Vincenzo Innocente

“Software Architectures For Parallel Programming”

Implemented using

- std::thread
- OpenMP
- MPI

Goal of Today

- Learn a methodology to analyze a computational problem and provide a (optimal) parallel solution
- Review structural building blocks
  - Architectural & Design patterns
  - Algorithmic structures
  - Implementations
- Study few use cases
- Identify pitfalls, use measurement tools, apply optimization strategies

Resources

This lectures is largely based on

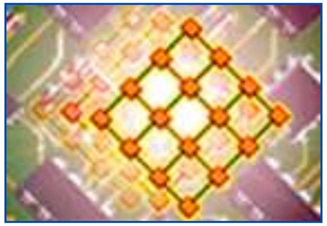
- the excellent 2009 Par Lab Boot Camp – “Short Course on Parallel Programming”

http://parlab.eecs.berkeley.edu/bootcampagenda

- The OpenLab/Intel courses at CERN
- Examples and exercises use the latest C++0x proposed standard as implemented in gcc 4.4.1
  - Not finalized, implementation incomplete and buggy…
  - Very little doc (best: Anthony Williams *ongoing* blog)

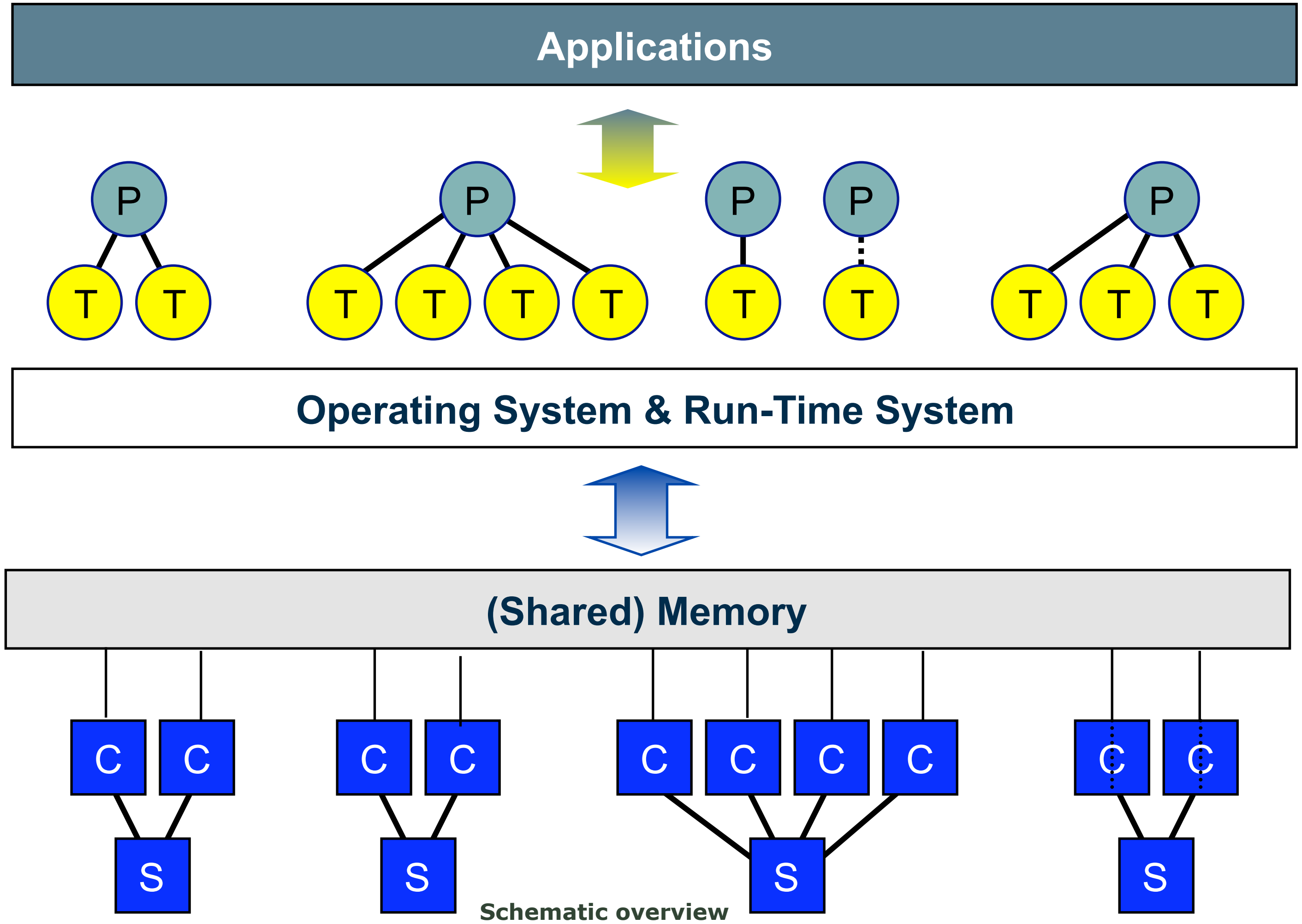
Top-to-Bottom Parallelism

- Cluster/Grid/Cloud
- Multi-socket
- Multi-core
- Hyperthreading
- SIMD/Wide execution
- Pipelining
- Superscalar execution



Our 7 dimensions

Parallel Environments



P: process
T: thread
C: core
S: socket

(Parallel) Software Engineering

Engineering Parallel software follows the "usual" software development process with one difference: **Think Parallel!**

- **Analyze, Find & Design**
  - Analyze problem, Finding and designing parallelism
- **Specify & Implement**
  - How will you express the parallelism (in detail)?
- **Check correctness**
  - How will you determine if the parallelism is right or wrong?
- **Check performance**
  - How will you determine if the parallelism improves over sequential performance?

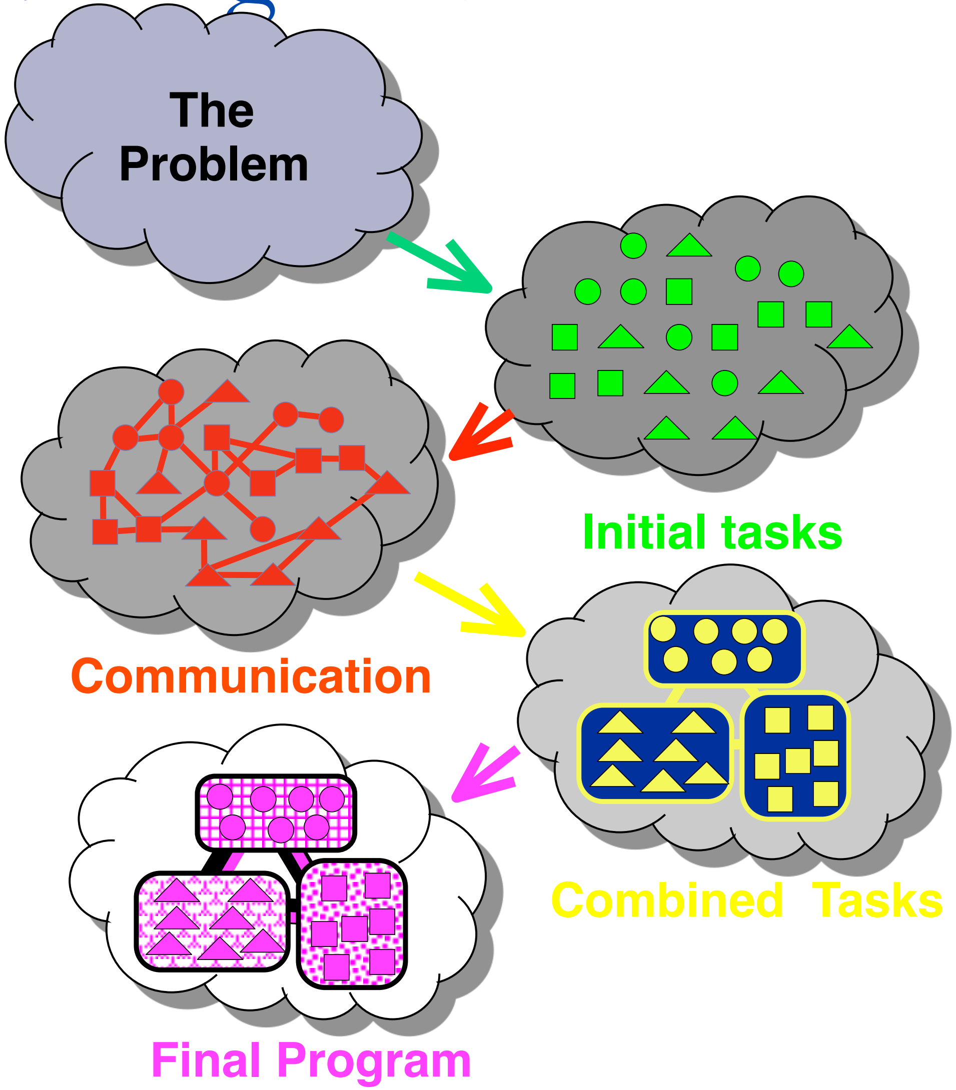
Foster's Design Methodology

- Four Steps:
  - Partitioning
    - Dividing computation and data
  - Communication
    - Sharing data between computations
  - Agglomeration
    - Grouping tasks to improve performance
  - Mapping
    - Assigning tasks to processors/threads

From "*Designing and Building Parallel Programs*" by Ian Foster

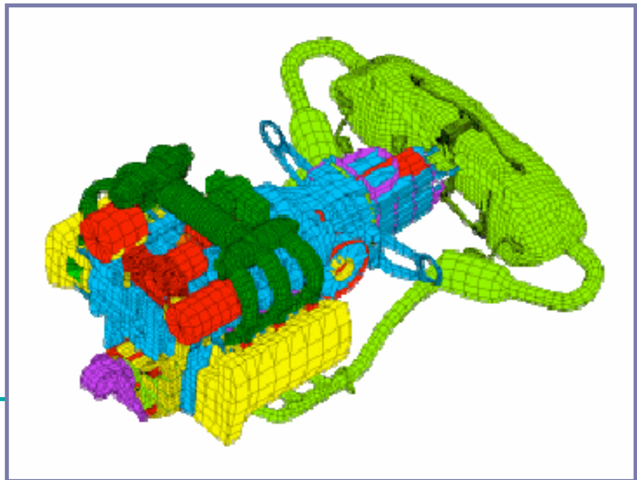
Designing Threaded Programs

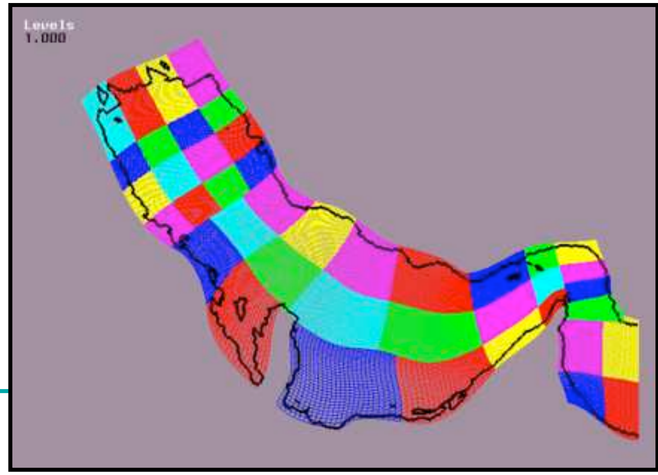
- **Partition**
  - Divide problem into tasks
- **Communicate**
  - Determine amount and pattern of communication
- **Agglomerate**
  - Combine tasks
- **Map**
  - Assign agglomerated tasks to created threads



Domain (Data) Decomposition

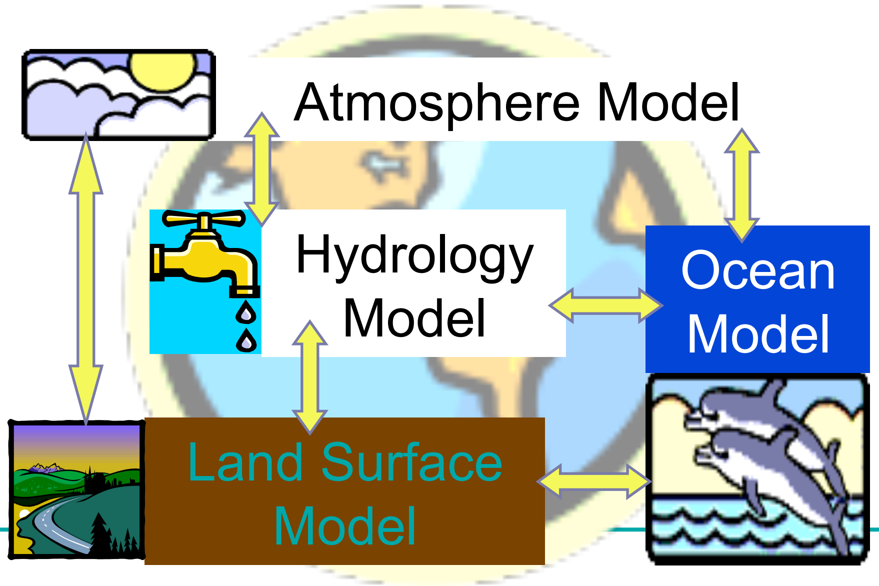
- Exploit large datasets whose elements can be computed independently
  - Divide data and associated computation amongst threads
  - Focus on largest or most frequently accessed data structures
  - Data parallelism: same operations(s) applied to all





