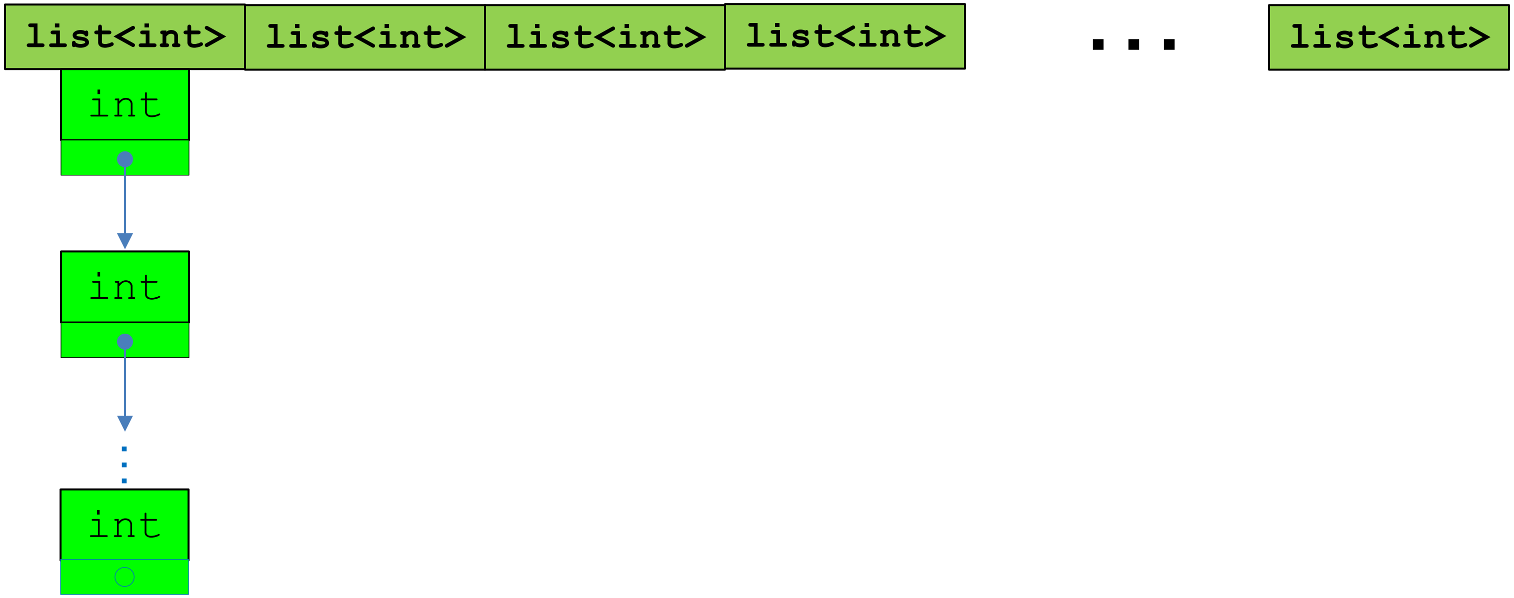
`list<int>` `list<int>` `list<int>` `list<int>` . . . `list<int>`

`int`

`int`

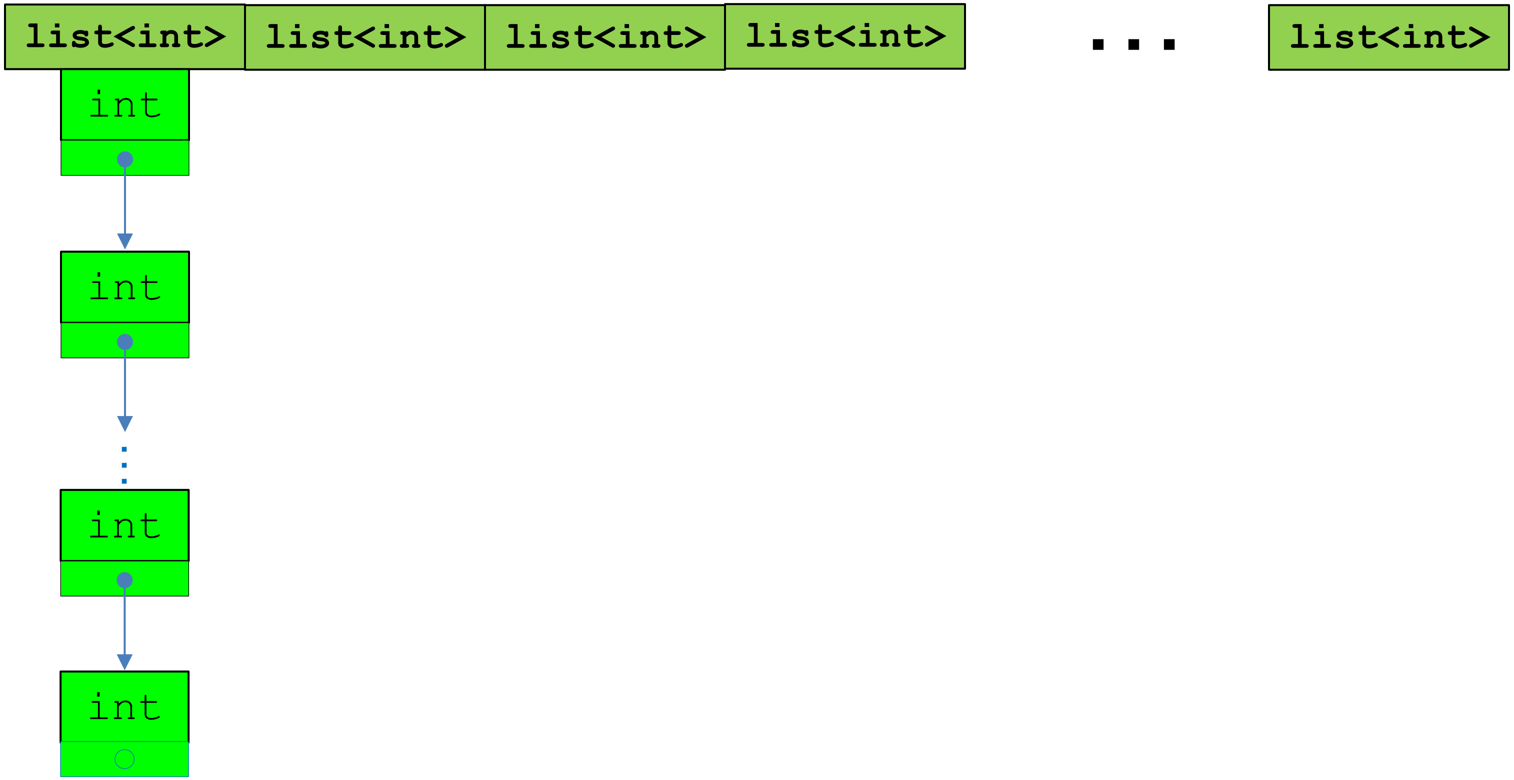
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



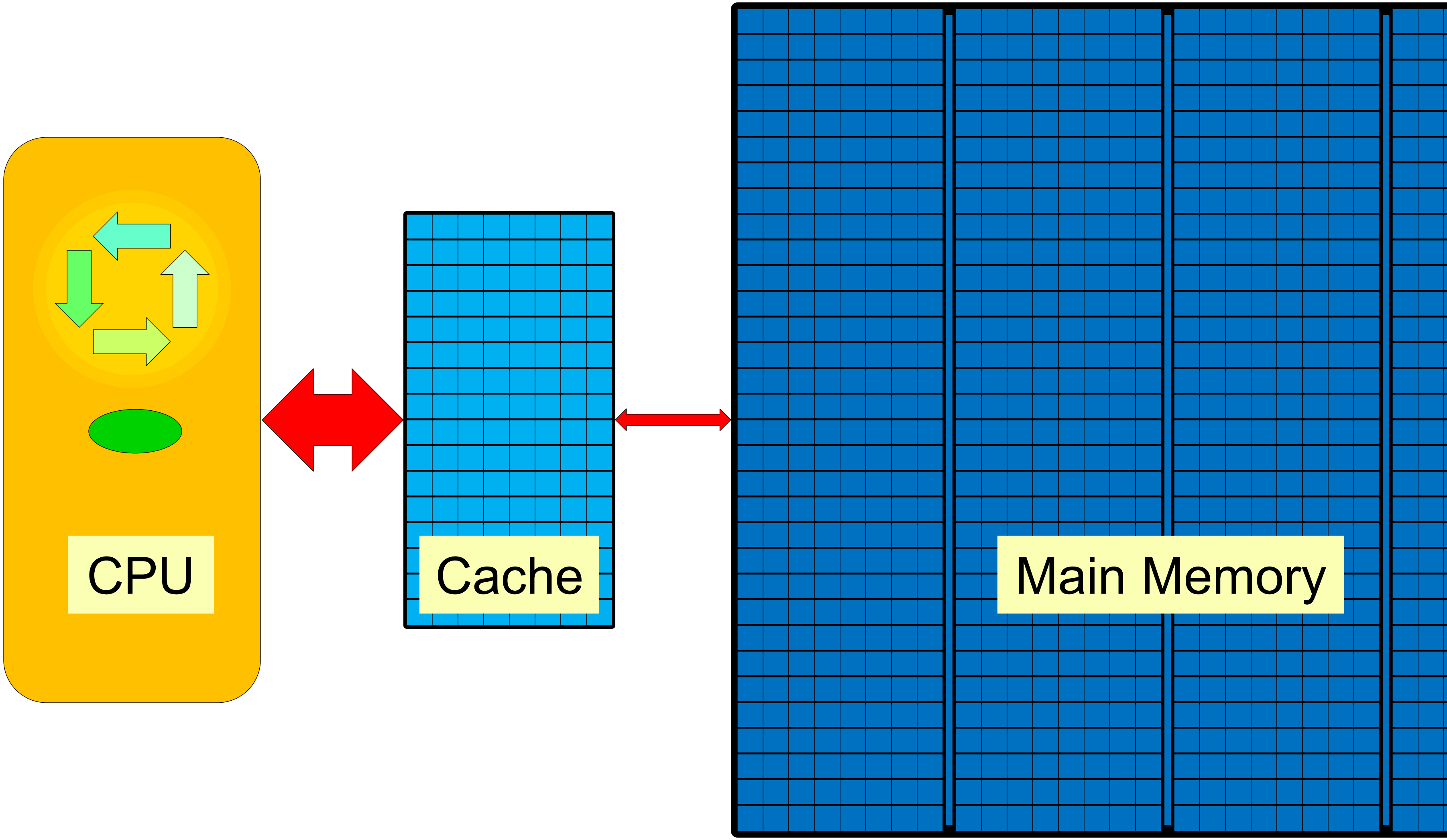
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



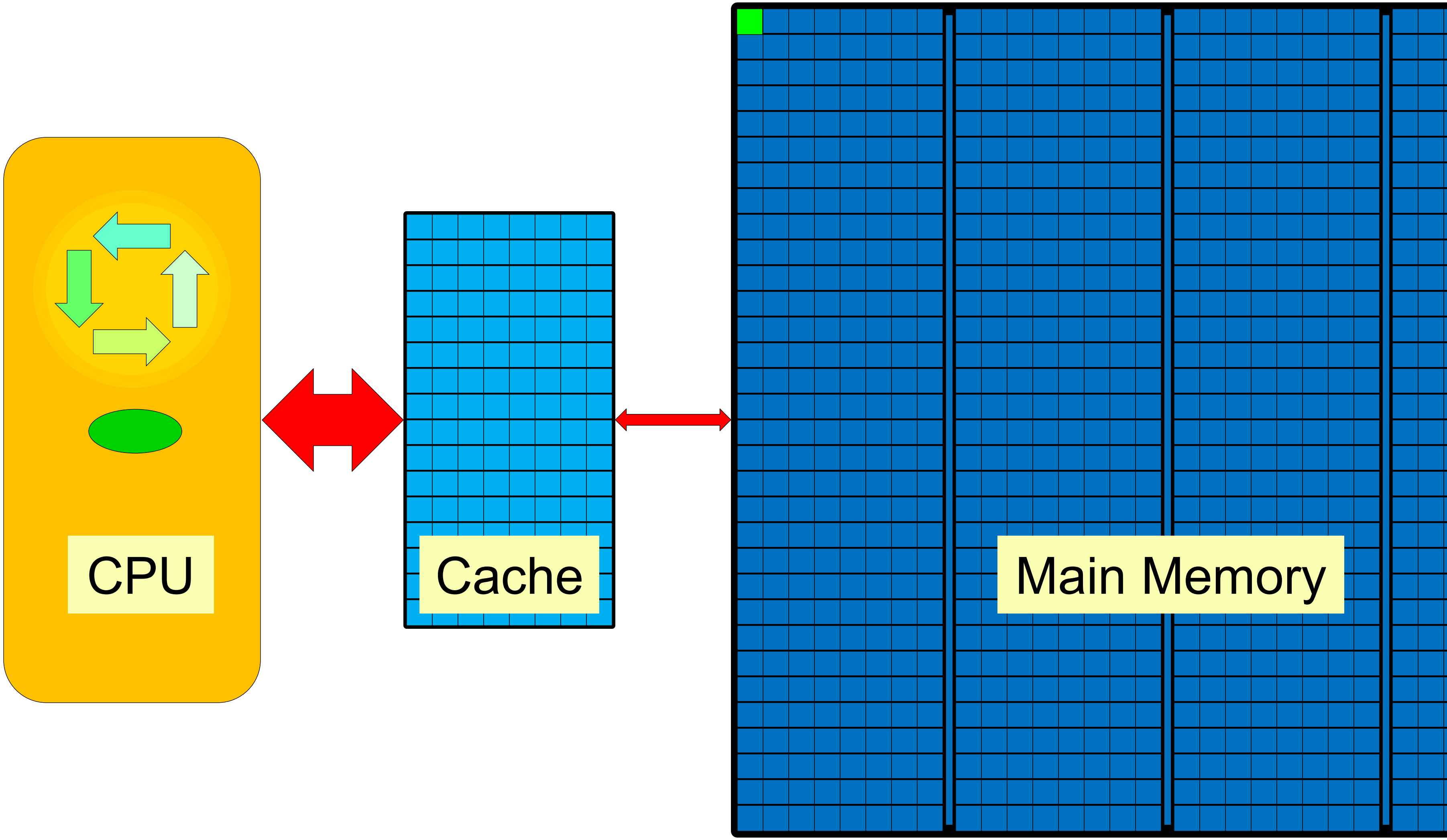
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



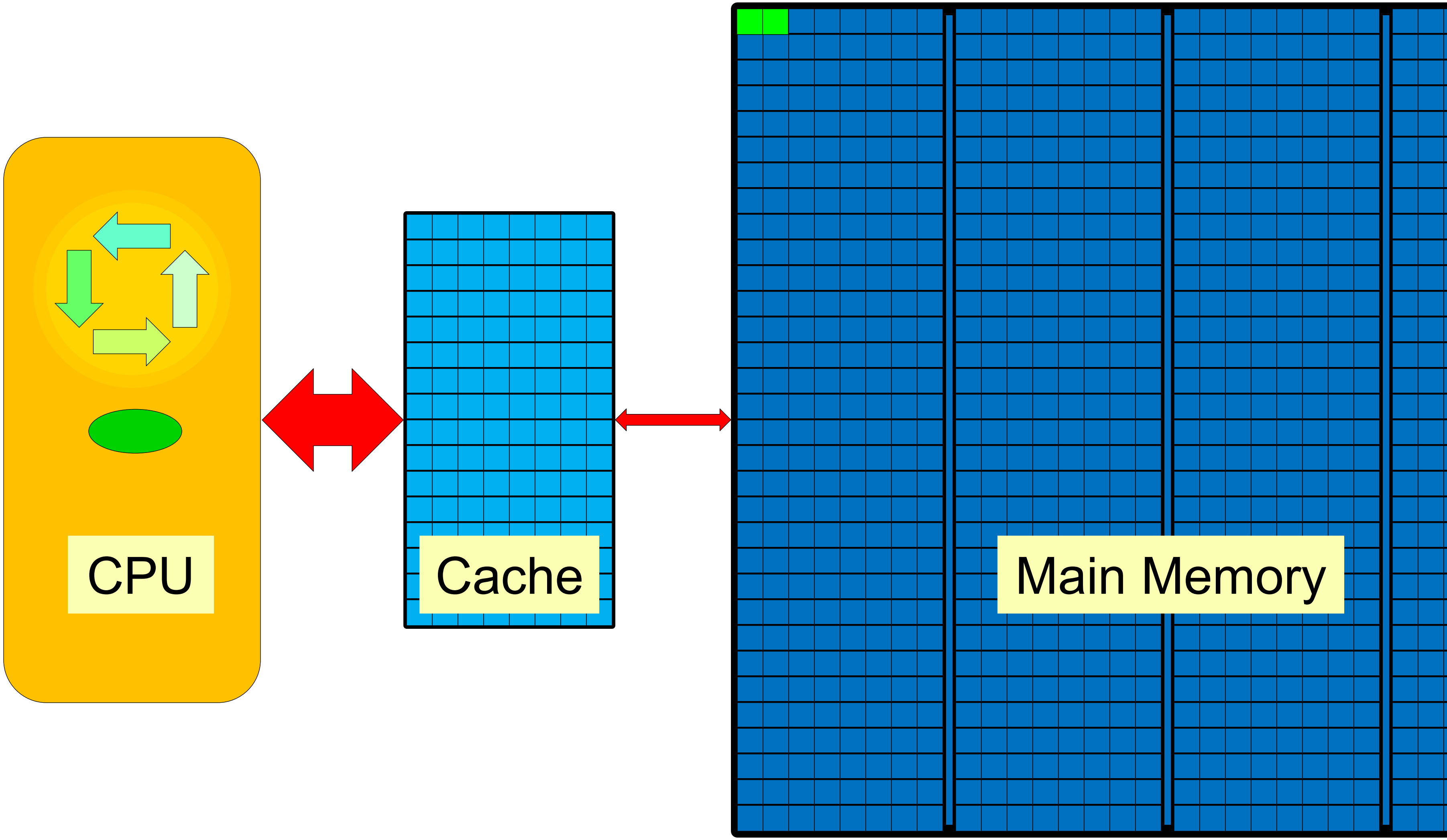
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

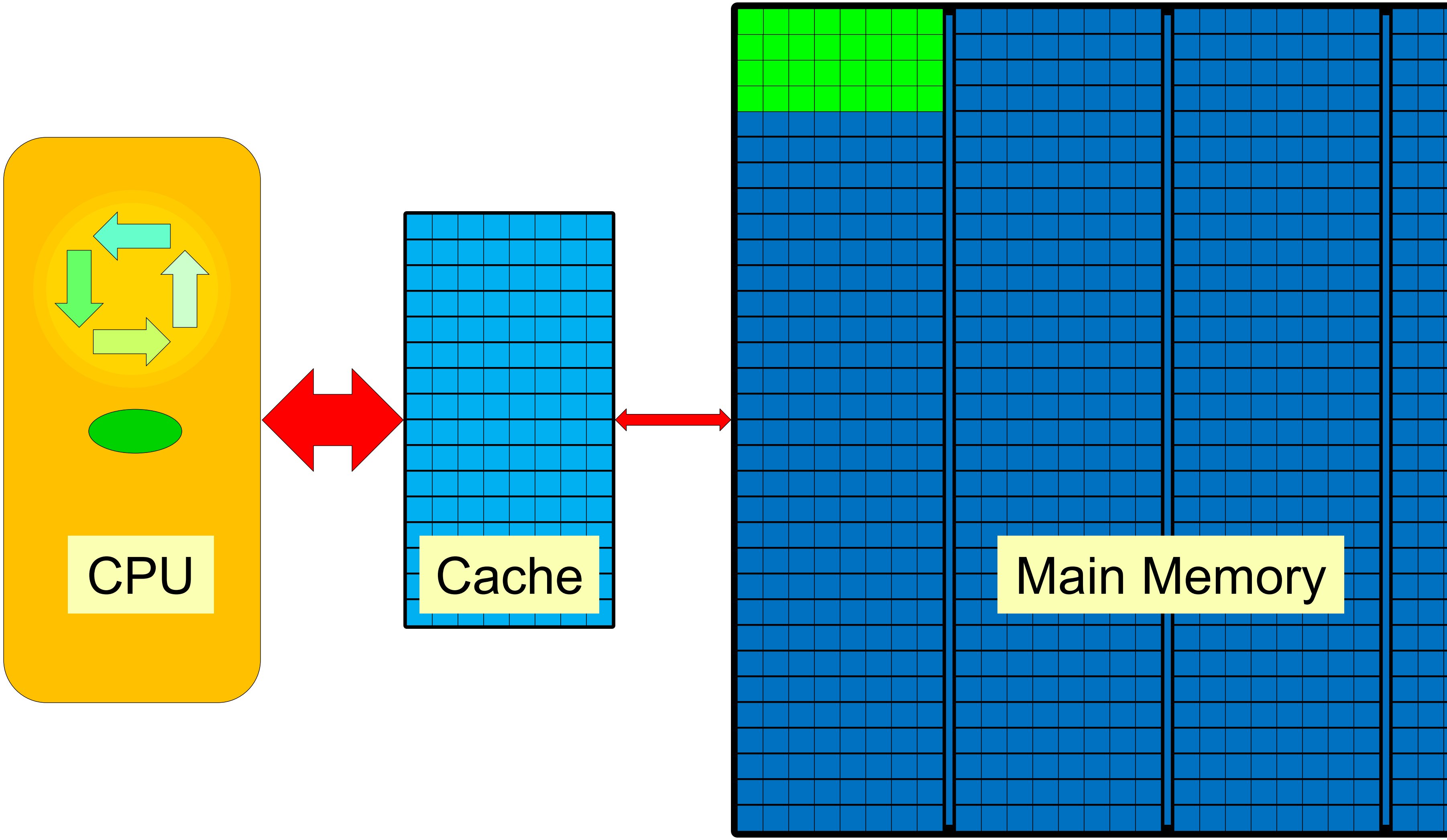


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

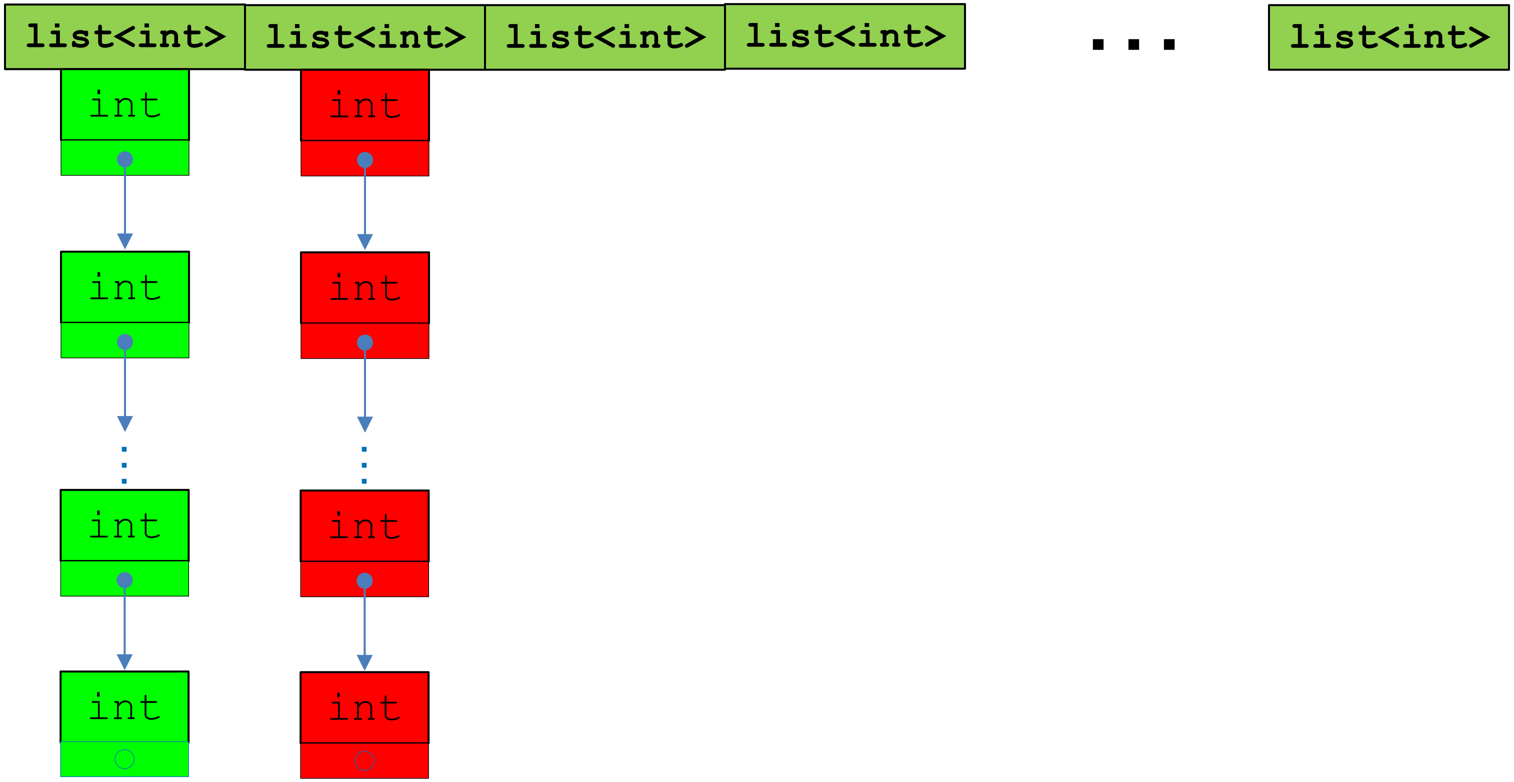


Benchmark II: LOCALITY



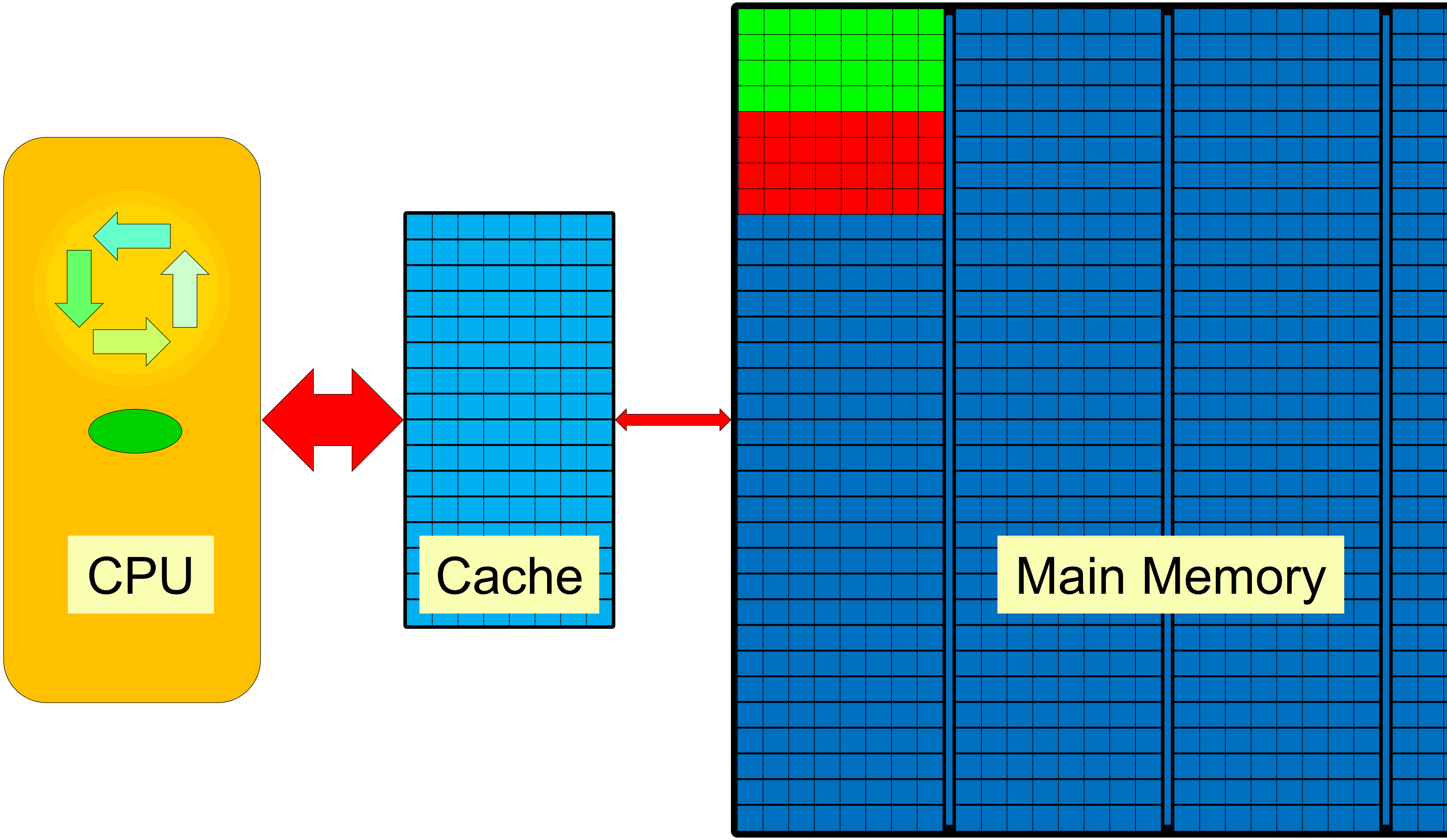
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



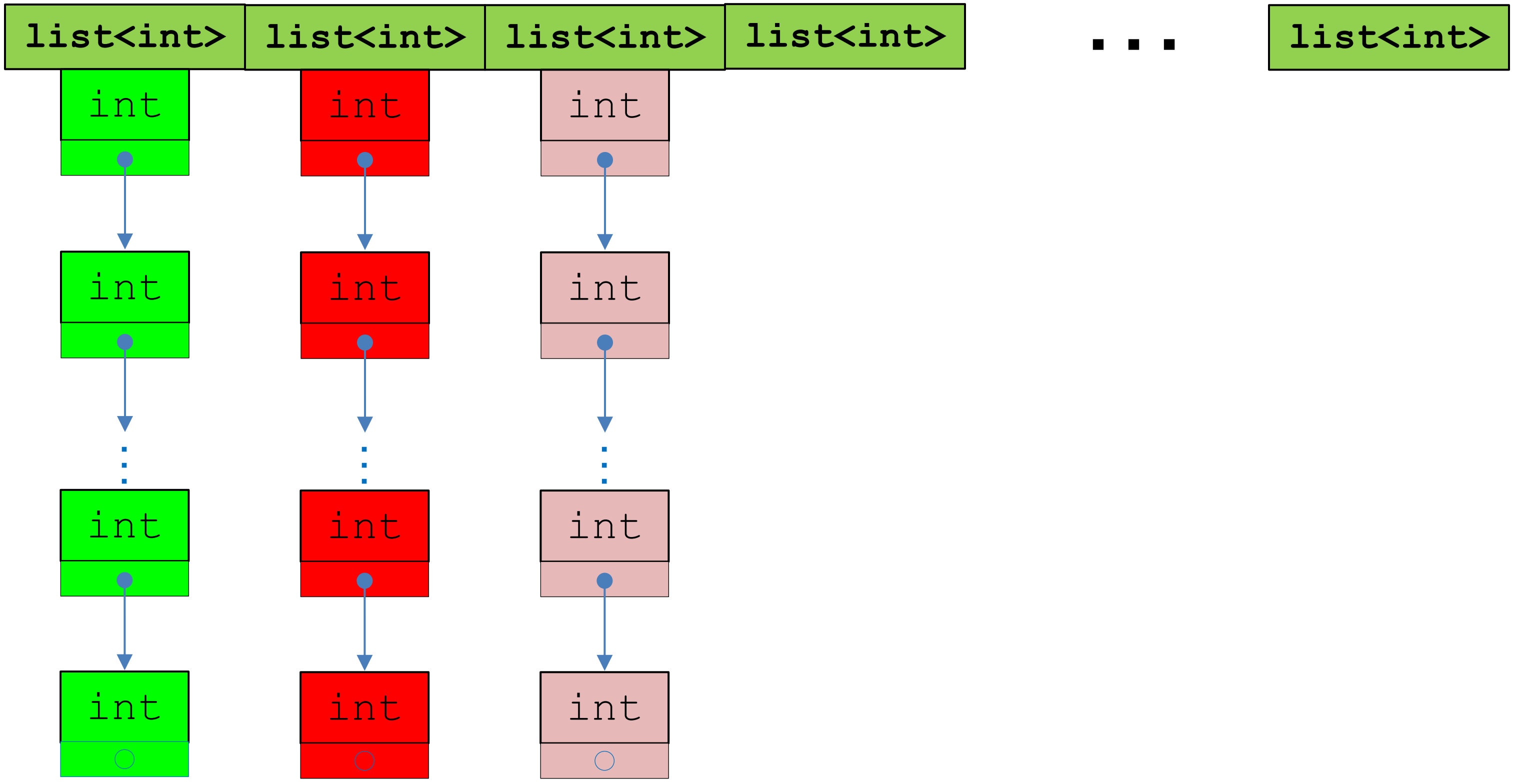
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

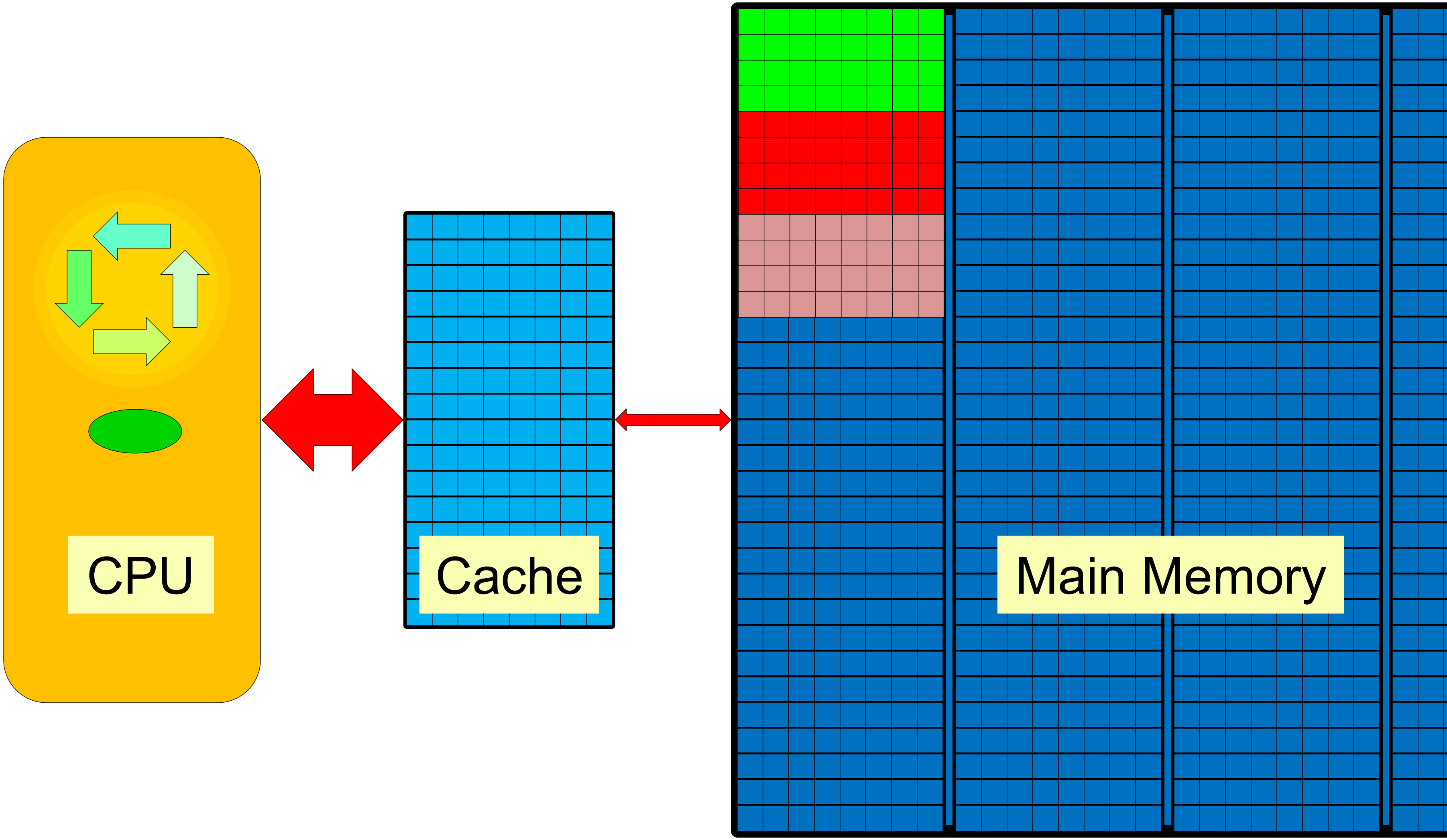


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

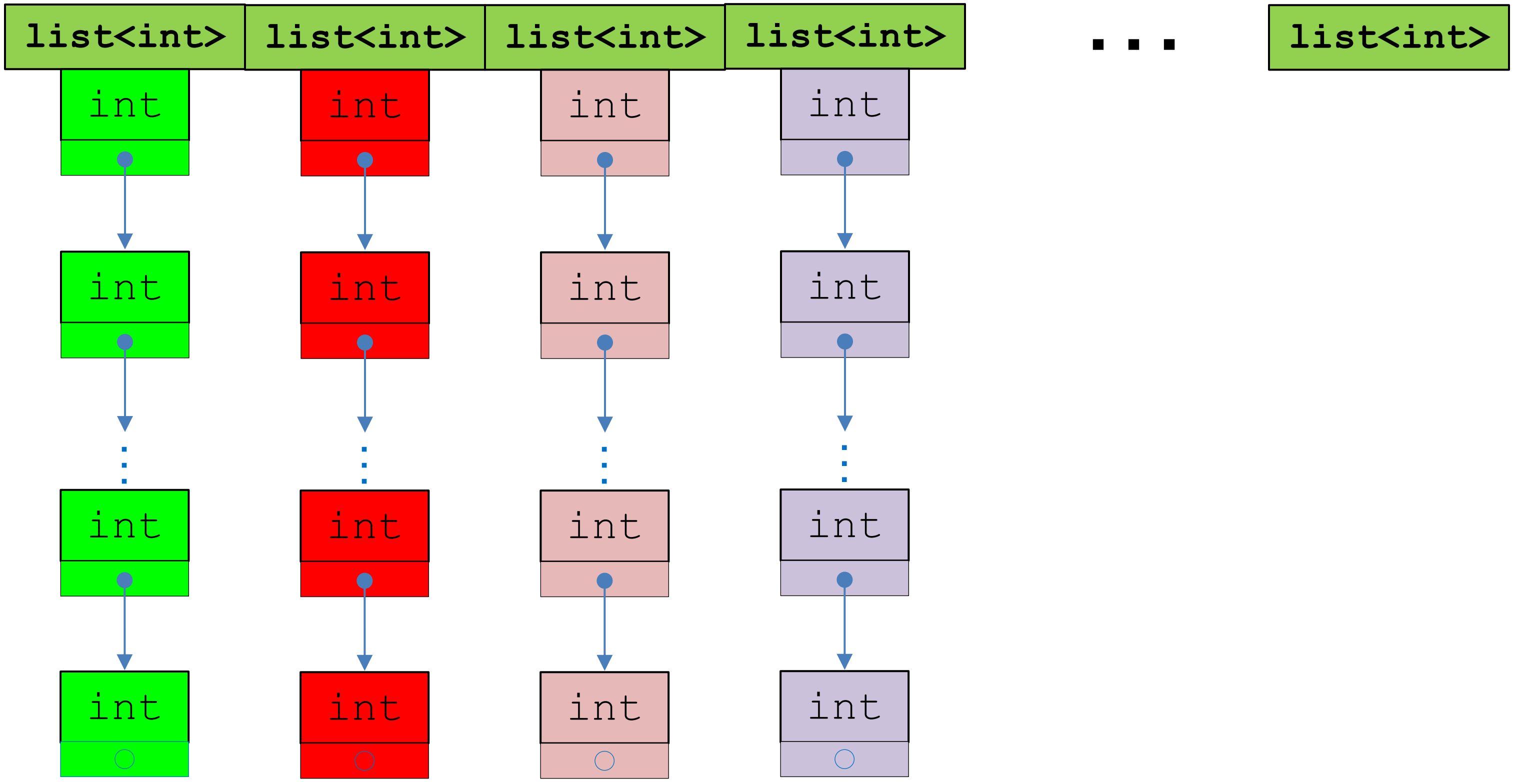


Benchmark II: LOCALITY



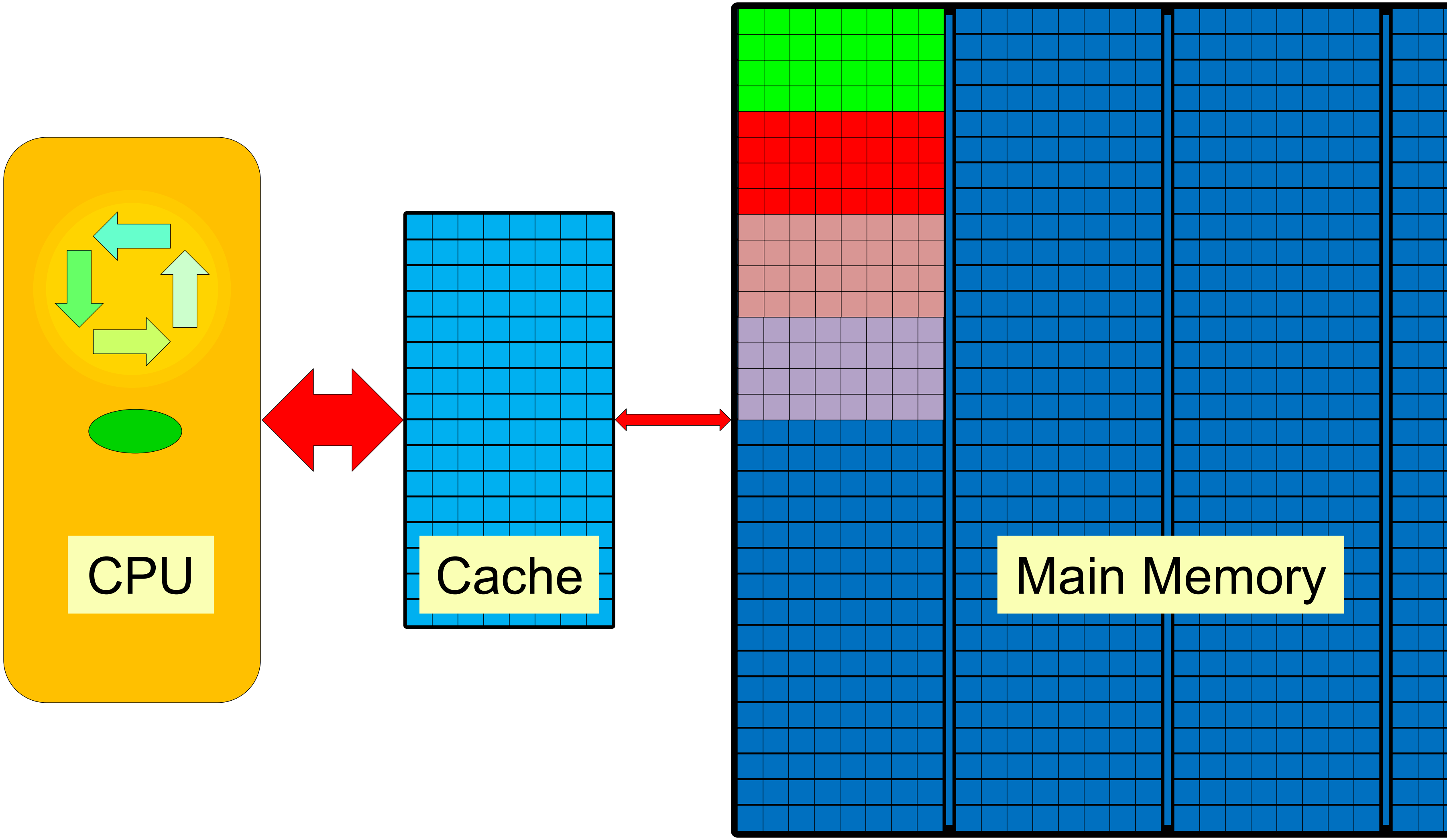
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



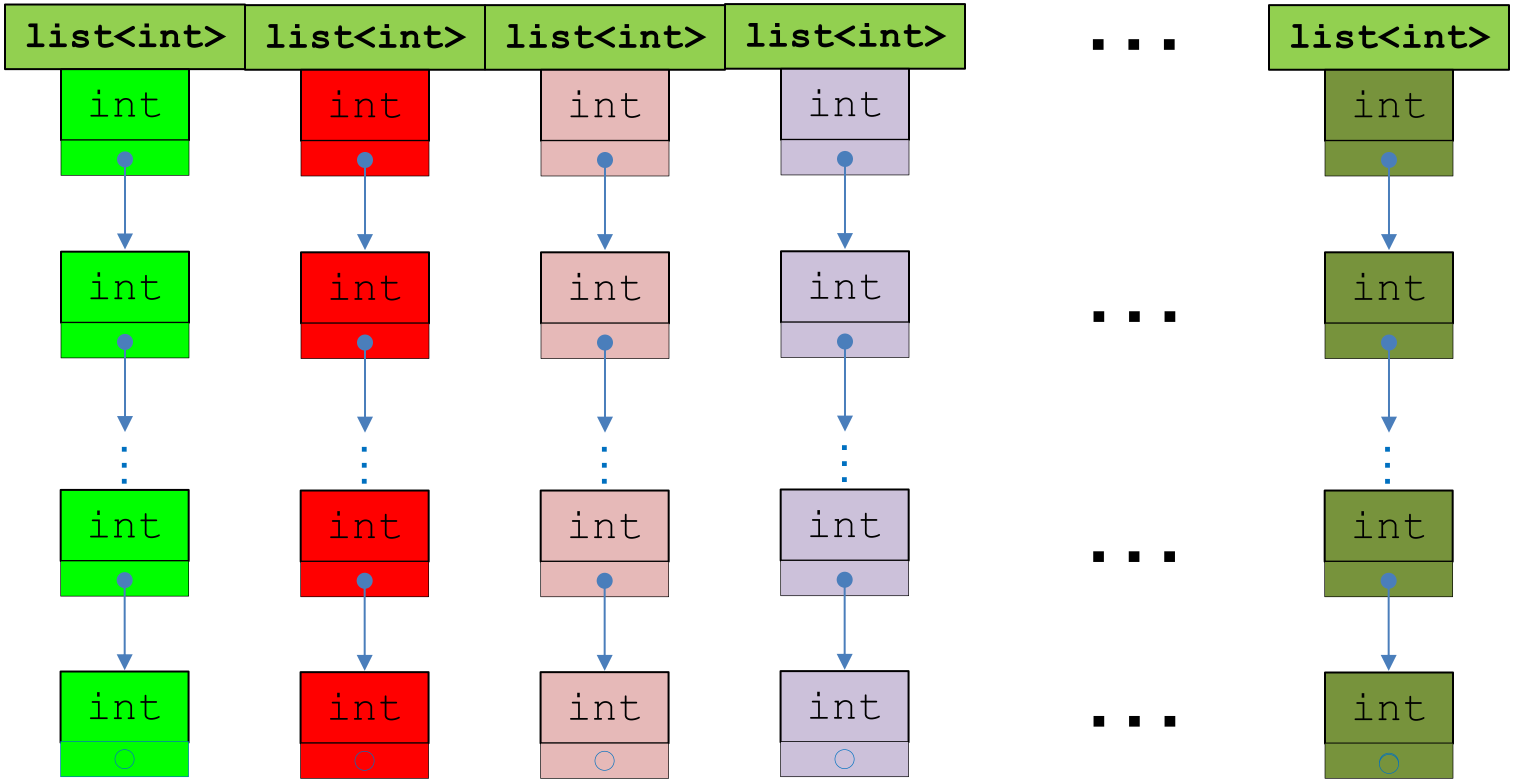
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



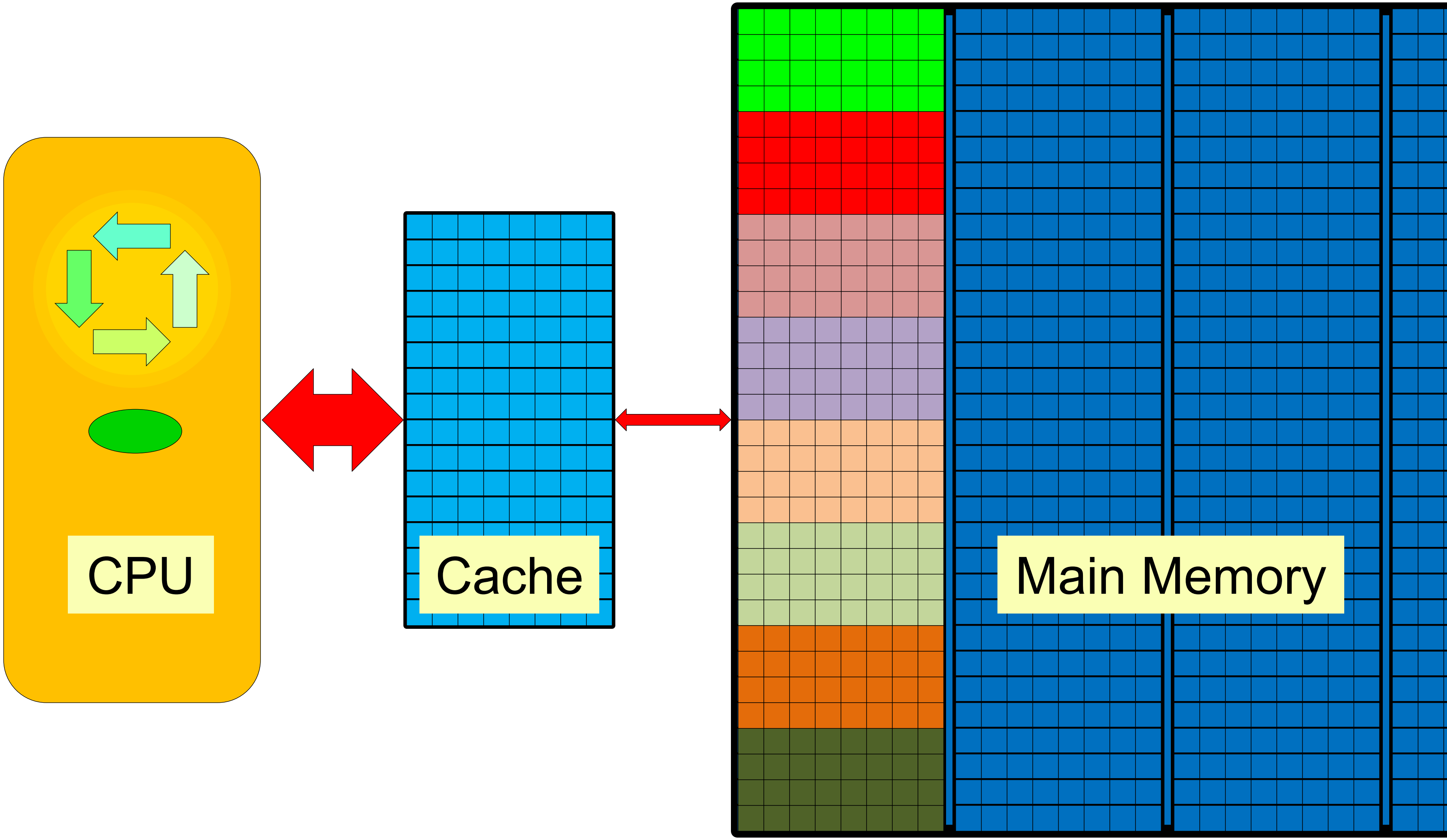
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Access Plan:

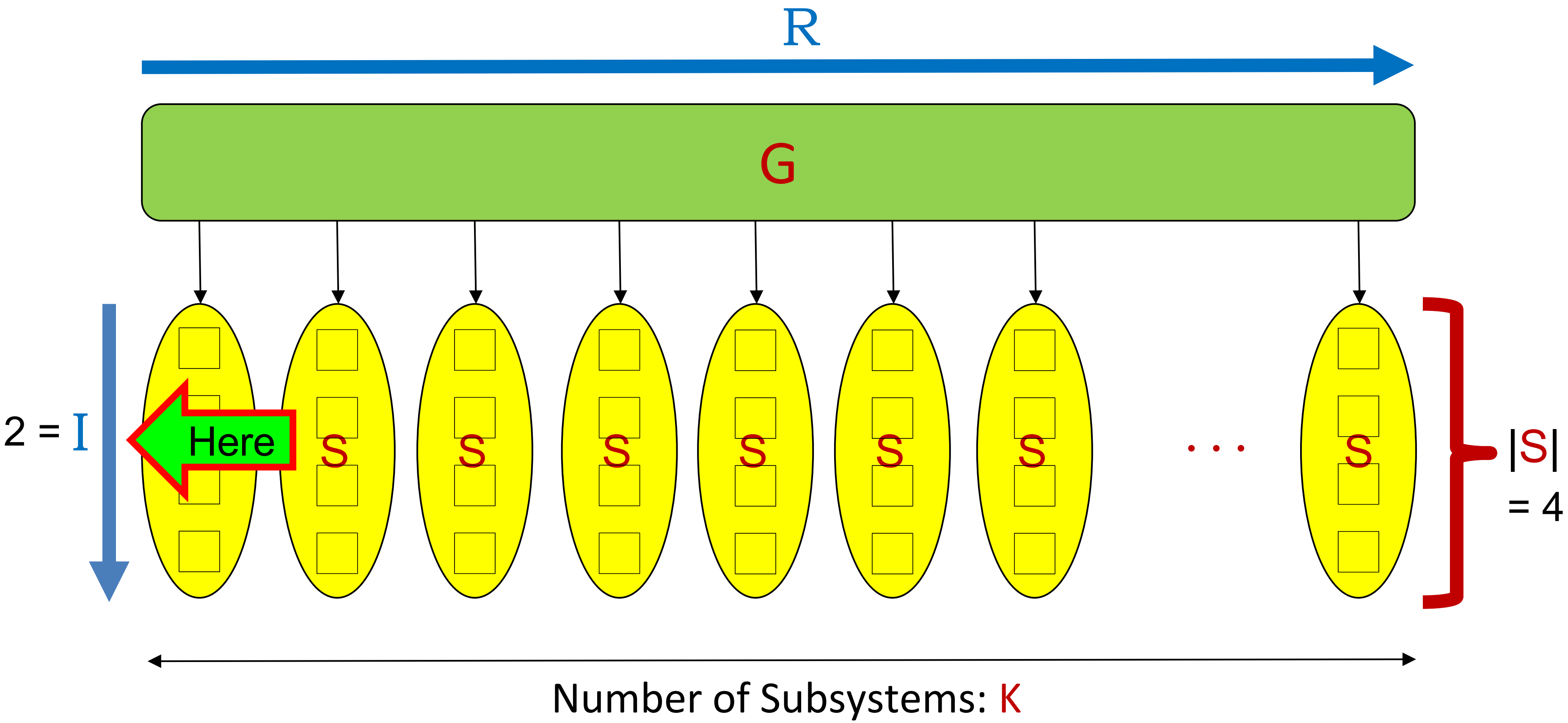
- Visit each of the subsystems $\mathbf{S}_\mathbf{j}$ of $\mathbf{G}$ in turn.
  - Traverse $\mathbf{S}_\mathbf{j}$ – $\mathbf{I}$ times – before moving to the next.
    - Increment the `int` value of each link of the list in $\mathbf{S}_\mathbf{j}$
  - Repeat (until the problem size $\mathbf{N}$ is reached).
    - We chose the overall problem size $\mathbf{N} >> 2^{30}$ (very c large).
- The result of this experiment is its (wall) **RUNTIME.**
  - No memory is allocated or deallocated.

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

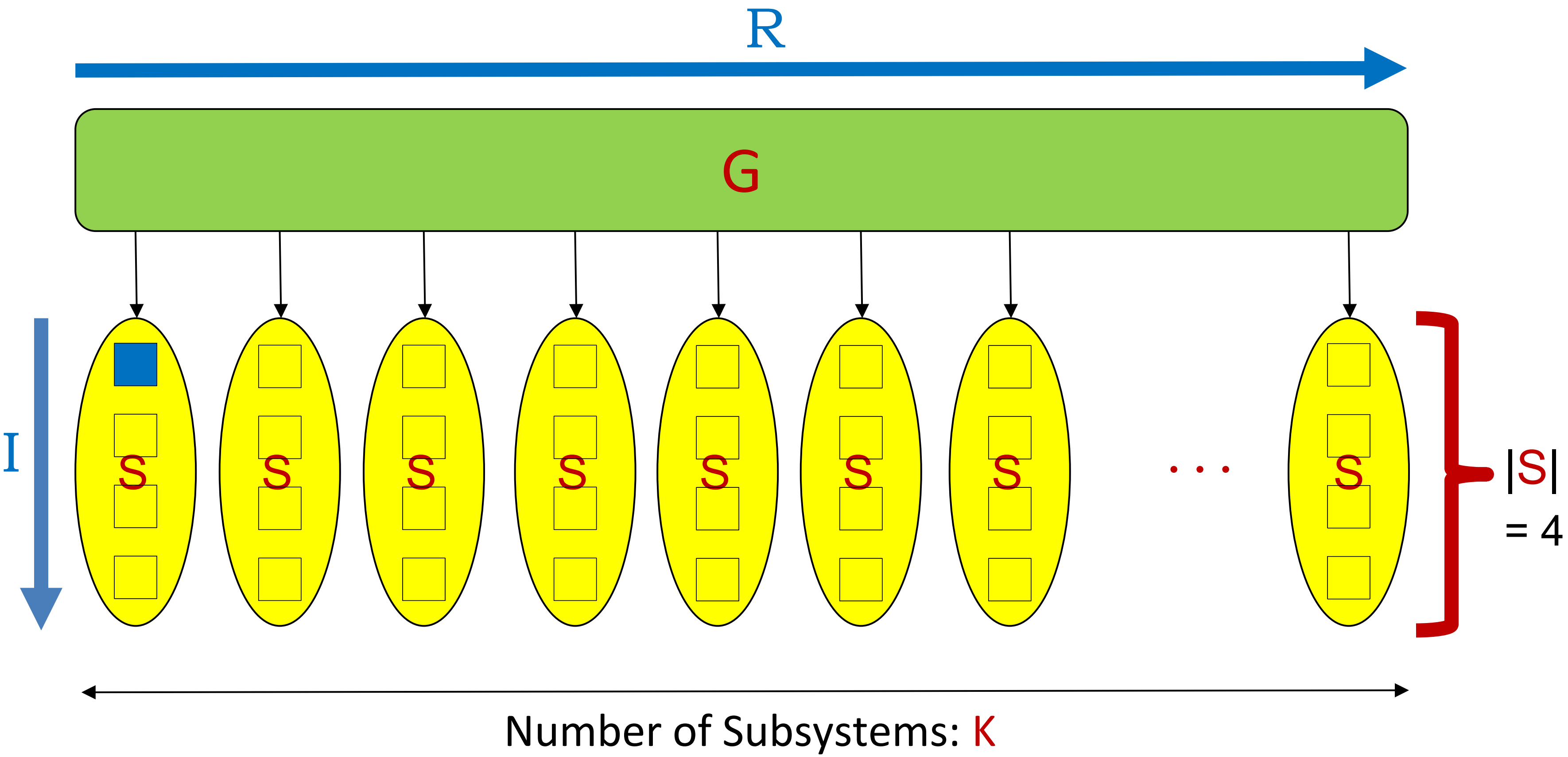


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

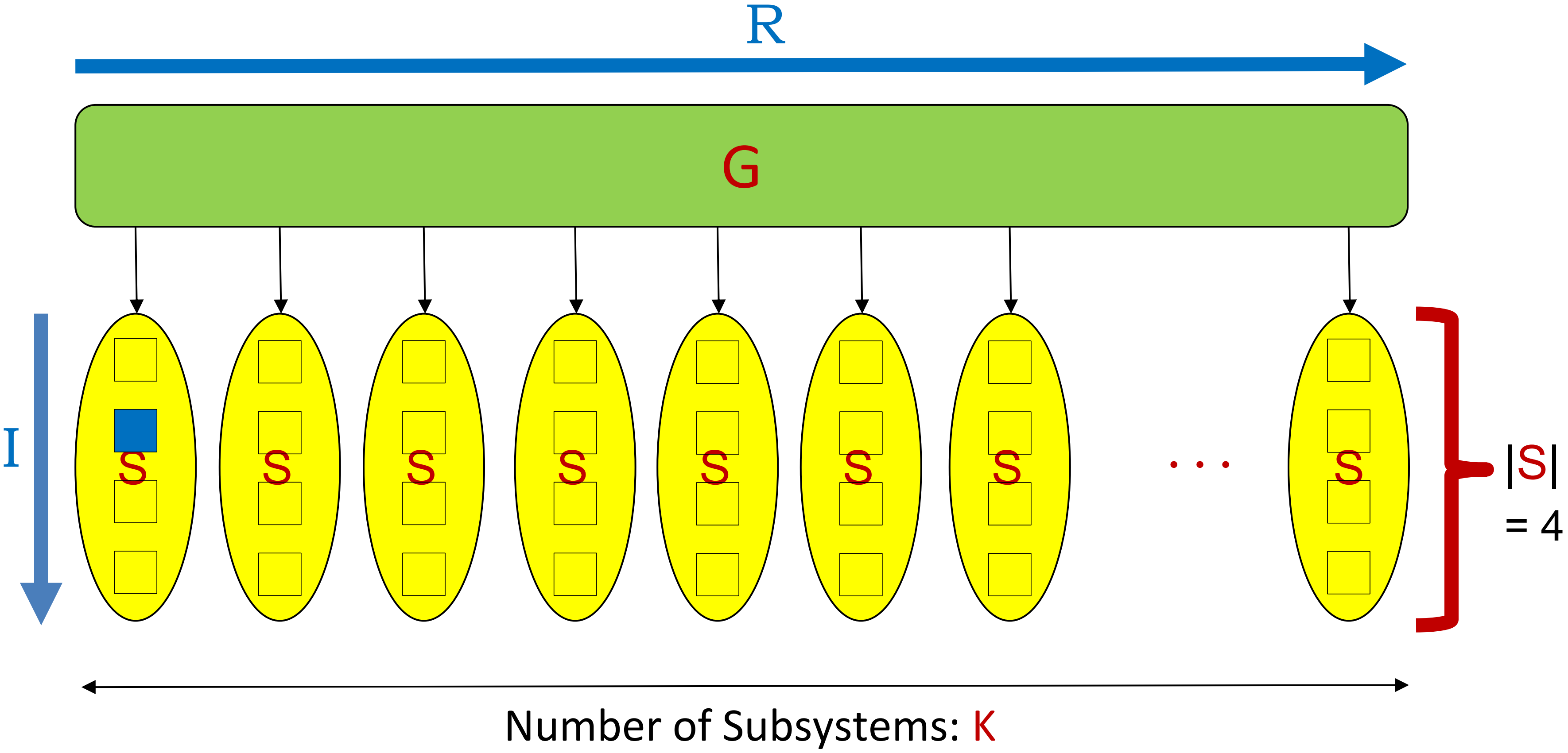


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

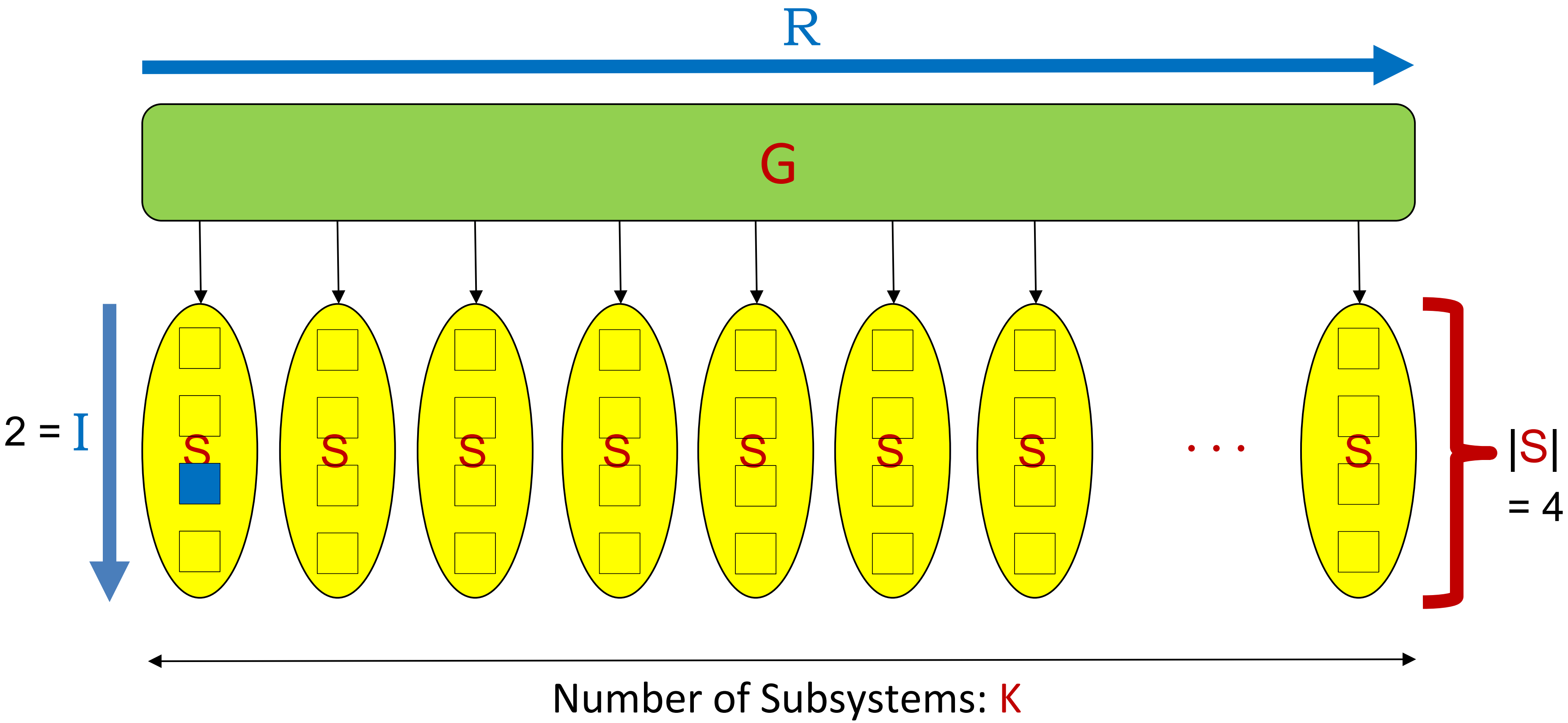


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$



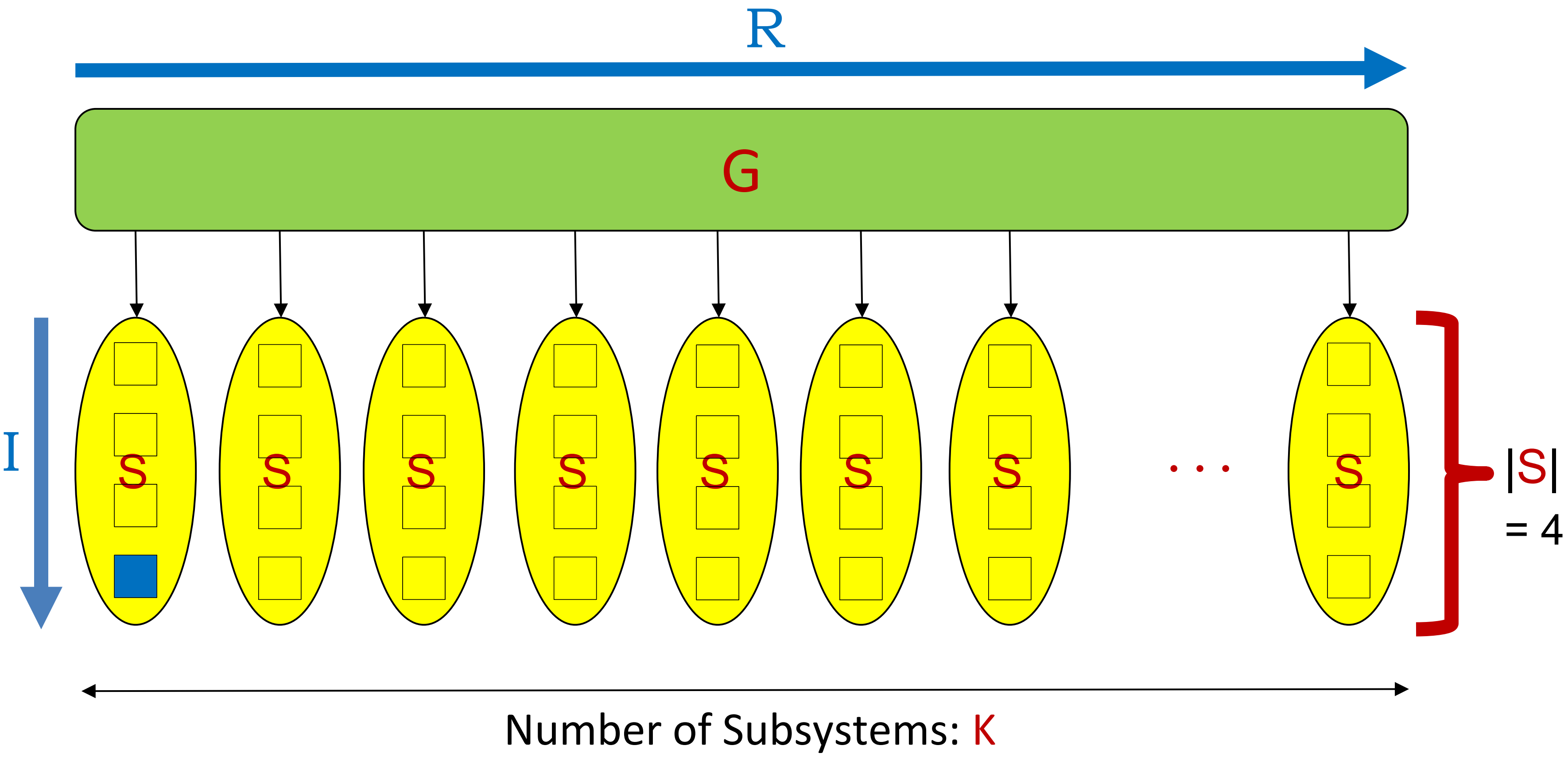
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$



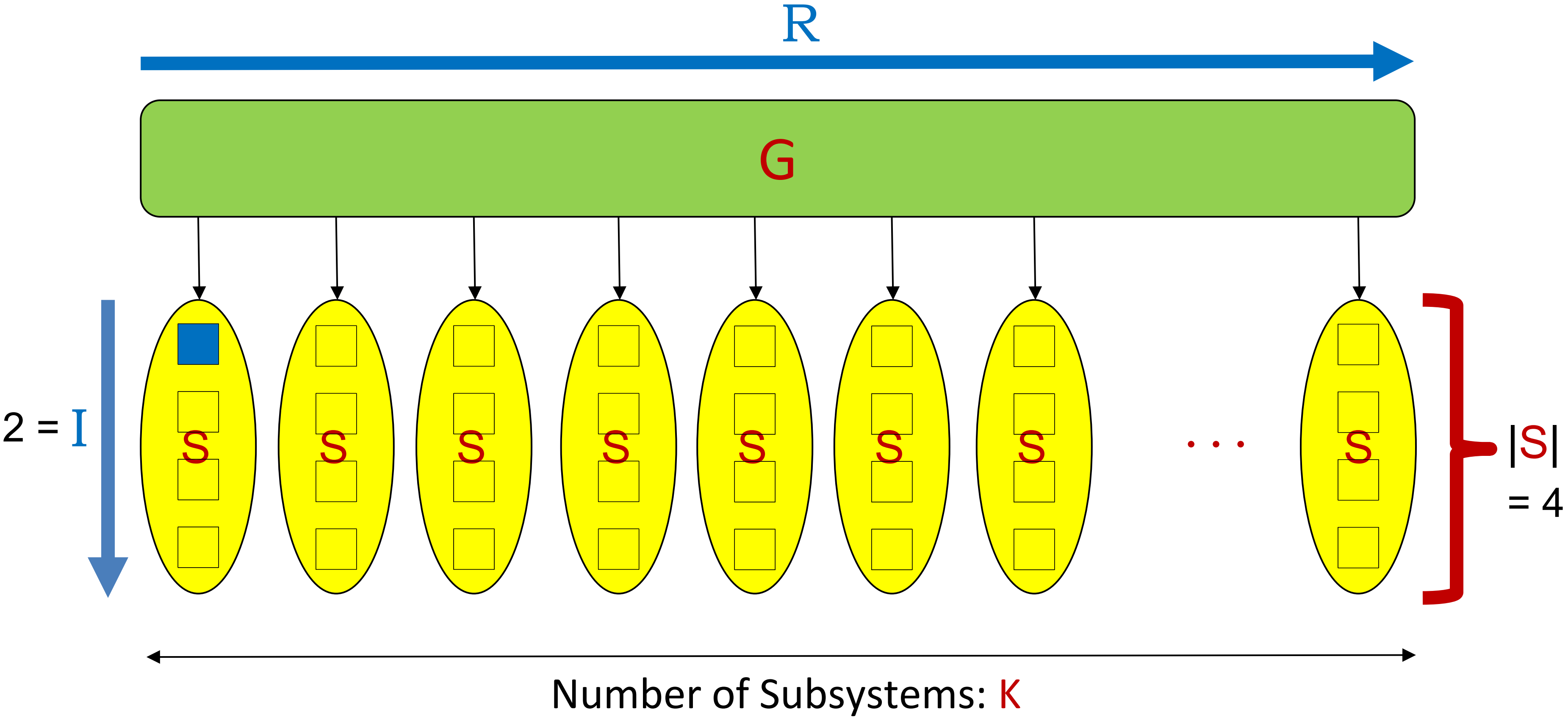
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