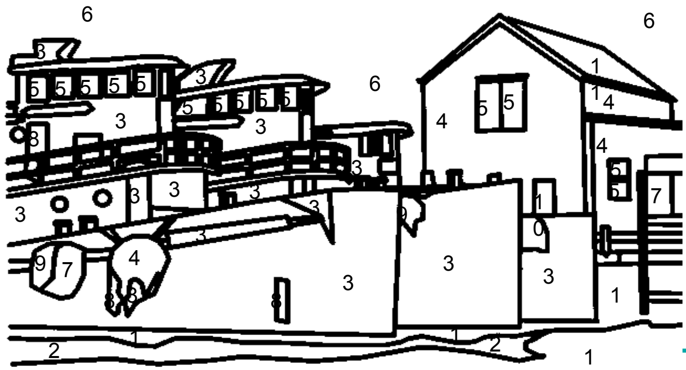
Functional Decomposition

- Divide computation based on a natural set of independent functions
  - Predictable organization and dependencies
  - Assign data for each task as needed
    - Conceptually a single data value or transformation is performed repeatedly



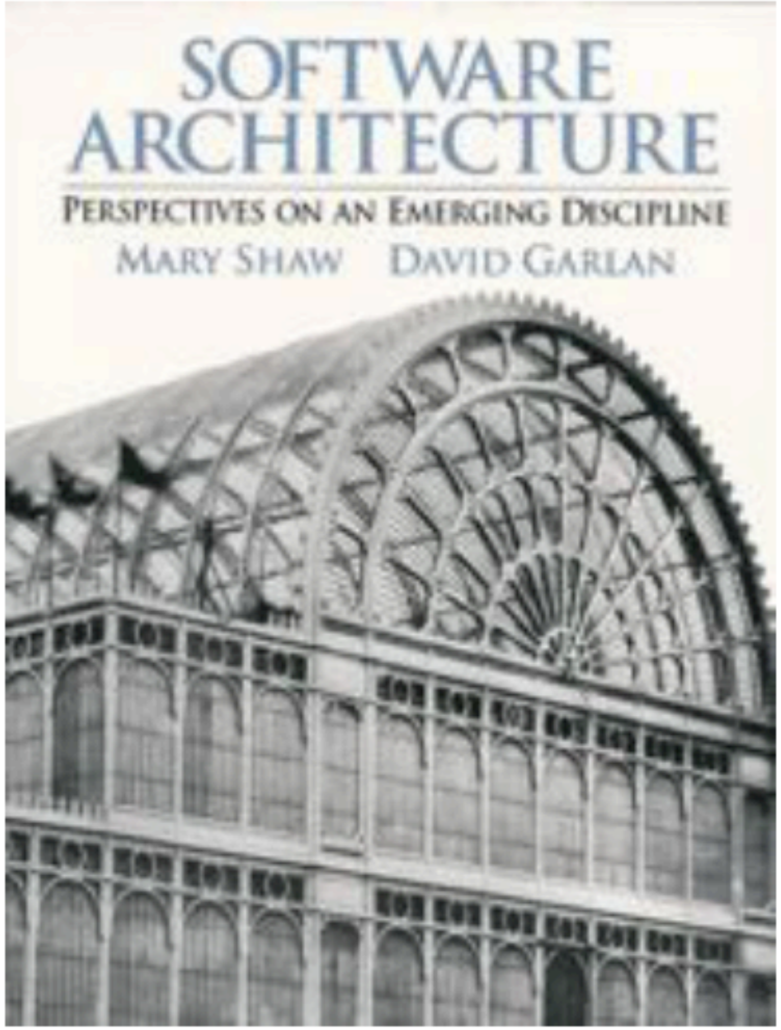
Activity (Task) Decomposition

- Divide computation based on a natural set of independent tasks
  - Non deterministic transformation
  - Assign data for each task as needed
  - Little communication
- Example: Paint-by-numbers
  - Painting a single colour is a single task



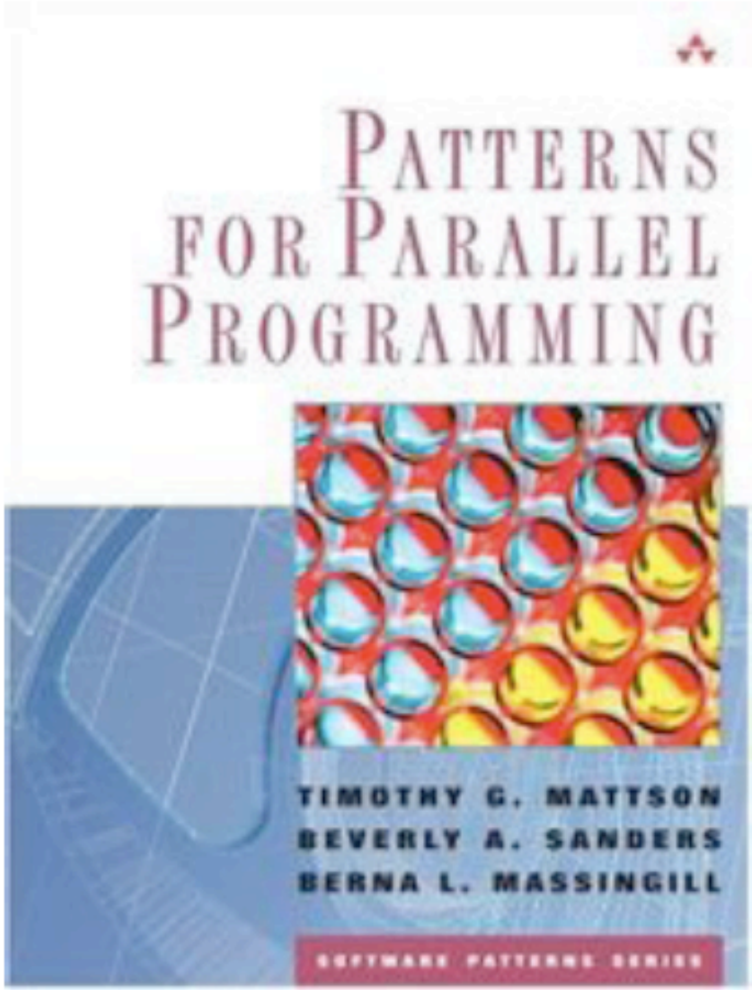
Structural programming patterns

- In order to create more complex software it is necessary to compose programming patterns
- For this purpose, it has been useful to induct a set of patterns known as "architectural styles"
- Examples:
  - pipe and filter
  - event based/event driven
  - layered
  - Agent and repository/ blackboard
  - process control
  - Model-view-controller

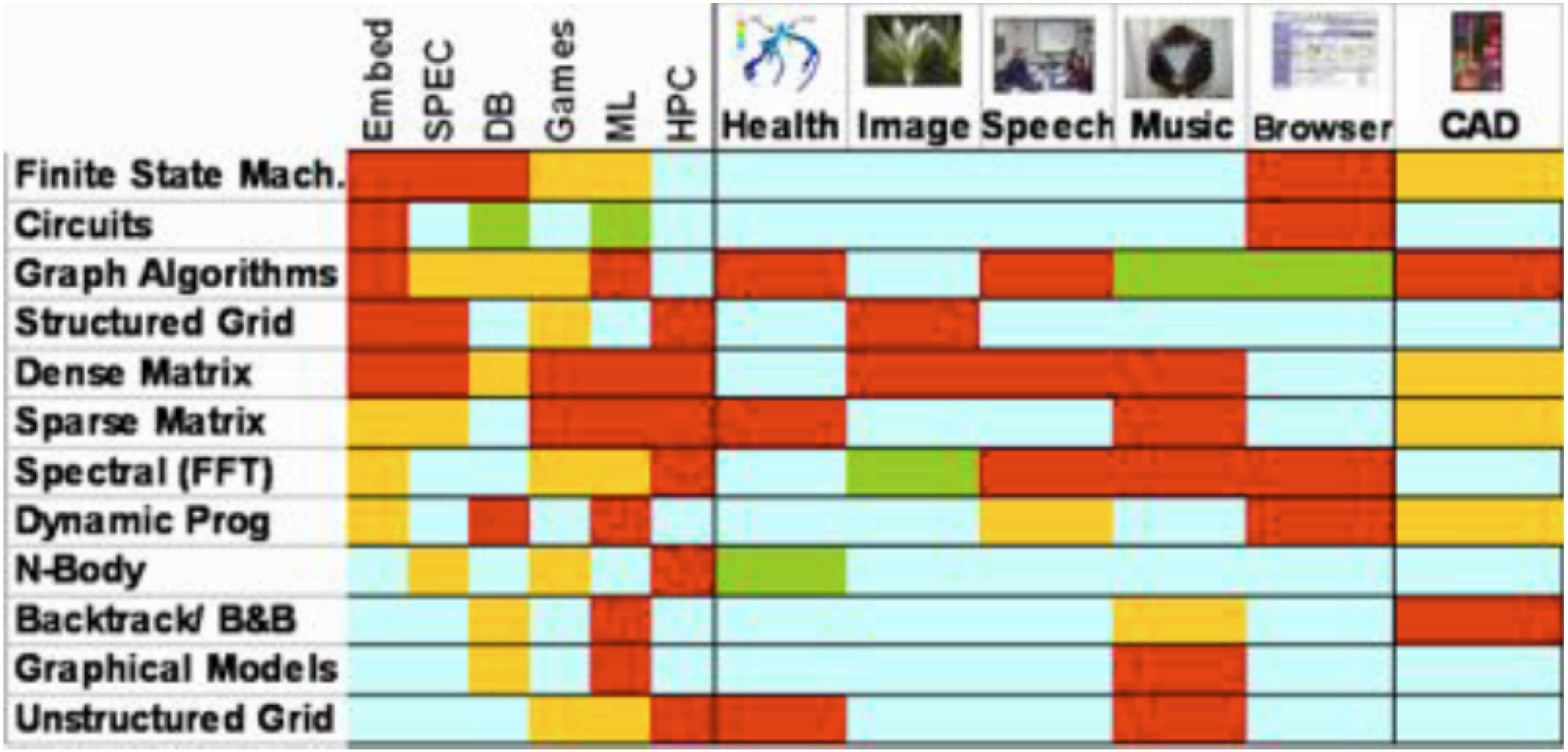


Patterns for Parallel Programming

- PLPP is the first attempt to develop a complete *pattern language* for parallel software development.
- PLPP is a great model for a pattern language for parallel software
- PLPP mined scientific applications that utilize a monolithic application style
- PLPP doesn't help us much with horizontal composition
- Much more useful to us than: *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson & Vlissides, Addison-Wesley, 1995.