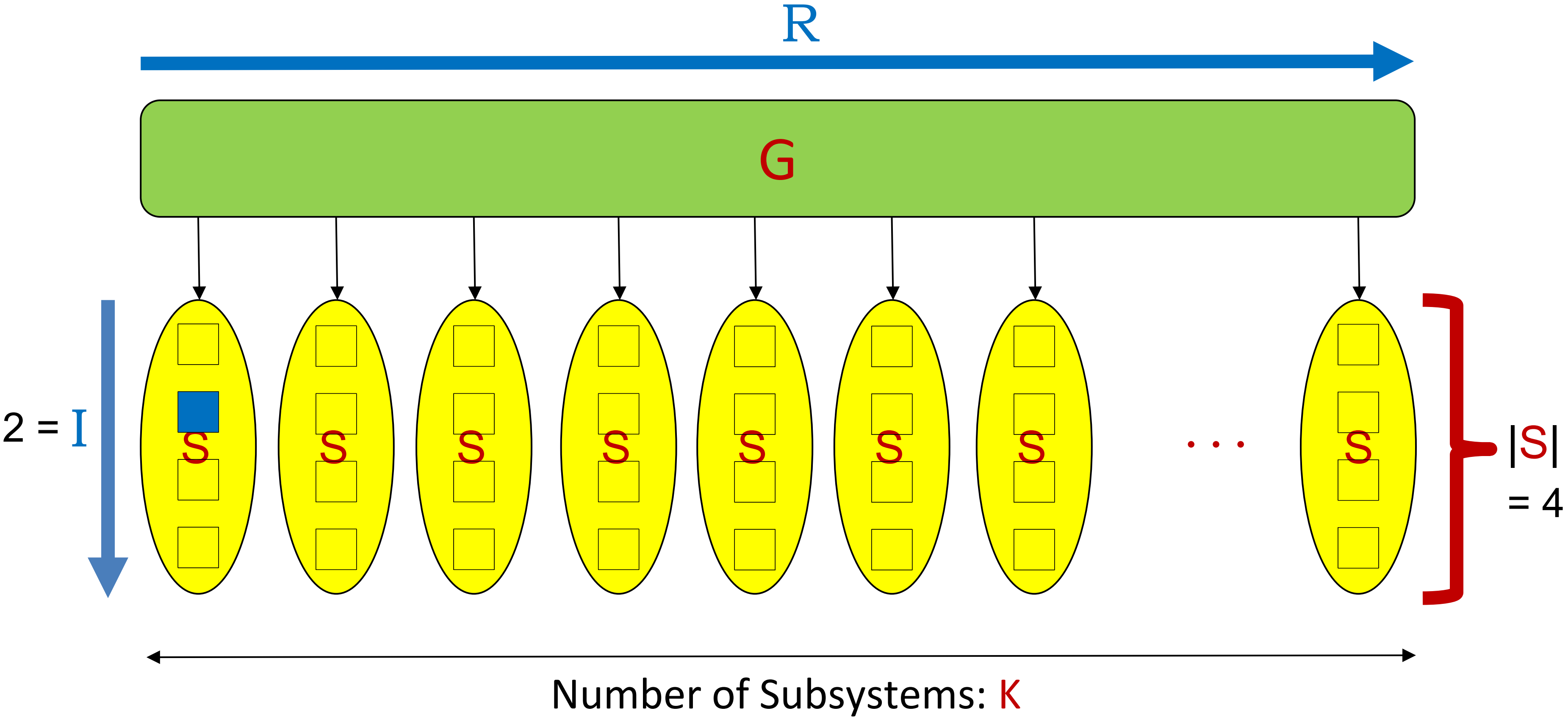


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

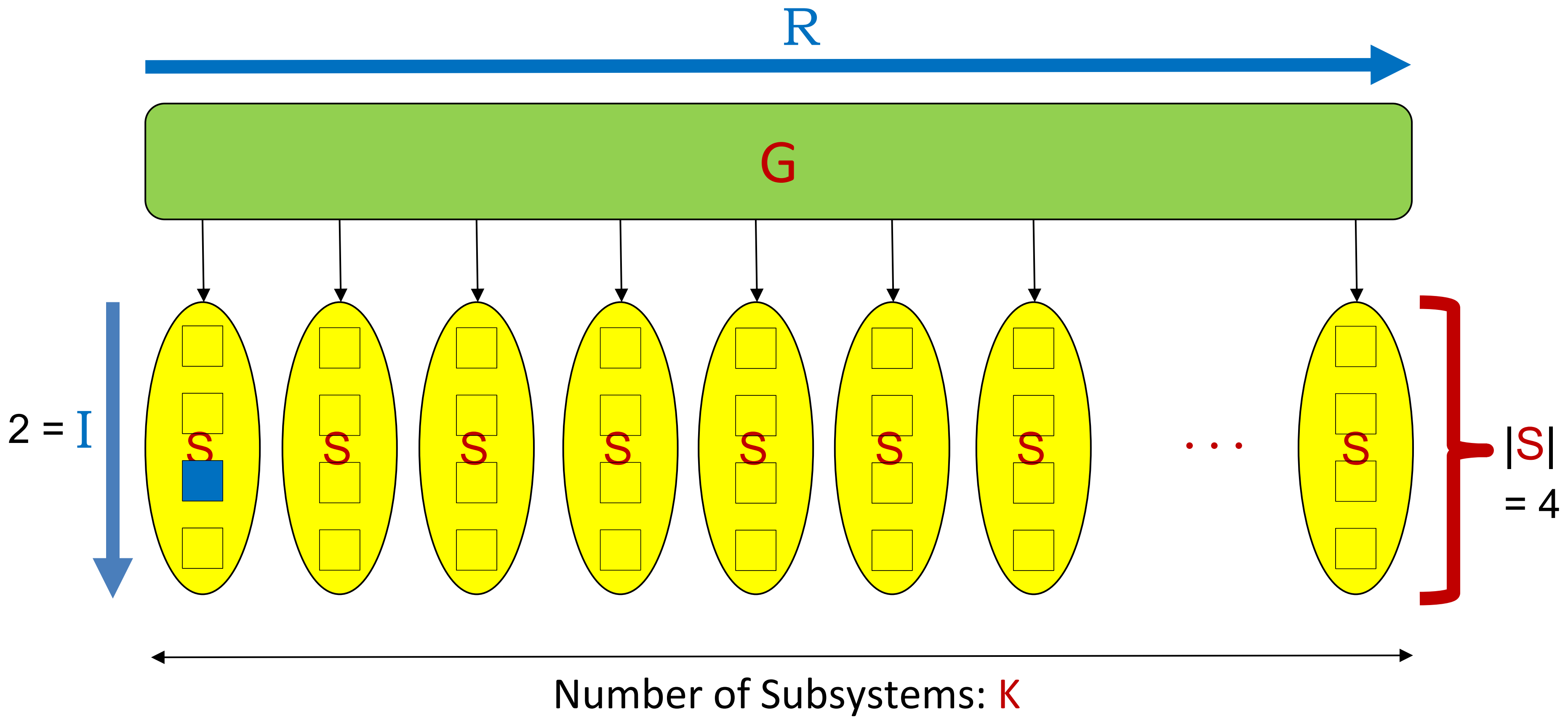


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

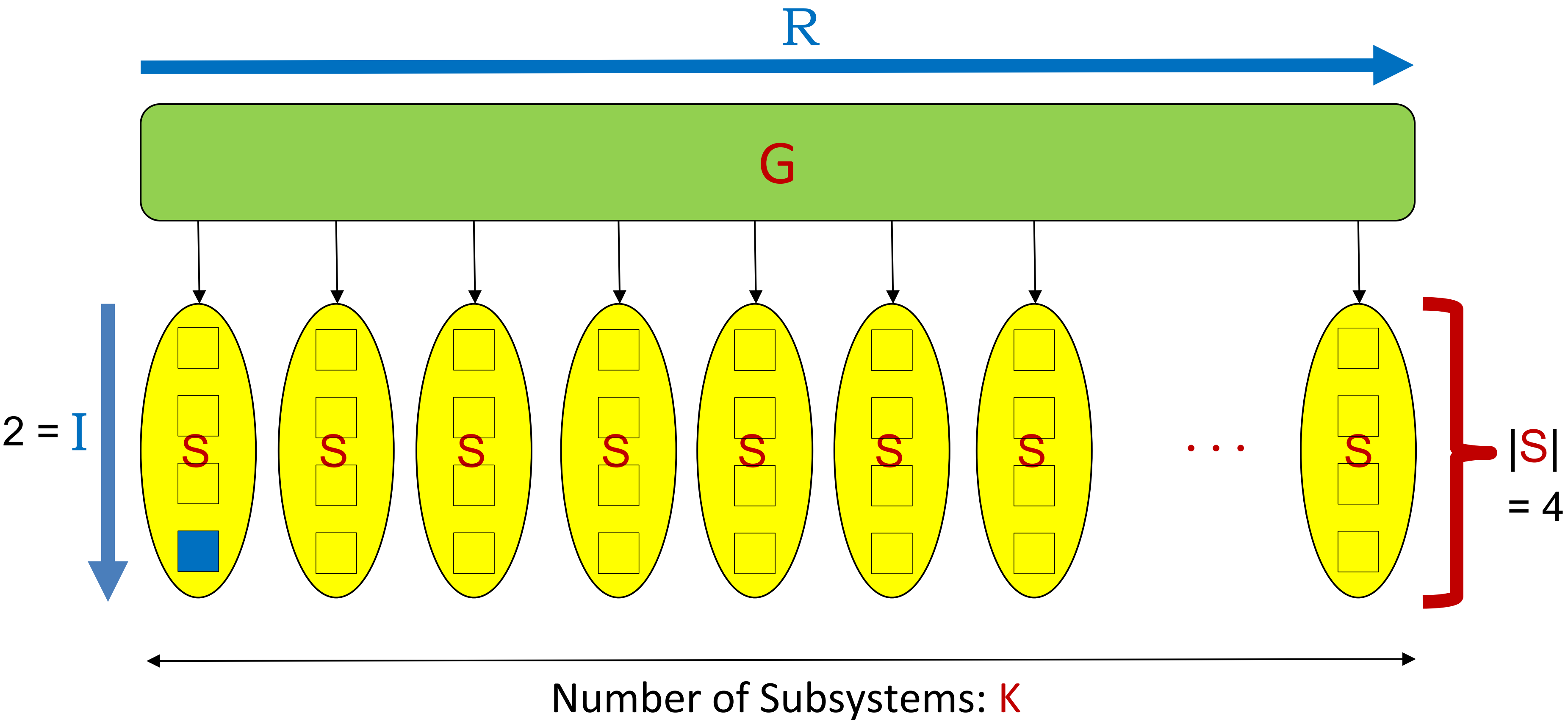
Physical System Size |G| = K · |S|



Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

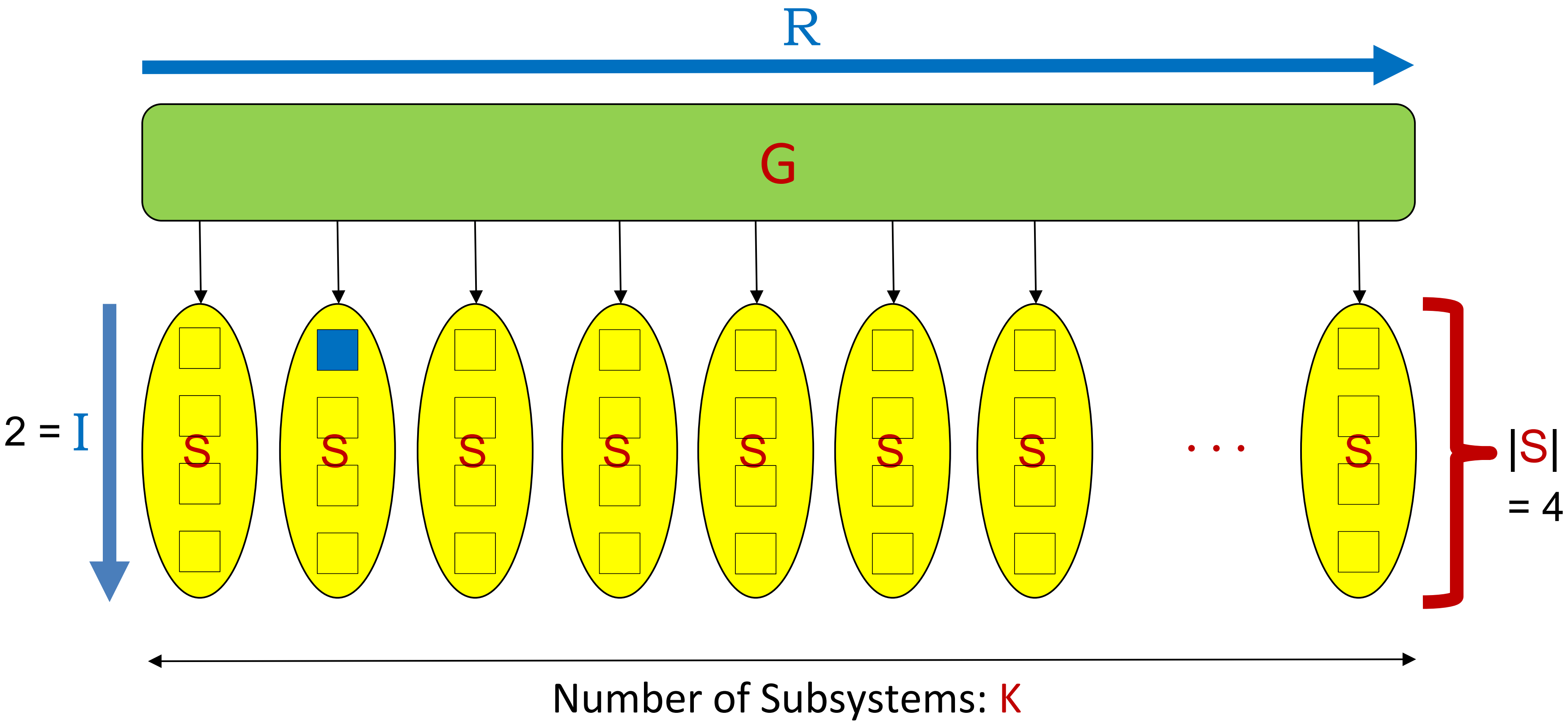


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

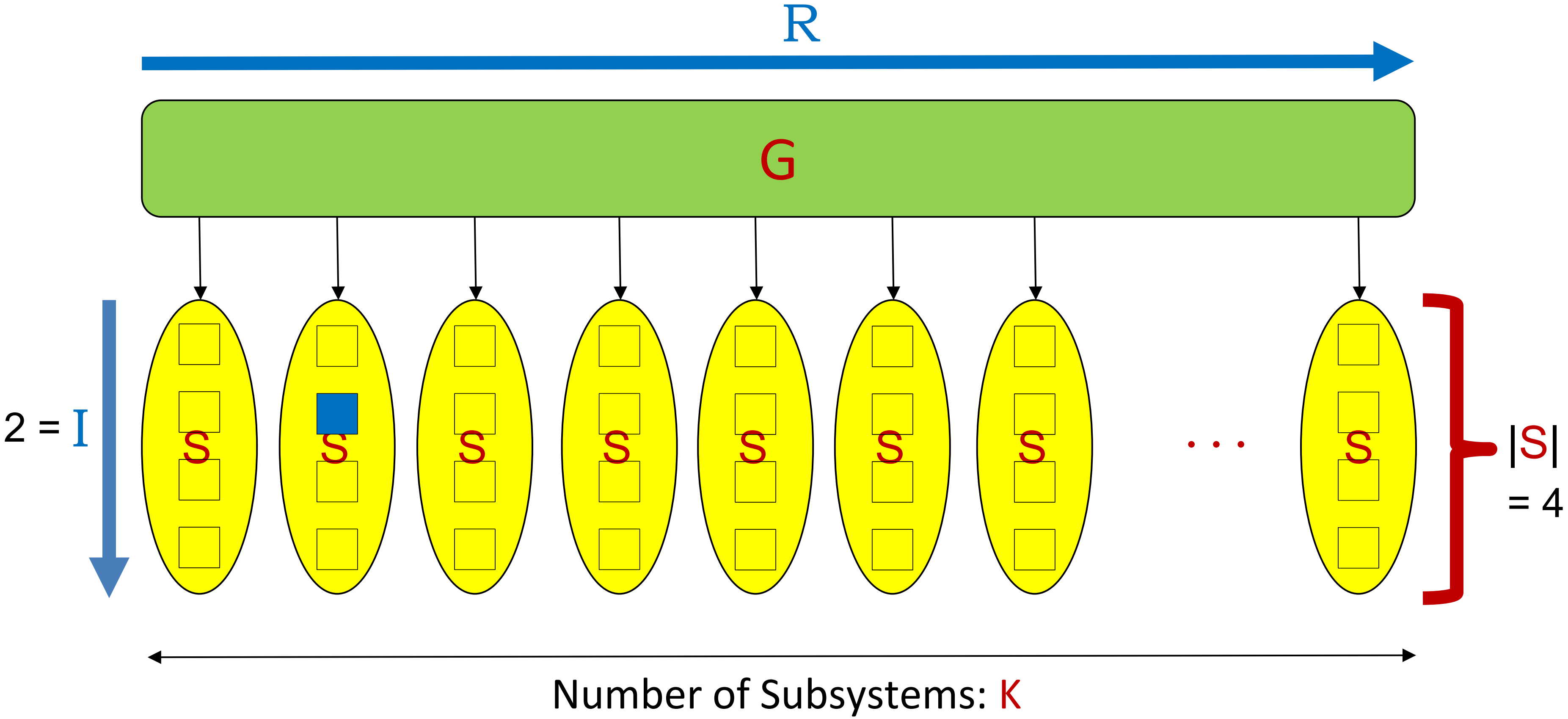


Number of Subsystems: K

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

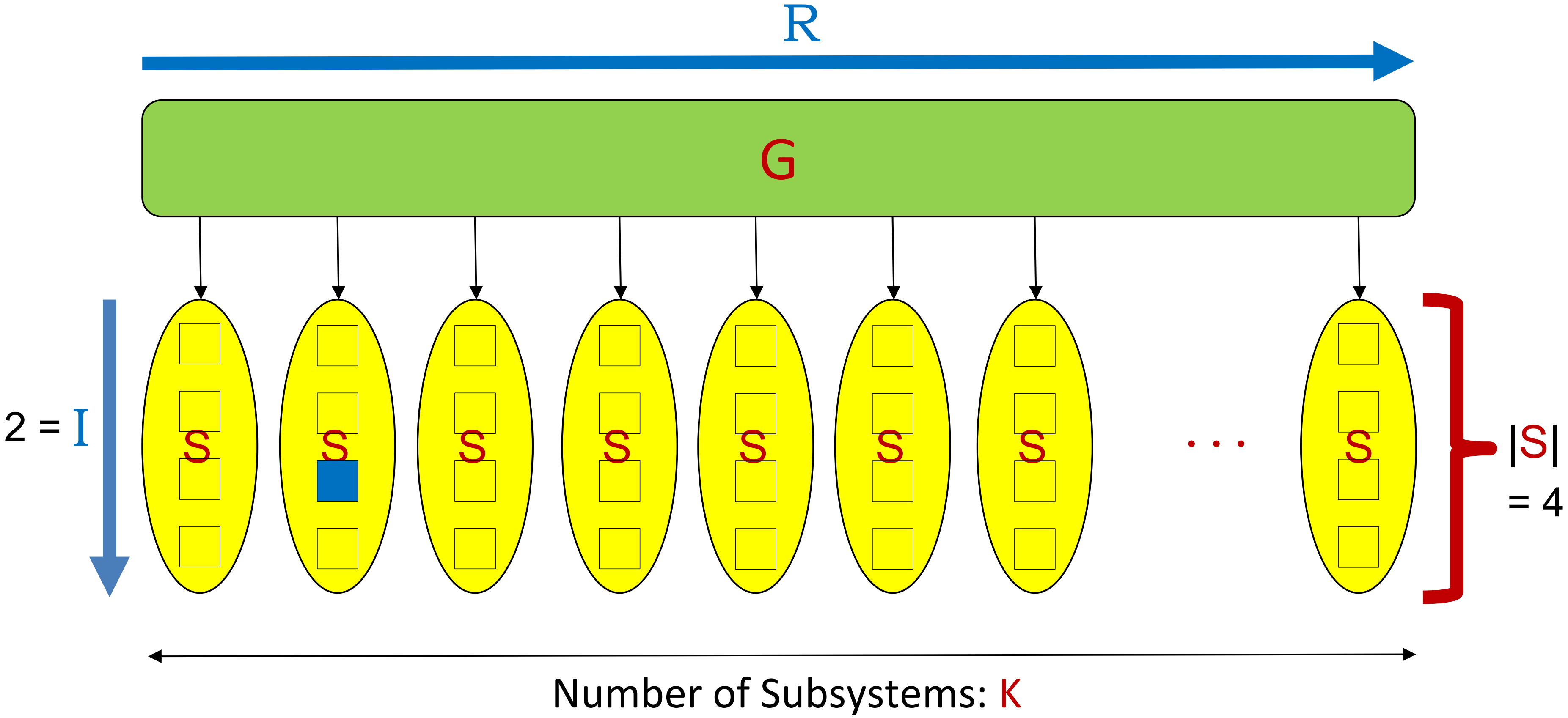


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

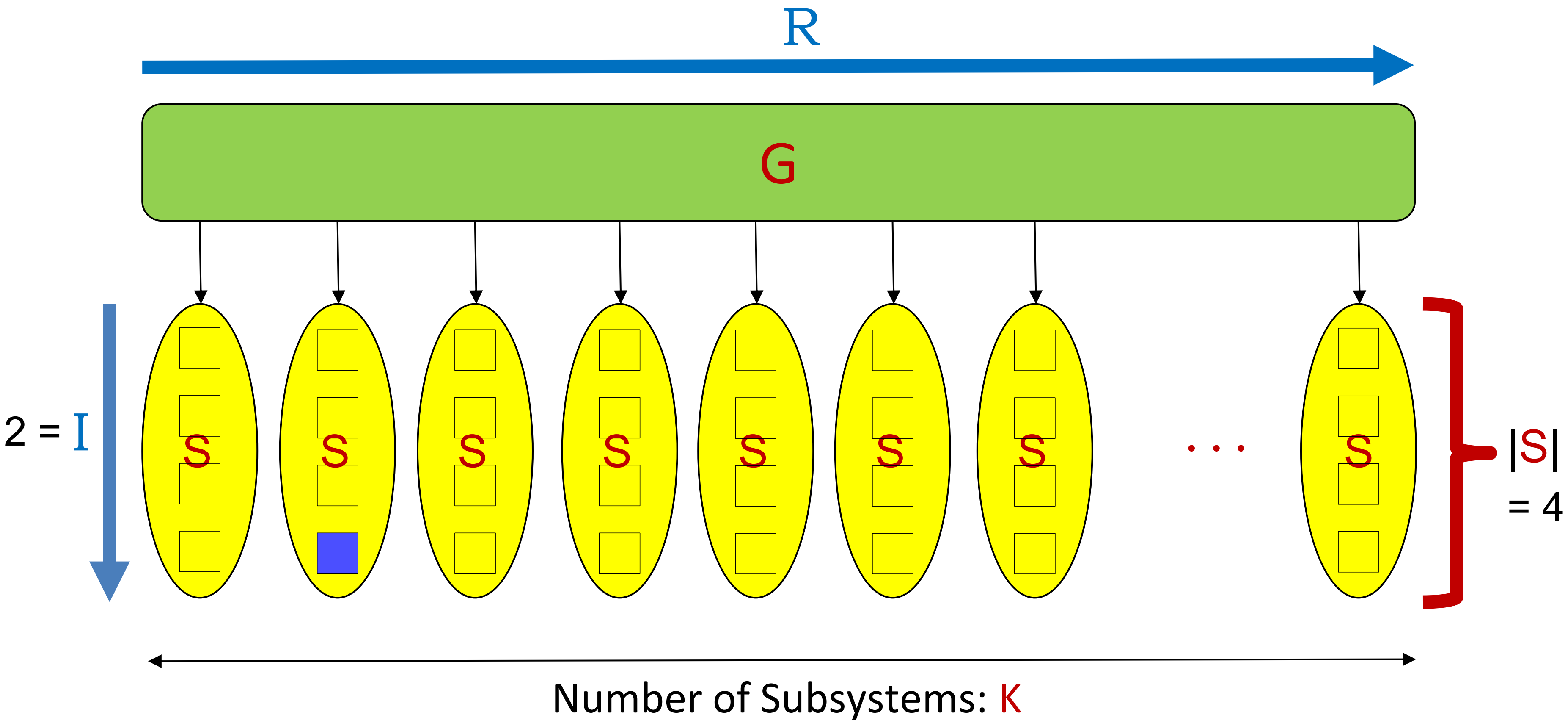
Physical System Size $|G| = K \cdot |S|$



Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|



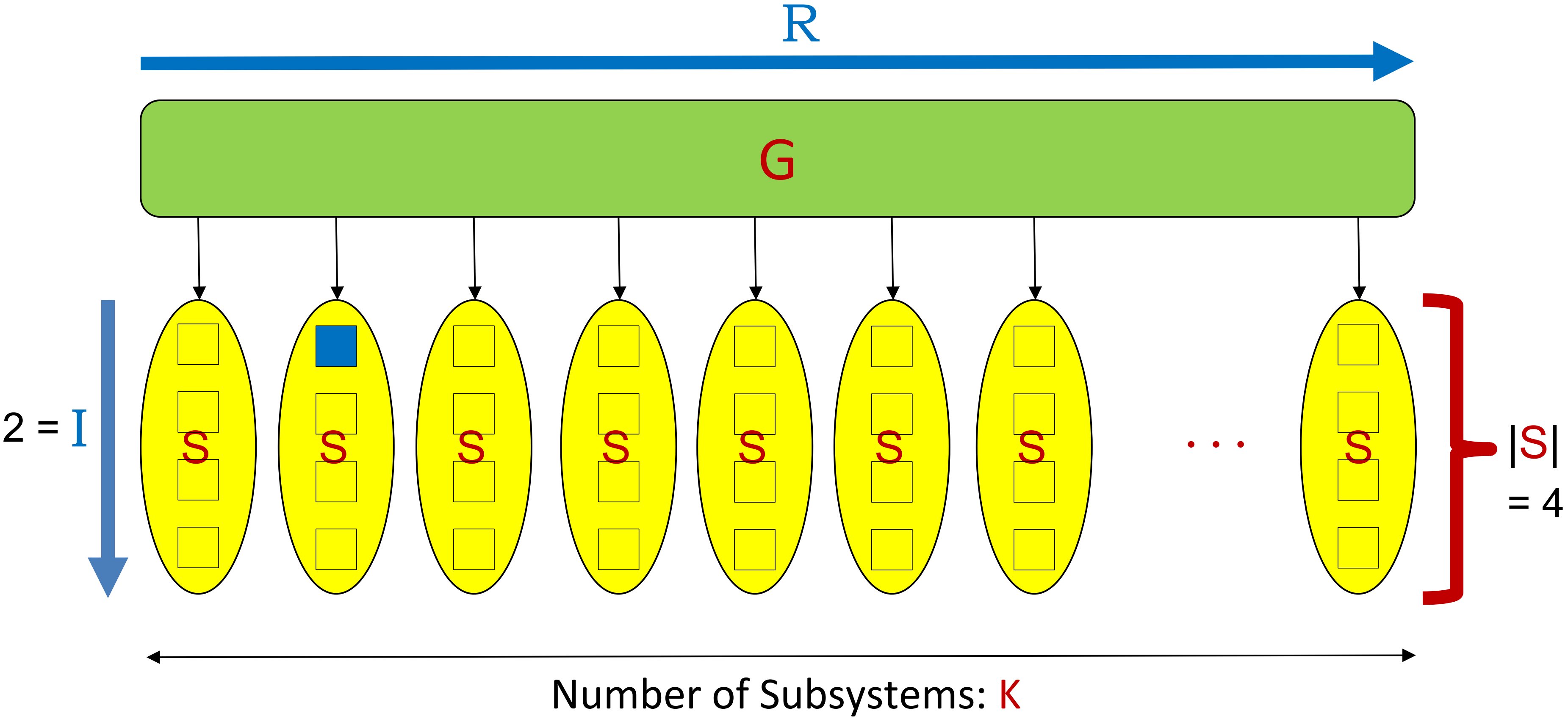
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

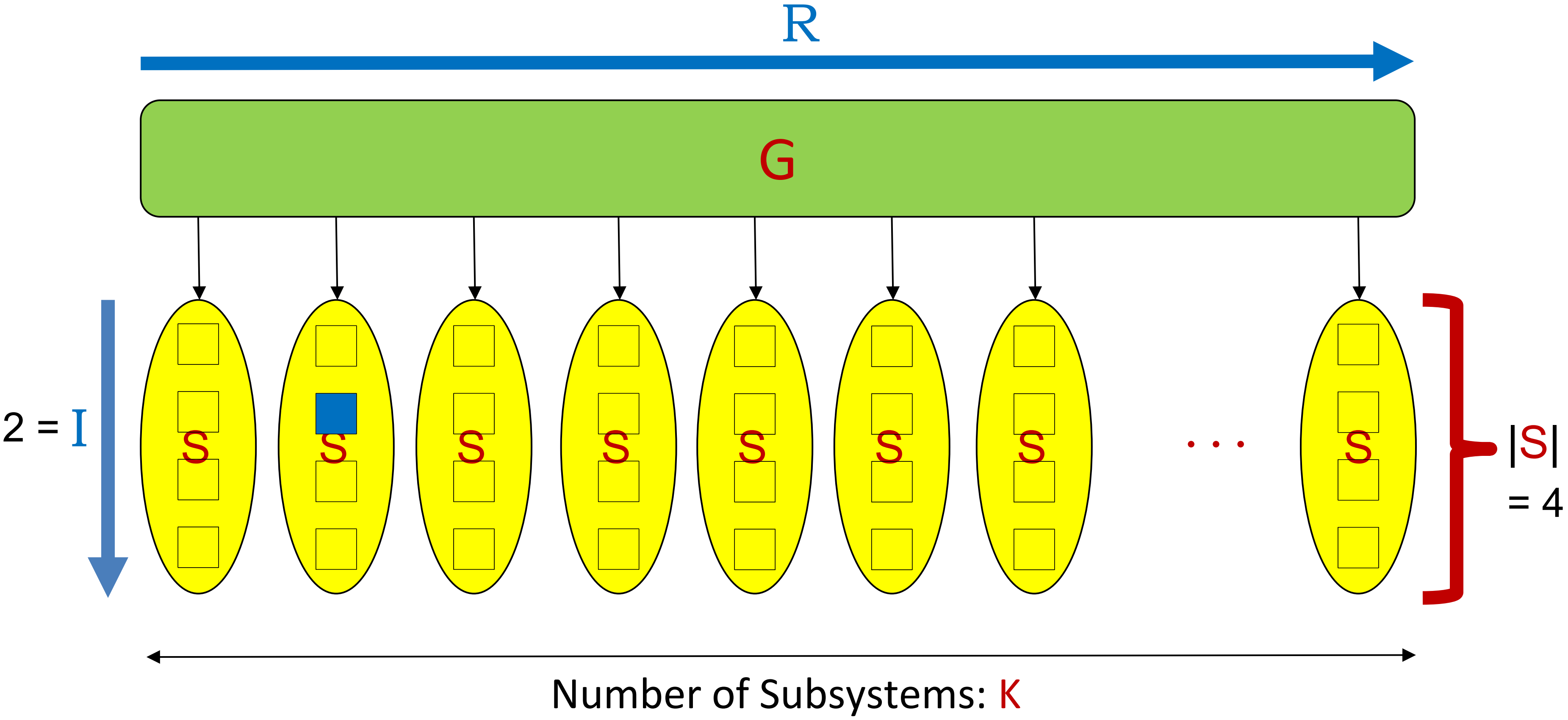


Number of Subsystems: K

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$

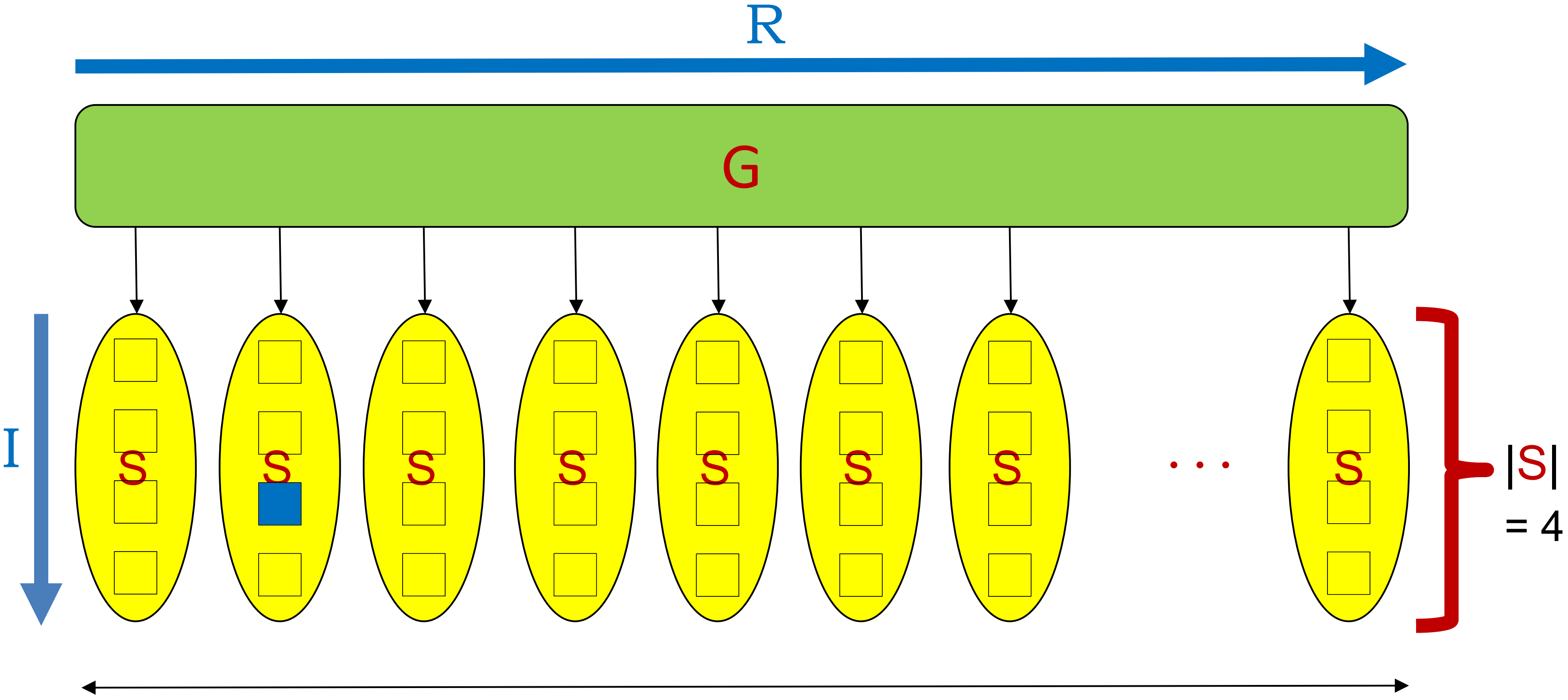


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$



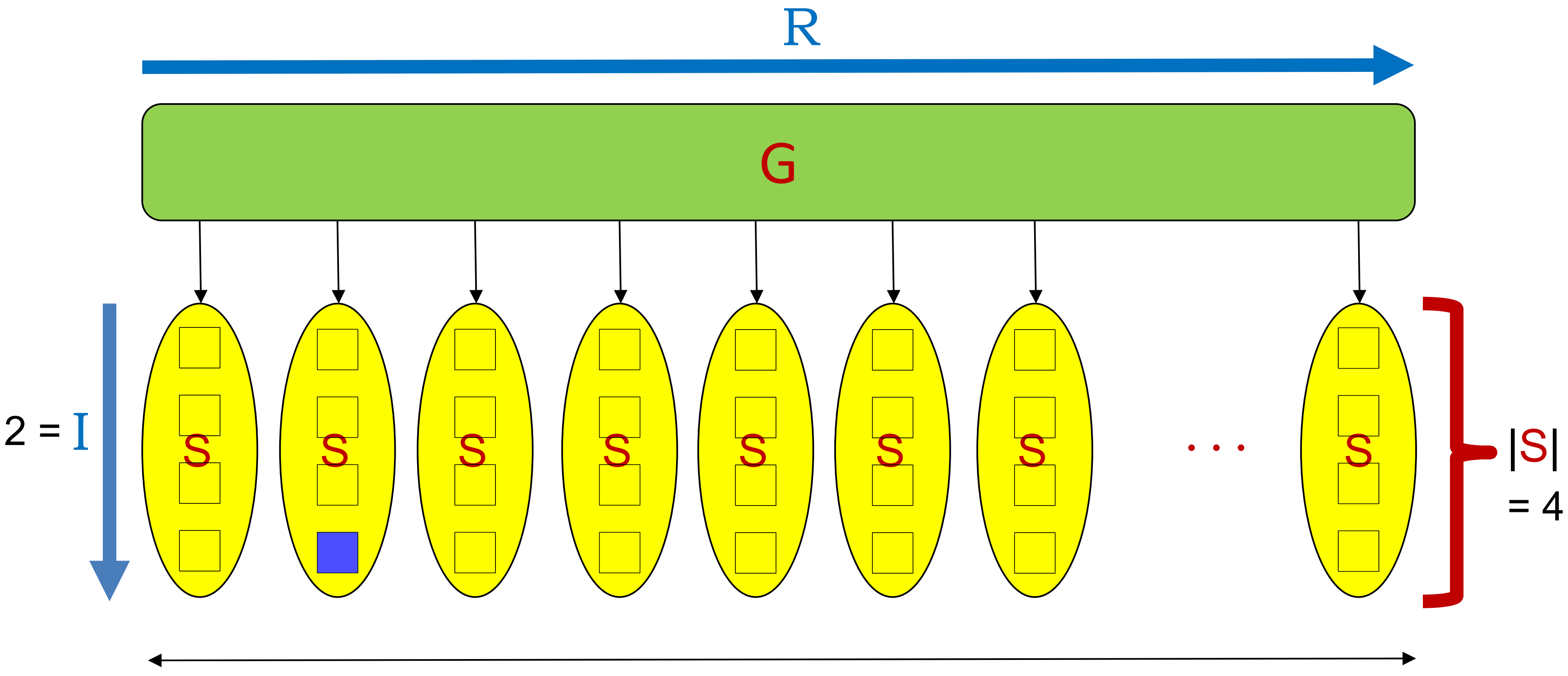
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|



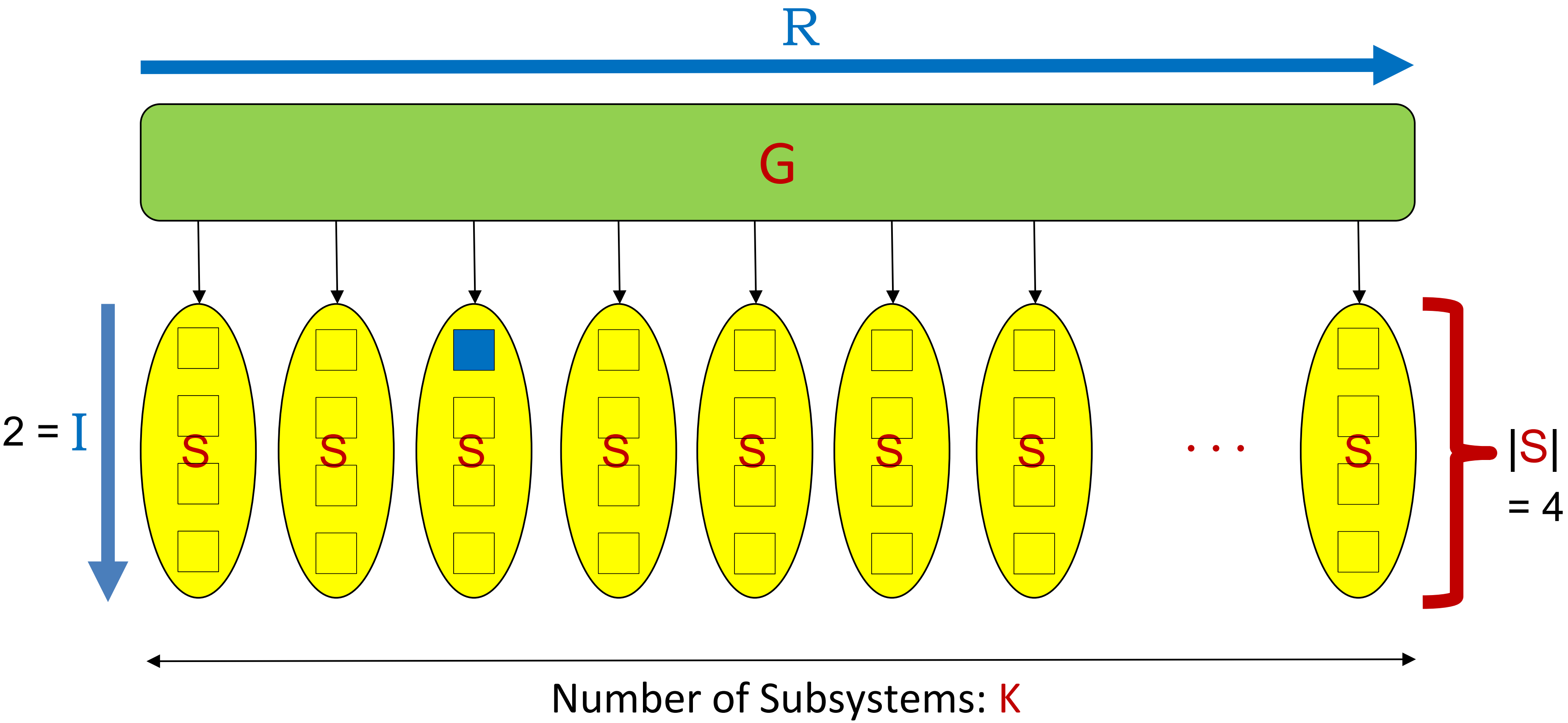
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

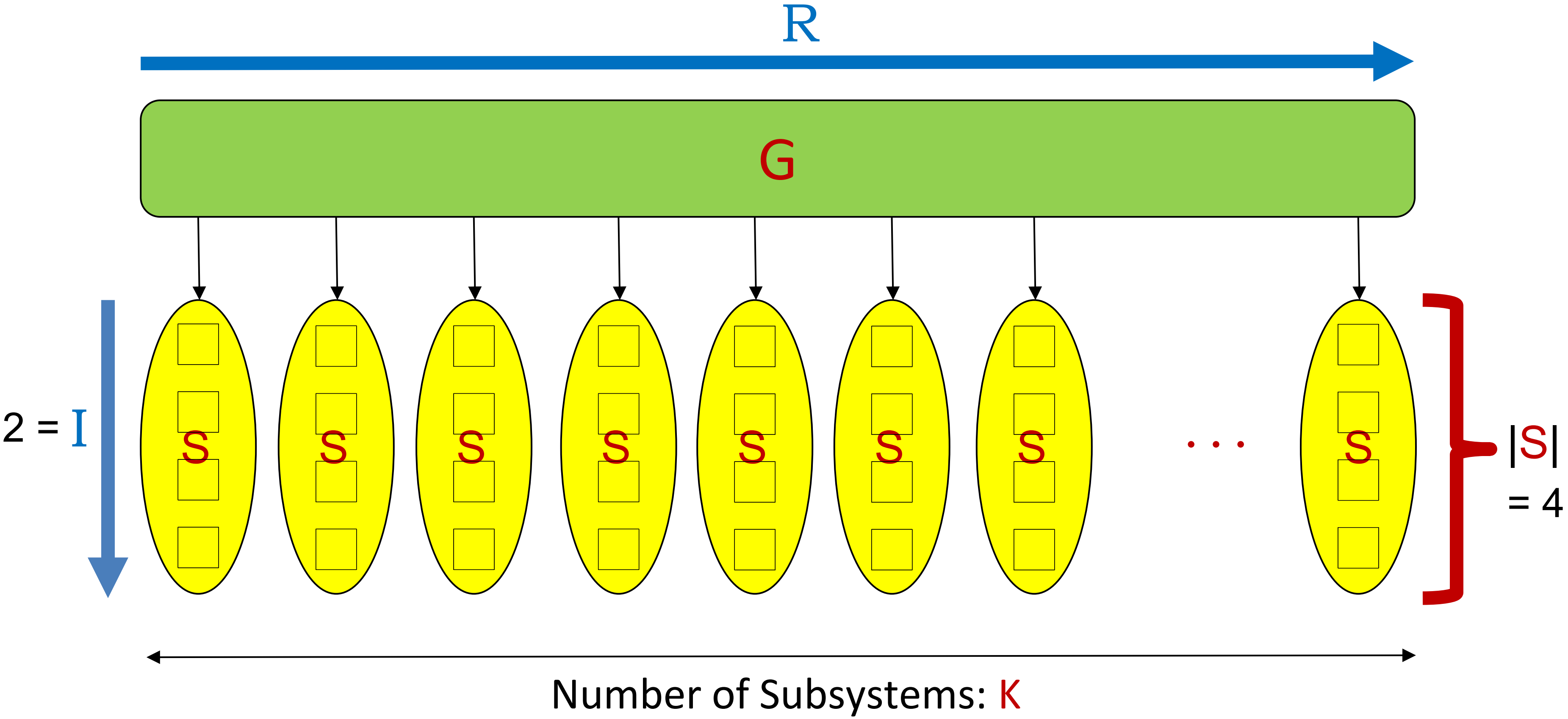


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|

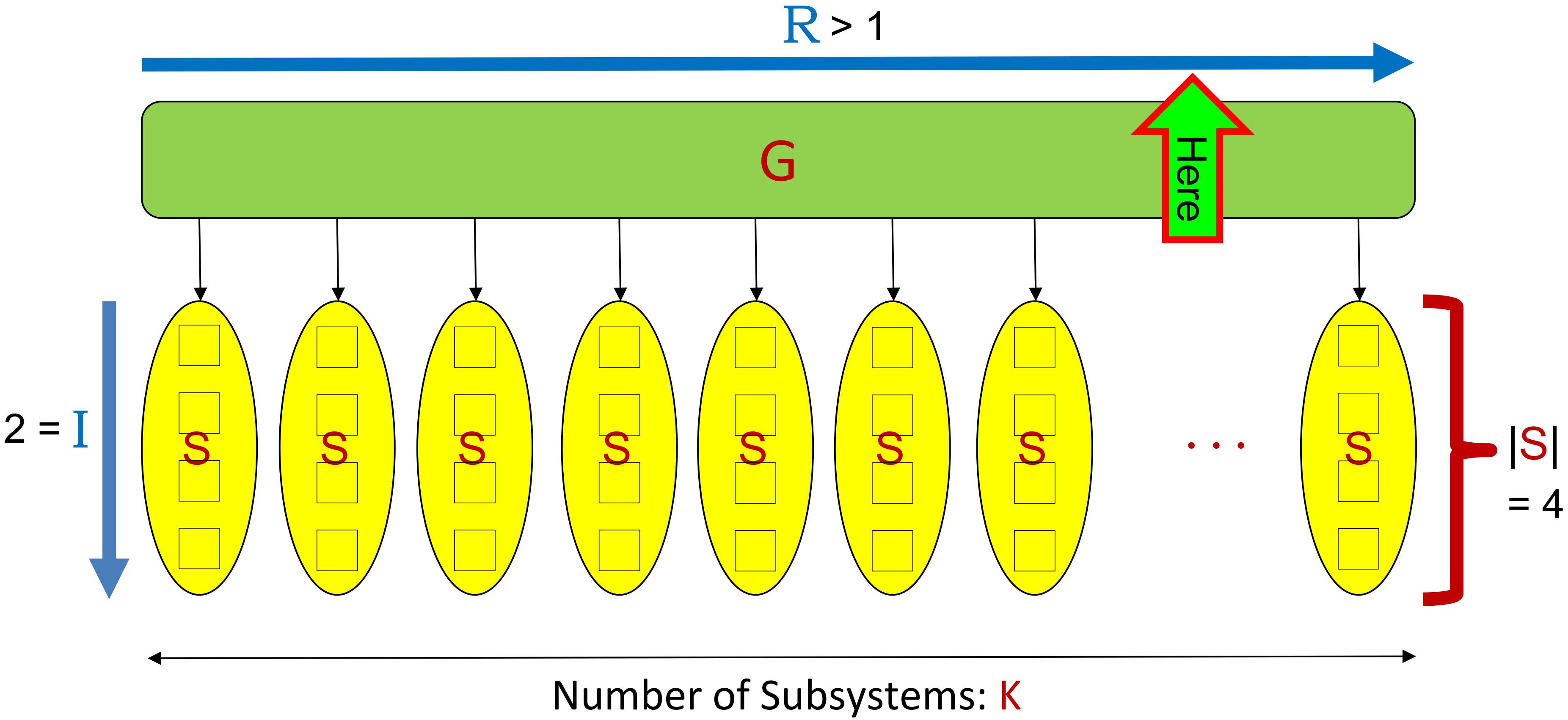


3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: **N** = |G| · I · R

Physical System Size |G| = K · |S|



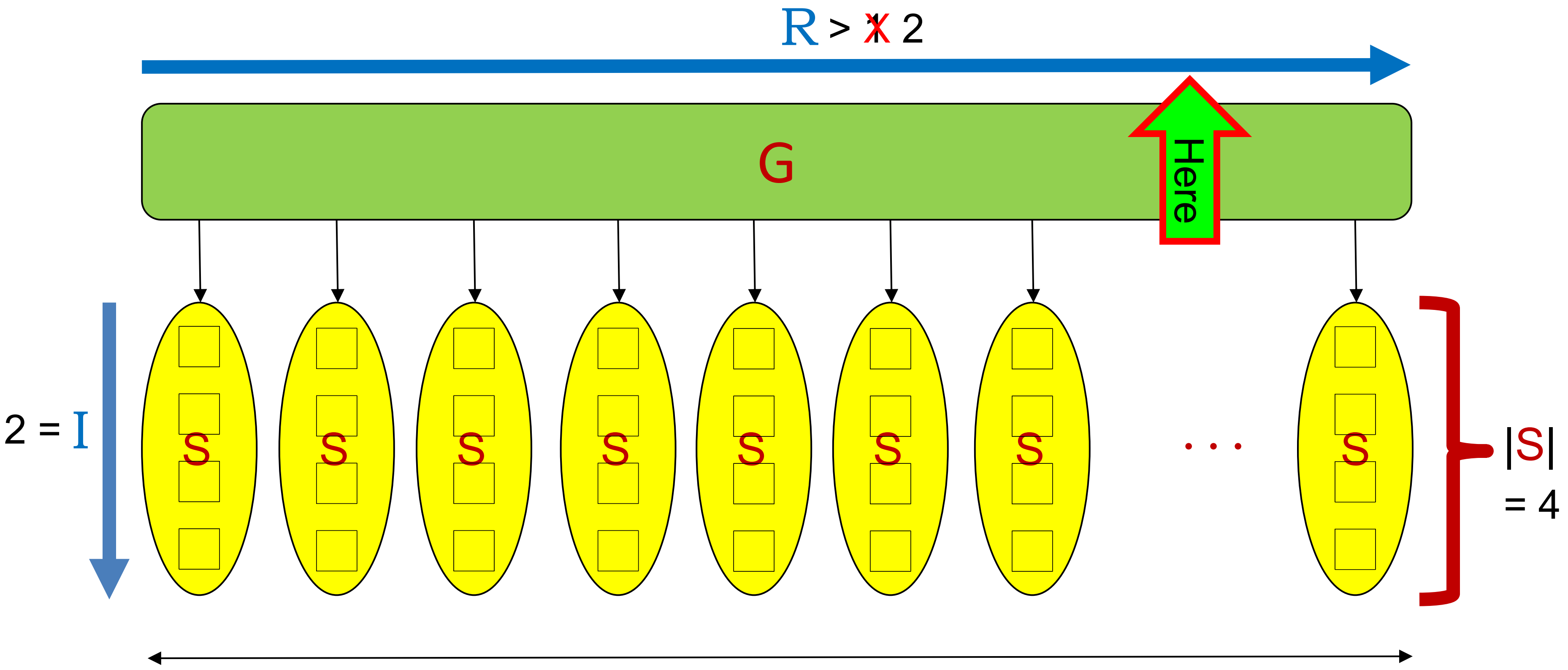
Number of Subsystems: K

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Problem Size: $\mathbf{N} = |G| \cdot I \cdot R$

Physical System Size $|G| = K \cdot |S|$



Number of Subsystems: K

3. Analyzing the Benchmark Data

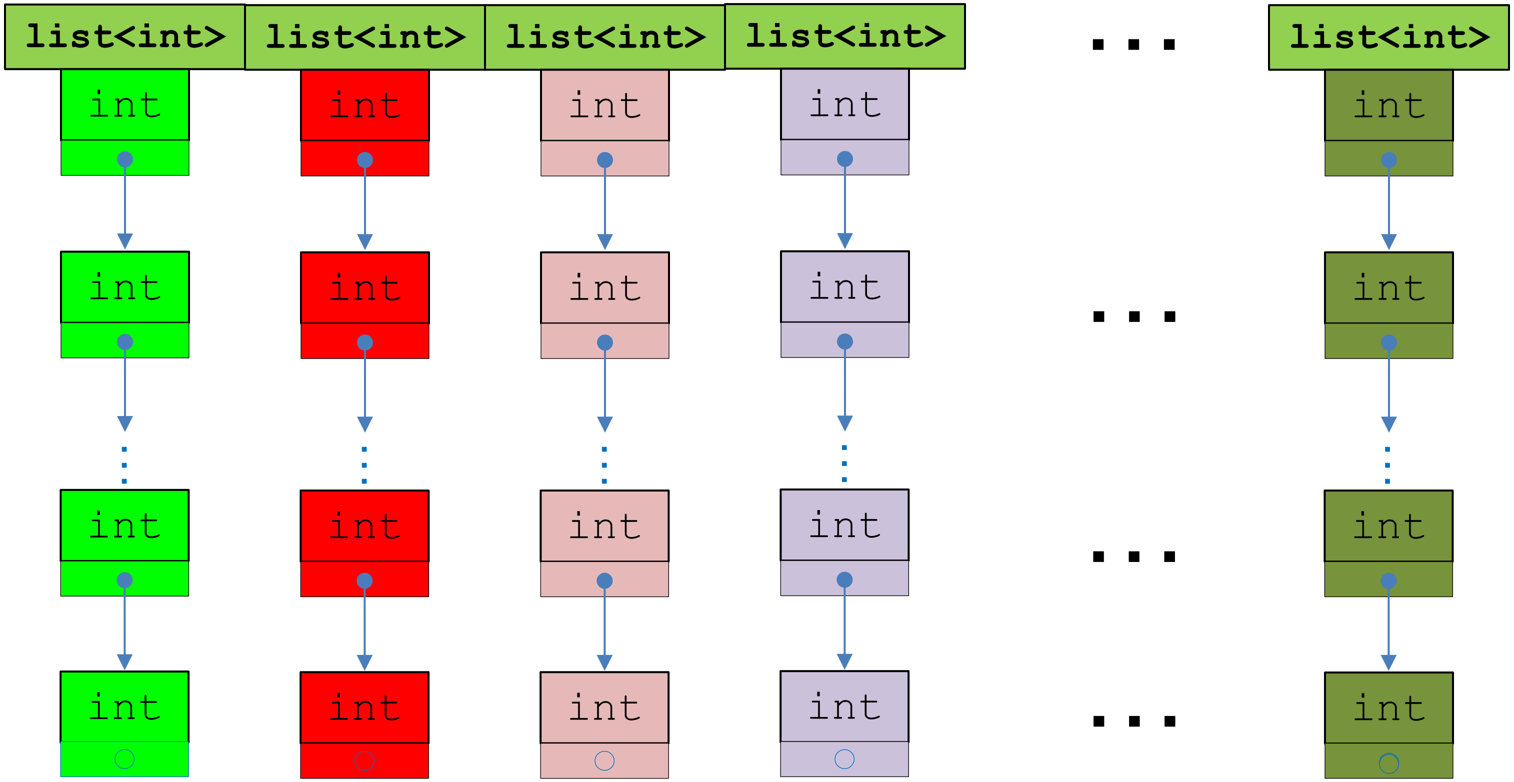
Benchmark II: LOCALITY

Shuffle Plan:

- Build up a data structure $\mathbf{G}$: `vector<list<int>>`
- Visit each of the subsystems $\mathbf{S}_\mathbf{j}$ of $\mathbf{G}$ in turn.
  - Pop the last element of $\mathbf{S}_\mathbf{j}$ after saving its `int` value.
  - Push that value onto the front of a random $\mathbf{S}_\mathbf{i}$.
  - Repeat until the number of moves reaches $\mathbf{sf} \cdot |\mathbf{G}|$.
  - The non-negative integer $\mathbf{sf}$ represents the **s**huffle **f**actor.
- The result of this experiment is a shuffled system $\mathbf{G}$.
  - There is also the added runtime to do the shuffle.
- Additional consideration: The shuffle is heuristic …
  - This to keep the time to shuffle as small as possible.

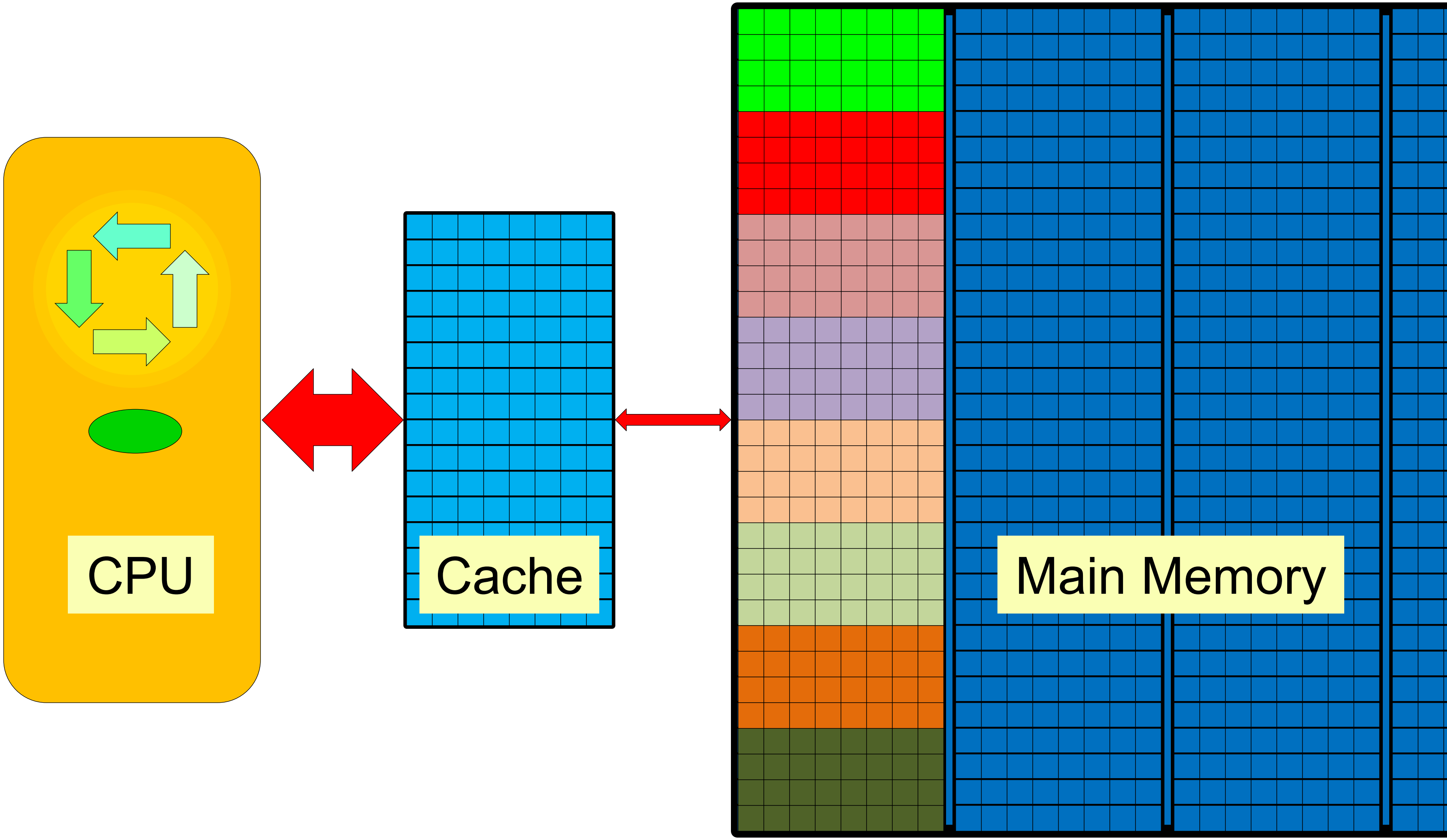
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Overall (Multiprogram) Plan:

A. Create and Access **After** Shuffle:

B. Create and Access **Before** Shuffle.

C. Create and Shuffle **Only**

The result of each program is its (wall) **RUNTIME.**

☐ **Shuffled**-Memory Access Time: $A - C$

☐ **Unshuffled**-Memory Access Time: $B - C$

☐ **Degradation Ratio** due to memory diffusion (DR):

$$\boldsymbol{Degredation\ Ratio\ (DR) = \frac{A - C}{B - C}}$$

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Contrasting access times across (sub)system size

Overall System Size = 10^7

$\log_{10} |\mathbf{G}| = 7$

These
are all
exponents
of **10**.

Subsystem Size	Number of Subsystems
0	7
1	6
2	5
3	4
4	3
5	2
6	1
7	0

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

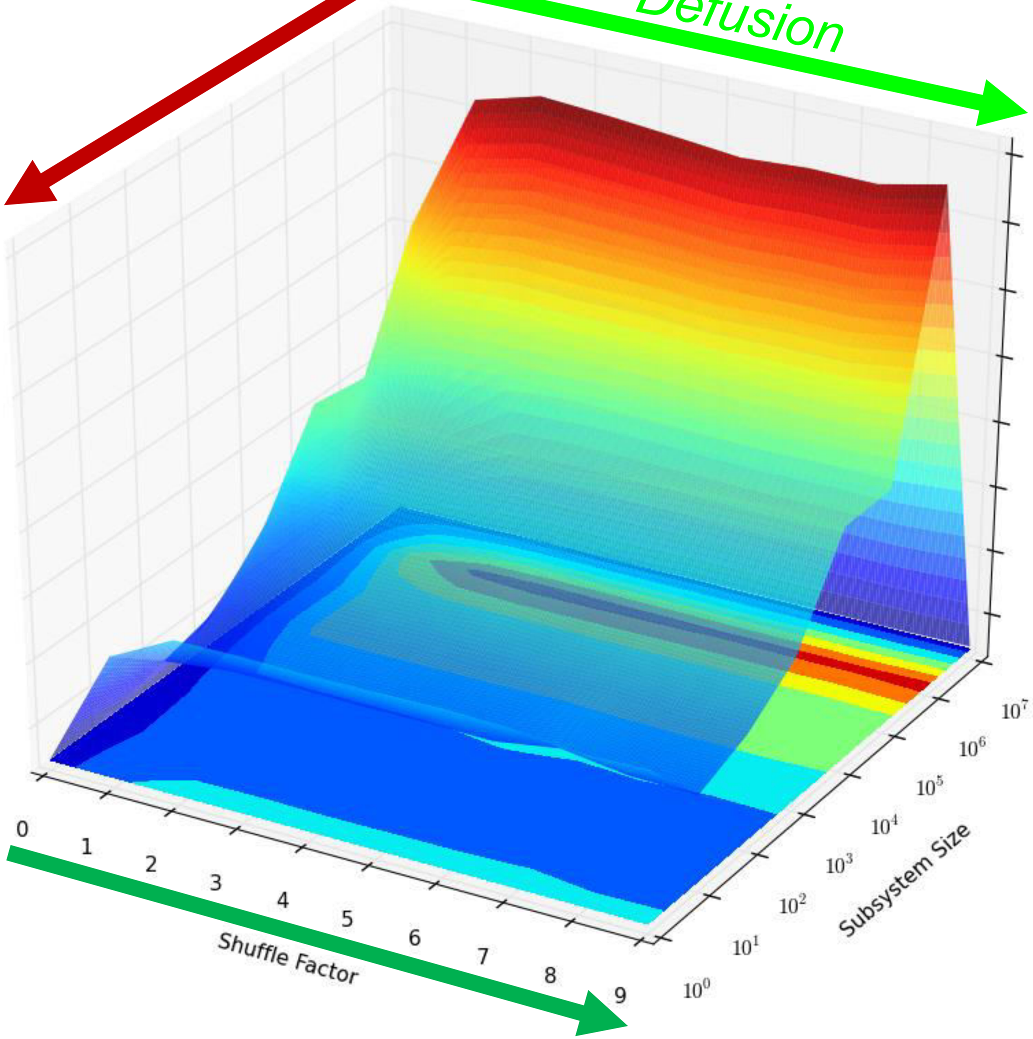
Defusion

Number of Subsystems (K)

Degradation Ratio (DR)

(due to diffusion)

Number of Shuffles (sf)



3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

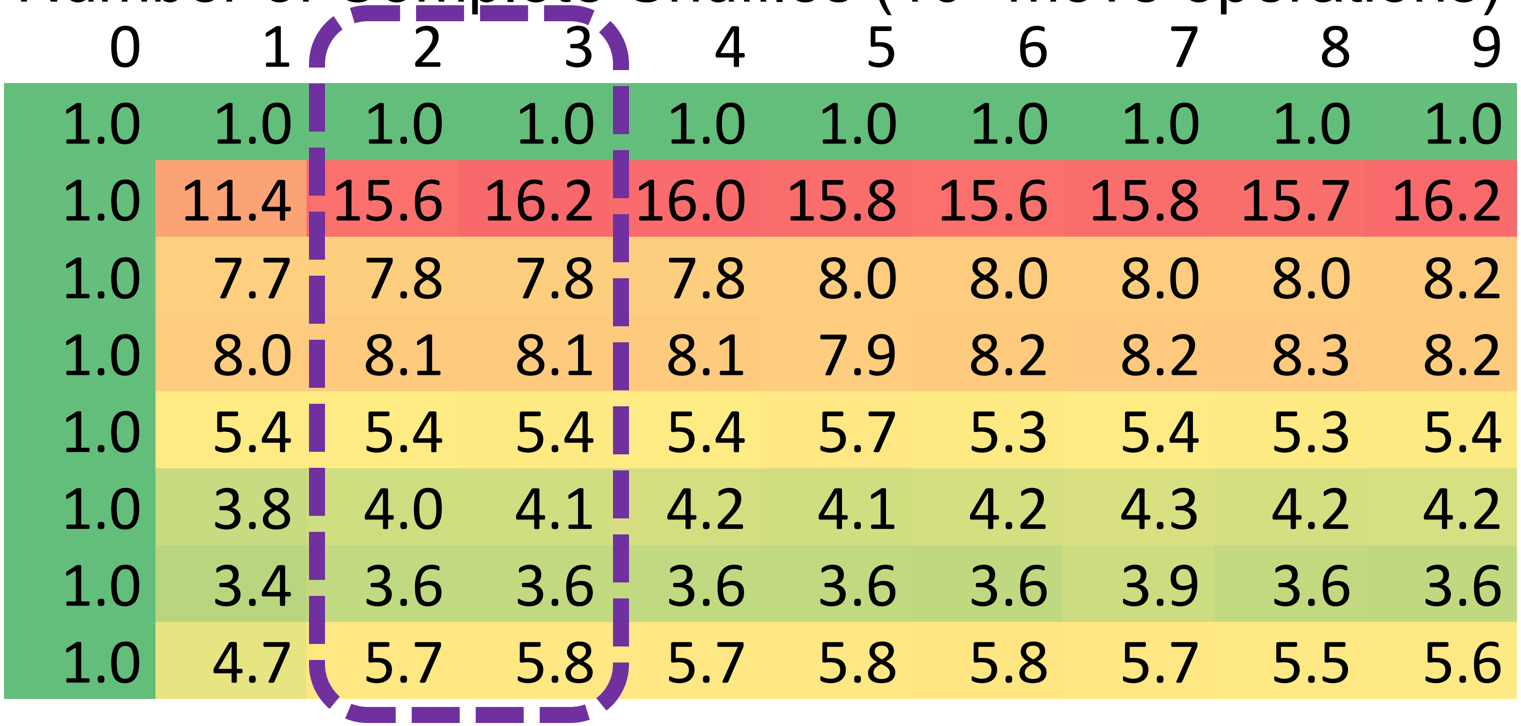
Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Defusion



Number of Complete Shuffles (10^7 move operations)

Subsystem Size

	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

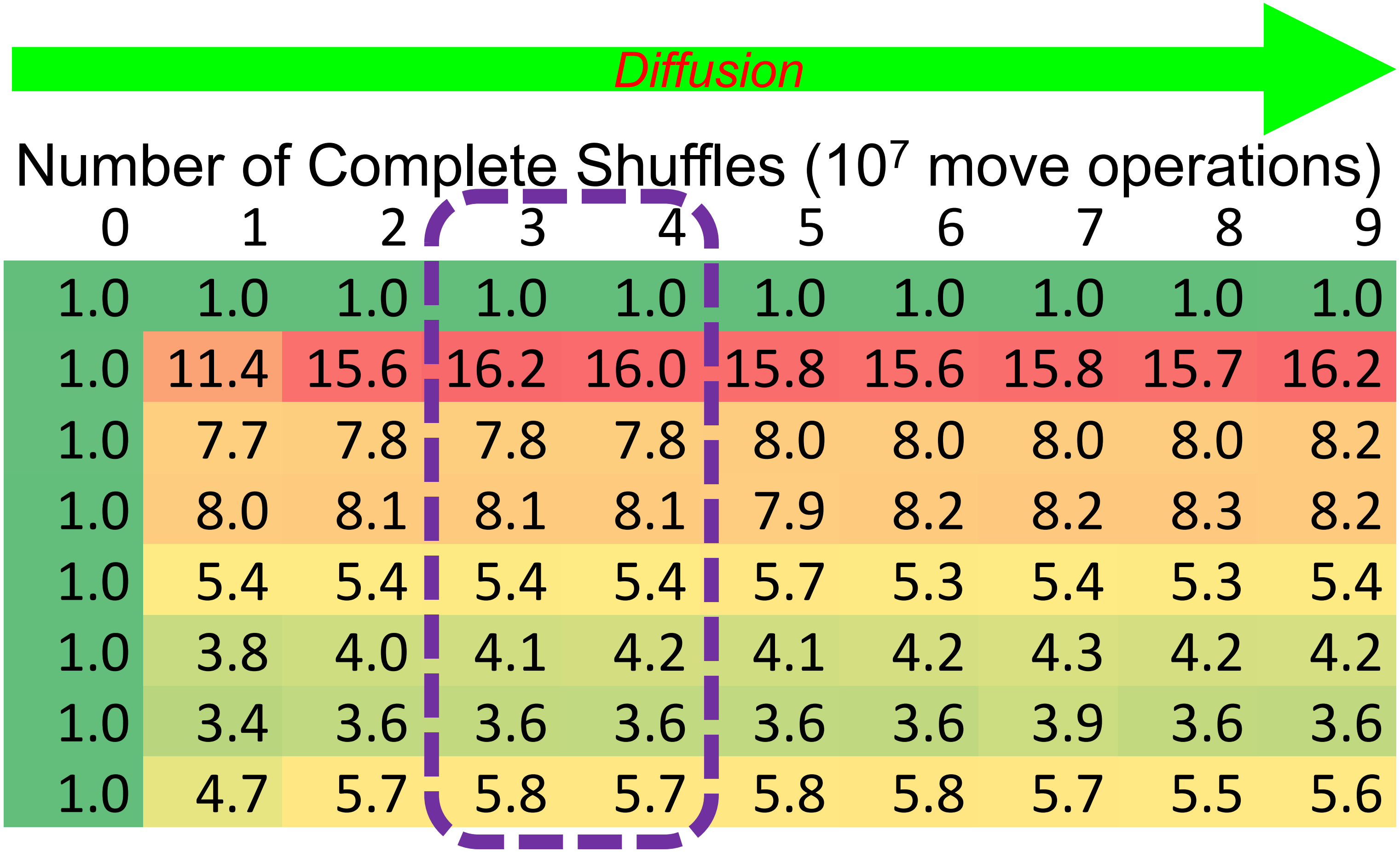
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$



Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Diffusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

We'll use *shuffle factor* **sf** = 5 from now on.