Computational Patters



- Computational patterns describe the key computations but not how they are implemented

Our Pattern Language 2.0: Keutzer and Mattson

Applications

Productivity Layer

Choose your high level structure – what is the structure of my application? Guided expansion

- Pipe-and-filter
- Agent and Repository
- Process Control
- Event based, implicit invocation
- Model-view controller
- Iteration
- Map reduce
- Layered systems
- Arbitrary Static Task Graph

Choose you high level architecture? Guided decomposition

Task Decomposition ↔ Data Decomposition

Group Tasks — Order groups — data sharing — data access

Identify the key computational patterns – what are my key computations? Guided instantiation

- Graph Algorithms
- Dynamic Programming
- Dense Linear Algebra
- Sparse Linear Algebra
- Unstructured Grids
- Structured Grids
- Graphical models
- Finite state machines
- Backtrack Branch and Bound
- N-Body methods
- Circuits
- Spectral Methods

Refine the structure – what concurrent approach do I use? Guided re-organization

- Event Based
- Divide and Conquer
- Data Parallelism
- Geometric Decomposition
- Pipeline
- Discrete Event
- Task Parallelism
- Graph algorithms
- Digital Circuits

Efficiency Layer

Utilize Supporting Structures – how do I implement my concurrency? Guided mapping

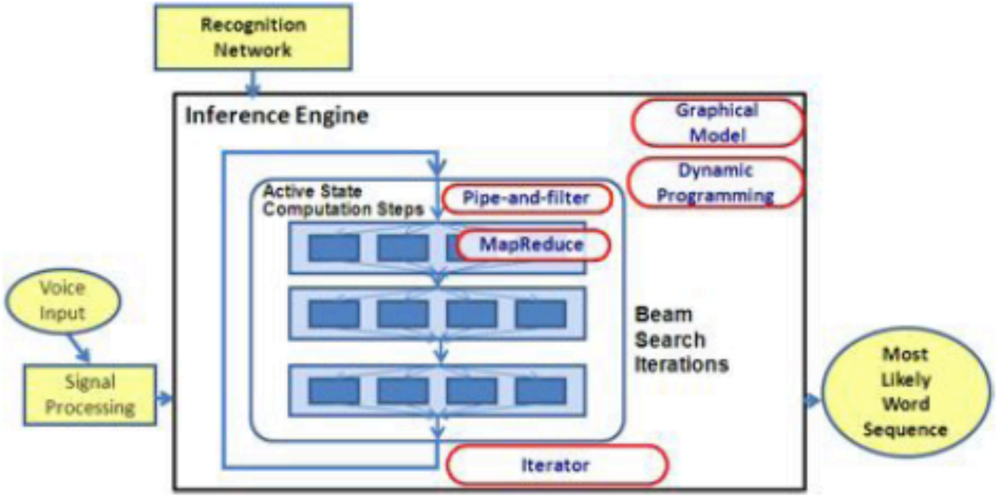
- Fork/Join
- CSP
- Distributed Array
- Shared Data
- Shared Queue
- Shared Hash Table
- SPMD
- Master/worker
- Loop Parallelism
- BSP

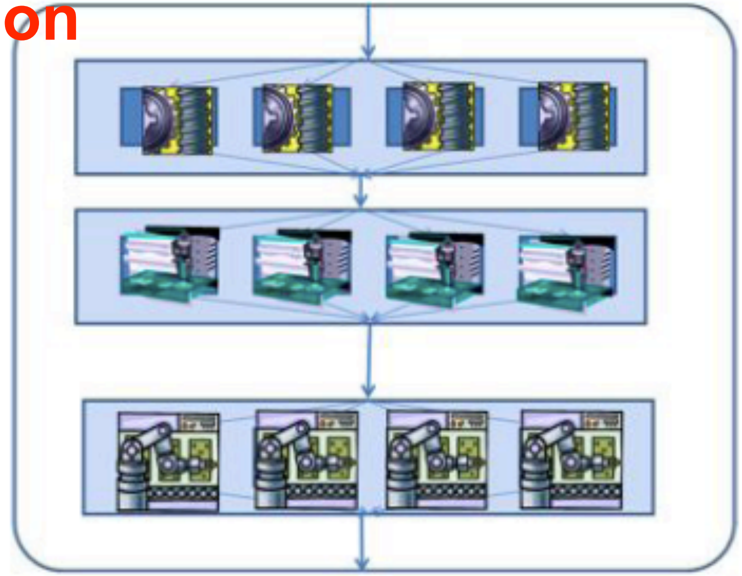
Implementation methods – what are the building blocks of parallel programming? Guided implementation

- Thread Creation/destruction
- Process Creation/destruction
- Message passing
- Collective communication
- Speculation
- Transactional memory
- Barriers
- Mutex
- Semaphores

Architecting the Whole Application

Main Challenge:
Build an architecture that scales and will survive hardware evolution





- SW Architecture of Large-Vocabulary Continuous Speech Recognition

Analogous to the design of an entire manufacturing plant

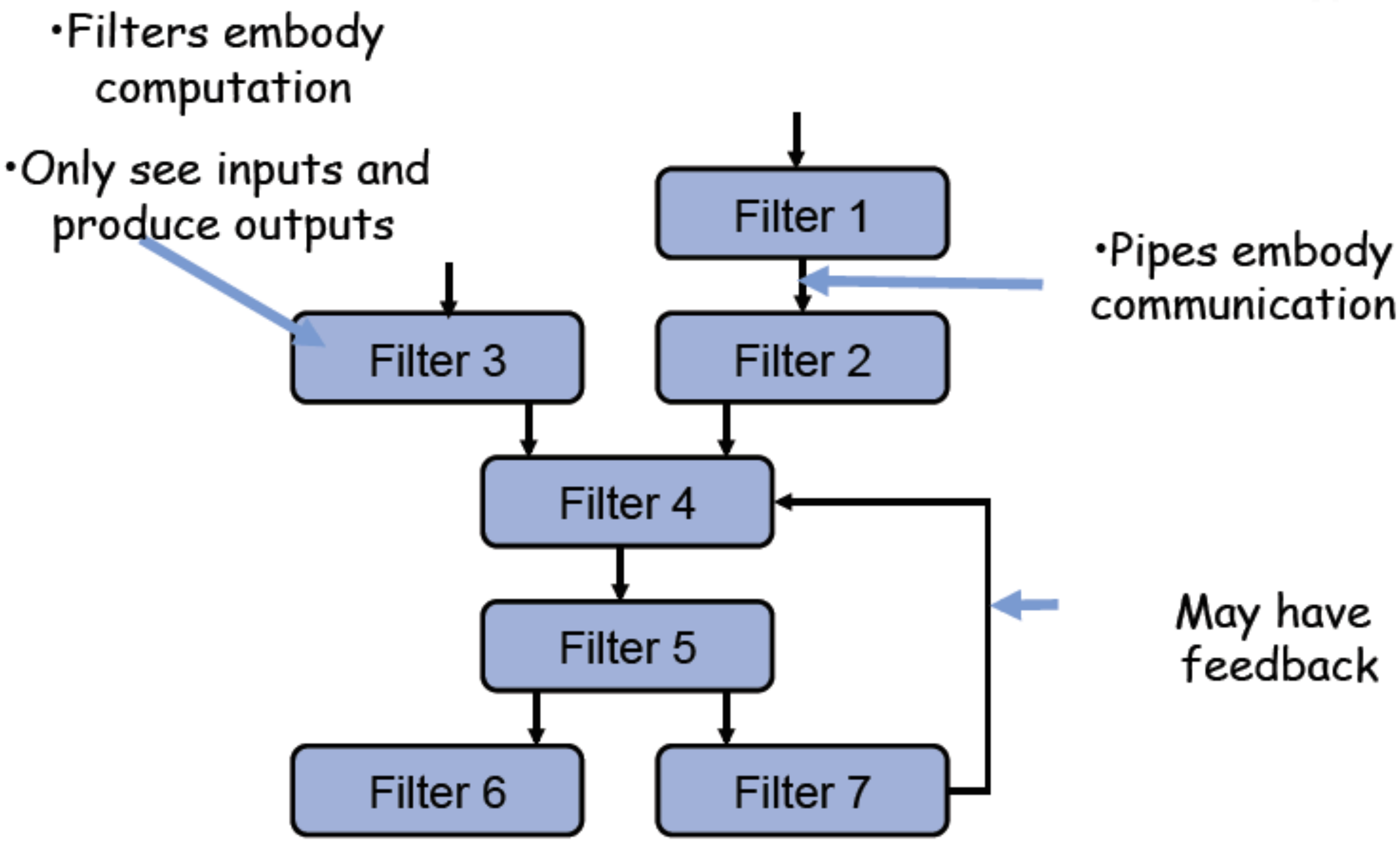
- Raises appropriate issues like scheduling, latency, throughput, workflow, resource management, capacity etc.

31

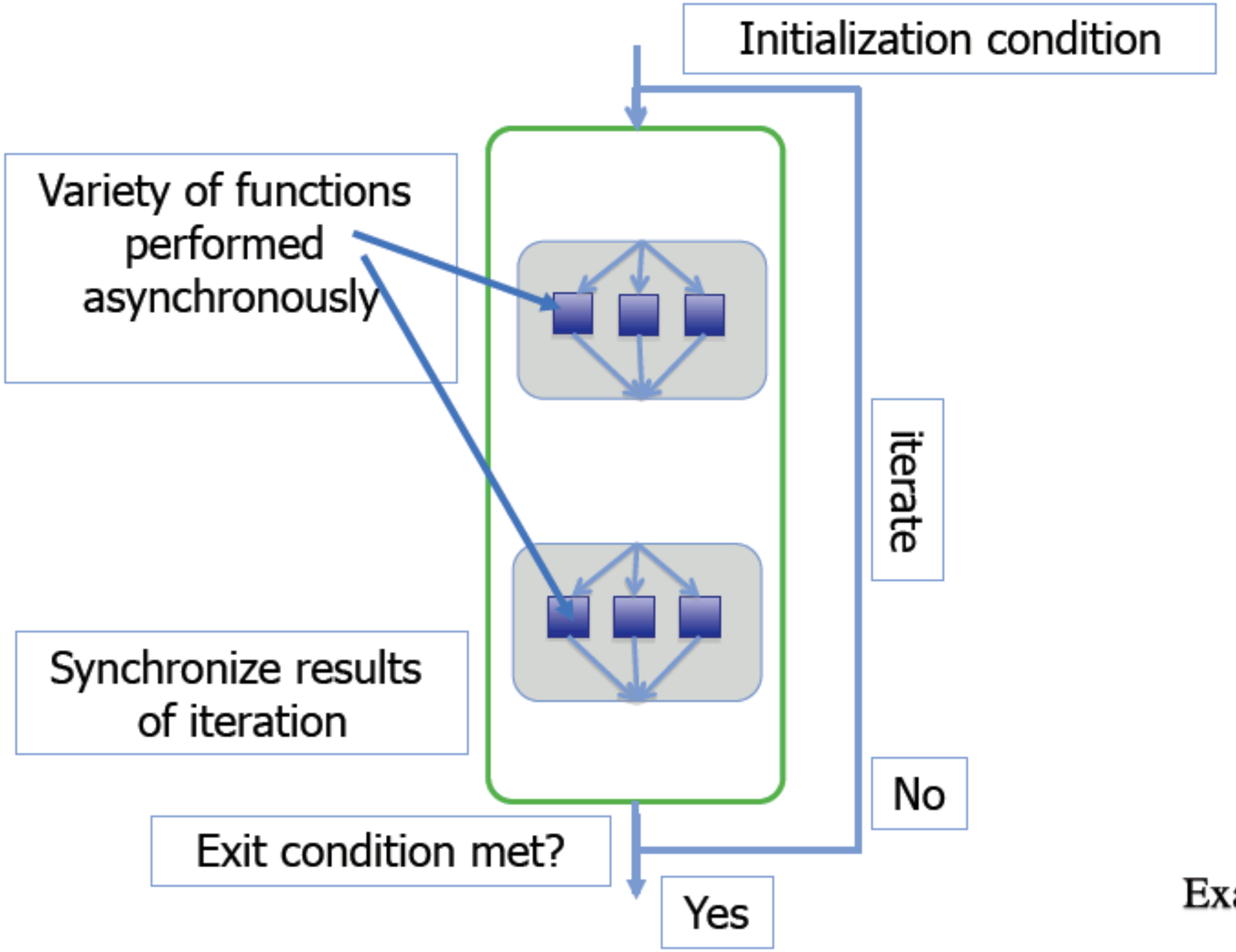
Think Parallel

PATTERNS

Pipes and Filters

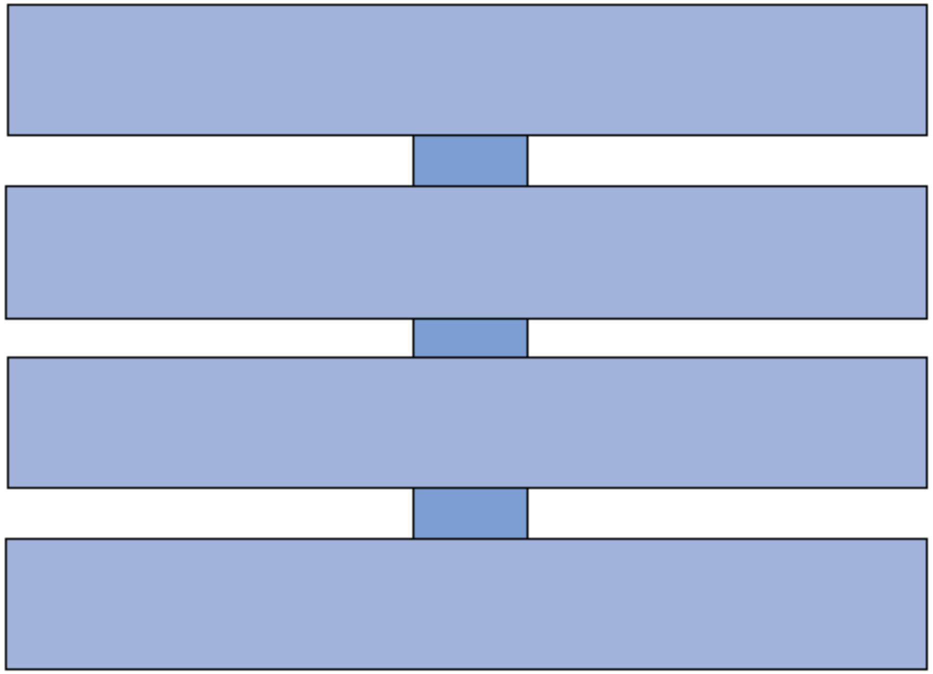


Iteration



Exa

Layered Systems



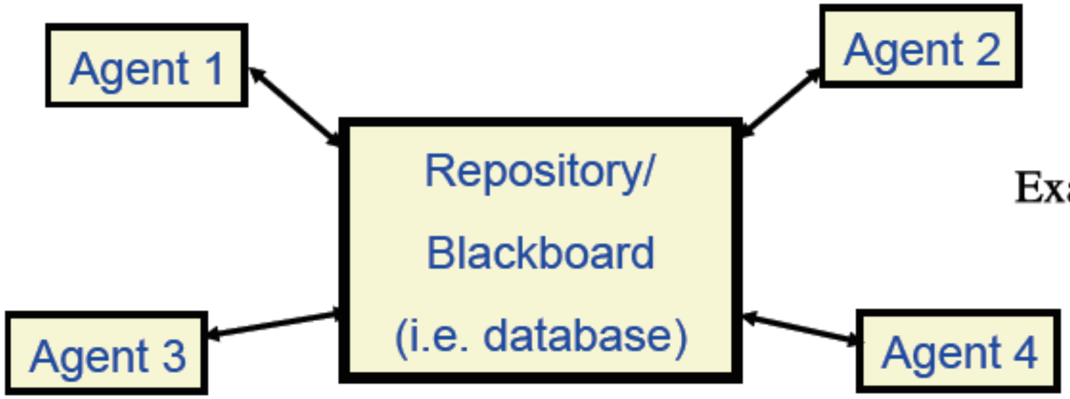
Delegation pattern: Lower layers "work" for the upper ones

- Individual Layers are big
- Interface between two adjacent layer is narrow
- No communication among not adjacent layers

Challenge:

- where parallelization shall occur?
- Often lower layers is legacy software

Agents and Repository



Examples?