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

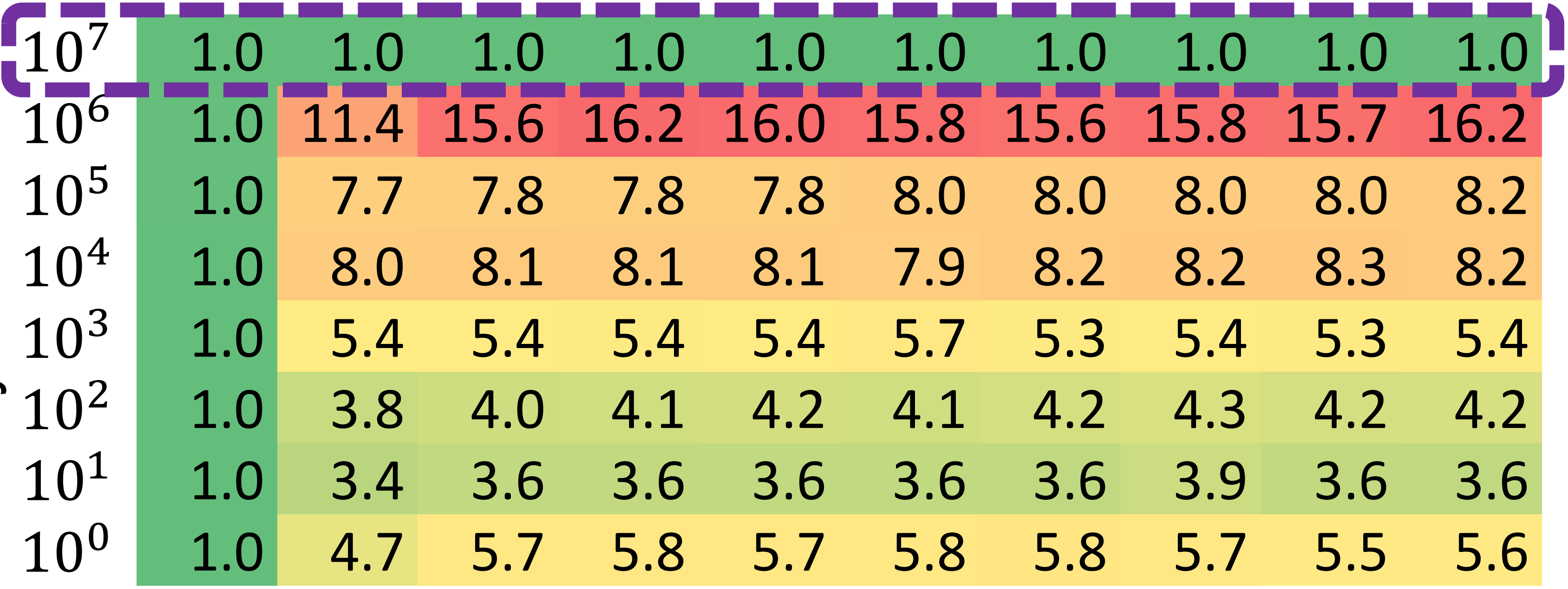
I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Diffusion

Number of Complete Shuffles (10^7 move operations)



Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

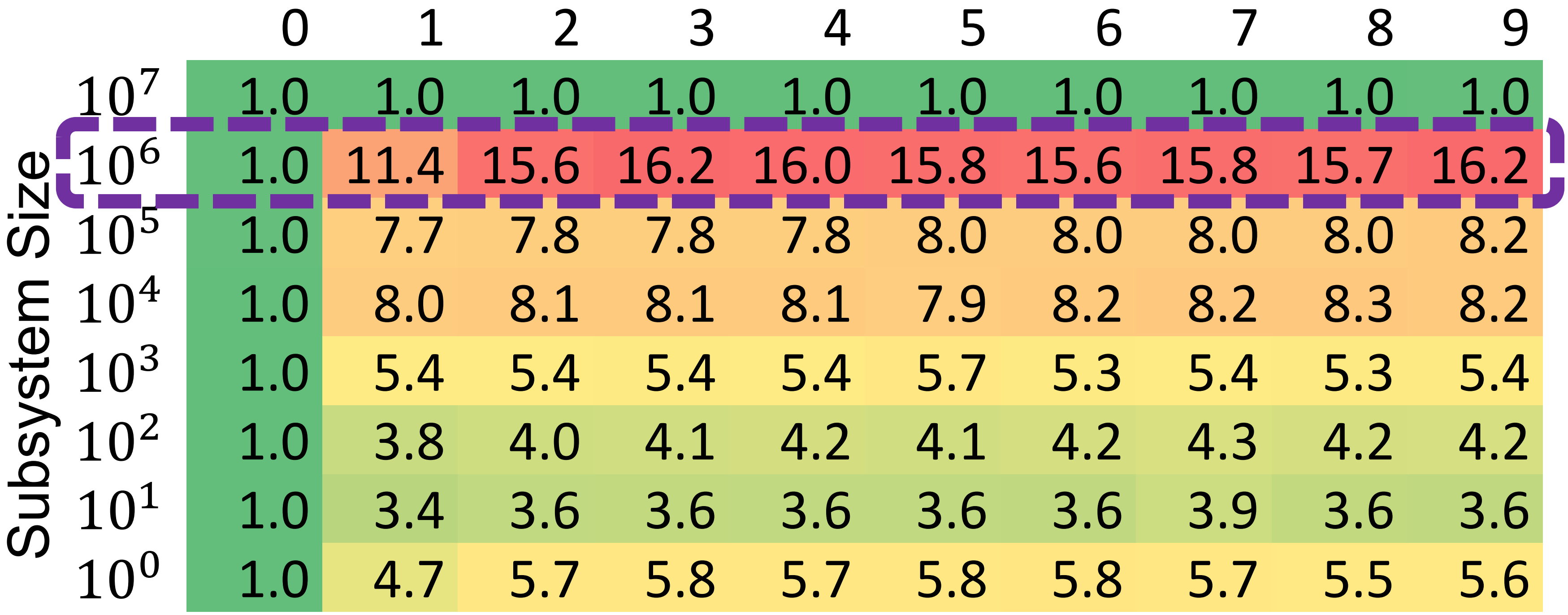
I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Diffusion

Number of Complete Shuffles (10^7 move operations)



Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Diffusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

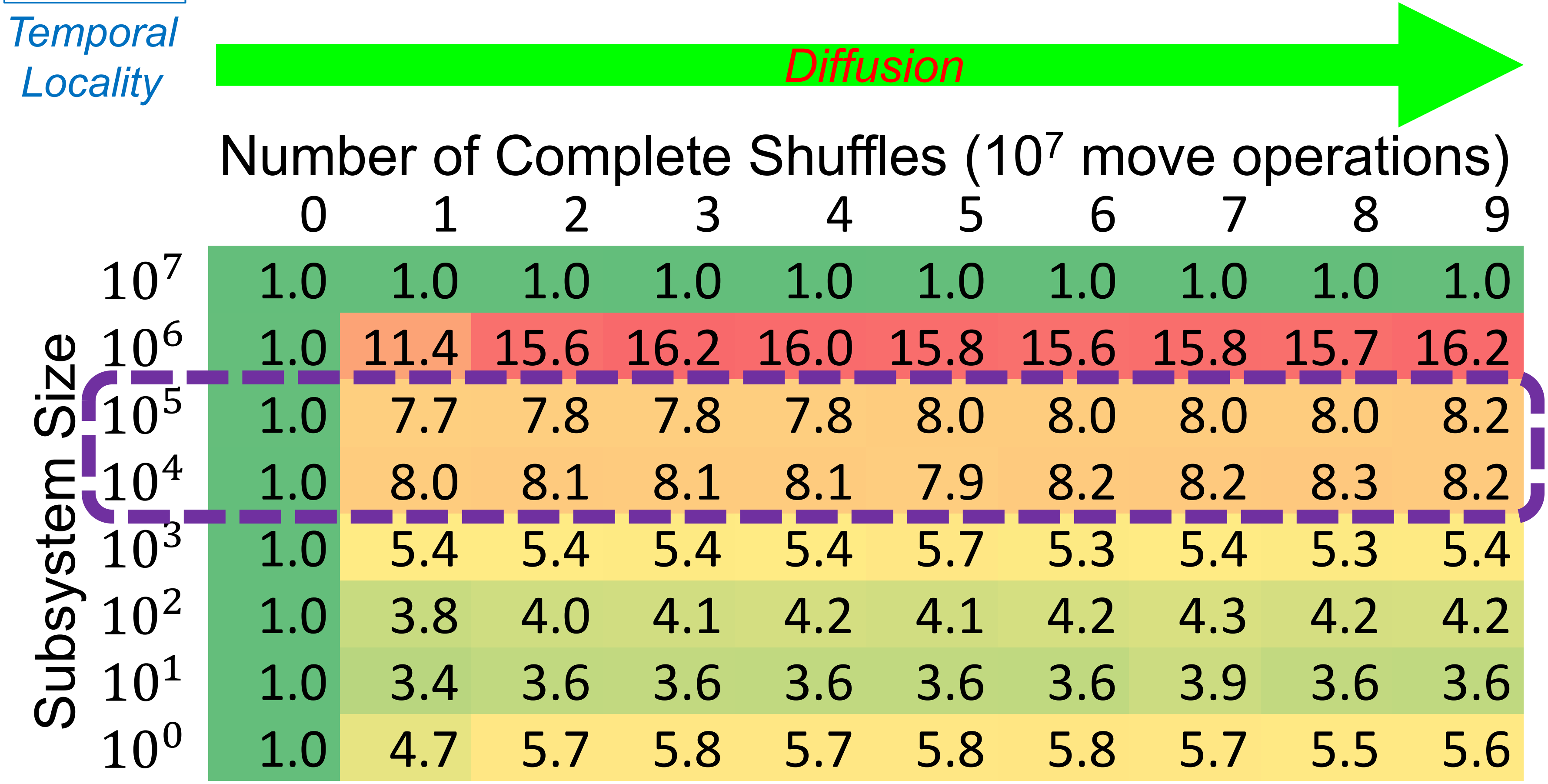
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7



Number of Complete Shuffles (10^7 move operations)

Subsystem Size

	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

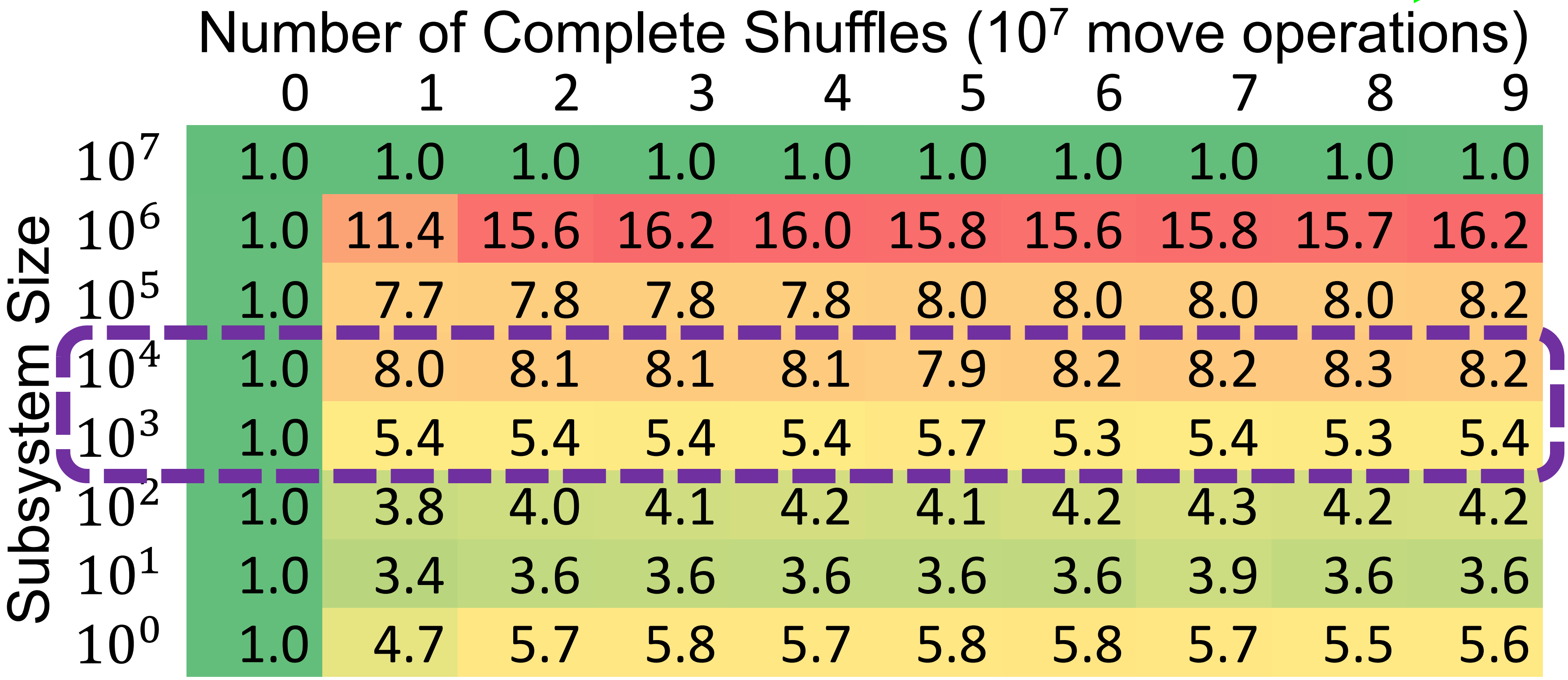
Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion



Number of Complete Shuffles (10^7 move operations)

Subsystem Size

	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

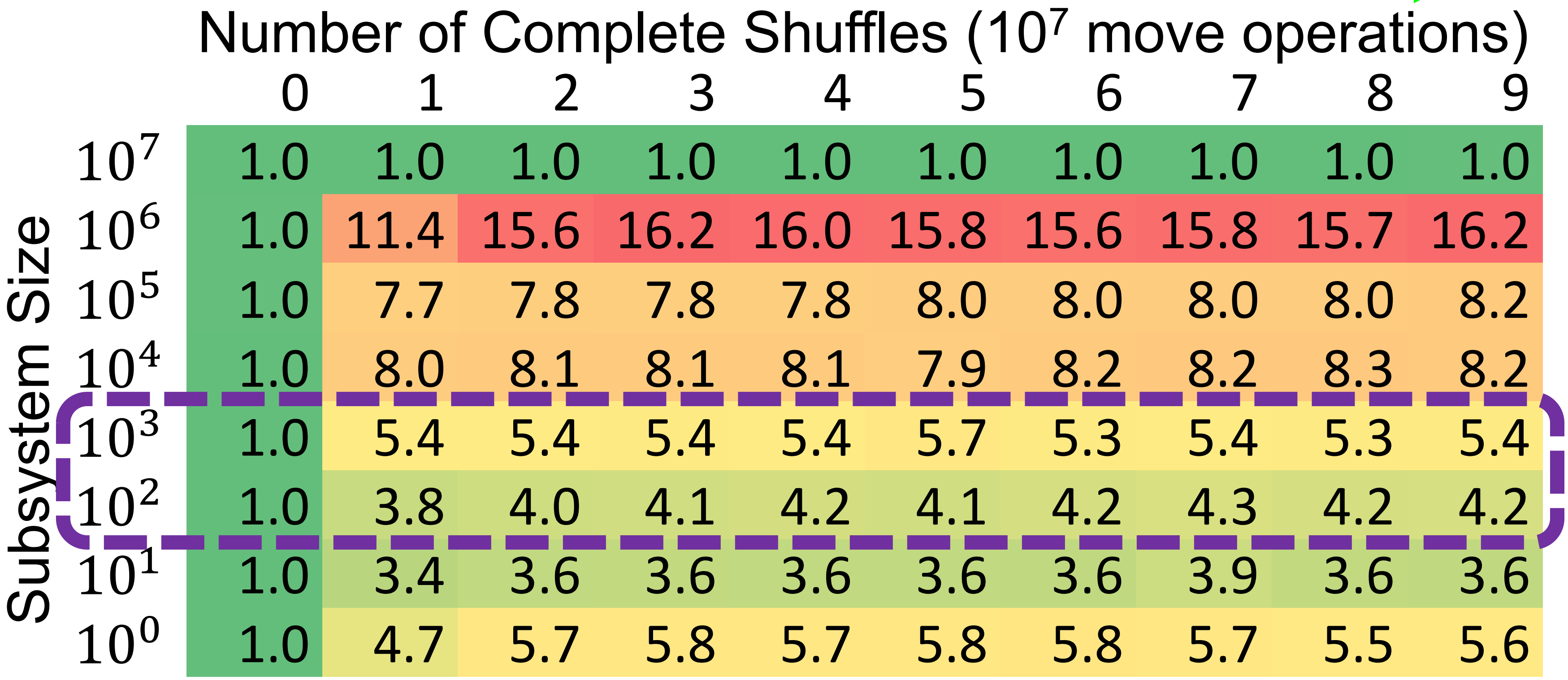
Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion



Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size |G| = K · |S| = 10^7

Defusion

Number of Complete Shuffles (10^7 move operations)

Subsystem Size	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Subsystem size
Is just one link!

Degradation Ratio (DR)

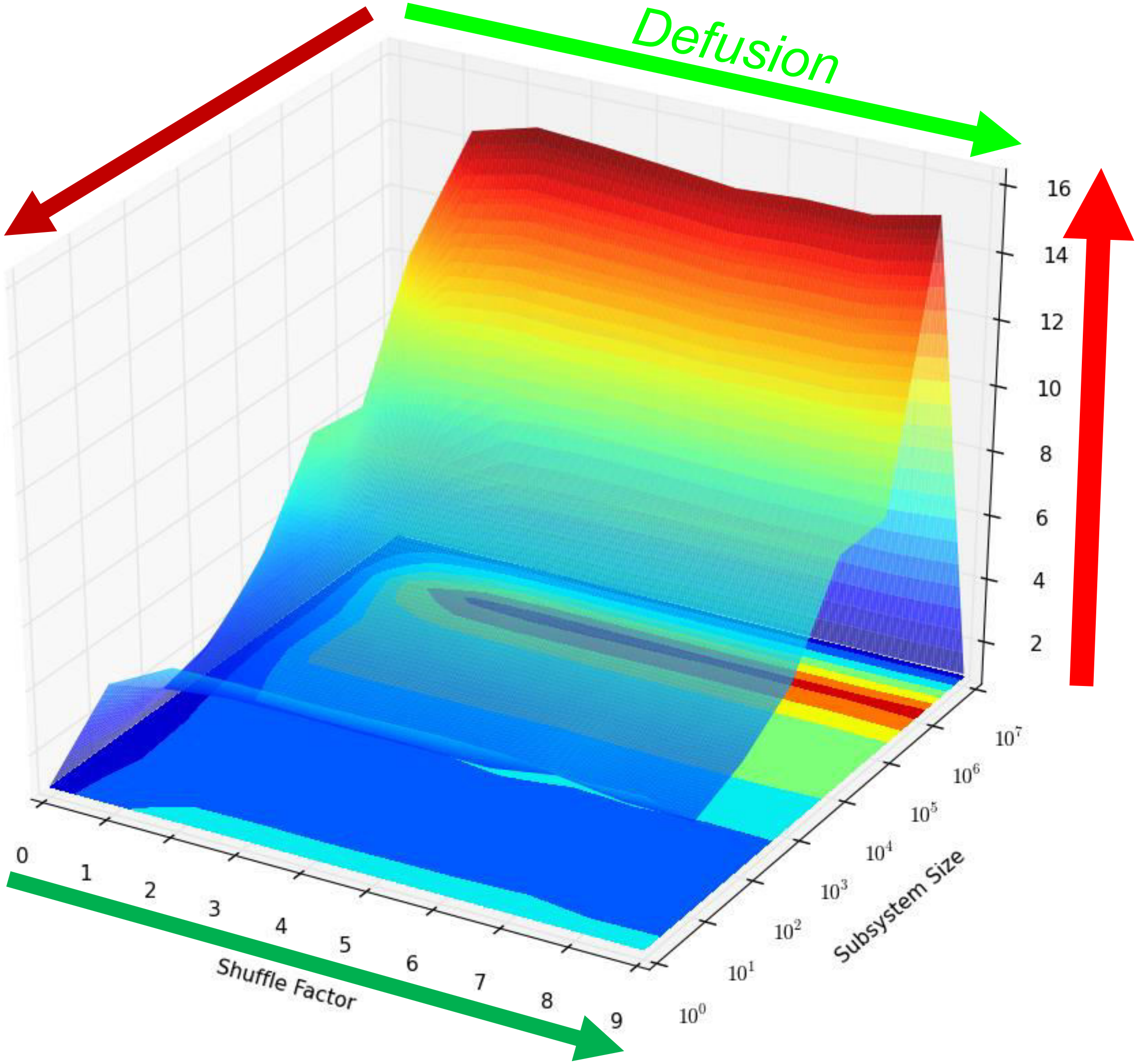
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

I = 10

Temporal Locality

Physical System Size $|G| = K \cdot |S| = 10^7$



Defusion

Number of Subsystems (K)

Degradation Ratio (DR)

(due to diffusion)

Number of Shuffles (sf)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Questions and/or Discussion?

Benchmark II: **L**OCALITY

Which local Allocation Strategy should we use?

Benchmark II: LOCALITY

Which <u>local</u> Allocation Strategy should we use?

AS7, AS9, AS11, or AS13

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Contrasting access times across (sub)system size

Overall System Size = 2^{21}

$\log_2 |\mathbf{G}| = 21$

These are all *exponents* of **2**.

Subsystem Size	Number of Subsystems
0	21
1	20
2	19
3	18
4	17
5	16
6	15
:	:

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

Contrasting access times across access rates

Overall Access Size = 2^{32}

$\log_2 \mathbf{N} = 32$

These
are all
exponents
of **2**.

Number of Iterations (I)	Number of Repitions (R)
8	24
7	25
6	26
5	27
4	28
3	29
2	30
1	31
0	32

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21}

Without Local Allocators

$\mathbf{sf} = 5$

Diffusion

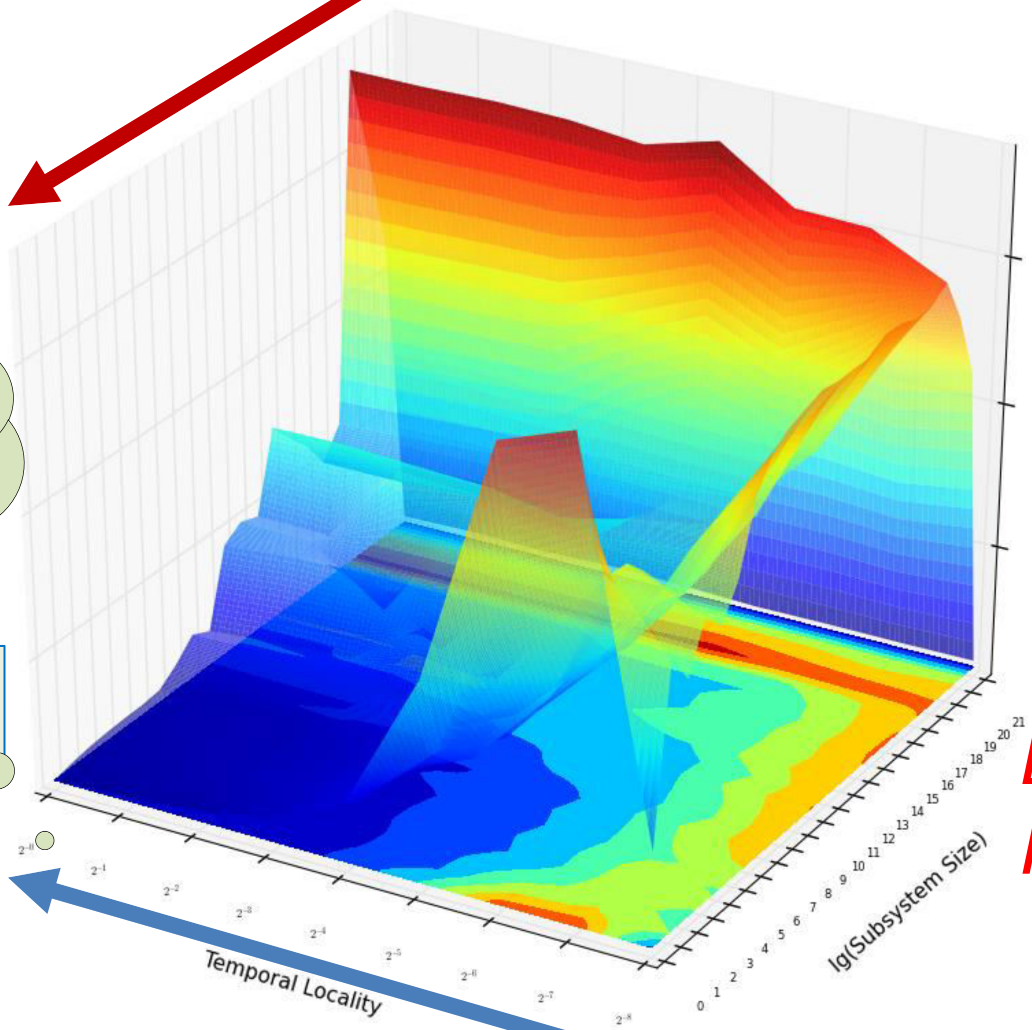
Physical Locality

$\mathbf{I} \cdot \mathbf{R}$ is fixed.

$\mathbf{I}$ ranges from $[\ 2^{8}\ ..\ 2^{0}\ ]$.

Max *Temporal Locality* = 256

Temporal Locality



Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21}

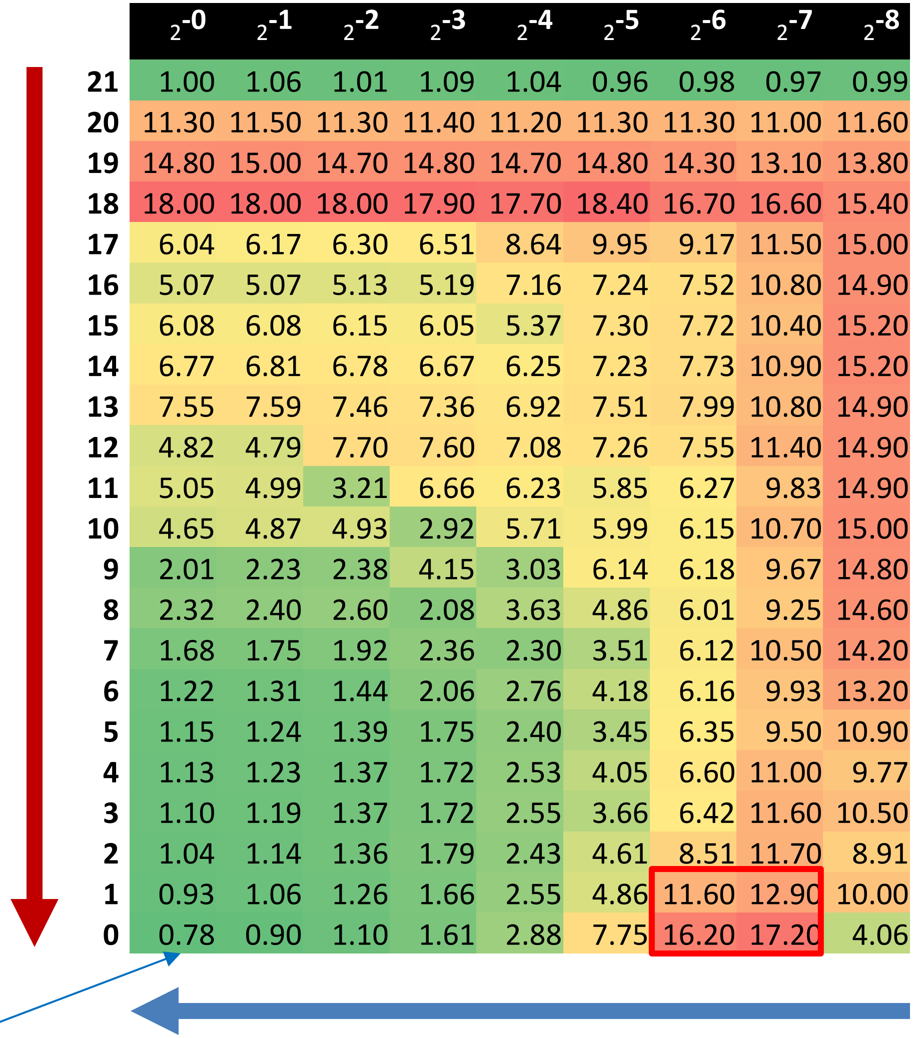
Without Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)

Physical Locality



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	1.00	1.06	1.01	1.09	1.04	0.96	0.98	0.97	0.99
20	11.30	11.50	11.30	11.40	11.20	11.30	11.30	11.00	11.60
19	14.80	15.00	14.70	14.80	14.70	14.80	14.30	13.10	13.80
18	18.00	18.00	18.00	17.90	17.70	18.40	16.70	16.60	15.40
17	6.04	6.17	6.30	6.51	8.64	9.95	9.17	11.50	15.00
16	5.07	5.07	5.13	5.19	7.16	7.24	7.52	10.80	14.90
15	6.08	6.08	6.15	6.05	5.37	7.30	7.72	10.40	15.20
14	6.77	6.81	6.78	6.67	6.25	7.23	7.73	10.90	15.20
13	7.55	7.59	7.46	7.36	6.92	7.51	7.99	10.80	14.90
12	4.82	4.79	7.70	7.60	7.08	7.26	7.55	11.40	14.90
11	5.05	4.99	3.21	6.66	6.23	5.85	6.27	9.83	14.90
10	4.65	4.87	4.93	2.92	5.71	5.99	6.15	10.70	15.00
9	2.01	2.23	2.38	4.15	3.03	6.14	6.18	9.67	14.80
8	2.32	2.40	2.60	2.08	3.63	4.86	6.01	9.25	14.60
7	1.68	1.75	1.92	2.36	2.30	3.51	6.12	10.50	14.20
6	1.22	1.31	1.44	2.06	2.76	4.18	6.16	9.93	13.20
5	1.15	1.24	1.39	1.75	2.40	3.45	6.35	9.50	10.90
4	1.13	1.23	1.37	1.72	2.53	4.05	6.60	11.00	9.77
3	1.10	1.19	1.37	1.72	2.55	3.66	6.42	11.60	10.50
2	1.04	1.14	1.36	1.79	2.43	4.61	8.51	11.70	8.91
1	0.93	1.06	1.26	1.66	2.55	4.86	11.60	12.90	10.00
0	0.78	0.90	1.10	1.61	2.88	7.75	16.20	17.20	4.06

Temporal Locality

Max *Temporal Locality* = 256

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

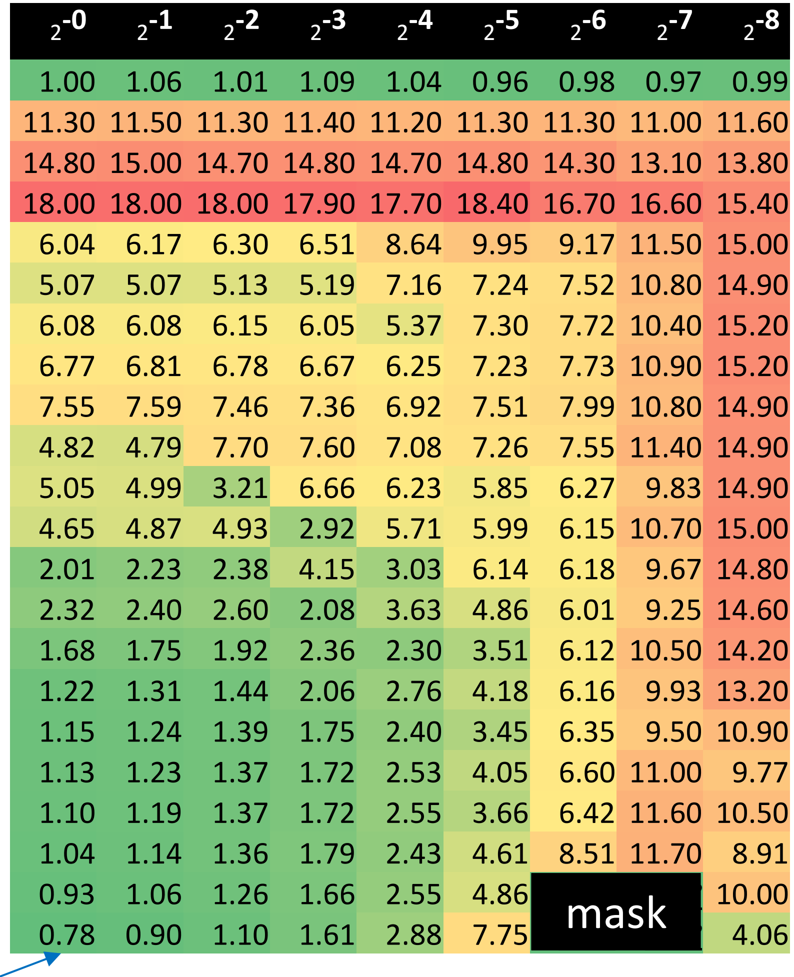
System Size = 2^{21} **Without** Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)

Physical Locality



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	1.00	1.06	1.01	1.09	1.04	0.96	0.98	0.97	0.99
20	11.30	11.50	11.30	11.40	11.20	11.30	11.30	11.00	11.60
19	14.80	15.00	14.70	14.80	14.70	14.80	14.30	13.10	13.80
18	18.00	18.00	18.00	17.90	17.70	18.40	16.70	16.60	15.40
17	6.04	6.17	6.30	6.51	8.64	9.95	9.17	11.50	15.00
16	5.07	5.07	5.13	5.19	7.16	7.24	7.52	10.80	14.90
15	6.08	6.08	6.15	6.05	5.37	7.30	7.72	10.40	15.20
14	6.77	6.81	6.78	6.67	6.25	7.23	7.73	10.90	15.20
13	7.55	7.59	7.46	7.36	6.92	7.51	7.99	10.80	14.90
12	4.82	4.79	7.70	7.60	7.08	7.26	7.55	11.40	14.90
11	5.05	4.99	3.21	6.66	6.23	5.85	6.27	9.83	14.90
10	4.65	4.87	4.93	2.92	5.71	5.99	6.15	10.70	15.00
9	2.01	2.23	2.38	4.15	3.03	6.14	6.18	9.67	14.80
8	2.32	2.40	2.60	2.08	3.63	4.86	6.01	9.25	14.60
7	1.68	1.75	1.92	2.36	2.30	3.51	6.12	10.50	14.20
6	1.22	1.31	1.44	2.06	2.76	4.18	6.16	9.93	13.20
5	1.15	1.24	1.39	1.75	2.40	3.45	6.35	9.50	10.90
4	1.13	1.23	1.37	1.72	2.53	4.05	6.60	11.00	9.77
3	1.10	1.19	1.37	1.72	2.55	3.66	6.42	11.60	10.50
2	1.04	1.14	1.36	1.79	2.43	4.61	8.51	11.70	8.91
1	0.93	1.06	1.26	1.66	2.55	4.86	mask		10.00
0	0.78	0.90	1.10	1.61	2.88	7.75			4.06

Max *Temporal Locality* = 256

Temporal Locality

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{25}

Without Local Allocators

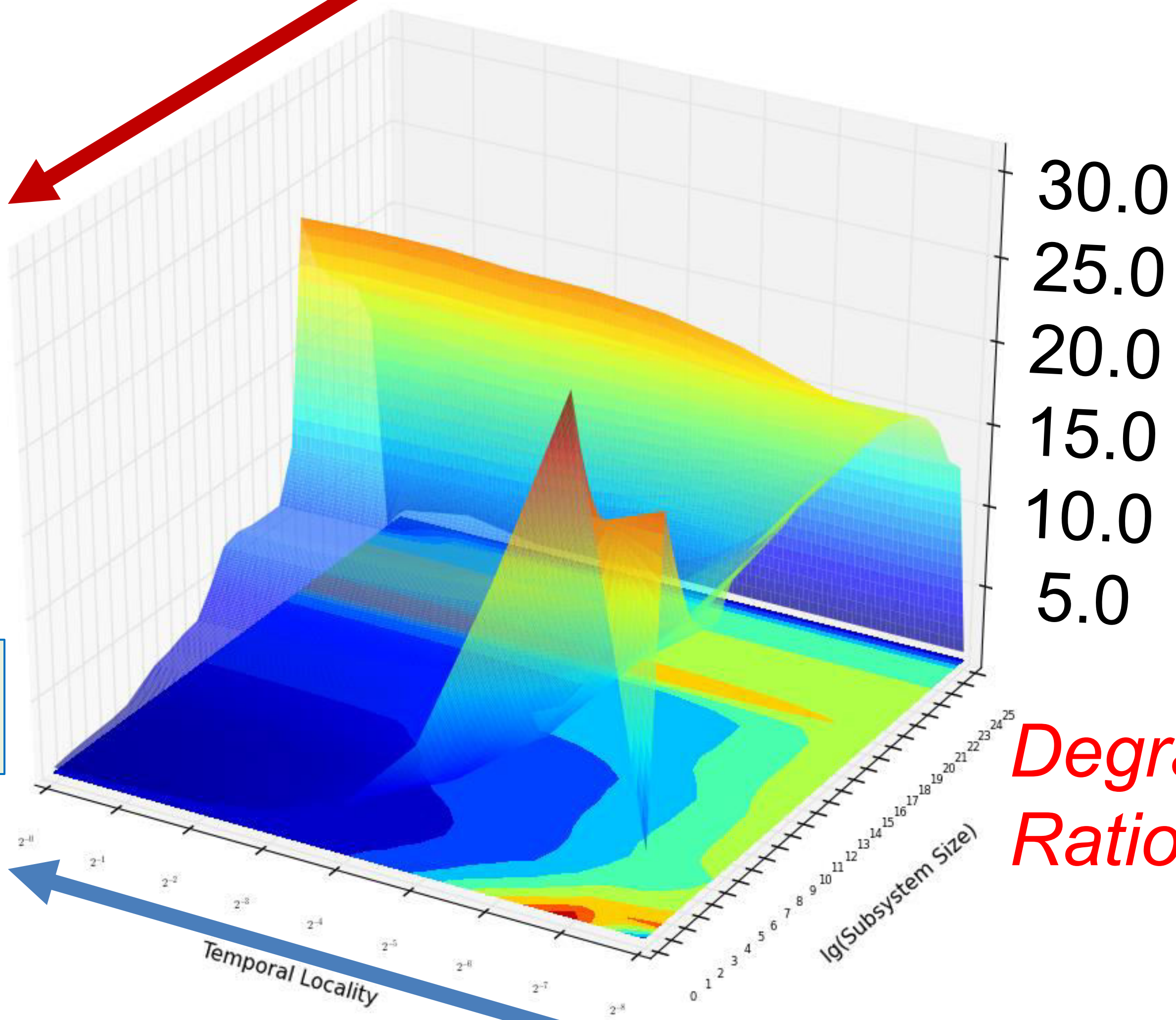
sf = 5

Diffusion

Physical Locality

Max *Temporal Locality* = 256

Temporal Locality



Degradation Ratio (DR)

3. Analyzing the Benchmark Data

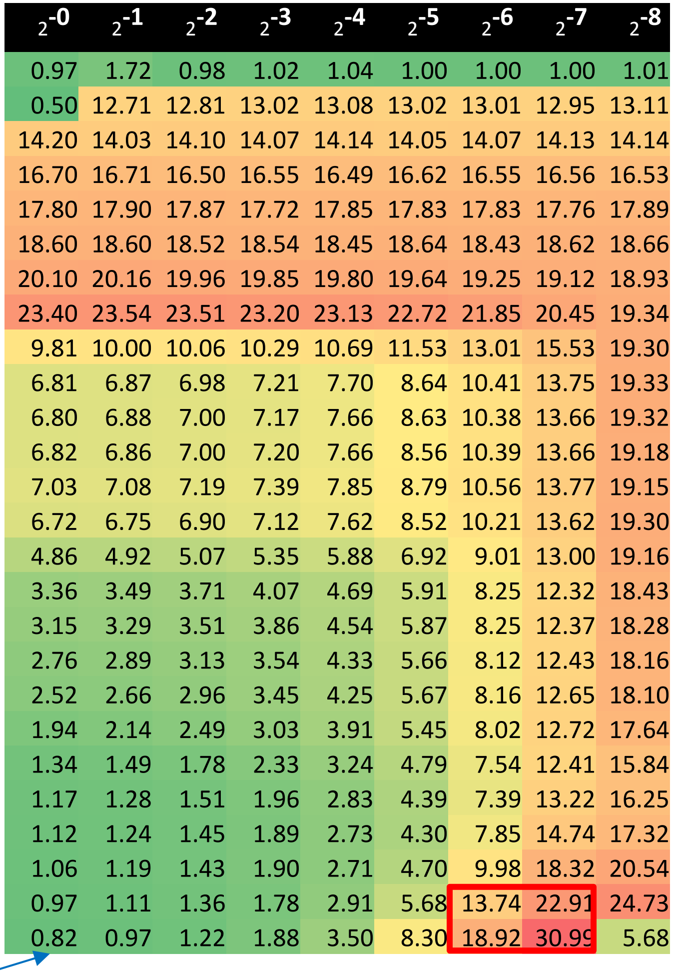
Benchmark II: LOCALITY

System Size = 2^{25} **Without** Local Allocators

sf = 5

Diffusion

Physical Locality



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.72	0.98	1.02	1.04	1.00	1.00	1.00	1.01
24	0.50	12.71	12.81	13.02	13.08	13.02	13.01	12.95	13.11
23	14.20	14.03	14.10	14.07	14.14	14.05	14.07	14.13	14.14
22	16.70	16.71	16.50	16.55	16.49	16.62	16.55	16.56	16.53
21	17.80	17.90	17.87	17.72	17.85	17.83	17.83	17.76	17.89
20	18.60	18.60	18.52	18.54	18.45	18.64	18.43	18.62	18.66
19	20.10	20.16	19.96	19.85	19.80	19.64	19.25	19.12	18.93
18	23.40	23.54	23.51	23.20	23.13	22.72	21.85	20.45	19.34
17	9.81	10.00	10.06	10.29	10.69	11.53	13.01	15.53	19.30
16	6.81	6.87	6.98	7.21	7.70	8.64	10.41	13.75	19.33
15	6.80	6.88	7.00	7.17	7.66	8.63	10.38	13.66	19.32
14	6.82	6.86	7.00	7.20	7.66	8.56	10.39	13.66	19.18
13	7.03	7.08	7.19	7.39	7.85	8.79	10.56	13.77	19.15
12	6.72	6.75	6.90	7.12	7.62	8.52	10.21	13.62	19.30
11	4.86	4.92	5.07	5.35	5.88	6.92	9.01	13.00	19.16
10	3.36	3.49	3.71	4.07	4.69	5.91	8.25	12.32	18.43
9	3.15	3.29	3.51	3.86	4.54	5.87	8.25	12.37	18.28
8	2.76	2.89	3.13	3.54	4.33	5.66	8.12	12.43	18.16
7	2.52	2.66	2.96	3.45	4.25	5.67	8.16	12.65	18.10
6	1.94	2.14	2.49	3.03	3.91	5.45	8.02	12.72	17.64
5	1.34	1.49	1.78	2.33	3.24	4.79	7.54	12.41	15.84
4	1.17	1.28	1.51	1.96	2.83	4.39	7.39	13.22	16.25
3	1.12	1.24	1.45	1.89	2.73	4.30	7.85	14.74	17.32
2	1.06	1.19	1.43	1.90	2.71	4.70	9.98	18.32	20.54
1	0.97	1.11	1.36	1.78	2.91	5.68	13.74	22.91	24.73
0	0.82	0.97	1.22	1.88	3.50	8.30	18.92	30.99	5.68

Degradation Ratio (DR)

Max *Temporal Locality* = 256

Temporal Locality

3. Analyzing the Benchmark Data

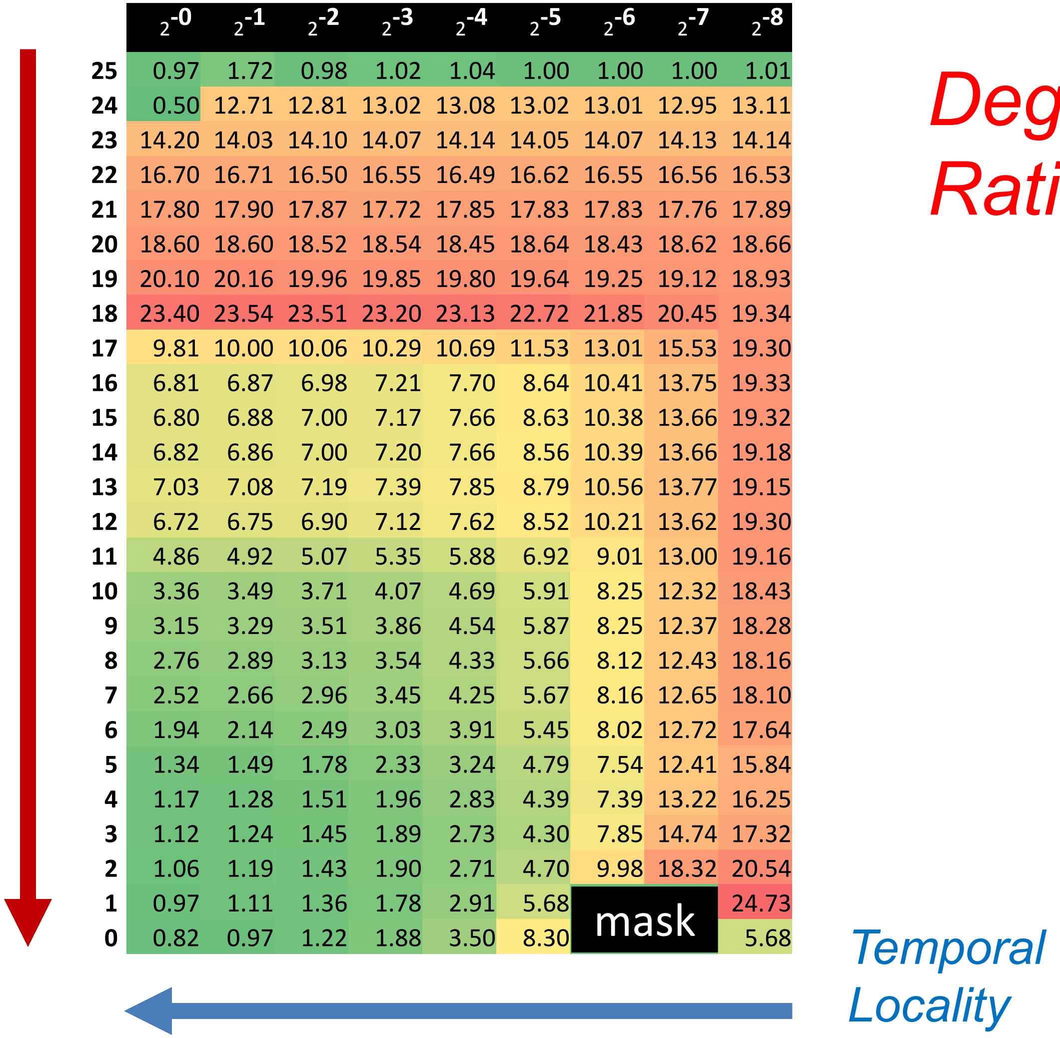
Benchmark II: LOCALITY

System Size = 2^{25} **Without** Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.72	0.98	1.02	1.04	1.00	1.00	1.00	1.01
24	0.50	12.71	12.81	13.02	13.08	13.02	13.01	12.95	13.11
23	14.20	14.03	14.10	14.07	14.14	14.05	14.07	14.13	14.14
22	16.70	16.71	16.50	16.55	16.49	16.62	16.55	16.56	16.53
21	17.80	17.90	17.87	17.72	17.85	17.83	17.83	17.76	17.89
20	18.60	18.60	18.52	18.54	18.45	18.64	18.43	18.62	18.66
19	20.10	20.16	19.96	19.85	19.80	19.64	19.25	19.12	18.93
18	23.40	23.54	23.51	23.20	23.13	22.72	21.85	20.45	19.34
17	9.81	10.00	10.06	10.29	10.69	11.53	13.01	15.53	19.30
16	6.81	6.87	6.98	7.21	7.70	8.64	10.41	13.75	19.33
15	6.80	6.88	7.00	7.17	7.66	8.63	10.38	13.66	19.32
14	6.82	6.86	7.00	7.20	7.66	8.56	10.39	13.66	19.18
13	7.03	7.08	7.19	7.39	7.85	8.79	10.56	13.77	19.15
12	6.72	6.75	6.90	7.12	7.62	8.52	10.21	13.62	19.30
11	4.86	4.92	5.07	5.35	5.88	6.92	9.01	13.00	19.16
10	3.36	3.49	3.71	4.07	4.69	5.91	8.25	12.32	18.43
9	3.15	3.29	3.51	3.86	4.54	5.87	8.25	12.37	18.28
8	2.76	2.89	3.13	3.54	4.33	5.66	8.12	12.43	18.16
7	2.52	2.66	2.96	3.45	4.25	5.67	8.16	12.65	18.10
6	1.94	2.14	2.49	3.03	3.91	5.45	8.02	12.72	17.64
5	1.34	1.49	1.78	2.33	3.24	4.79	7.54	12.41	15.84
4	1.17	1.28	1.51	1.96	2.83	4.39	7.39	13.22	16.25
3	1.12	1.24	1.45	1.89	2.73	4.30	7.85	14.74	17.32
2	1.06	1.19	1.43	1.90	2.71	4.70	9.98	18.32	20.54
1	0.97	1.11	1.36	1.78	2.91	5.68	mask		24.73
0	0.82	0.97	1.22	1.88	3.50	8.30			5.68

Physical Locality

Temporal Locality

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21}

With Local Allocators

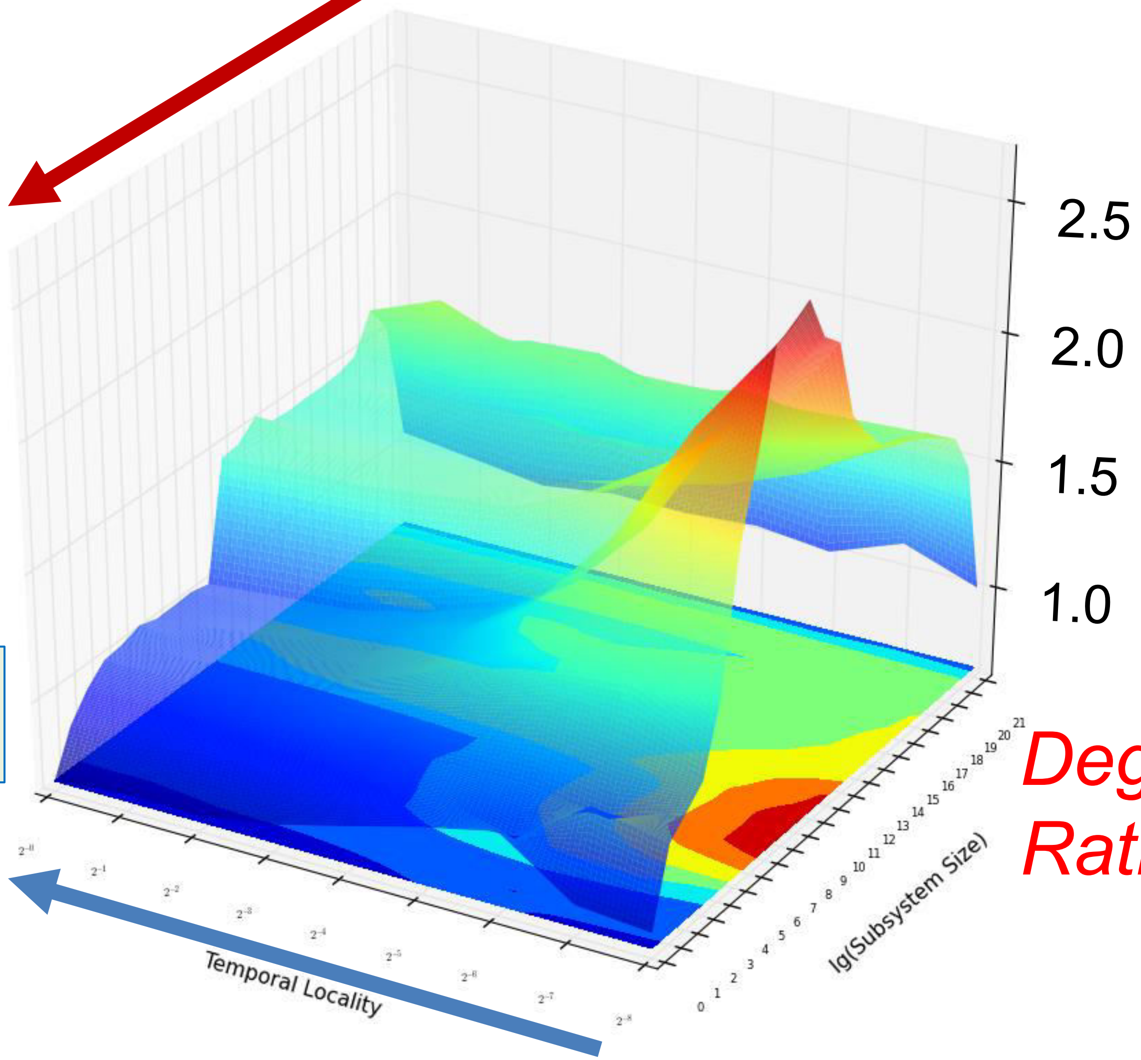
sf = 5

Diffusion

Physical Locality

Max *Temporal Locality* = 256

Temporal Locality



Degradation Ratio (DR)

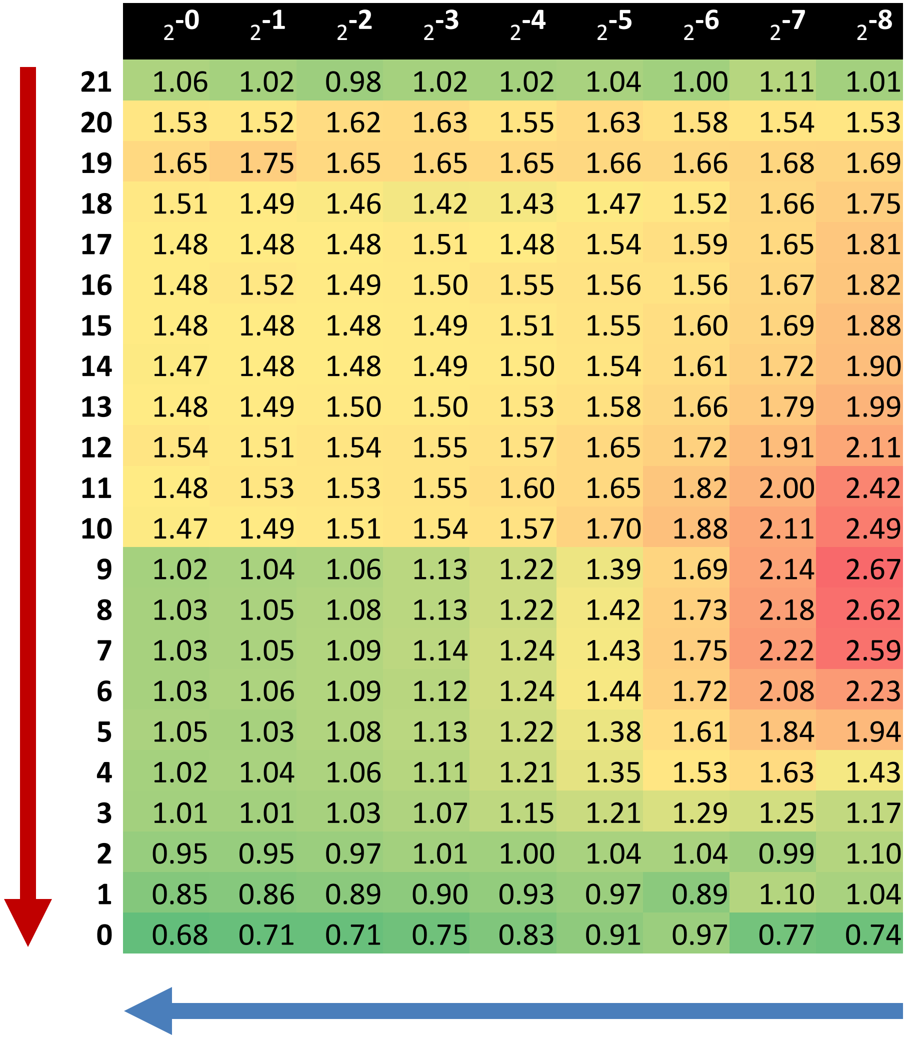
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21} **With** Local Allocators

sf = 5

Diffusion



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	1.06	1.02	0.98	1.02	1.02	1.04	1.00	1.11	1.01
20	1.53	1.52	1.62	1.63	1.55	1.63	1.58	1.54	1.53
19	1.65	1.75	1.65	1.65	1.65	1.66	1.66	1.68	1.69
18	1.51	1.49	1.46	1.42	1.43	1.47	1.52	1.66	1.75
17	1.48	1.48	1.48	1.51	1.48	1.54	1.59	1.65	1.81
16	1.48	1.52	1.49	1.50	1.55	1.56	1.56	1.67	1.82
15	1.48	1.48	1.48	1.49	1.51	1.55	1.60	1.69	1.88
14	1.47	1.48	1.48	1.49	1.50	1.54	1.61	1.72	1.90
13	1.48	1.49	1.50	1.50	1.53	1.58	1.66	1.79	1.99
12	1.54	1.51	1.54	1.55	1.57	1.65	1.72	1.91	2.11
11	1.48	1.53	1.53	1.55	1.60	1.65	1.82	2.00	2.42
10	1.47	1.49	1.51	1.54	1.57	1.70	1.88	2.11	2.49
9	1.02	1.04	1.06	1.13	1.22	1.39	1.69	2.14	2.67
8	1.03	1.05	1.08	1.13	1.22	1.42	1.73	2.18	2.62
7	1.03	1.05	1.09	1.14	1.24	1.43	1.75	2.22	2.59
6	1.03	1.06	1.09	1.12	1.24	1.44	1.72	2.08	2.23
5	1.05	1.03	1.08	1.13	1.22	1.38	1.61	1.84	1.94
4	1.02	1.04	1.06	1.11	1.21	1.35	1.53	1.63	1.43
3	1.01	1.01	1.03	1.07	1.15	1.21	1.29	1.25	1.17
2	0.95	0.95	0.97	1.01	1.00	1.04	1.04	0.99	1.10
1	0.85	0.86	0.89	0.90	0.93	0.97	0.89	1.10	1.04
0	0.68	0.71	0.71	0.75	0.83	0.91	0.97	0.77	0.74

Physical Locality

Degradation Ratio (DR)

Temporal Locality

Benchmark II: LOCALITY

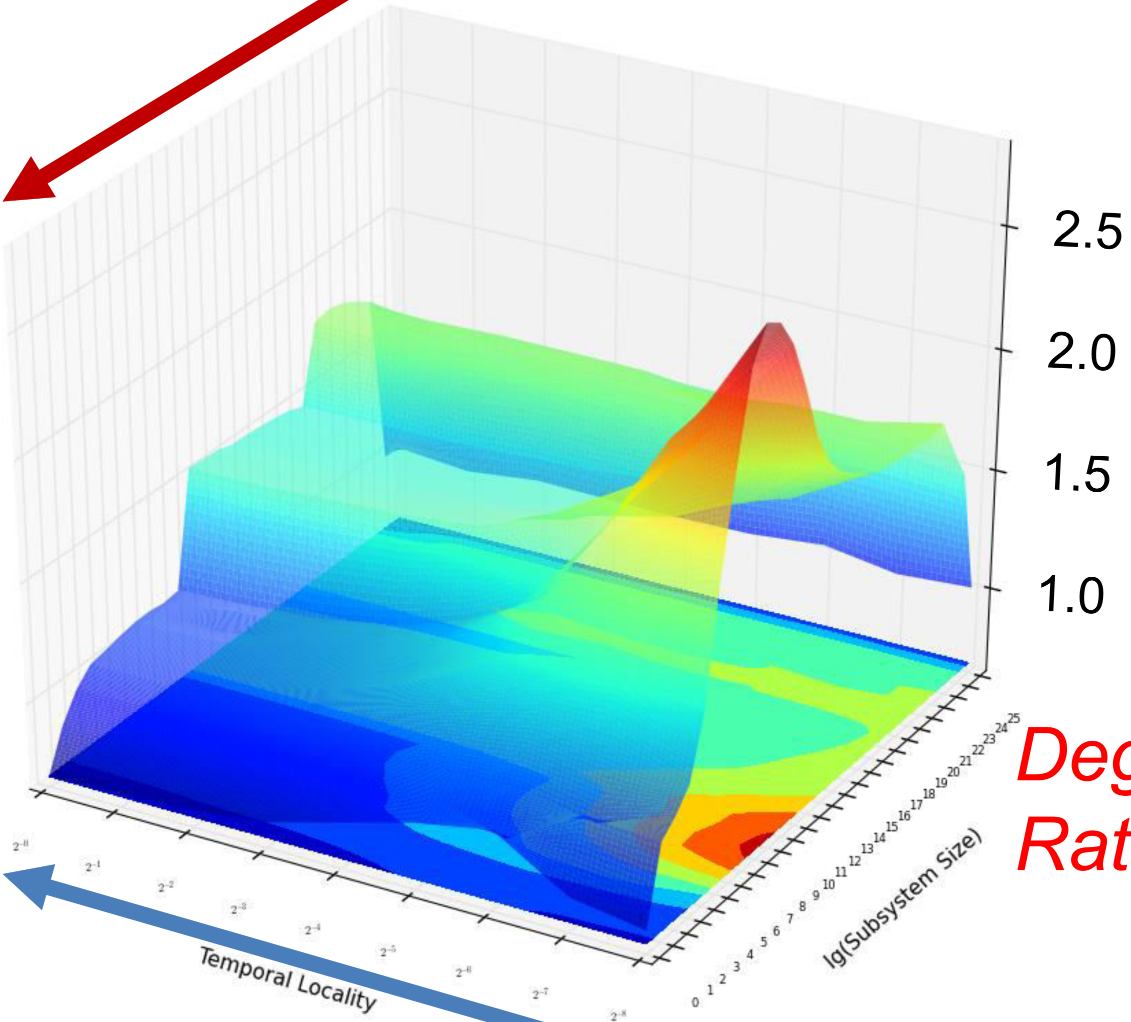
System Size = 2^{25}

With Local Allocators

sf = 5

Diffusion

Physical Locality



Max *Temporal Locality* = 256

Temporal Locality

Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{25}

With Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)

Physical Locality

Temporal Locality

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	1.00	0.97	1.01	1.00	1.00	1.01	1.04	0.99	1.01
24	1.00	1.54	1.57	1.55	1.55	1.53	1.55	1.56	1.53
23	1.71	1.67	1.70	1.69	1.68	1.68	1.68	1.69	1.67
22	1.75	1.75	1.76	1.76	1.75	1.72	1.76	1.76	1.83
21	1.79	1.78	1.78	1.80	1.79	1.74	1.80	1.80	1.80
20	1.80	1.80	1.80	1.81	1.81	1.82	1.81	1.81	1.82
19	1.79	1.78	1.79	1.80	1.79	1.80	1.80	1.82	1.82
18	1.47	1.47	1.47	1.49	1.50	1.53	1.58	1.67	1.83
17	1.49	1.49	1.49	1.50	1.51	1.54	1.59	1.67	1.84
16	1.50	1.50	1.53	1.51	1.53	1.55	1.61	1.70	1.88
15	1.51	1.51	1.51	1.52	1.53	1.56	1.63	1.74	1.92
14	1.51	1.51	1.52	1.52	1.54	1.58	1.65	1.78	1.97
13	1.51	1.52	1.52	1.53	1.55	1.60	1.67	1.82	2.05
12	1.53	1.54	1.54	1.56	1.59	1.64	1.74	1.92	2.20
11	1.54	1.54	1.55	1.57	1.60	1.67	1.84	2.08	2.43
10	1.54	1.55	1.56	1.58	1.61	1.74	1.93	2.25	2.63
9	1.07	1.08	1.11	1.16	1.26	1.44	1.87	2.22	2.76
8	1.06	1.10	1.12	1.18	1.27	1.47	1.85	2.29	2.80
7	1.07	1.06	1.12	1.17	1.28	1.48	1.82	2.32	2.67
6	1.07	1.08	1.10	1.16	1.26	1.46	1.75	2.13	2.31
5	1.05	1.06	1.09	1.14	1.23	1.40	1.62	1.86	1.93
4	1.04	1.05	1.07	1.12	1.22	1.38	1.54	1.65	1.44
3	1.02	1.03	1.05	1.08	1.15	1.23	1.30	1.29	1.20
2	0.96	0.97	0.99	1.02	1.02	1.04	1.05	1.00	1.12
1	0.85	0.86	0.89	0.90	0.94	0.99	0.90	1.08	1.06
0	0.69	0.70	0.72	0.75	0.84	0.92	0.98	0.79	0.74

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21} **Without**/With Local Allocators

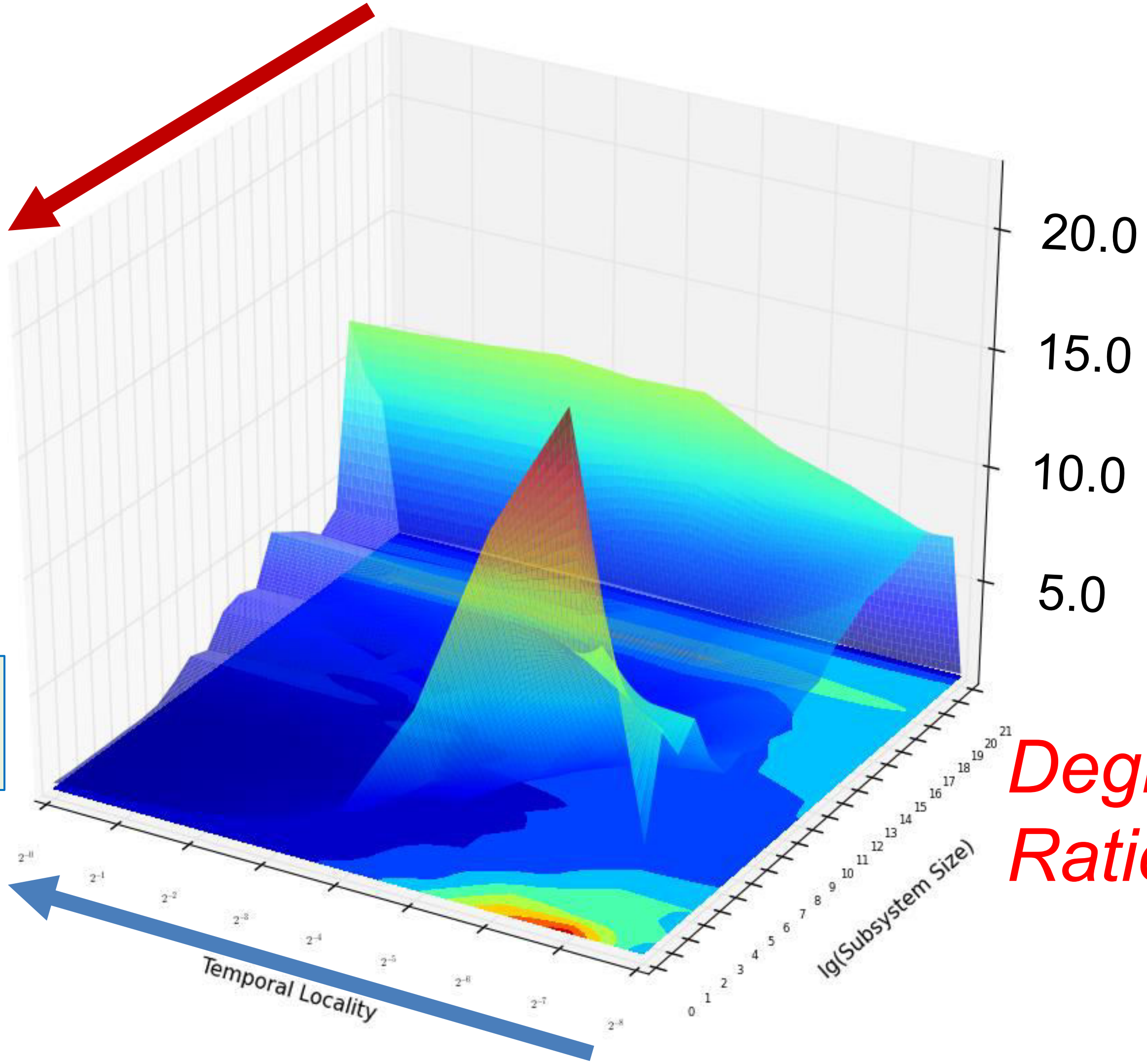
sf = 5

Diffusion

Physical Locality

Max *Temporal Locality* = 256

Temporal Locality



Degradation Ratio (DR)

3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{21} **Without**/With Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)

Physical Locality

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	0.95	1.03	1.03	1.07	1.02	0.92	0.97	0.87	0.98
20	7.40	7.56	6.97	6.96	7.27	6.94	7.15	7.10	7.56
19	9.02	8.56	8.95	8.99	8.92	8.92	8.61	7.82	8.17
18	11.90	12.10	12.30	12.60	12.40	12.50	10.90	10.00	8.80
17	4.07	4.16	4.26	4.31	5.85	6.48	5.78	6.97	8.27
16	3.43	3.34	3.45	3.45	4.61	4.63	4.82	6.49	8.19
15	4.11	4.11	4.16	4.06	3.56	4.71	4.84	6.14	8.09
14	4.59	4.61	4.59	4.47	4.15	4.69	4.81	6.34	8.02
13	5.11	5.08	4.96	4.90	4.52	4.76	4.82	6.03	7.47
12	3.12	3.16	5.01	4.89	4.51	4.41	4.40	5.96	7.08
11	3.41	3.27	2.09	4.31	3.90	3.54	3.45	4.91	6.15
10	3.16	3.26	3.27	1.89	3.63	3.53	3.28	5.08	6.04
9	1.98	2.15	2.24	3.68	2.48	4.43	3.66	4.51	5.56
8	2.24	2.29	2.42	1.83	2.97	3.43	3.48	4.24	5.59
7	1.64	1.67	1.77	2.07	1.85	2.45	3.49	4.70	5.49
6	1.19	1.24	1.32	1.83	2.23	2.91	3.59	4.78	5.91
5	1.10	1.20	1.29	1.55	1.97	2.50	3.96	5.17	5.61
4	1.11	1.18	1.30	1.55	2.09	3.00	4.31	6.73	6.82
3	1.09	1.18	1.33	1.60	2.23	3.02	4.99	9.30	9.03
2	1.10	1.21	1.40	1.76	2.42	4.43	8.19	11.80	8.09
1	1.09	1.24	1.41	1.85	2.73	5.00	13.00	11.70	9.70
0	1.14	1.26	1.56	2.14	3.47	8.54	16.70	22.40	5.46

Temporal Locality

3. Analyzing the Benchmark Data

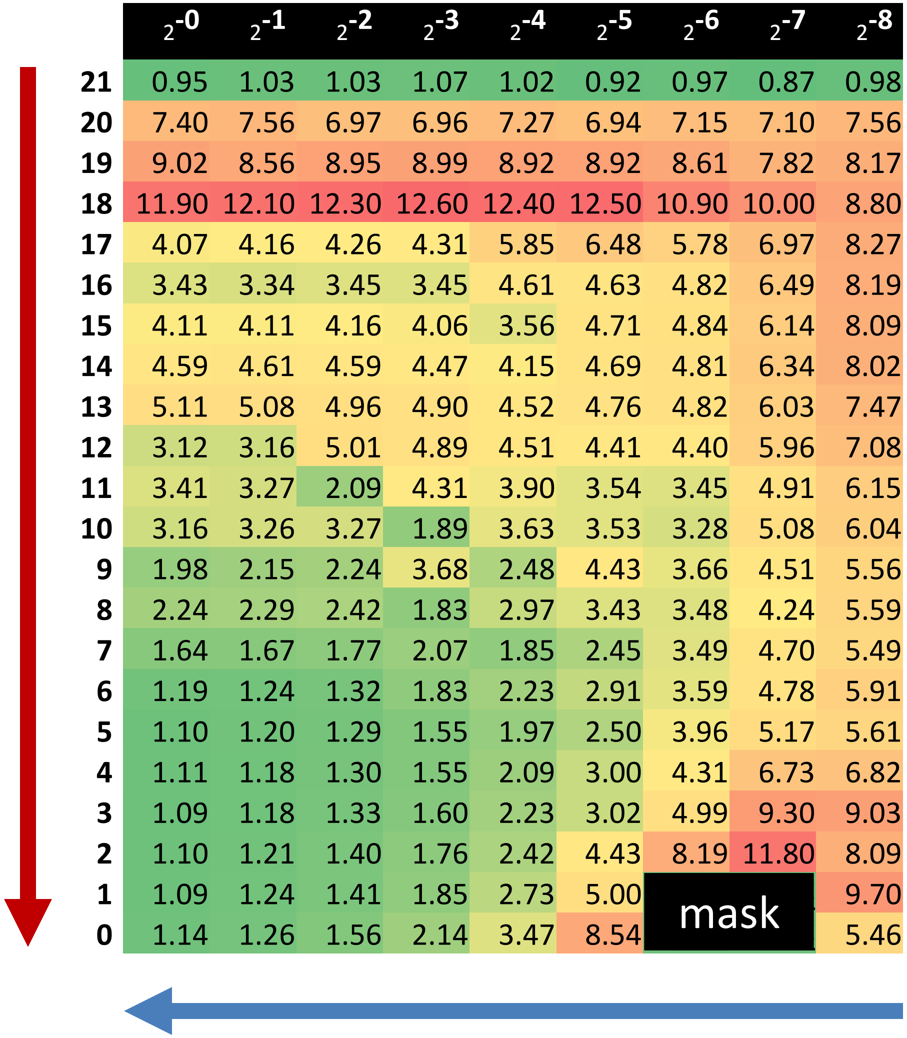
Benchmark II: LOCALITY

System Size = 2^{21} **Without**/With Local Allocators

sf = 5

Diffusion

Degradation Ratio (DR)



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	0.95	1.03	1.03	1.07	1.02	0.92	0.97	0.87	0.98
20	7.40	7.56	6.97	6.96	7.27	6.94	7.15	7.10	7.56
19	9.02	8.56	8.95	8.99	8.92	8.92	8.61	7.82	8.17
18	11.90	12.10	12.30	12.60	12.40	12.50	10.90	10.00	8.80
17	4.07	4.16	4.26	4.31	5.85	6.48	5.78	6.97	8.27
16	3.43	3.34	3.45	3.45	4.61	4.63	4.82	6.49	8.19
15	4.11	4.11	4.16	4.06	3.56	4.71	4.84	6.14	8.09
14	4.59	4.61	4.59	4.47	4.15	4.69	4.81	6.34	8.02
13	5.11	5.08	4.96	4.90	4.52	4.76	4.82	6.03	7.47
12	3.12	3.16	5.01	4.89	4.51	4.41	4.40	5.96	7.08
11	3.41	3.27	2.09	4.31	3.90	3.54	3.45	4.91	6.15
10	3.16	3.26	3.27	1.89	3.63	3.53	3.28	5.08	6.04
9	1.98	2.15	2.24	3.68	2.48	4.43	3.66	4.51	5.56
8	2.24	2.29	2.42	1.83	2.97	3.43	3.48	4.24	5.59
7	1.64	1.67	1.77	2.07	1.85	2.45	3.49	4.70	5.49
6	1.19	1.24	1.32	1.83	2.23	2.91	3.59	4.78	5.91
5	1.10	1.20	1.29	1.55	1.97	2.50	3.96	5.17	5.61
4	1.11	1.18	1.30	1.55	2.09	3.00	4.31	6.73	6.82
3	1.09	1.18	1.33	1.60	2.23	3.02	4.99	9.30	9.03
2	1.10	1.21	1.40	1.76	2.42	4.43	8.19	11.80	8.09
1	1.09	1.24	1.41	1.85	2.73	5.00	mask		9.70
0	1.14	1.26	1.56	2.14	3.47	8.54			5.46