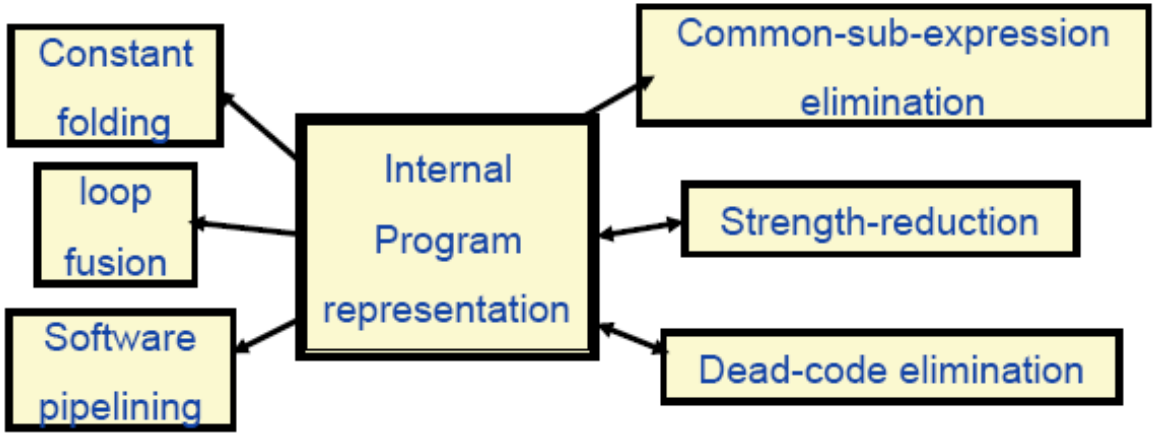
Agent and repository : Blackboard structural pattern

Agents cooperate on a shared medium to produce a result

Key elements:

- Blackboard: repository of the resulting creation that is shared by all agents ~~(circuit database)~~
- Agents: intelligent agents that will act on blackboard ~~(optimizations)~~
- Manager: orchestrates agents access to the blackboard and creation of the aggregate results (scheduler)

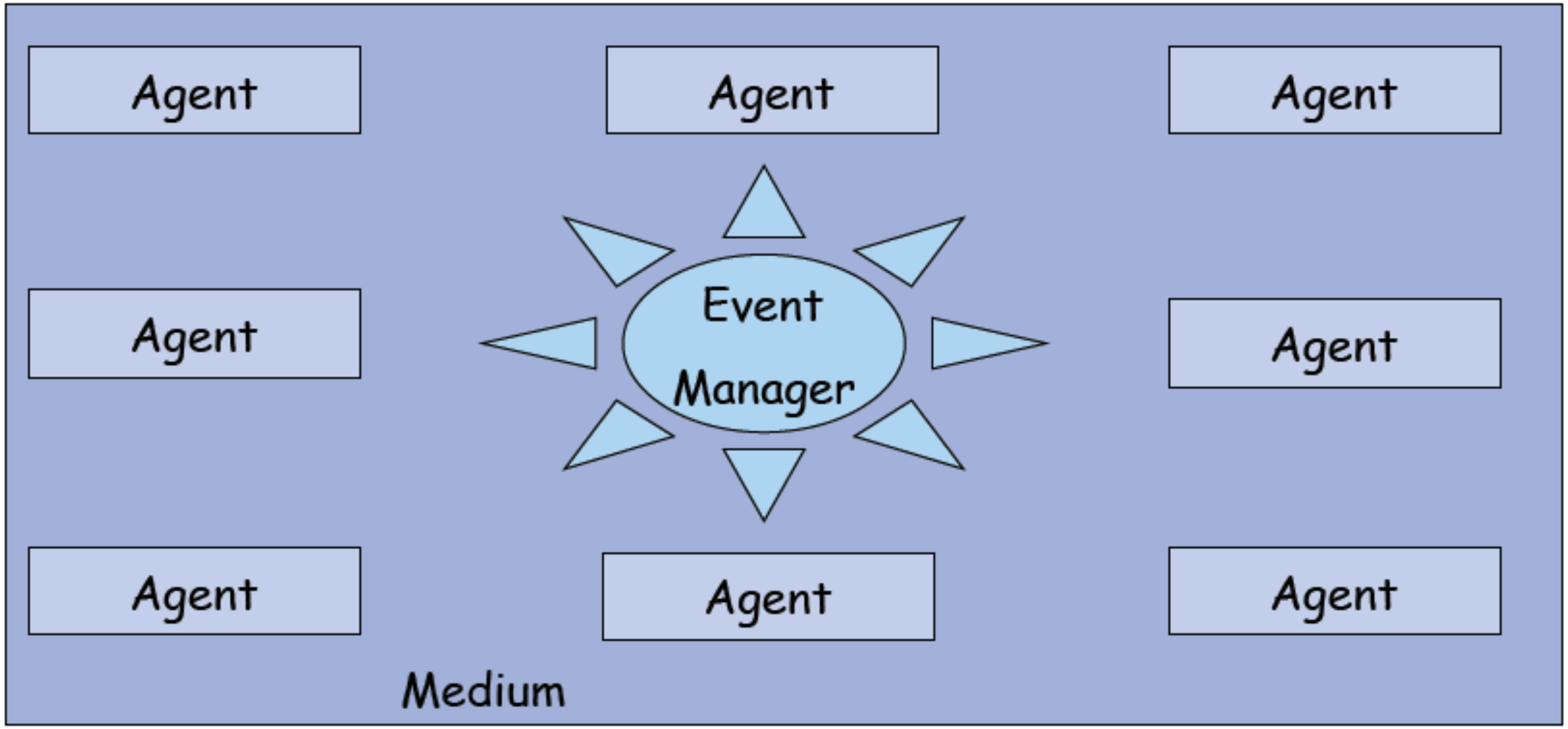
Example: Compiler Optimization



Optimization of a software program

- Intermediate representation of program is stored in the repository
  - Individual agents have heuristics to optimize the program
- Manager orchestrates the access of the optimization agents to the program in the repository
  - Resulting program is left in the repository

Event-Based Systems

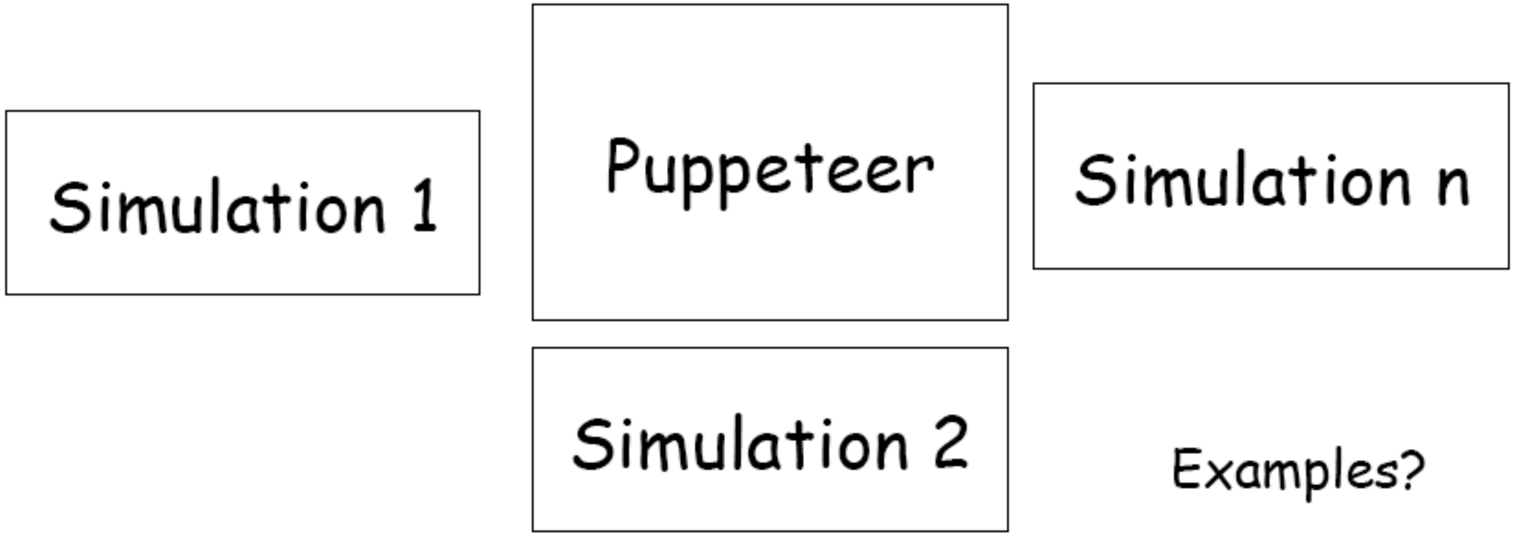


- Agents interact via events/signals in a medium
- Event manager manages events
- Interaction among agents is dynamic - no fixed connection

Examples?

Puppeteer

- Need an efficient way to manage and control the interaction of multiple simulators/computational agents
- **Puppeteer Pattern** – guides the interaction between the simulation codes to guarantee correctness of the overall simulation
- Difference with agent and repository?
    - No central repository
    - Data transfer between simulators



Examples?

Map/Reduce

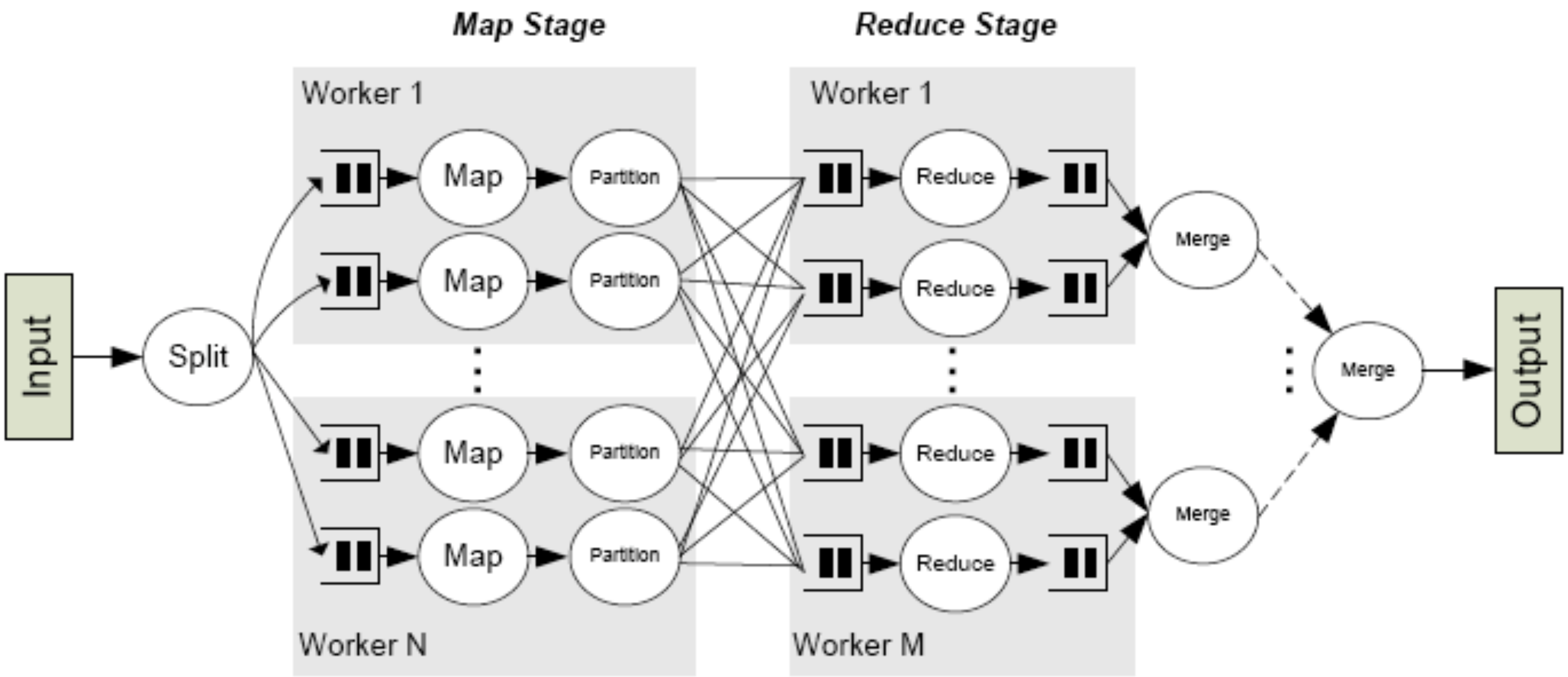
Original (google, Hadhoop) Map/Reduce takes a set of *input key/value pairs, and* produces a set of *output key/value pairs.*