Physical Locality

Temporal Locality

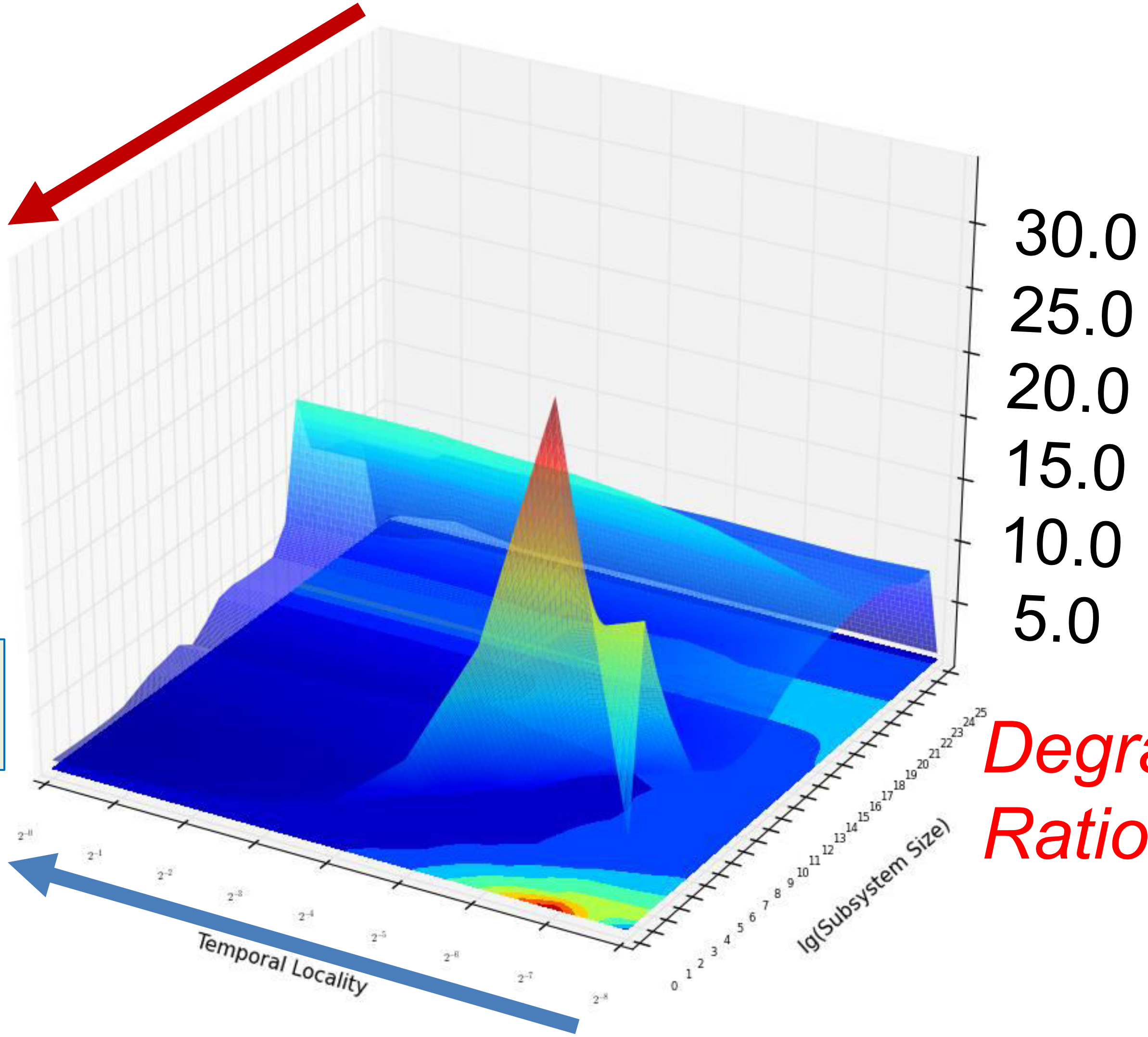
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{25} **Without**/With Local Allocators

sf = 5

Diffusion



Physical Locality

Max *Temporal Locality* = 256

Temporal Locality

Degradation Ratio (DR)

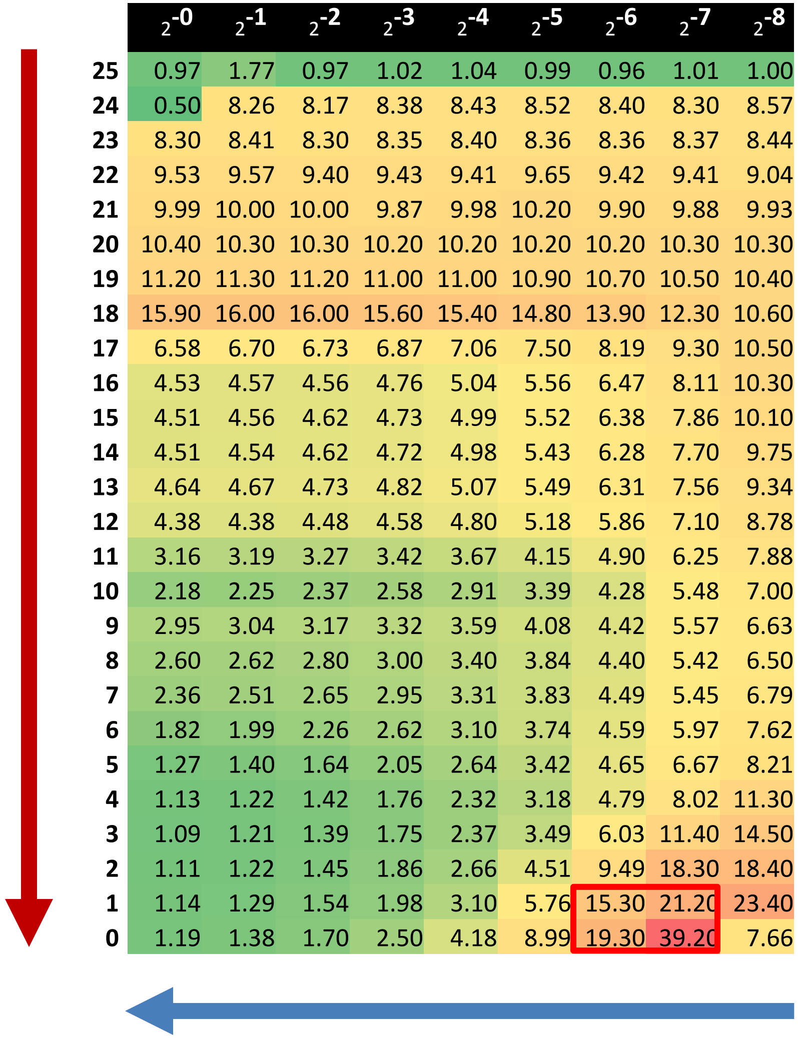
3. Analyzing the Benchmark Data

Benchmark II: LOCALITY

System Size = 2^{25} **Without**/With Local Allocators

sf = 5

Diffusion



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.77	0.97	1.02	1.04	0.99	0.96	1.01	1.00
24	0.50	8.26	8.17	8.38	8.43	8.52	8.40	8.30	8.57
23	8.30	8.41	8.30	8.35	8.40	8.36	8.36	8.37	8.44
22	9.53	9.57	9.40	9.43	9.41	9.65	9.42	9.41	9.04
21	9.99	10.00	10.00	9.87	9.98	10.20	9.90	9.88	9.93
20	10.40	10.30	10.30	10.20	10.20	10.20	10.20	10.30	10.30
19	11.20	11.30	11.20	11.00	11.00	10.90	10.70	10.50	10.40
18	15.90	16.00	16.00	15.60	15.40	14.80	13.90	12.30	10.60
17	6.58	6.70	6.73	6.87	7.06	7.50	8.19	9.30	10.50
16	4.53	4.57	4.56	4.76	5.04	5.56	6.47	8.11	10.30
15	4.51	4.56	4.62	4.73	4.99	5.52	6.38	7.86	10.10
14	4.51	4.54	4.62	4.72	4.98	5.43	6.28	7.70	9.75
13	4.64	4.67	4.73	4.82	5.07	5.49	6.31	7.56	9.34
12	4.38	4.38	4.48	4.58	4.80	5.18	5.86	7.10	8.78
11	3.16	3.19	3.27	3.42	3.67	4.15	4.90	6.25	7.88
10	2.18	2.25	2.37	2.58	2.91	3.39	4.28	5.48	7.00
9	2.95	3.04	3.17	3.32	3.59	4.08	4.42	5.57	6.63
8	2.60	2.62	2.80	3.00	3.40	3.84	4.40	5.42	6.50
7	2.36	2.51	2.65	2.95	3.31	3.83	4.49	5.45	6.79
6	1.82	1.99	2.26	2.62	3.10	3.74	4.59	5.97	7.62
5	1.27	1.40	1.64	2.05	2.64	3.42	4.65	6.67	8.21
4	1.13	1.22	1.42	1.76	2.32	3.18	4.79	8.02	11.30
3	1.09	1.21	1.39	1.75	2.37	3.49	6.03	11.40	14.50
2	1.11	1.22	1.45	1.86	2.66	4.51	9.49	18.30	18.40
1	1.14	1.29	1.54	1.98	3.10	5.76	15.30	21.20	23.40
0	1.19	1.38	1.70	2.50	4.18	8.99	19.30	39.20	7.66

Physical Locality

Degradation Ratio (DR)

Temporal Locality

3. Analyzing the Benchmark Data

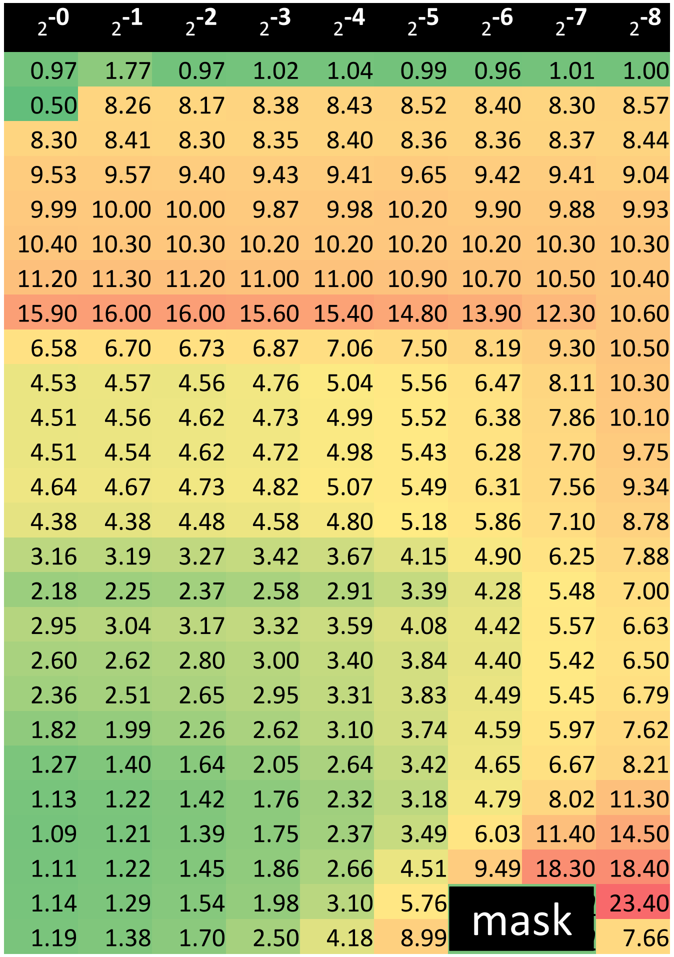
Benchmark II: LOCALITY

System Size = 2^{25} **Without**/With Local Allocators

sf = 5

Diffusion

Physical Locality



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.77	0.97	1.02	1.04	0.99	0.96	1.01	1.00
24	0.50	8.26	8.17	8.38	8.43	8.52	8.40	8.30	8.57
23	8.30	8.41	8.30	8.35	8.40	8.36	8.36	8.37	8.44
22	9.53	9.57	9.40	9.43	9.41	9.65	9.42	9.41	9.04
21	9.99	10.00	10.00	9.87	9.98	10.20	9.90	9.88	9.93
20	10.40	10.30	10.30	10.20	10.20	10.20	10.20	10.30	10.30
19	11.20	11.30	11.20	11.00	11.00	10.90	10.70	10.50	10.40
18	15.90	16.00	16.00	15.60	15.40	14.80	13.90	12.30	10.60
17	6.58	6.70	6.73	6.87	7.06	7.50	8.19	9.30	10.50
16	4.53	4.57	4.56	4.76	5.04	5.56	6.47	8.11	10.30
15	4.51	4.56	4.62	4.73	4.99	5.52	6.38	7.86	10.10
14	4.51	4.54	4.62	4.72	4.98	5.43	6.28	7.70	9.75
13	4.64	4.67	4.73	4.82	5.07	5.49	6.31	7.56	9.34
12	4.38	4.38	4.48	4.58	4.80	5.18	5.86	7.10	8.78
11	3.16	3.19	3.27	3.42	3.67	4.15	4.90	6.25	7.88
10	2.18	2.25	2.37	2.58	2.91	3.39	4.28	5.48	7.00
9	2.95	3.04	3.17	3.32	3.59	4.08	4.42	5.57	6.63
8	2.60	2.62	2.80	3.00	3.40	3.84	4.40	5.42	6.50
7	2.36	2.51	2.65	2.95	3.31	3.83	4.49	5.45	6.79
6	1.82	1.99	2.26	2.62	3.10	3.74	4.59	5.97	7.62
5	1.27	1.40	1.64	2.05	2.64	3.42	4.65	6.67	8.21
4	1.13	1.22	1.42	1.76	2.32	3.18	4.79	8.02	11.30
3	1.09	1.21	1.39	1.75	2.37	3.49	6.03	11.40	14.50
2	1.11	1.22	1.45	1.86	2.66	4.51	9.49	18.30	18.40
1	1.14	1.29	1.54	1.98	3.10	5.76	mask		23.40
0	1.19	1.38	1.70	2.50	4.18	8.99			7.66

Degradation Ratio (DR)

Temporal Locality

3. Analyzing the Benchmark Data

Benchmark II: **L**OCALITY

Questions

and/or

Discussion?

3. Analyzing the Benchmark Data

Benchmark III: **U**TILIZATION

I. **Short Running:** Build Up, Use, Tear Down

- Allocation **D**ENSITY and **V**ARIATION in Allocated Sizes

II. **Long Running:** Time-Multiplexed Subsystems

- Access **L**OCALITY – both *Physical* and *Temporal*

III. **Short Running:** Varying Memory Reusability

- Memory **U**TILIZATION

IV. **Multithreaded:** Varying Numbers of Threads

- Allocator **C**ONTENTION

3. Analyzing the Benchmark Data

Benchmark III: **U**TILIZATION

Considerations:

- Investigate memory **U**TILIZATION.
  - Focus on allocation/deallocation costs themselves.
- Access **L**OCALITY should *NOT* dominate results.
  - Write to just first byte of each newly allocated element.
- Identify sub-dimensions of **U** = **M/T**
  - **T** is the total memory allocated.
  - **M** is the maximum concurrently active memory.
  - **S** is the (atomic) memory “chunk” size.

Benchmark III: UTILIZATION

Plan:

- Allocate chunks of memory of size ***S*** until memory of size ***M*** is concurrently allocated.
- Deallocate **L**east **R**ecently **U**sed chunk and reallocate it until the total memory of size ***T*** has been allocated.
- Deallocate the remaining allocated memory (of size ***M***), one chunk (of size ***S***) at a time.
- The results of this experiment are absolute runtimes.
  - The entries in each row, other than AS1, are relative to AS1.

3. Analyzing the Benchmark Data

Benchmark III: **U**TILIZATION

Two more Considerations:

- We chose to omit **AS**s that “winking-out” memory.
  - I.e., we omitted **AS4**, **AS6**, **AS8**, **AS10**, **AS12**, and **AS14**.
  - Virtually all of the memory deallocated individually.
  - (No mathematical possibility of any noticeable effect.)
- We anticipated trouble with *any* use of **monotonic**.
  - Total memory allocated (T) is large (2^{35} bytes).
  - Artificial limit of 2^{12} for pooled memory chunks (**AS6-AS14**).
    - Memory larger than that passes through to the backing allocator.

These
are all
exponents
of **2**.

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{30}

Inputs			**Global**		**Monotonic**		**Multipool**		**Multi<Mono>**	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
30	15	10	0.063s	103	440	435	46	43	46	47
30	16	10	0.069s	102	401	395	42	42	41	45
30	17	10	0.064s	110	435	428	46	44	47	46
30	18	10	0.063s	102	440	434	46	39	54	47
30	19	10	0.063s	104	439	434	51	46	47	47
30	20	10	0.064s	110	433	430	46	42	46	52
30	20	11	0.035s	125	758	747	54	37	49	37
30	20	12	0.022s	101	1216	1206	51	31	52	32
30	20	13	0.013s	60	1985	1961	110	67	1996	1979
30	20	14	0.008s	77	3356	3304	110	58	3276	3314
30	20	15	0.004s	74	5985	6288	60	111	6016	6057

Maximum Active Memory (M)

Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{31}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
31	15	10	0.127s	104	428	434	39	38	39	41
31	16	10	0.123s	102	442	446	42	42	41	40
31	17	10	0.124s	102	439	442	45	45	42	45
31	18	10	0.123s	102	442	447	47	46	41	42
31	19	10	0.123s	107	441	446	42	41	46	43
31	20	10	0.127s	99	431	434	44	42	41	41
31	20	11	0.064s	102	815	824	48	40	52	48
31	20	12	0.038s	93	1369	1387	57	51	47	54
31	20	13	0.021s	102	2368	2392	108	80	2376	2401
31	20	14	0.013s	61	3787	3833	109	67	3797	3844
31	20	15	0.007s	54	6621	6706	112	59	6651	6708

Maximum Active Memory (M)

Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{32}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
32	15	10	0.248s	103	FAIL	FAIL	38	39	38	41
32	16	10	0.248s	102	FAIL	FAIL	38	41	38	39
32	17	10	0.246s	102	FAIL	FAIL	40	39	39	39
32	18	10	0.246s	102	FAIL	FAIL	40	40	39	40
32	19	10	0.246s	102	FAIL	FAIL	40	42	40	40
32	20	10	0.246s	102	FAIL	FAIL	40	41	41	41
32	20	11	0.124s	102	FAIL	FAIL	46	44	41	47
32	20	12	0.062s	102	FAIL	FAIL	44	45	46	56
32	20	13	0.034s	108	FAIL	FAIL	127	110	FAIL	FAIL
32	20	14	0.022s	72	FAIL	FAIL	105	78	FAIL	FAIL
32	20	15	0.015s	87	FAIL	FAIL	99	60	FAIL	FAIL

Maximum Active Memory (M) Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{33}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
33	15	10	0.495s	102	FAIL	FAIL	41	39	39	39
33	16	10	0.493s	102	FAIL	FAIL	38	39	38	41
33	17	10	0.492s	102	FAIL	FAIL	38	41	38	40
33	18	10	0.492s	102	FAIL	FAIL	40	41	39	40
33	19	10	0.492s	102	FAIL	FAIL	40	41	40	41
33	20	10	0.492s	102	FAIL	FAIL	40	40	40	41
33	20	11	0.248s	102	FAIL	FAIL	42	43	41	42
33	20	12	0.122s	101	FAIL	FAIL	43	47	45	47
33	20	13	0.062s	102	FAIL	FAIL	112	112	FAIL	FAIL
33	20	14	0.040s	89	FAIL	FAIL	96	88	FAIL	FAIL
33	20	15	0.022s	102	FAIL	FAIL	107	80	FAIL	FAIL

Maximum Active Memory (M)

Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{34}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
34	15	10	0.990s	103	FAIL	FAIL	41	39	41	39
34	16	10	0.986s	102	FAIL	FAIL	38	39	38	40
34	17	10	0.985s	102	FAIL	FAIL	38	39	39	40
34	18	10	0.984s	102	FAIL	FAIL	40	40	39	40
34	19	10	0.983s	102	FAIL	FAIL	40	41	40	40
34	20	10	0.984s	102	FAIL	FAIL	40	41	40	41
34	20	11	0.494s	102	FAIL	FAIL	42	42	41	42
34	20	12	0.241s	102	FAIL	FAIL	43	42	47	44
34	20	13	0.120s	107	FAIL	FAIL	114	113	FAIL	FAIL
34	20	14	0.064s	102	FAIL	FAIL	117	112	FAIL	FAIL
34	20	15	0.038s	96	FAIL	FAIL	103	95	FAIL	FAIL

Maximum Active Memory (M)

Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: UTILIZATION

Total Allocated Memory (T) = 2^{35}

Inputs			**Global**		**Monotonic**		**Multipool**		**Multi<Mono>**	
				Virtual		Virtual		Virtual		Virtual
T	**M**	**S**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
35	15	10	1.981s	102	FAIL	FAIL	38	41	39	39
35	16	10	1.975s	102	FAIL	FAIL	39	40	38	39
35	17	10	1.970s	102	FAIL	FAIL	39	40	39	40
35	18	10	1.967s	102	FAIL	FAIL	39	39	39	40
35	19	10	1.967s	102	FAIL	FAIL	39	41	40	41
35	20	10	1.968s	102	FAIL	FAIL	40	41	40	40
35	20	11	0.988s	102	FAIL	FAIL	41	42	41	41
35	20	12	0.481s	102	FAIL	FAIL	42	42	44	44
35	20	13	0.240s	102	FAIL	FAIL	113	113	FAIL	FAIL
35	20	14	0.125s	102	FAIL	FAIL	113	112	FAIL	FAIL
35	20	15	0.070s	94	FAIL	FAIL	103	110	FAIL	FAIL

Maximum Active Memory (M) Size of Each Allocation (S)

3. Analyzing the Benchmark Data

Benchmark III: **U**TILIZATION

Questions and/or Discussion?

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

I. **Short Running:** Build Up, Use, Tear Down

- Allocation **D**ENSITY and **V**ARIATION in Allocated Sizes

II. **Long Running:** Time-Multiplexed Subsystems

- Access **L**OCALITY – both *Physica*l and *Temporal*

III. **Short Running:** Varying Memory Reusability

- Memory **U**TILIZATION

IV. **Multithreaded:** Varying Numbers of Threads

- Allocator **C**ONTENTION

3. Analyzing the Benchmark Data

Benchmark IV: **C**ONTENTION

Considerations:

- Investigate allocator **C**ONTENSION.
    - Focus on allocation/deallocation costs themselves.
- Access **L**OCALITY should *NOT* dominate results.
    - Increment just first byte of each newly allocated element.
- Identify sub-dimensions of **C:**
    - **C**: Expected number of concurrent allocations per thread.
    - **I**: Number of alloc/access/dealloc sequences per thread.
    - **S:** Atomic memory “chunk” size (in bytes).
    - **W**: Number of active threads.

3. Analyzing the Benchmark Data

Benchmark IV: **C**ONTENTION

Plan:

- For each of ***W*** threads:
  - Start the thread.
- For a total of **I** iterations.
  - Allocate a chunk of memory of size ***S*** bytes.
  - Increment the first byte of the allocated memory.
  - Deallocate the chunk of memory (of size ***S***).
  - Join all threads.
- The results of this experiment are absolute runtimes.
  - The entries in each row, other than AS1, are relative to AS1.

3. Analyzing the Benchmark Data

Benchmark IV: **C**ONTENTION

Additional Considerations:

- The allocation **D**ENSITY (**D**) is very high.
- Each thread has access to its own *private unsynchronized* allocator.
- No contention occurs unless the local allocator goes to its backing (e.g., global) allocator.
- Unlike other targeted benchmarks, this experiment doesn't vary **C**ONTENTION over the range from 0 to 1.
  - **C**ONTENTION was kept high (versus 0 for other benchmarks).
- # processors was greater than the max **W** = **8**.

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{15}

Allocation Size (S) = 2^{6}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
15	6	1	0.041s	91	40	39	26	26	24	24
15	6	2	0.037s	100	42	43	27	26	26	29
15	6	3	0.038s	105	41	43	15	16	17	16
15	6	4	0.032s	93	56	58	31	32	25	24
15	6	5	0.032s	91	46	52	26	23	22	24
15	6	6	0.030s	95	51	53	24	27	26	27
15	6	7	0.033s	96	47	49	23	28	21	26
15	6	8	0.029s	96	71	63	33	30	31	25

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{15}

Allocation Size (S) = 2^{7}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
15	7	1	0.023s	100	114	116	44	47	47	48
15	7	2	0.043s	101	46	69	26	26	26	26
15	7	3	0.041s	103	51	68	25	25	22	25
15	7	4	0.033s	121	78	95	26	19	20	23
15	7	5	0.031s	102	81	86	20	26	26	25
15	7	6	0.032s	99	84	84	18	23	19	25
15	7	7	0.029s	114	111	110	23	27	21	31
15	7	8	0.029s	117	114	120	27	35	31	29

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{15}

Allocation Size (S) = 2^{8}

Inputs			**Global**		**Monotonic**		**Multipool**		**Multi<Mono>**	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
15	8	1	0.043s	101	87	89	23	23	22	23
15	8	2	0.042s	102	61	59	23	23	27	26
15	8	3	0.046s	90	85	111	23	25	24	25
15	8	4	0.040s	84	100	98	18	18	19	22
15	8	5	0.028s	136	190	200	30	30	30	38
15	8	6	0.024s	125	209	201	33	33	31	29
15	8	7	0.033s	108	162	162	24	29	26	26
15	8	8	0.031s	114	184	188	34	33	36	42

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{16}

Allocation Size (S) = 2^{8}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
16	8	1	0.085s	97	109	107	23	23	23	23
16	8	2	0.091s	101	104	106	22	21	22	21
16	8	3	0.093s	100	105	104	22	21	21	21
16	8	4	0.097s	94	93	121	20	20	18	17
16	8	5	0.078s	118	108	130	24	18	17	18
16	8	6	0.059s	87	138	136	21	26	22	26
16	8	7	0.063s	93	137	135	17	27	21	20
16	8	8	0.057s	109	162	164	29	28	28	26

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{17}

Allocation Size (S) = 2^{8}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
17	8	1	0.090s	100	206	206	45	42	42	42
17	8	2	0.179s	101	107	106	22	22	22	22
17	8	3	0.179s	101	104	104	22	23	22	22
17	8	4	0.209s	109	89	70	16	15	11	11
17	8	5	0.177s	100	85	78	12	15	15	15
17	8	6	0.108s	142	147	178	27	28	25	25
17	8	7	0.140s	85	116	132	24	22	22	22
17	8	8	0.118s	100	142	150	22	21	25	26

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{18}

Allocation Size (S) = 2^{8}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
18	8	1	0.177s	109	177	177	45	45	45	46
18	8	2	0.339s	100	95	95	24	24	24	24
18	8	3	0.333s	102	99	95	24	25	24	25
18	8	4	0.304s	98	93	93	24	21	26	21
18	8	5	0.311s	94	97	86	22	24	25	20
18	8	6	0.276s	95	118	122	16	17	18	17
18	8	7	0.297s	79	109	108	18	18	21	18
18	8	8	0.219s	114	176	186	26	21	21	23

Number of Active Threads (W)

Benchmark IV: CONTENTION

Number of Iterations (I) = 2^{19}

Allocation Size (S) = 2^{8}

Inputs			Global		Monotonic		Multipool		Multi<Mono>	
				Virtual		Virtual		Virtual		Virtual
I	**S**	**W**	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
19	8	1	0.421s	89	134	134	28	23	21	25
19	8	2	0.615s	101	93	93	25	26	26	26
19	8	3	0.631s	99	93	93	25	25	25	25
19	8	4	0.565s	107	95	103	28	28	29	28
19	8	5	0.575s	119	106	101	27	28	27	27
19	8	6	0.499s	114	126	113	17	25	28	22
19	8	7	0.558s	100	113	115	18	18	15	16
19	8	8	0.460s	105	149	148	19	21	18	21

Number of Active Threads (W)

3. Analyzing the Benchmark Data

Benchmark III: **U**TILIZATION

Questions
and/or
Discussion?

3. Analyzing the Benchmark Data

Another Use For Local Allocators

- http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206?pgno=1#

The Little Parallel Counter That Couldn't

Consider this sequential code to count the number of odd numbers in a matrix:

If our job is to parallelize existing code, this is just what the doctor ordered: An embarrassingly parallel problem where it should be trivial to achieve linear speedups simply by assigning 1/*P*-th of the independent workload to each of *P* parallel workers. Here's a simple way to do it:

Quick: How well would you expect Example 1 to scale as *P* increases from 1 to the available hardware parallelism on the machine?

```
int odds = 0;
for( int i = 0; i < DIM; ++i )
  for( int j = 0; j < DIM; ++j )
    if( matrix[i*DIM + j] % 2 != 0 )
      ++odds;
```

3. Analyzing the Benchmark Data

Another Use For Local Allocators

```
// Example 1: Simple parallel version (flawed)
//
int result[P];
// Each of P parallel workers processes 1/P-th
// of the data; the p-th worker records its
// partial count in result[p]
for( int p = 0; p < P; ++p )
  pool.run( [&,p] {
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p];
  } );
// Wait for the parallel work to complete…
pool.join();
// Finally, do the sequential "reduction" step
// to combine the results
odds = 0;
for( int p = 0; p < P; ++p )
  odds += result[p];
```

3. Analyzing the Benchmark Data

Another Use For Local Allocators

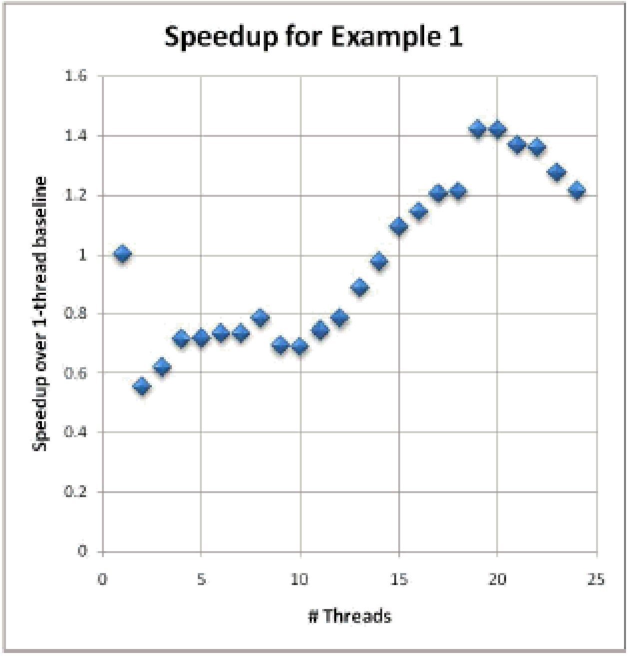


Figure 1: Example 1 seems to be about how to use more cores to get less total work done.

3. Analyzing the Benchmark Data

Another Use For Local Allocators

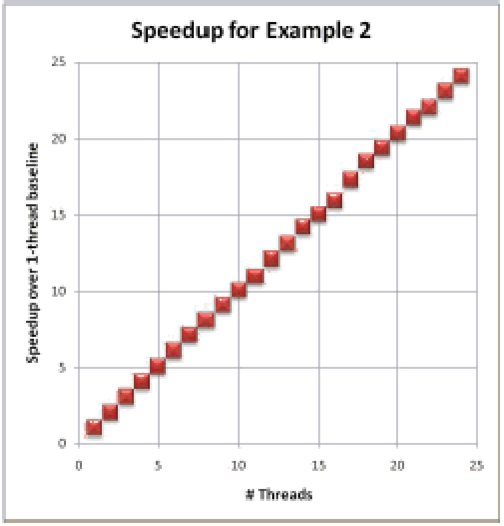
```
// Example 2: Simple parallel version
// (de-flawed using a local variable)
//
int result[P];

// Each of P parallel workers processes 1/P-th
// of the data; the p-th worker records its
// partial count in result[p]
for( int p = 0; p < P; ++p )
  pool.run( [&,p] {
  <font color="#FF0000">int count = 0;</font>
  int chunkSize = DIM/P + 1;
  int myStart = p * chunkSize;
  int myEnd = min( myStart+chunkSize, DIM );
  for( int i = myStart; i < myEnd; ++i )
    for( int j = 0; j < DIM; ++j )
      if( matrix[i*DIM + j] % 2 != 0 )
        <font color="#FF0000">++count;
  result[p] = count;</font>
} );
// … etc. as before …
```

3. Analyzing the Benchmark Data

Another Use For Local Allocators

Figure 4: Removing cache line contention on the result array takes us from zero scaling to linear scaling, up to the available hardware parallelism (test run on a 24-core machine).



Houston, we have a problem!

But there was a problem…

- Some of the original data was unexplainable.
- We needed a strategy to understand why.
- More work was needed.
- We needed to "confirm" the data!
- How we solved the problem (next slide):
  - A revised paper: **Doc No:** P0089R1
  - A new paper: **Doc No:** P0213R0

3. Analyzing the Benchmark Data

The Solution: Graham Bleaney

Enter Graham as Co-Op at Bloomberg (May, 2014).

3. Analyzing the Benchmark Data

The Solution: Graham Bleaney

Enter Graham as Co-Op at Bloomberg (May, 2014).

- From P0089R1:

3. Analyzing the Benchmark Data

The Solution: Graham Bleaney

Enter Graham as Co-Op at Bloomberg (May, 2014).

- From P0089R1:

  "… a separate effort has recently been made to recreate our experiments in order to confirm these results (P0213 by Graham Bleaney). We anticipate that paper will appear at approximately the same time as this revision."

3. Analyzing the Benchmark Data

The Solution: Graham Bleaney

Enter Graham as Co-Op at Bloomberg (May, 2014).

- From P0089R1:

  "… a separate effort has recently been made to recreate our experiments in order to confirm these results (P0213 by Graham Bleaney). We anticipate that paper will appear at approximately the same time as this revision."

- New *allocator-use dimension,* **F**RAGMENTABILITY (**F**):

The Solution: Graham Bleaney

Enter Graham as Co-Op at Bloomberg (May, 2014).

Joins us as FTE (Summer, 2017)!

- From P0089R1:

  "… a separate effort has recently been made to recreate our experiments in order to confirm these results (P0213 by Graham Bleaney). We anticipate that paper will appear at approximately the same time as this revision."

- New *allocator-use dimension,* **F**RAGMENTABILITY (**F**):
  "A measure of the potential of a subsystem's allocated memory to become diffused throughout physical memory, as a result of the interference of other subsystems' memory allocation. If a subsystem is fragmentable (i.e., other subsystems are present in the process and the subsystem allocates more than one chunk of memory), (**F**) is greater than zero."

The C++ Standardization Process

C++ Standards Committee Meeting

- Jacksonville, Florida.
  - February 29 thru March 5, 2016.
- Graham presents his findings to the LEWG.
- Polymorphic Memory Resources (PMR) was adopted into C++17 on March 5, 2016!
  - Along with ***both*** of ***our*** local (“arena”) allocators:
    - **Monotonic**
    - **Multipool**

References

- **References**

- [1] The Bloomberg BDE Library open source distribution, https://github.com/bloomberg/bde
- [2] John Lakos, *Large Scale C++ Software Design*, Addison-Wesley, 1996.
- [3] Pablo Halpern, *N3916: Polymorphic Memory Resources*.
- [4] Memory Allocator Benchmark Data, https://github.com/bloomberg/bde-allocator-benchmarks
- [5] Graham Bleaney, *P0213R0: Reexamining the Performance of Memory-Allocation Strategies.*
- [6] John Lakos, Jeffrey Mendelsohn, Alisdair Meredith, Nathan Myers, *P0089R1: On Quantifying Memory-Allocation Strategies (Revision 2).*

References

- **References**
- [1] The Bloomberg BDE Library open source distribution, https://github.com/bloomberg/bde
- [2] John Lakos, *Large Scale C++ Software Design*, Addison-Wesley, 1996.
- [3] Pablo Halpern, *N3916: Polymorphic Memory Resources*.
- [4] Memory Allocator Benchmark Data, https://github.com/bloomberg/bde-allocator-benchmarks
- [5] Graham Bleaney, *P0213R0: Reexamining the Performance of Memory-Allocation Strategies.*
- [6] John Lakos, Jeffrey Mendelsohn, Alisdair Meredith, Nathan Myers, *P0089R1: On Quantifying Memory-Allocation Strategies (Revision 2).*

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

Outline

1. Introduction and Background

   What are memory allocators, and why are they useful?