- *Map (written by the user)*
  - *takes an input pair and produces* a set of *intermediate key/value pairs.*
- *The MapReduce* library
  - groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce function.*
- *Reduce , also written by the user,*
  - *accepts* an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

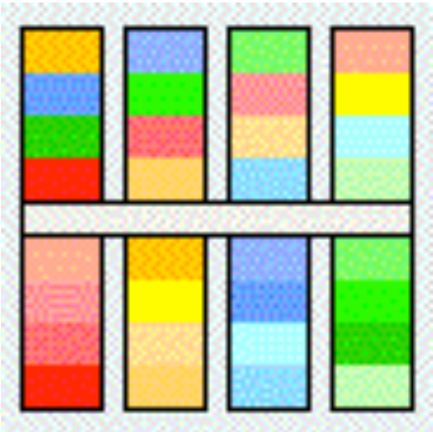
Word count

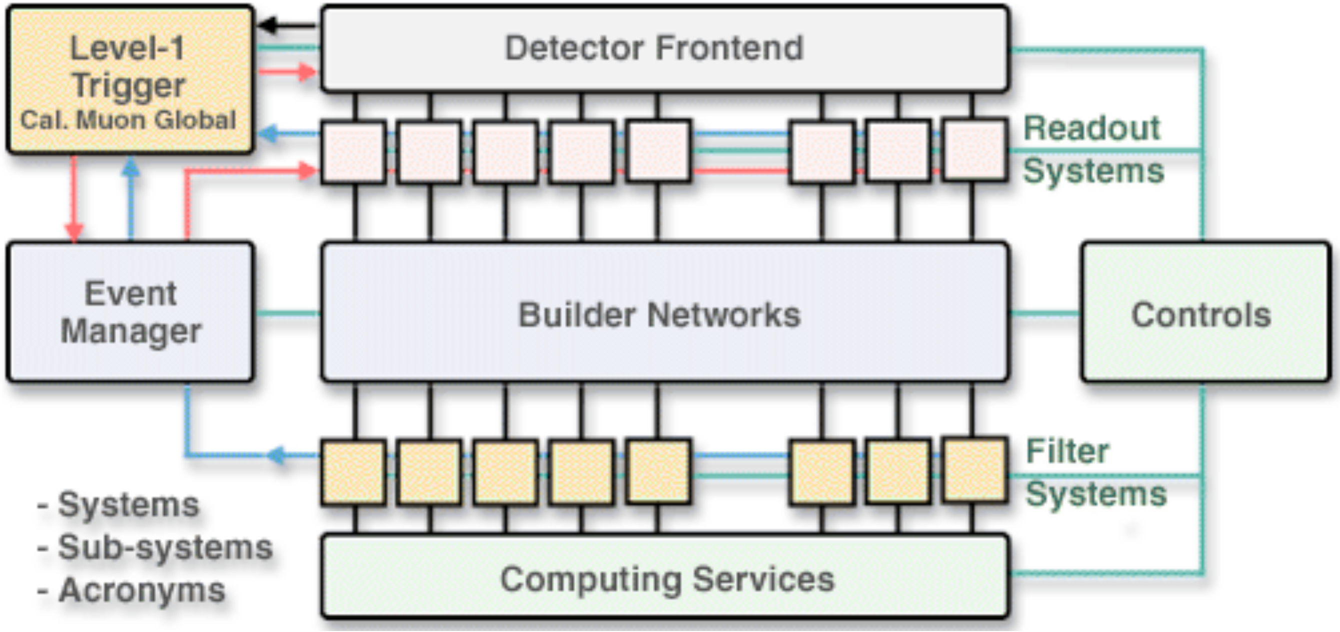
```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```



Event Building!

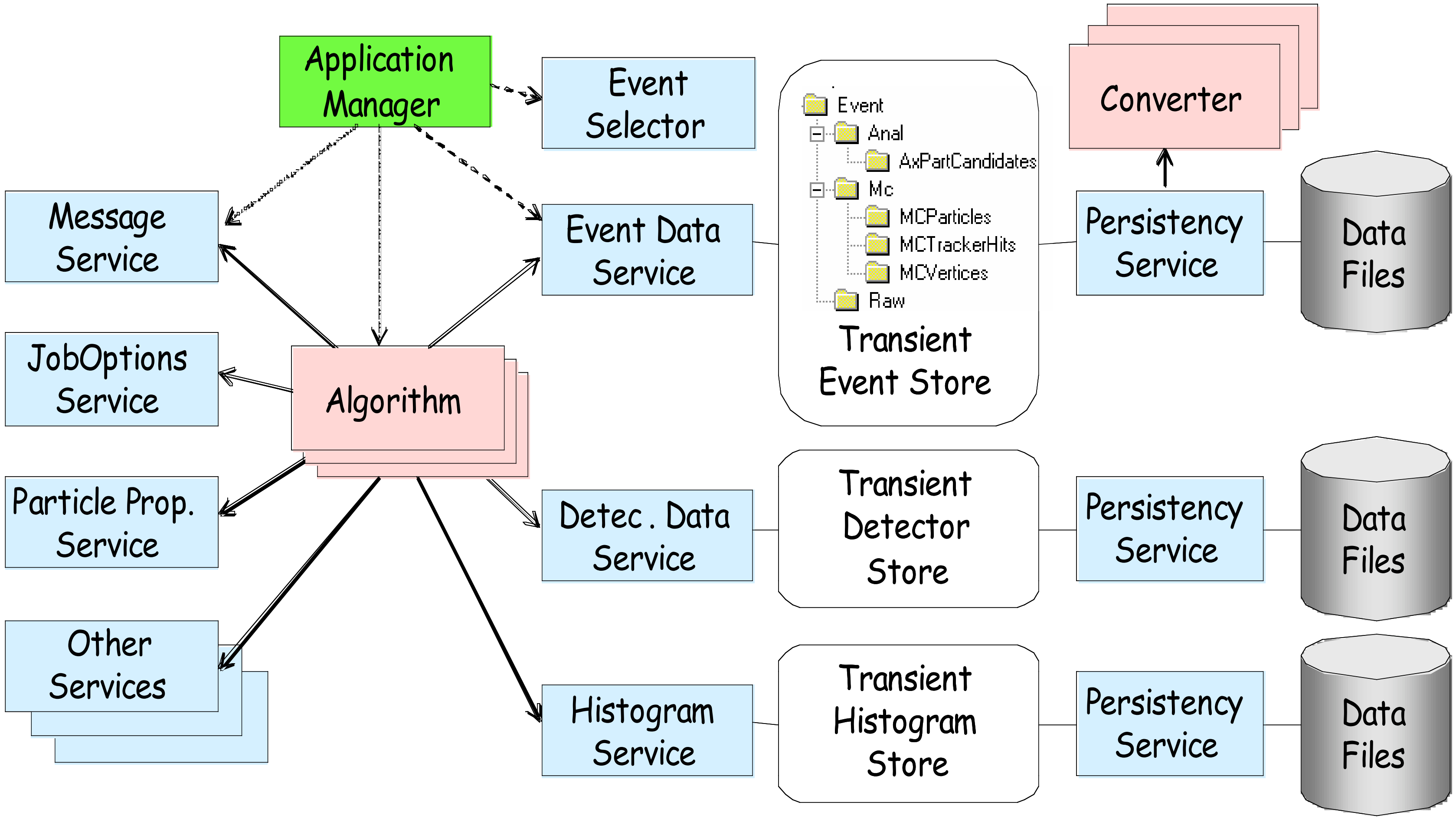




Map: Detector frontend assign event-id to each fragment
DAQ dispatch all fragment with same id to a given filter node
Reduce: filter node assemble the event and process it

PARALLEL ARCHITECTURES FOR HEP EVENT PROCESSING

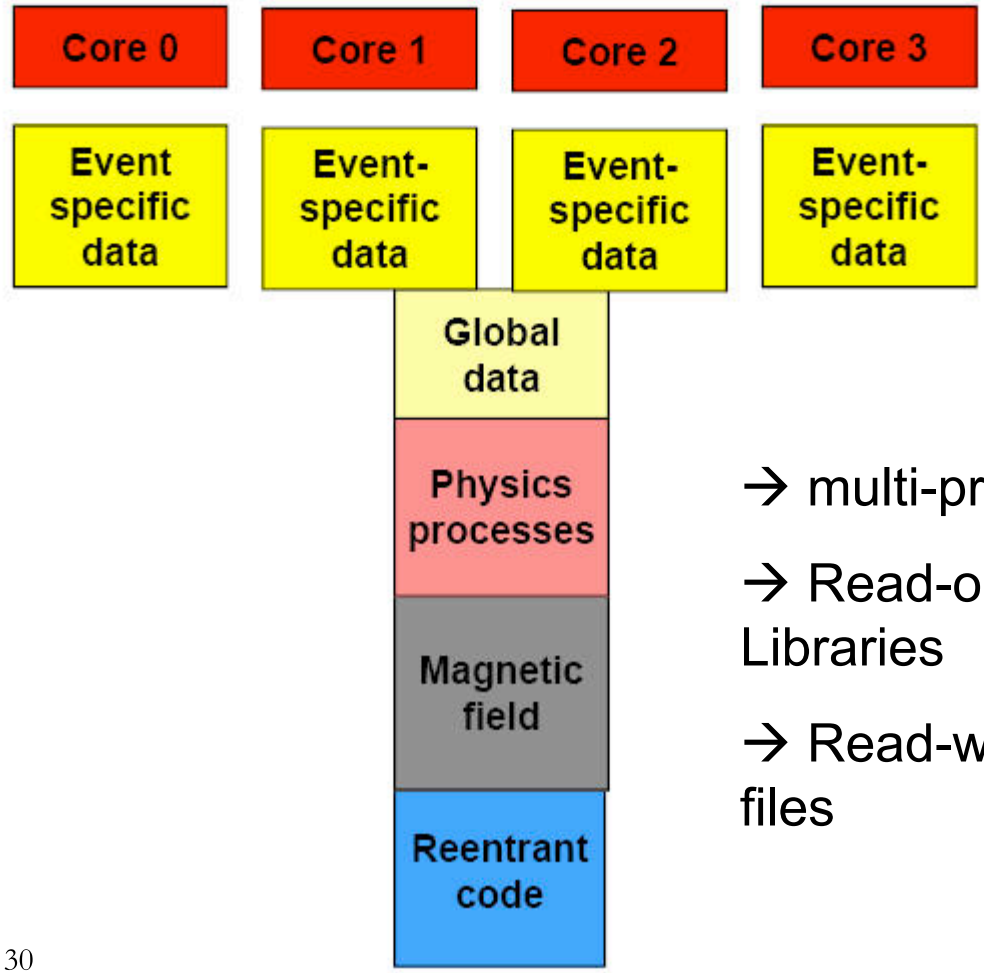
HEP Application



Event parallelism

Opportunity: Reconstruction Memory-Footprint shows large condition data

How to share common data between different process?



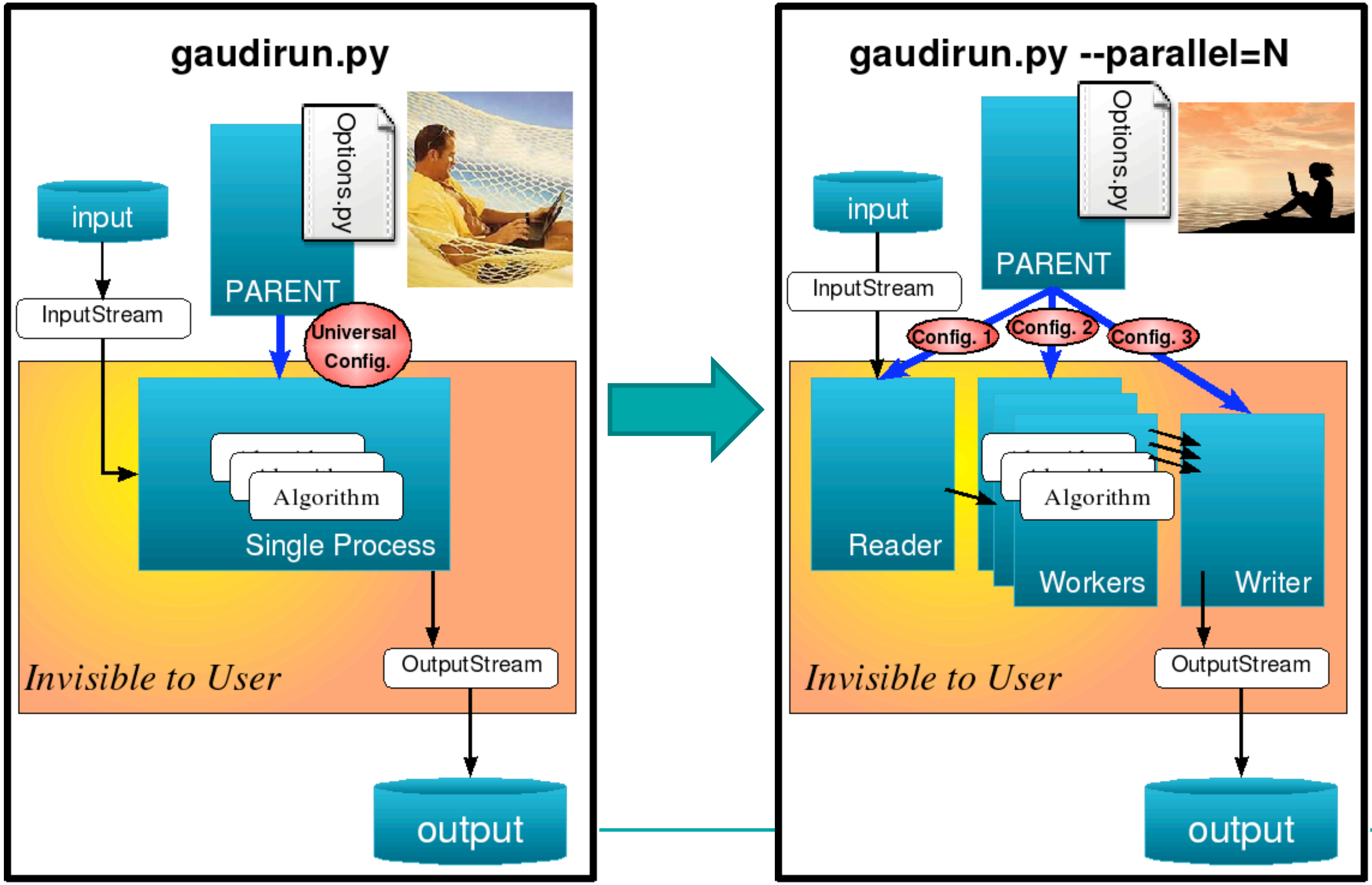
CMS:
1GB total Memory Footprint
Event Size 1 MB
Sharable data 250MB
Shared code 130MB
Private Data 400MB !!

→ multi-process vs multi-threaded

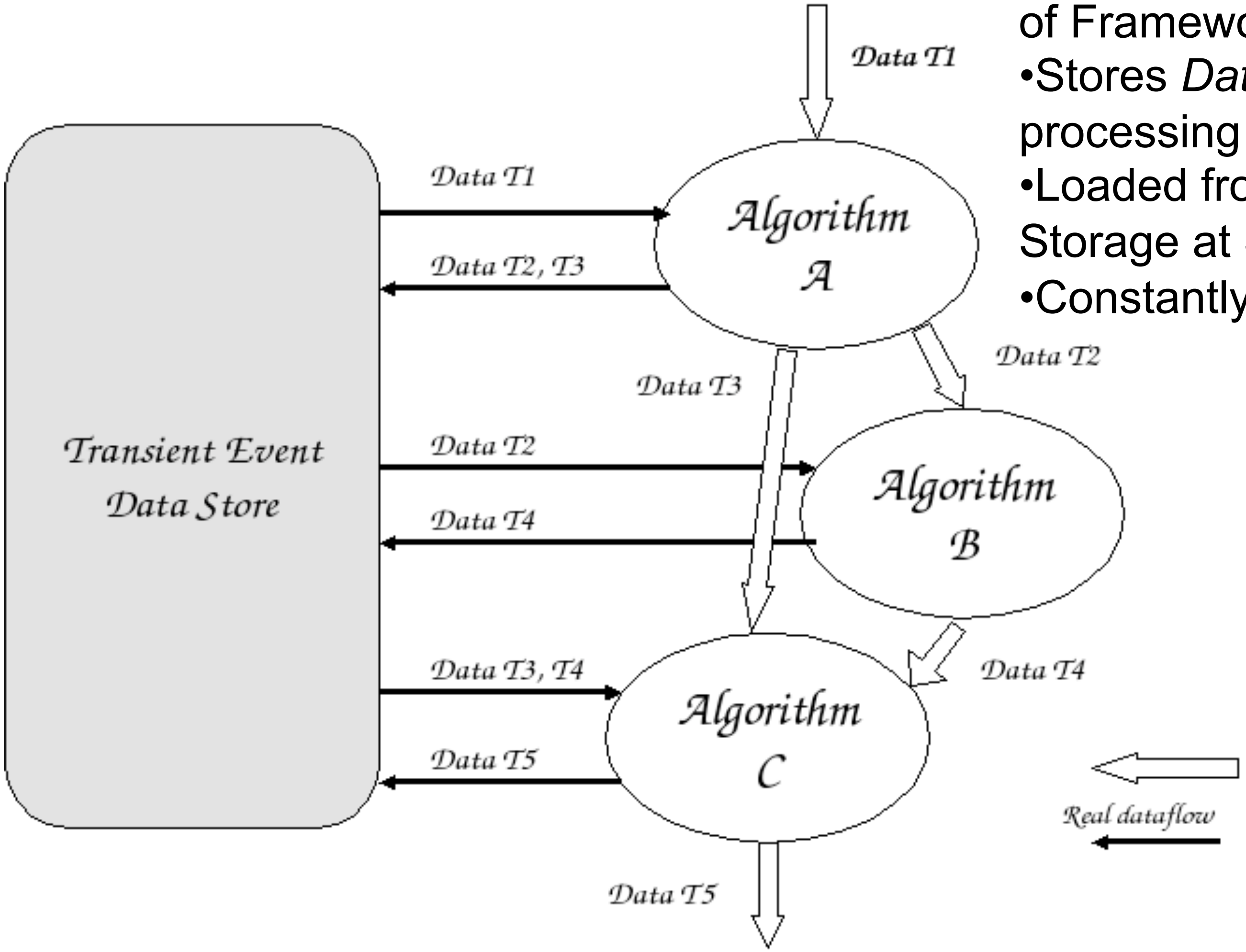
→ Read-only: Copy-on-write, Shared Libraries

→ Read-write: Shared Memory, sockets, files

Parallelization of Gaudi Framework



Gaudi : HEP Event Processing



- **Transient Event Store** : Part of Framework
- Stores *DataObjects* during processing
- Loaded from Persistent Storage at Start
- Constantly modified during run

HEP data processing

- No need of a coherent event state:
  - Algorithms
    - read specific event-fragments, store new fragments: never modify existing ones
  - Storage:
    - Fragments map root branches: independent of each other
- Conditions shared among events and (some) algorithms
  - Event parallelism will profit of coherent shared conditions
  - Algorithm parallelism can make them private to each of them

Act Parallel

ALGORITHMIC STRUCTURE

Hello Word!

```
void hi() {
  std::cout << "Hello World from "
            << std::this_thread::get_id() << std::endl;
}
```

```
void hello() {
   std::thread t1(hi);
   std::thread t2(hi);
   hi();
   t1.join();
   t2.join();
}
```

Create (Fork) new Threads where hi will run

Run hi in the "main" thread

wait for Threads to terminate

OO Hello Word!

```
class Hi {
public:
  Hi() : j(0) {}
  explicit Hi(int i) : j(i){}
  void operator()() {
    ++j;
    std::cout << "Hello World from "
                        << std::this_thread::get_id()
                 << " where j is " << j << std::endl;
  }
  int j;
};
```

Create (Fork) new Threads where Hi::operator() will run

wait for Threads to terminate

```
void hello() {
  std::thread t1(hi);
  std::thread t2(Hi(3));
  hi();

  Hi oneHi;
  std::thread t3(std::ref(oneHi));
  std::thread t4(std::ref(oneHi));
  oneHi();

  t1.join();
  t2.join();
  t3.join();
  t4.join();
  std::cout << "j is " << oneHi.j <
std::endl;
}
```

Create local Hi object Pass it by copy to Thread

Create local Hi object Pass it by reference to Threads

What Happens?

```
[pcphsft60] ~/public/Bertinoro $ ./a.out
Hello World from Hello World from 140186020210544
Hello World from 1090701632 where j is 4 start is 0
1113024832
start is 3
Hello World Hello Hello World from from World from \
 1090701632 where j is 1113024832140186020210544 2 \
where j is  where j is 11 start is  start is  start is 00
0

start is 0
j is 2
[pcphsft60] ~/public/Bertinoro $ ./a.out
Hello World from 1091725632Hello World from
Hello World from 1102555456 where j is 4 start is 0
139953850718064
start is 2
Hello World from Hello 139953850718064Hello  where j is \
World 1 start is World 0from
start is 0
from 1102555456 where j is 2 start is 0
1091725632 where j is 3 start is 0
j is 3
[
```

```
[pcphsft60] ~/public/Bertinoro $ ./a.out
Hello World from Hello 139882834876272World from
1108351296 where j is 4 start is 0
Hello World from 1084438848
start is 2
Hello World Hello Hello World from from 1398828348762721108351296
where j is  where j is 1 start is 0World
from start is 0
3 start is 0
1084438848 where j is 2 start is 0
j is 3
[pcphsft60] ~/public/Bertinoro $ ./a.out
Hello World from Hello 140206101608304World from 1093429568 where j
is
4 start is Hello World from 0
1085036864
start is 2
Hello Hello Hello World World from World from 140206101608304 where j
is from 11093429568 1085036864 where j is  start is  where j is 0
start is 0
1 start is 0
2 start is 0
j is 2
```

Synchronization

- Critical sections
  - A critical section is a portion of code that shall be executed by only one thread at a time.
    - Used to protect access to shared resources (memory)
  - In C++0x, a critical section can be protected by a "guard" that takes care to lock and then release a "Mutual exclusion object (mutex)"

```
typedef std::mutex Mutex;
typedef std::unique_lock<std::mutex> Guard;
 Mutex lock;
{
  Guard guard(lock);
  std::cout….
}
```

Constructor locks the mutex
Destructor unlocks it

Optimize Critical Sessions (and avoid pitfalls)

- Critical sections may introduce a significant fraction of sequential (non parallel) operations
  - Granularity shall be chosen properly
    - Make critical sections small
    - Use different mutex to guard independent sections
  - Major Pitfall: DeadLock
    - Are sections really independent?

Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - » SPMD "Single Program Multiple Data"
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
  work1();
  barrier
} else { barrier }
```

- barriers are not provided in all thread libraries

No "barrier in C++0x: use modified boost::barrier

Explicit Synchronization

C++0x provides a simple explicit synchronization mechanism based on “condition_variable”s

```
typedef std::mutex Mutex;
typedef std::unique_lock<std::mutex> Guard;
typedef std::condition_variable Condition;

Mutex goLock;
Condition go;
```

```
{
  Guard guard(goLock);
  go.wait(guard);
  // do something
}
```

```
{
  Guard guard(goLock);
  // prepare data
  go.notify_all();
}
```

release mutex, put thread in wait
When notified re-acquire mutex

Barrier implementation

```
#include <thread>
#include <exception>
class barrier {
  public:
    typedef std::mutex Mutex;
    typedef std::unique_lock<std::mutex> Guard;
    typedef std::condition_variable Condition;

        barrier(unsigned int count)
                    : m_threshold(count),
m_count(count), m_generation(0) {
      if (count == 0)
                throw std::invalid_argument("count
cannot be zero.");
    }
```

```
bool wait() {
     Guard guard(m_mutex);
      unsigned int gen = m_generation;

      if (--m_count == 0) {
                    m_generation++;
                    m_count = m_threshold;
                    m_cond.notify_all();
                    return true;
      }
      while (gen == m_generation)
                  m_cond.wait(guard);
      return false;
    }
  private:
    Mutex m_mutex; Condition m_cond;
    unsigned int m_threshold,  m_count;
    unsigned int m_generation;
};
```