2. Understanding the Problem

   What aspects of software affect allocation strategy?

3. Analyzing the Benchmark Data

   When and how do you use which allocator, and why?

4. Conclusions

   What must we remember about memory allocators?

4. Conclusion

Important Recurring Questions

Are memory allocators really worth the trouble?

4. Conclusion

What situations merit their use?

There are a few qualitatively different use cases:

A. To improve and/or preserve performance:

- ➢ Ensure physical locality of allocated memory.
- ➢ Avoid memory diffusion in long-running systems.
- ➢ Obviate deallocation of individual objects.
- ➢ Sidestep contention during concurrent allocations.
- ➢ Separate unrelated data to avoid false sharing.
- ➢ Compose effective allocation strategies.

4. Conclusion

What situations merit their use?

There are a few qualitatively different use cases:

B. To place objects in a specific *kind* of memory:

- Static memory
- Memory-mapped memory
- Read/write protectable memory
- Fast memory (special architectures)
- Shared memory (special allocators)

4. Conclusion

What situations merit their use?

There are a few qualitatively different use cases:

C. To measure, test, control, or debug memory:

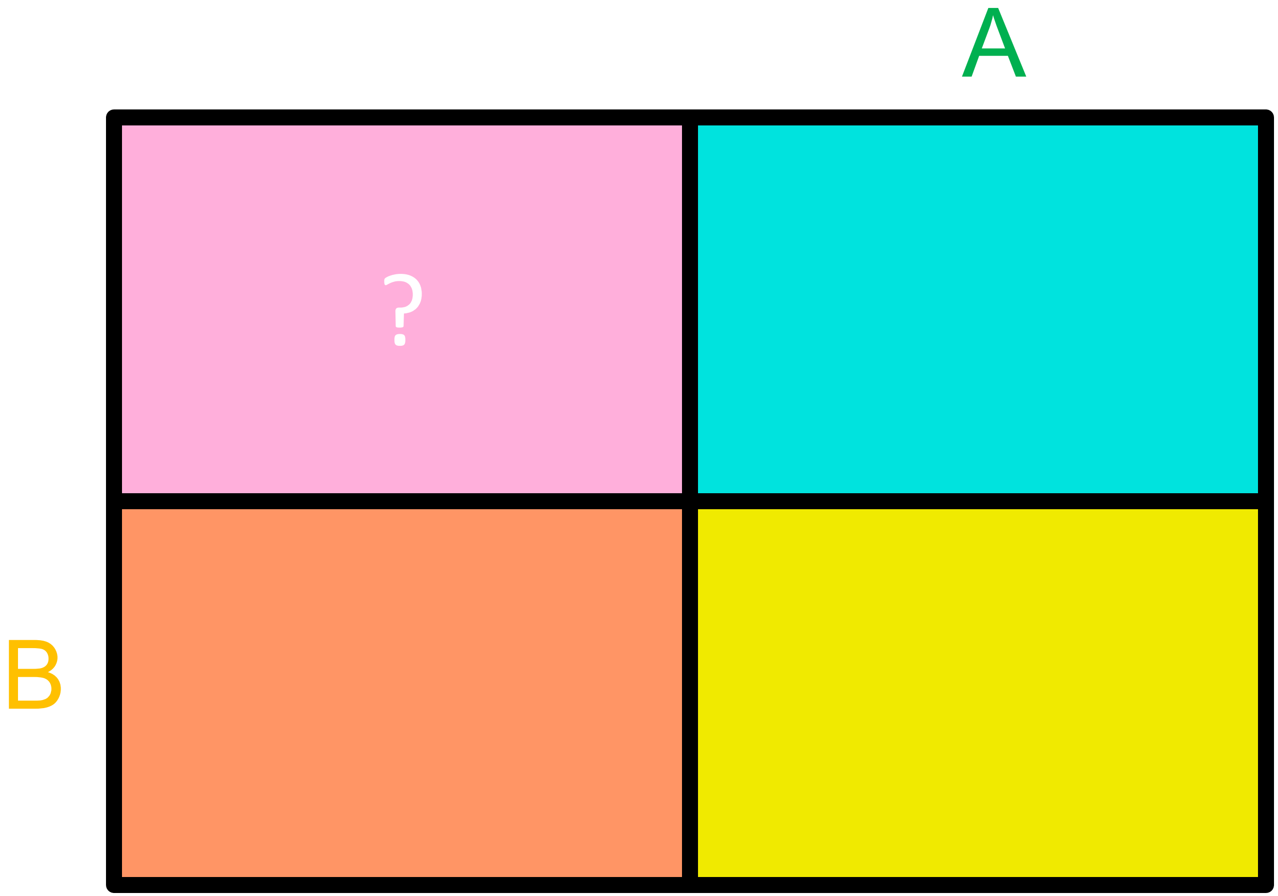
- Counting (auditing) allocator
- Test allocator
- Limit allocator
- Read/write protectable memory allocator

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems are accessed disproportionally often (L).

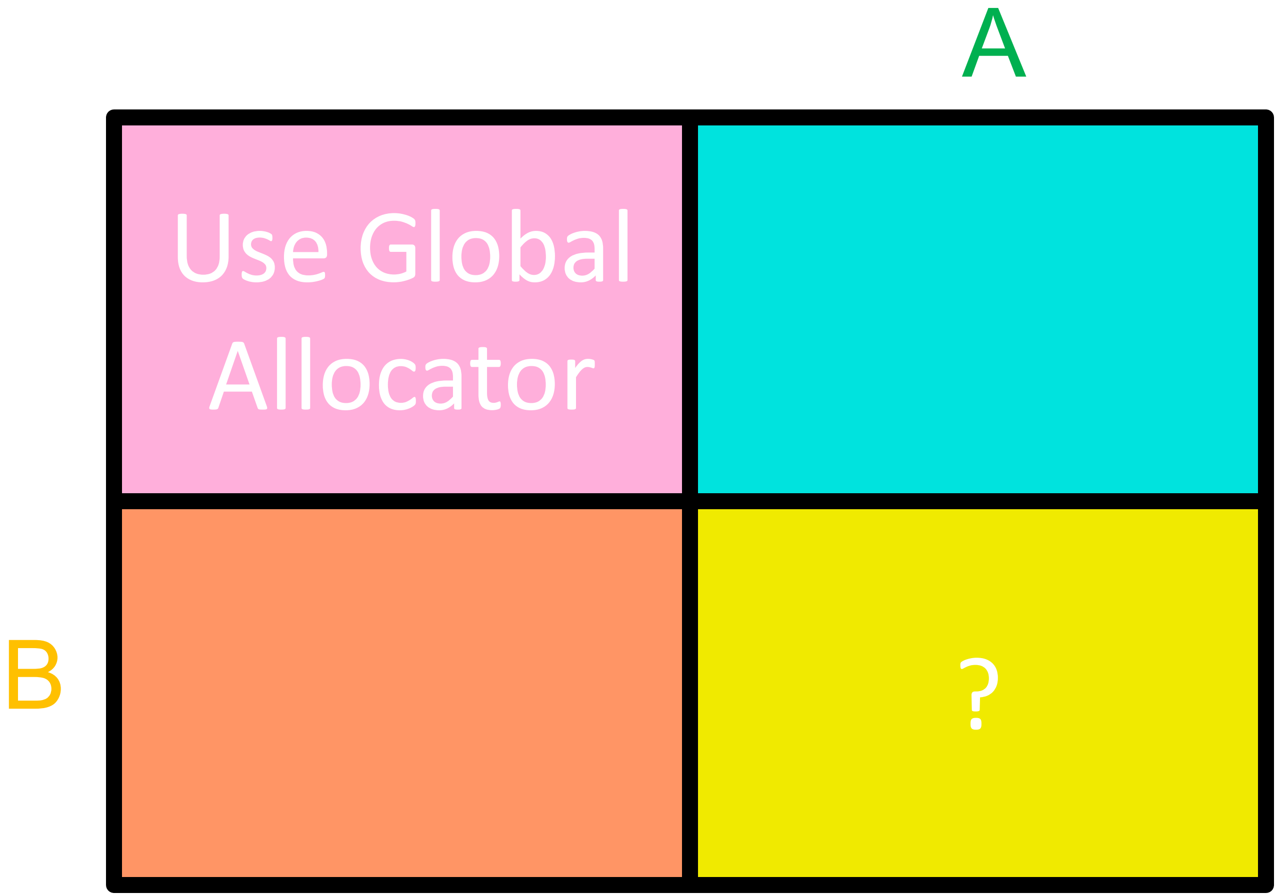


4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems are accessed disproportionally often (L).

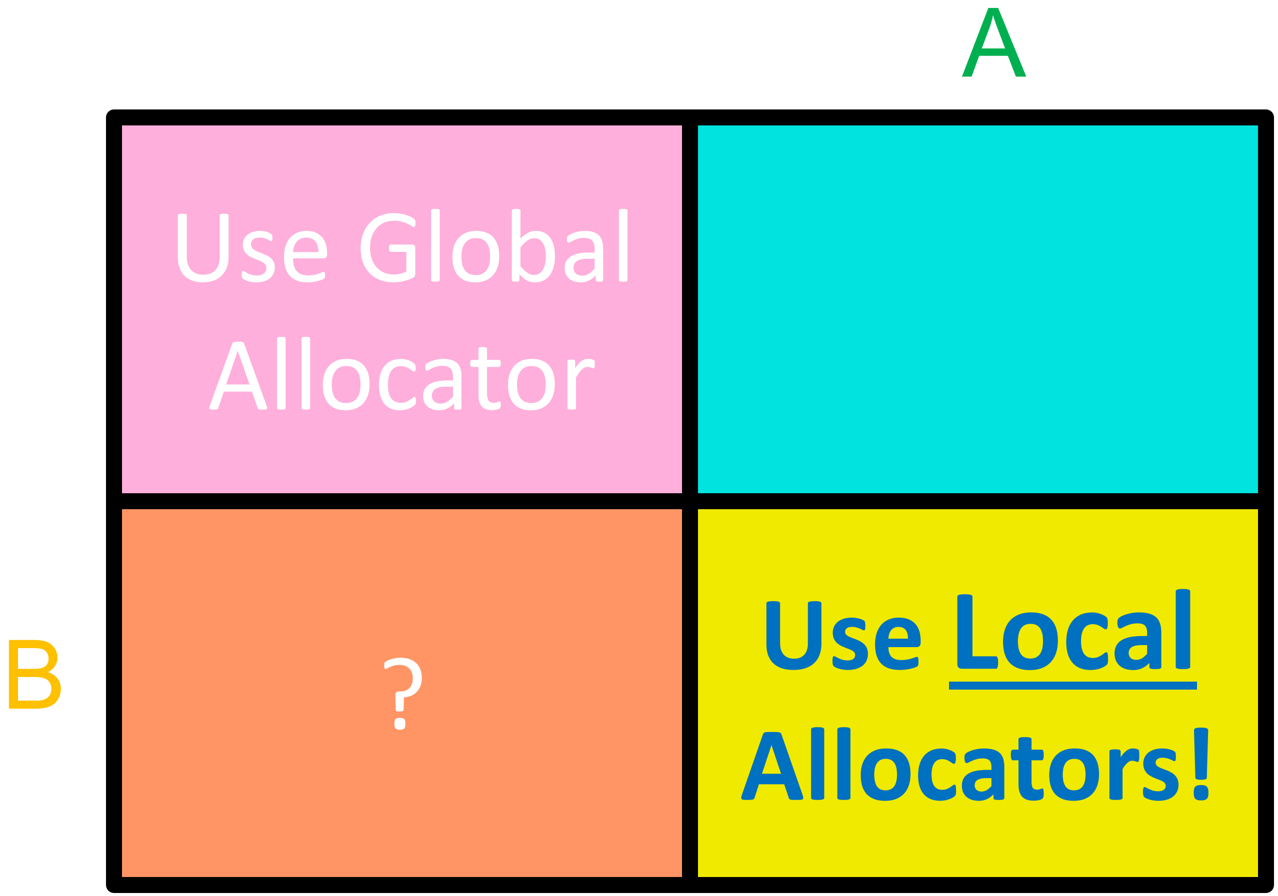


4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems are accessed disproportionally often (L).



		A
	Use Global Allocator	
B	?	**Use Local Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems are accessed disproportionally often (L).

		A
	Use Global Allocator	?
B	Use Local Allocators?	**Use Local Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems are accessed disproportionally often (L).

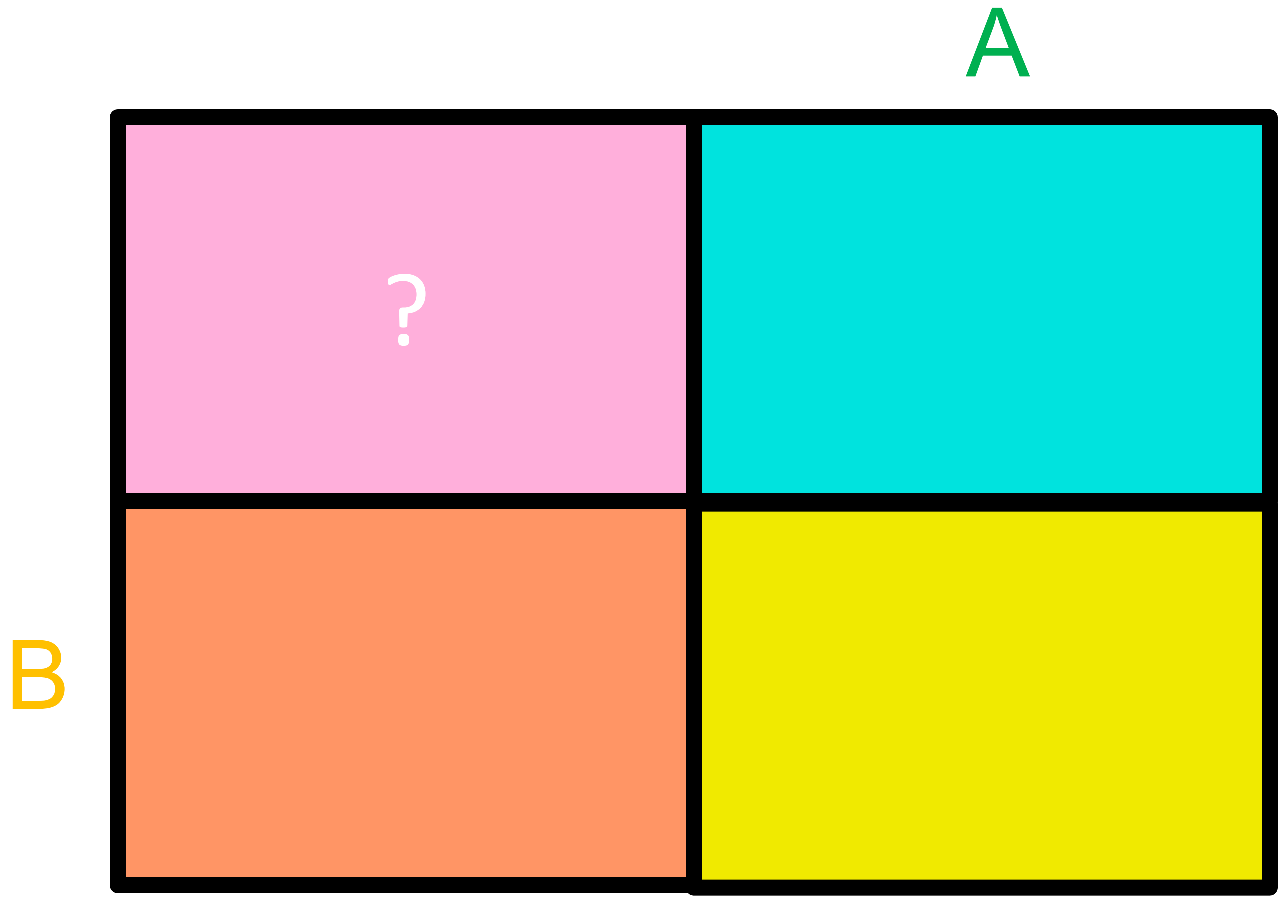
		A
	Use Global Allocator	**Find a new *Job!***
B	Use Local Allocators?	**Use Local Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

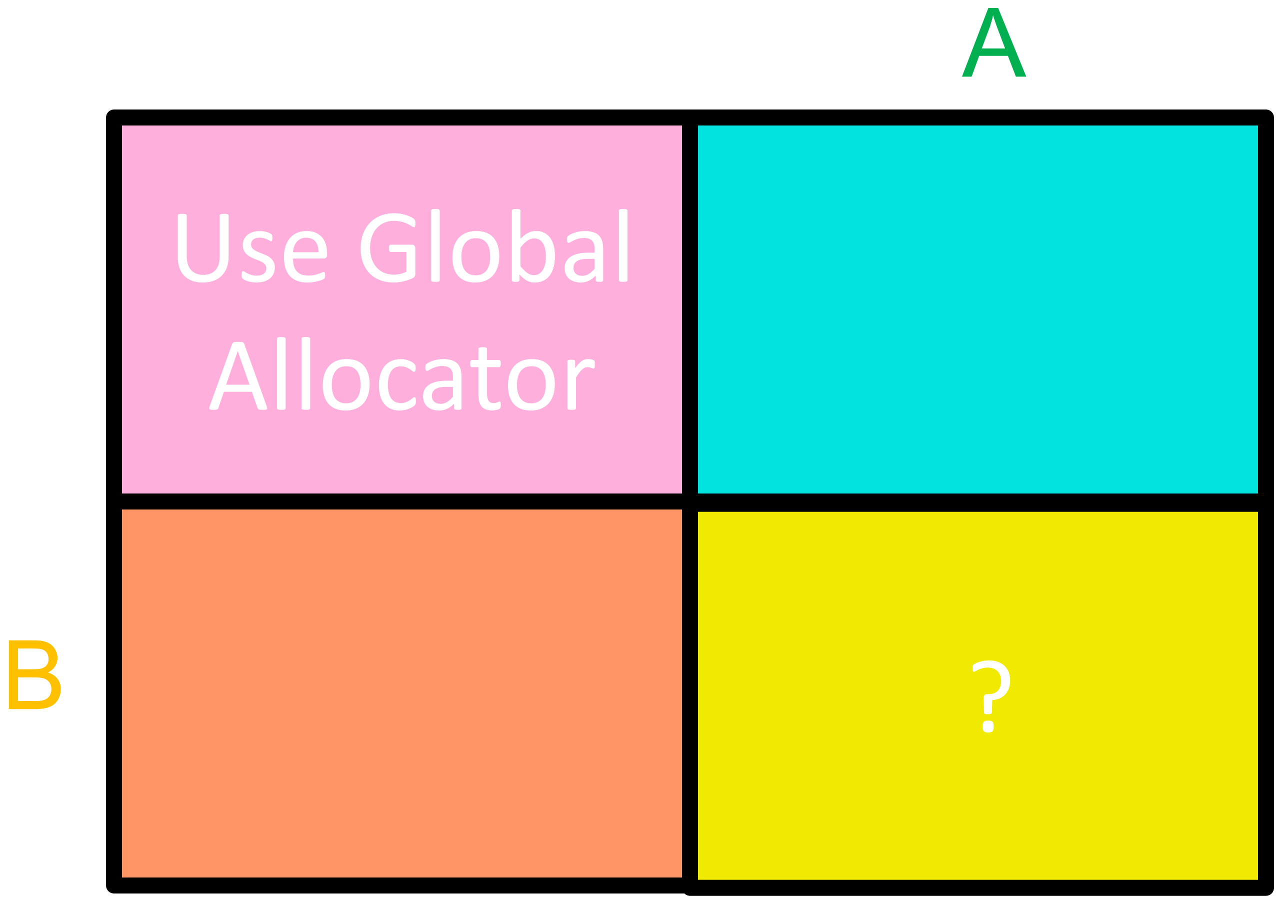
B: Subsystems exhibit high memory utilization (U).



4. Conclusion

How are they applied effectively?

A: Program is large, long running.
B: Subsystems exhibit high memory utilization (U).



A

B

Use Global Allocator

?

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems exhibit high memory utilization (U).

		A
	Use Global Allocator	
B	?	**Use <u>Local</u> *Monotonic* Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems exhibit high memory utilization (U).

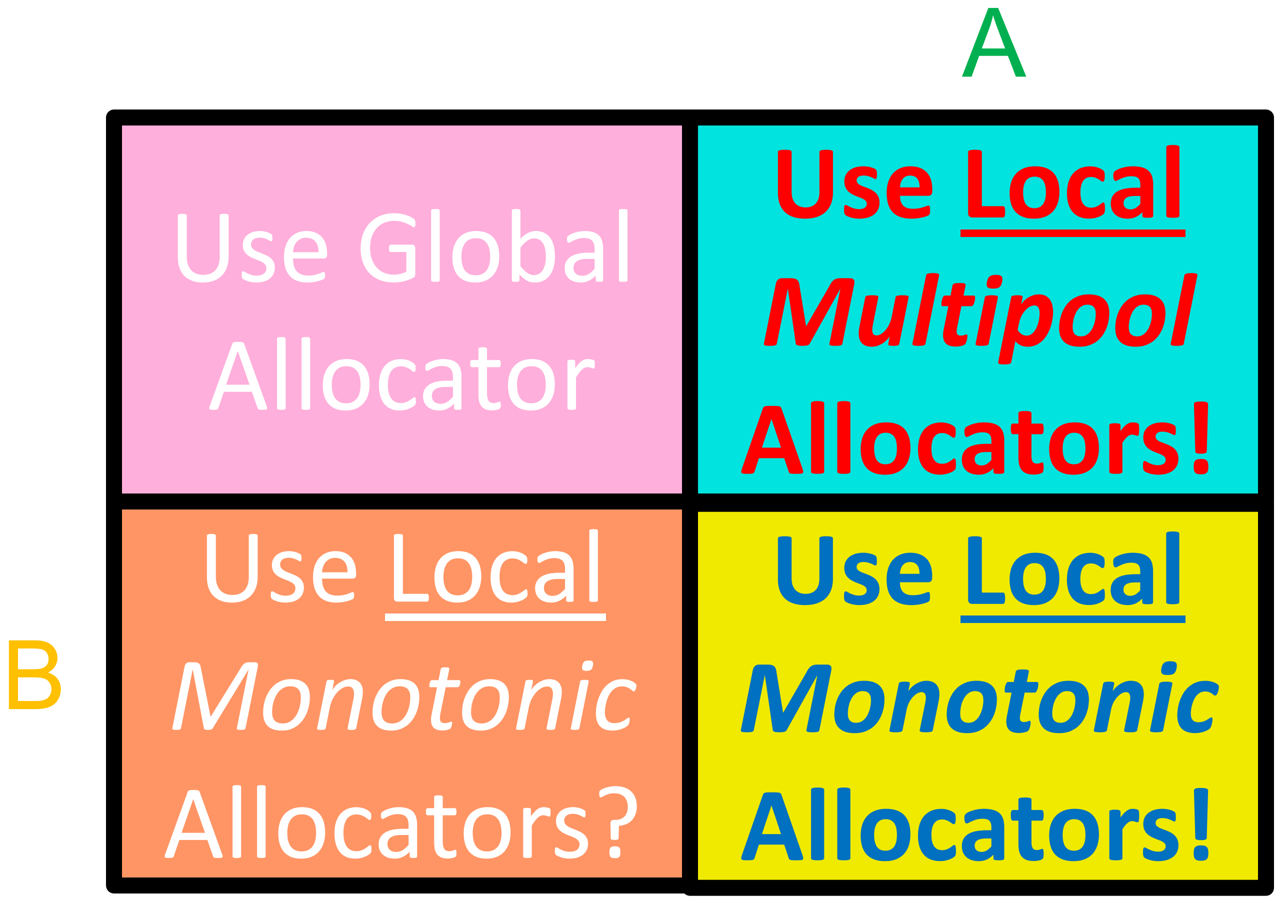
		A
	Use Global Allocator	?
B	Use Local *Monotonic* Allocators?	**Use Local *Monotonic* Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems exhibit high memory utilization (U).



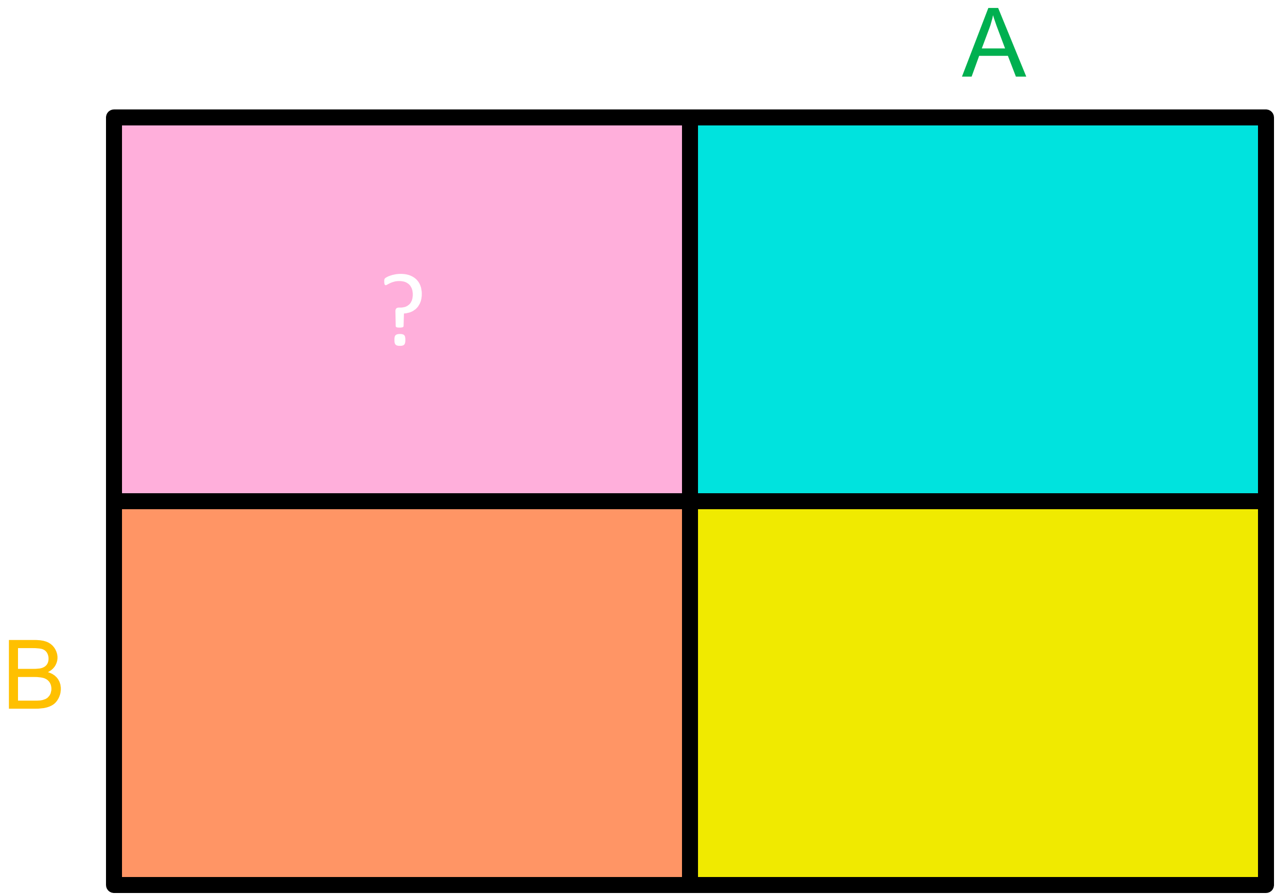
		A
	Use Global Allocator	**Use Local *Multipool* Allocators!**
B	Use Local *Monotonic* Allocators?	**Use Local *Monotonic* Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

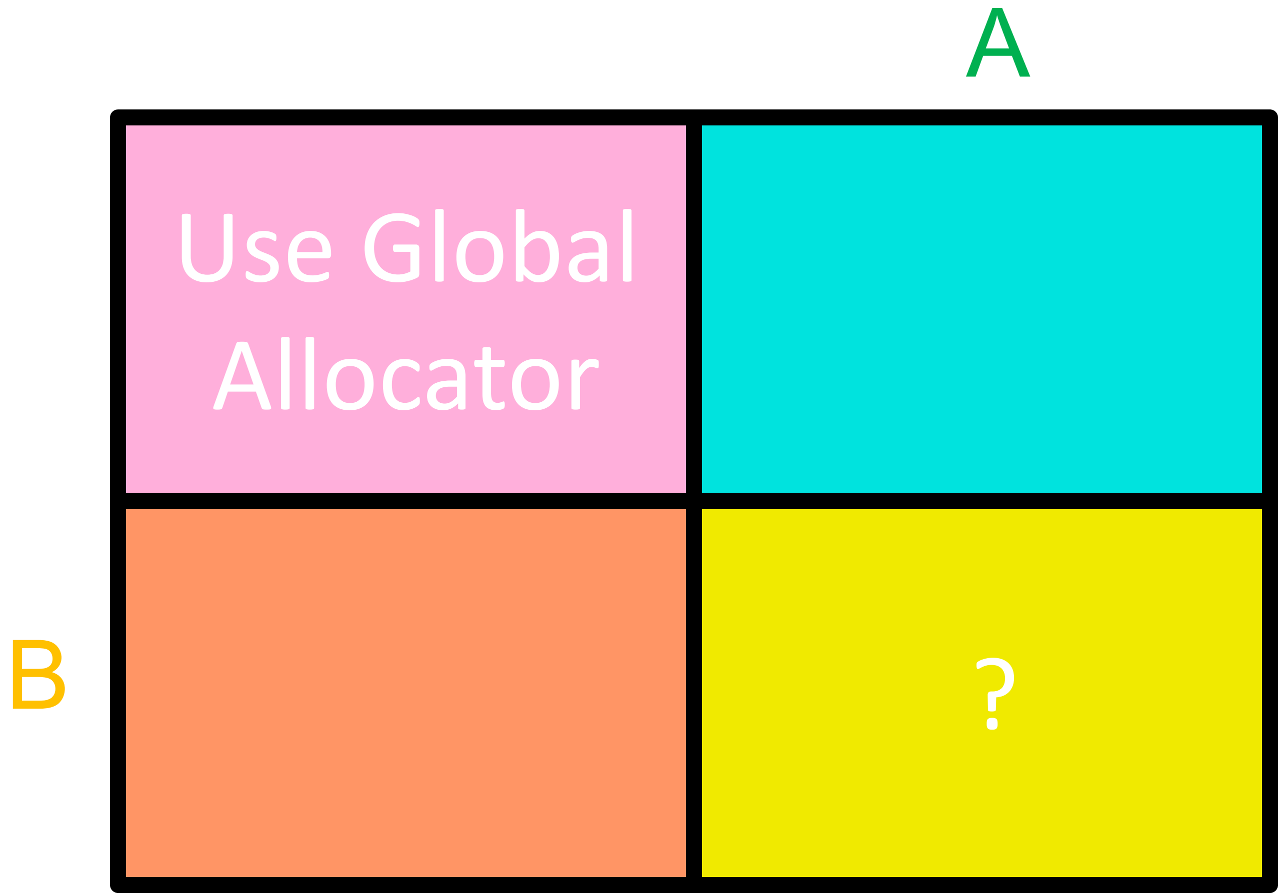
B: Subsystems run concurrently (C).



How are they applied effectively?

A: Program is large, long running.

B: Subsystems run concurrently (C).



4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems run concurrently (C).

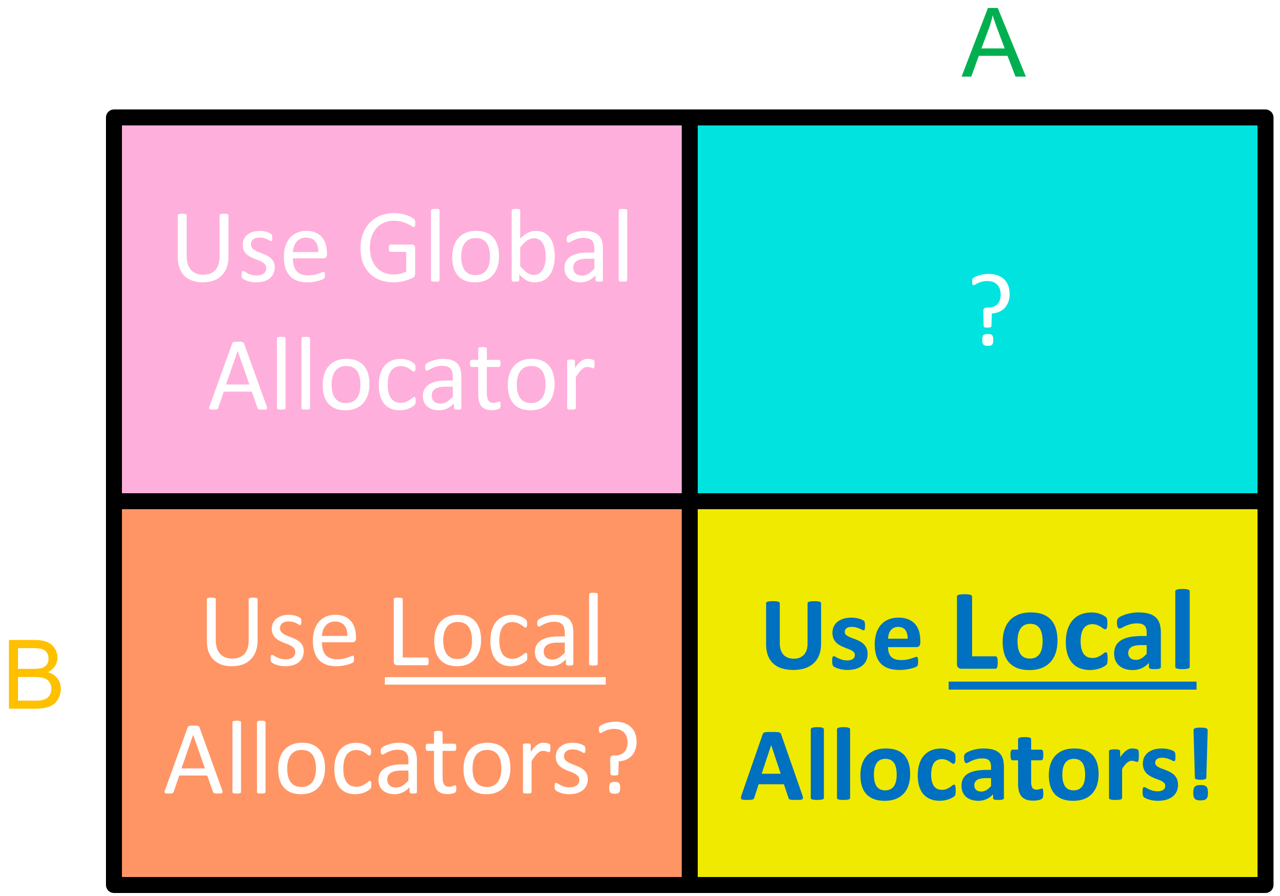
		A
	Use Global Allocator	
B	?	**Use Local Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems run concurrently (C).



		A
	Use Global Allocator	?
B	Use Local Allocators?	**Use Local Allocators!**

4. Conclusion

How are they applied effectively?

A: Program is large, long running.

B: Subsystems run concurrently (C).

		A
	Use Global Allocator	**Use Local Allocators!**
B	Use Local Allocators?	**Use Local Allocators!**

4. Conclusion

What’s the performance impact?

The performance impact can be substantial:

1. In every benchmark, use of some (if not all) of the local allocators performed *no worse*, and typically *much better* than the default allocator.
2. For long-running programs, improvements of as much as an *order of magnitude* were observed.
3. The overhead of using the virtual-function interface and (in the default case) holding an extra address was *minimal to non-existent*.

4. Conclusion

Final Thoughts

Object-level control over memory allocation is intrinsic to C++, and must always be so if we hope to maintain this language’s supremacy as the best-performing high-level “systems” language.

4. Conclusion

Final Thoughts

Object-level control over memory allocation is intrinsic to C++, and must always be so if we hope to maintain this language’s supremacy as the best-performing high-level “systems” language.

Supporting object-specific memory allocation is admittedly an added burden – exacerbated by an initially difficult-to-use model – which is finally being addressed in C++17 by *Polymorphic Memory Resources*.

4. Conclusion

Final Thoughts

Object-level control over memory allocation is intrinsic to C++, and must always be so if we hope to maintain this language's supremacy as the best-performing high-level "systems" language. Supporting object-specific memory allocation is admittedly an added burden – exacerbated by an initially difficult-to-use model – which is finally being addressed in C++17 by *Polymorphic Memory Resources*. Any future incarnation of STL should incorporate the lessons elucidated here.

4. Conclusion

The End

ENSITY ARIATION OCALITY TILIZATION ONTENTION

D V L U C

4. Conclusion

The End

RAGMENTABILITY ENSITY ARIATION OCALITY TILIZATION ONTENTION

F* D V L U C

* Graham Bleaney

4. Conclusion

The End

RAGMENTABILITY ENSITY ARIATION OCALITY TILIZATION ONTENTION

F* D V L U C !

* Graham Bleaney

But don’t mess with the duck!