Future

C++0x provides a data exchange mechanism among threads based on *promise-future* pattern

Able to transfer exceptions too!

```
std::promise<T> promise;
```

```
try {
  // prepare data
  promise.set_value(result);
 }catch (…){
m_promise.set_exception(std::current_exc
eption());
 }
  // continue processing
```

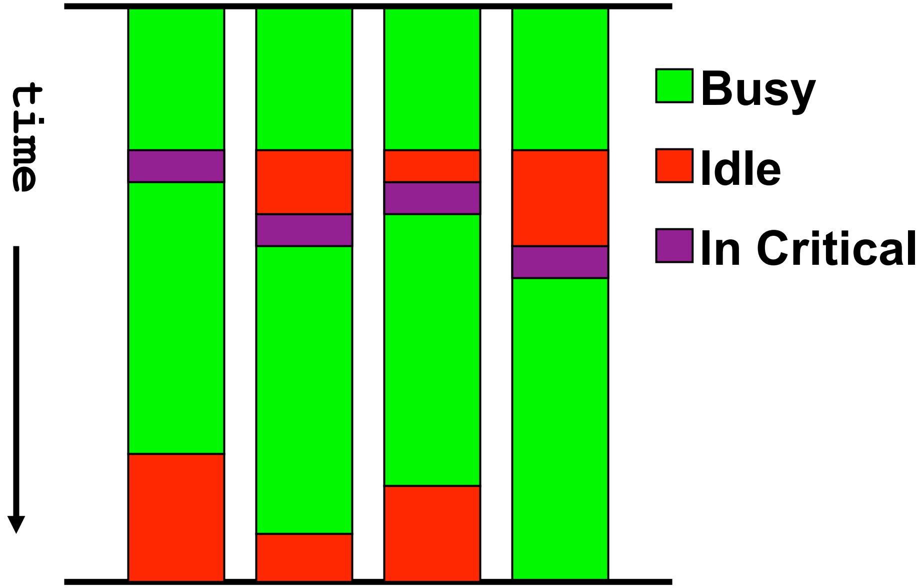
```
 unique_future<T> input = promise.get_future();
 // now I need the result of the other thread
try {
 T data =  input.get();
 // continue processing
}catch (…) { /*handle error*/}
```

Wait till other thread does not set value (or exception) in promise

Synchronization

- Lost time waiting for locks

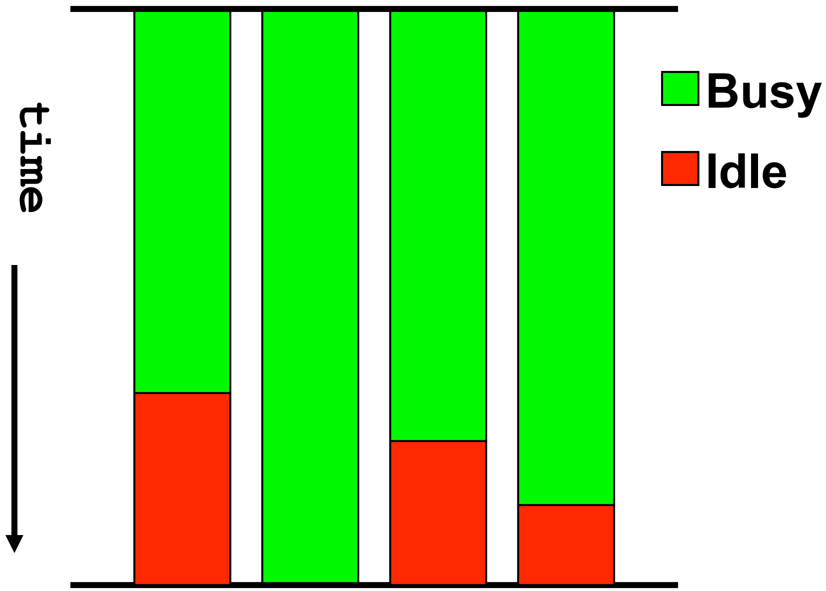
```
#pragma omp parallel
{

#pragma omp critical
   {
    ...
   }
   ...
}
```



Load Imbalance

- Unequal work loads lead to idle threads and wasted time

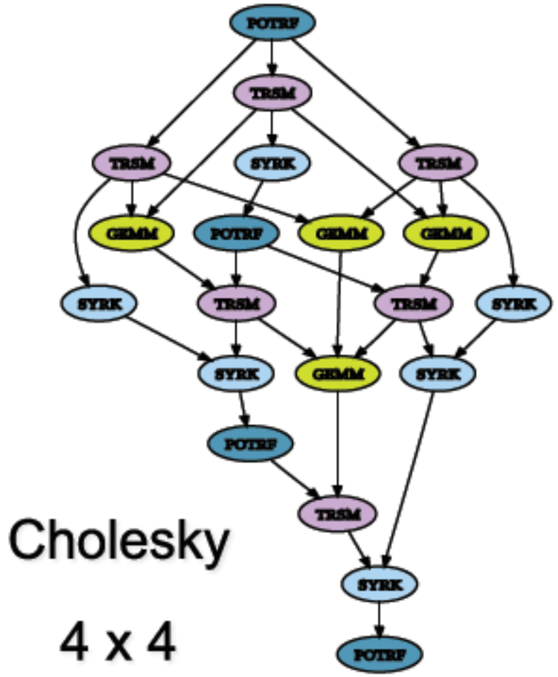
```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

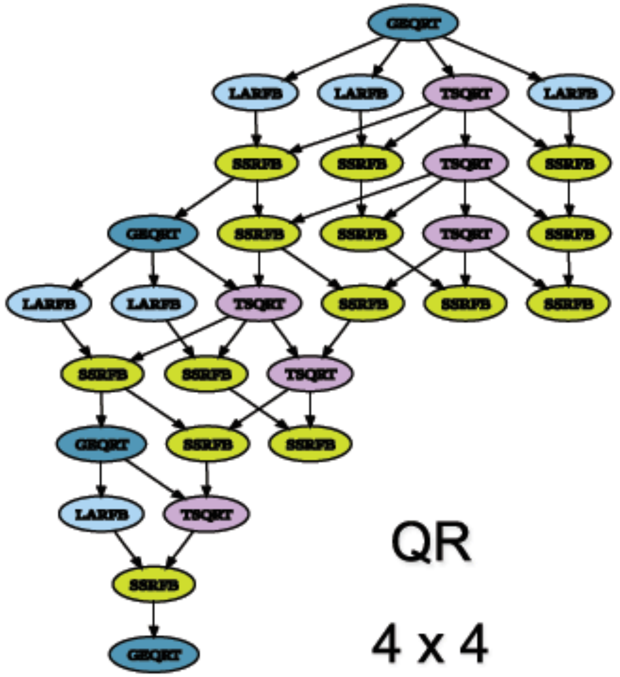


Performance Pitfall: too many Barriers

Computations as DAGs

View parallel executions as the directed acyclic graph of the computation

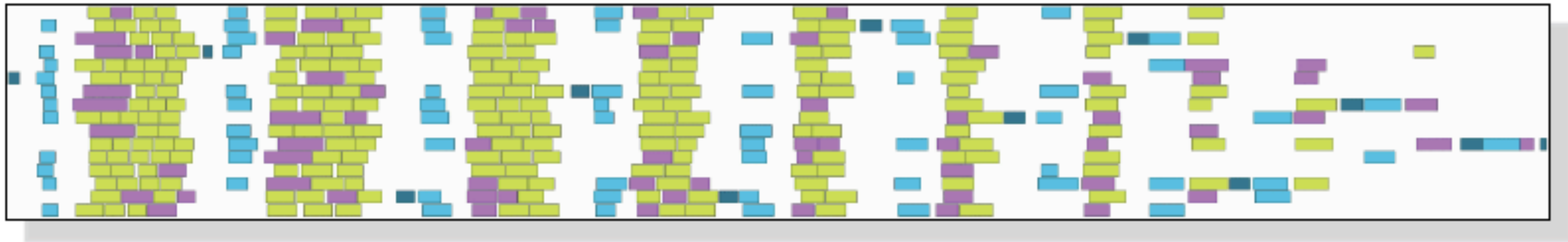




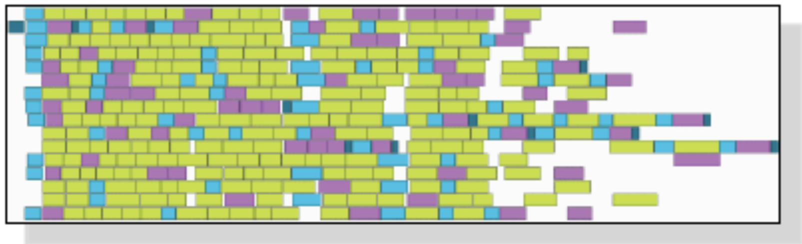
Slide source: Jack Dongarra

Illustration from PLASMA Project

Nested fork-join parallelism (e.g., Cilk, TBB)



Arbitrary DAG scheduling (e.g., PLASMA,



SuperMatrix)

Performance Calculations

T_s Best serial code timing (single thread/core)

$T_p(n)$ Parallel code timing using *n* threads/cores (*p* fraction parallel, $p \in [0,1]$)

$T_p(1)$ Parallel code timing using *1* thread/core

$\frac{T_p(1)}{T_s}$ Indication of parallel overhead

$$S_p(n) = \frac{T_s}{T_p(n)}$$

Actual Speedup over single thread using *n* threads/cores (*p* fraction parallel)

$$E_p(n) = \frac{S_p(n)}{n}$$

Efficiency using *n* threads/cores (*p* fraction parallel)

$$S_p^{\max}(n) = \frac{1}{1-p+\frac{p}{n}}$$

Max. Speedup over single thread using *n* threads/cores (*p* fraction parallel)

Speedup

- Maximum speedup defined by Amdahl's law:

$$S_p^{max}(n) = \frac{1}{1 - p + \frac{p}{n}}$$

n=#threads, p=parallel fraction

- Which just state the obvious:
  - A bare 10% non-parallel fraction limits the speedup to a factor 10!

Classical parallel Algorithms

- Single Program Multiple Data (SPMD)
- Loop parallelism

- Wait for OpenMP/MPI presentation!

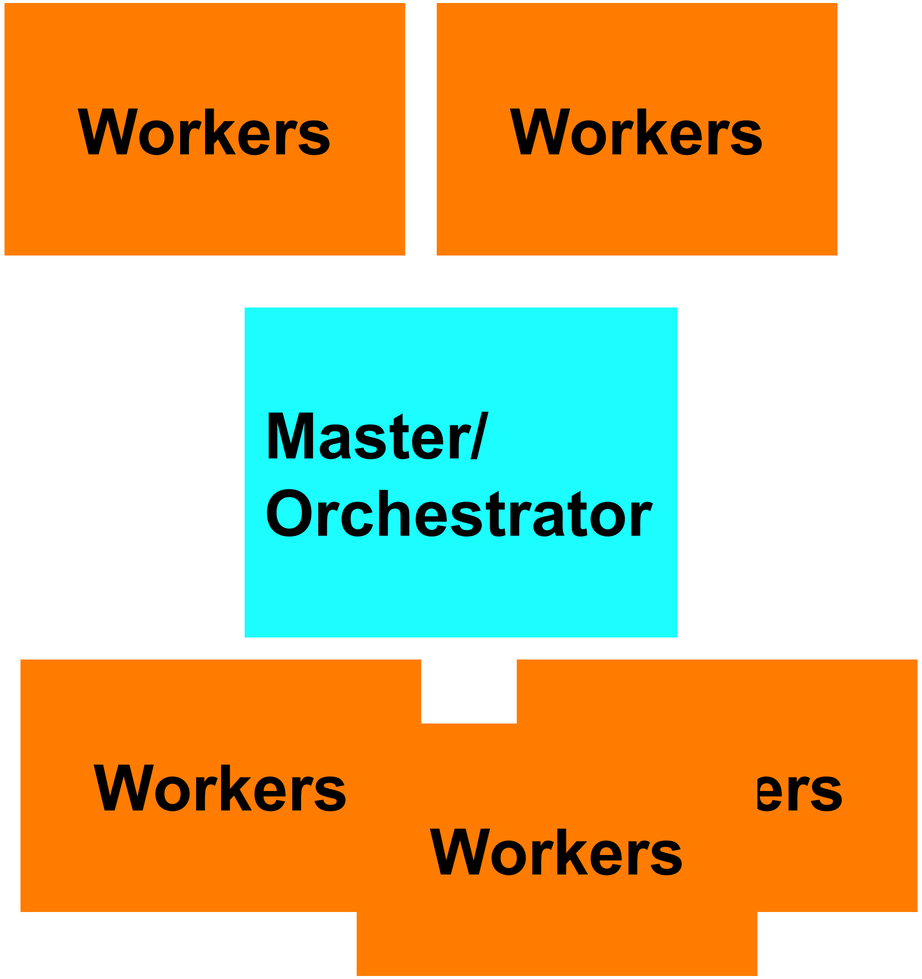
Divide&Conquer by Fork&Join

```
/* re-entrant function Mattson el al. 5.29 page 170 */
template<typename Iter, typename Compare>
void parallel_sort(Iter b, Iter e, Compare c) {
  size_t n = std::distance(b,e);
  // final exit
  if (n< SORT_THRESHOLD) return std::sort(b,e,c);
 // Divide
  Iter pivot = b +n/2;
  // Conquer
  // fork first half
  Thread forked(parallel_sort<Iter,Compare>,b,pivot,c);
 // process locally second half
  parallel_sort(pivot,e,c);
// wait for the other half
  forked.join();
  // merge...
  std::inplace_merge(b,pivot,e);
}
```

Exercise:
rewrite Map-Reduce-Like
Eventually using OpenMP

What about returning data,
error management,
exceptions?
Wait for future!

Master/Worker



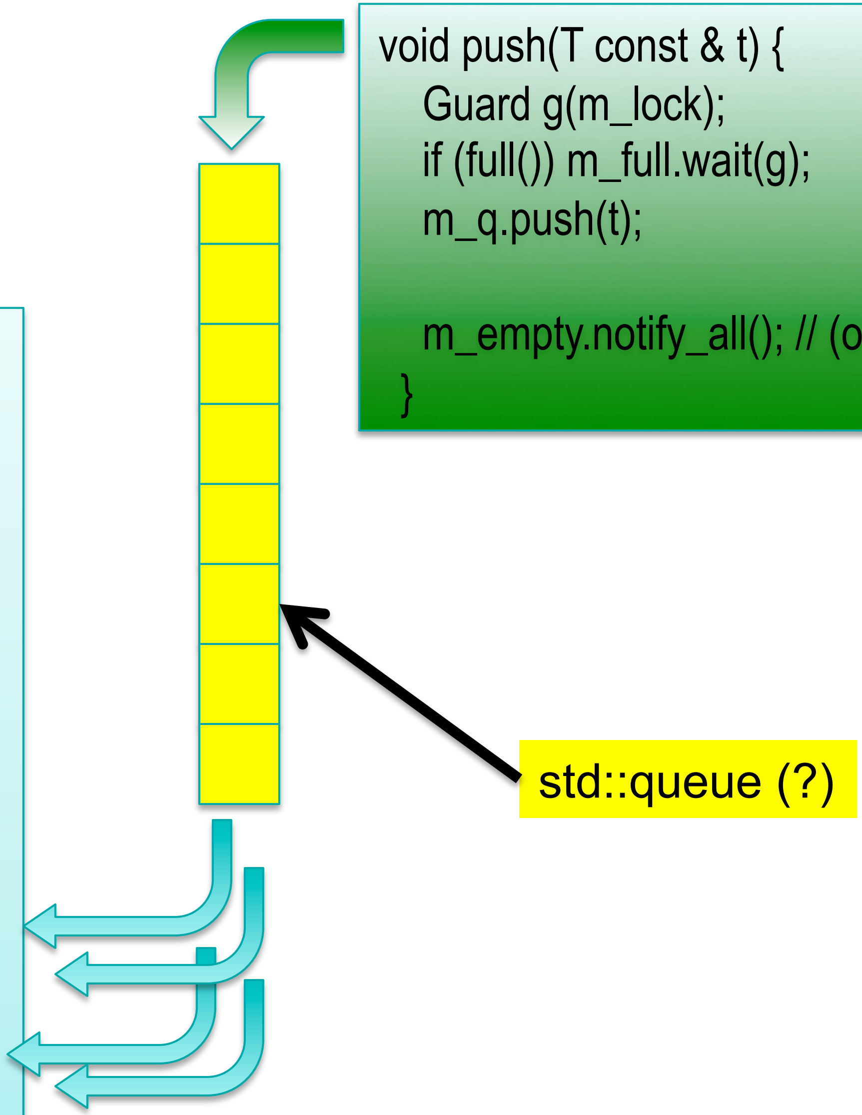
- Master distribute tasks to workers
  - No communication among workers
  - Many possible data access patterns
- Scheduling and queue theory applies!
  - Single queue multiple workers
  - Multiple queues
  - Dedicated queues/workers
  - Roundrobin, priorities,…

Shared Queue

Full chapter in Mattson el al.

```
void push(T const & t) {
    Guard g(m_lock);
    if (full()) m_full.wait(g);
    m_q.push(t);

    m_empty.notify_all(); // (or notify_one?)
  }
```

```
bool pop(T & t) {
    Guard g(m_lock);
    while (empty()) {
      if (m_drain) return false;
      m_empty.wait(g);
    }
    t =  m_q.front();
    m_q.pop();

    if ( !full() ) m_full.notify_all(); // (or
notify_one?)

    return true;
  }
```

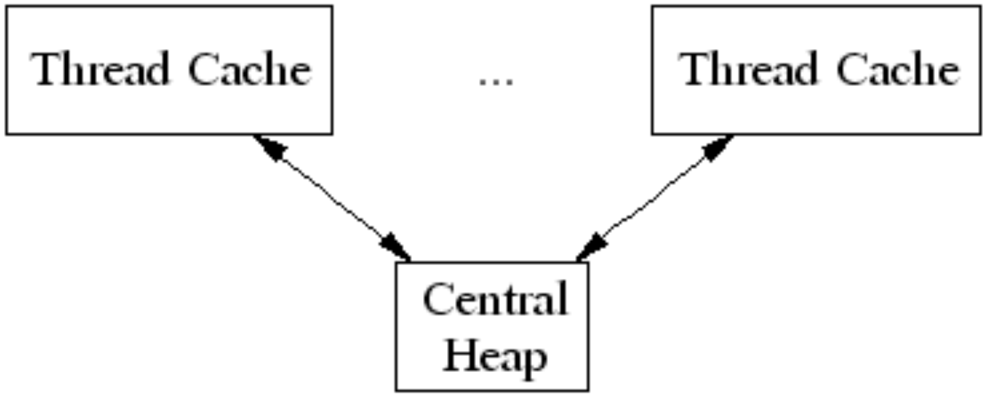


(managing) shared data

- Single Lock:
  - Last resort (in case of legacy data-structures)
- Distributed Locks
  - Optimize granularity
    - Heuristic: #locks = #threads
- Distributed data
  - Map-Reduce:
    - overhead of reduce vs overhead of locks
- Transactional memory access
  - Atomic operations
  - No lock, unroll if fails
  - Promising technology…

TCMalloc (Google Malloc)

- TCMalloc assigns each thread a thread-local cache.
  - Small allocations are satisfied from the thread-local cache.
  - Objects are moved from central data structures into a thread-local cache as needed,
  - Periodic garbage collections are used to migrate memory back from a thread-local cache into the central data structures.



Not the end of the story…

Read Phenix Rebirth about porting on a 256-thread UltraSPARC T2+ system.

http://csl.stanford.edu/~christos/publications/2009.scalable_phoenix.iiswc.pdf

Atomic operation

- Modern architectures provide atomic operations
  - Guaranteed to be fully completed by just one thread
- gcc intrinsics on x86_64

http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/Atomic-Builtins.html

  - CAS
    - __sync_bool_compare_and_swap(addr,expected,new)
    - __sync_val_compare_and_swap(addr,expected,new)
  - Op and fetch (also fetch and op)
    - __sync_add_and_fetch(addr,n)
  - Swap
    - __sync_lock_test_and_set(addr,n)

Atomic operation

- C++0x std::atomic<type>

  ```
  #include <cstdatomic>
  std::atomic<int> x;
  ```

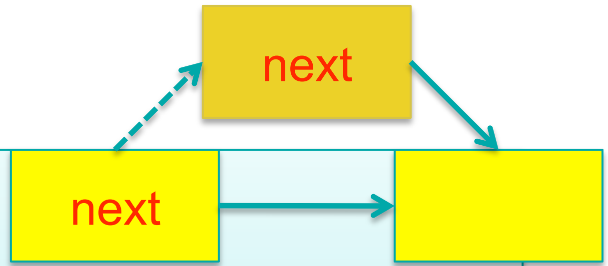
  - CAS
    - x.compare_exchange_strong(expected,new)
  - Fetch and op ; Op and fetch
    - x.fetch_add(n); x++; x+=n; etc
  - Swap
    - x.exchange(n); x=n;

Atomics in use

- Barrier with spinlock

```
struct barrier {
  void wait() {
   __sync_add_and_fetch(m_n,-1);
   while(m_n) {/* std::this_thread::yield(); */}
  }
  volatile long m_n;
};
```

- Shared linked list

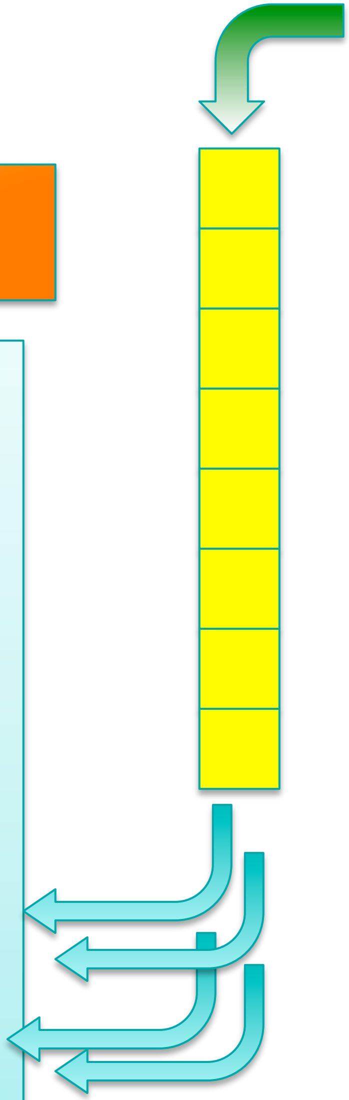



```
pointer insert(pointer p, value_type const & value) {
  Node * me = new Node; me->value=value;
  if (p==0) p=&head;
  while (true) {
   me->next = p->next;  // next sequential code p->next=me;
   if (__sync_bool_compare_and_swap(&(p->next),me->next,me)) break;
  }
```

Non Blocking Shared Queue

```
// circular buffer
 T  container[last+1];
```

```
bool pop(T&t) {
   while (true) {
      if(waitEmpty()) return false; //
include a signal to drain and terminate
      volatile size_t cur=tail;
      if (cur==head) continue;
      t = container[cur];
      if
(__sync_bool_compare_and_swap(&tai
l,cur,cur==0 ?last : cur-1)) break;
   }
   return true;
 }
```



```
// single producer only
void push(T const & t) {
    while (true) {
      waitFull(); // head always empty…
      volatile size_t cur=head;
      container[cur] = t; // shall be done first to
avoid popping wrong value
 if
(__sync_bool_compare_and_swap(&head,c
ur,cur==0 ? last : cur-1 )) {
              // container[cur] = t; // too late
pop already occured!
              break;
     }
   }
 }
```

Lock Free Hash Table
(Cliff Click at Google Camp 2007)

- Insert: CAS as before
- Delete: just mark
- The difficult part is to implement lock free resizing
  - Do not block operation from other threads
  - Allow other threads to collaborate in resizing
- Memory management is the other big issue
- Read slides and papers: very instructive

Singleton, Services

- Very unclear, static seems to introduce a full memory barrier in any case
- Will not be covered
- Slide will be deleted

(Sorting)
Word count
Histograming
Clustering
N-body dynamics

USE CASES

Histograming

Produce amplitude histogram (0-255) for each 10MPixel image

- Throughput: process images in parallel
  - Embarrassingly parallel
- Latency: process each image in parallel!

- Input: vector<uchar> image(10000000);
- Output: vector<int> histogram(256);
- Function: ++histogram[image[i]];

Word count

Count the number of occurrences of each word in a text

- Input: Sequence<char> text(N); N large
- Output: AssociativeContainer<string,int> words;
- Function: WordIterator word(text);

```
++words[*(++word)];
```

Clusterize

Clusterize points in 3D using Kmeans

- Input vector<Points3D> points
- Output vector<Cluster> clusters

Algorithm

- Start with a set of seeds
- Associate points to closest seed: (clusters)
- Compute mean (cluster position, new seed)
- iterate

Bibliography

- Google, wikipedia
- Architecture:
  - http://www.cs.berkeley.edu/~volkov/cs267.sp09/
  - http://parlab.eecs.berkeley.edu/wiki/patterns/pattern1_0
  - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.3594
  - http://www.sigsoft.org/phdDissertations/theses/JorgeOrtega.pdf
- Computational Patterns
  - http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
- Map-Reduce
  - http://hadoop.apache.org/ (distributed: cluster, grid, cloud)
  - Phoenix http://mapreduce.stanford.edu/ (multicore)
- C++0X
  - http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html
  - http://www.stdthread.co.uk/doc/