Parallel Programming Concepts

Software Programming Models

Peter Tröger

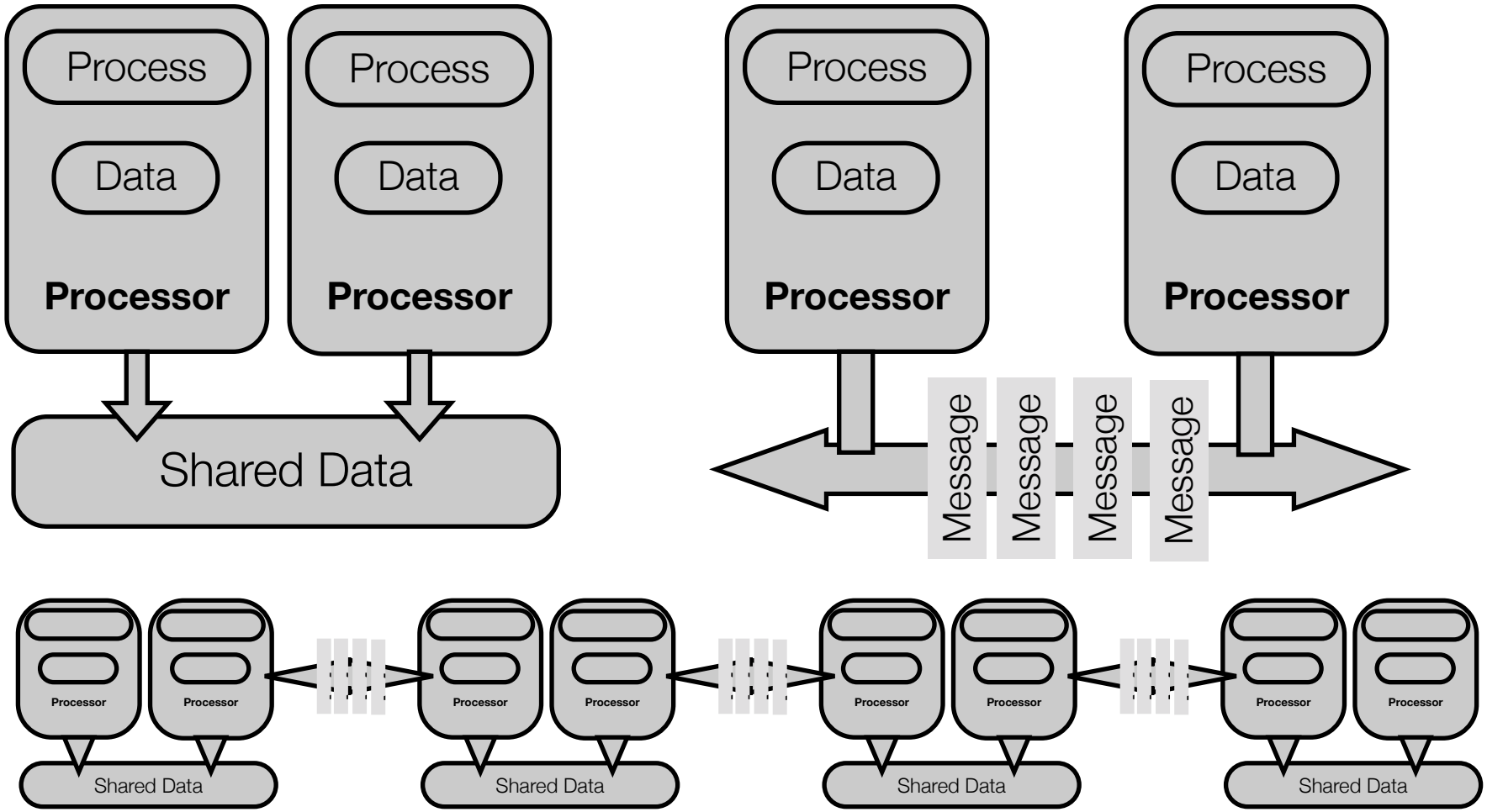
Sources:

Clay Breshears: The Art of Concurrency
Blaise Barney: Introduction to Parallel Computing
OpenMP 3.0 Specification
MPI2 Specification
Blaise Barney: OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/

Programming Models

- Almasi and Gottlieb: *„set of rules for a game“* - algorithms as game strategies
- View on data and execution, where architecture and application meet
- High-level view of the application on it‘s run time environment
  - Hardware might imply a programming model, but does not enforce it
  - Reflects on the design of the application
- For uni-processor, no question due to „von Neumann“
- Parallel systems: **Shared-memory** or **distributed-memory** architecture
- Intended as contract - if you follow the rules, scalability should be achievable
- Decouples software application and hardware architecture development

Shared Memory vs. Distributed Memory Systems



Terminology

- **Concurrency**
  - Supported to have two or more actions *in progress* at the same time
  - Classical operating system responsibility
    (resource sharing for better utilization of CPU, memory, network, ...)
  - Demands **scheduling** and **synchronization**
- **Parallelism**
  - Supported to have two or more actions executing *simultaneously*
  - Demands **parallel hardware**, **concurrency support**, (and **communication**)
  - Programming model relates to chosen hardware / communication approach
- Examples: Windows 3.1, threads, signal handlers, shared memory

Examples for Parallel Programming Support

	Task-Parallel Programming Model	Data-Parallel Programming Model	Actor Programming Model	Functional Programming Model	PGAS / DSM Programming Model
Shared Memory System	OpenMP, Threading Libs, Linda, Ada, Cilk	OpenMP, PLINQ, HPF	Scala, Erlang	Lisp, Clojure, Haskell, Scala, Erlang	-
Distributed Memory System	*Socket communication, MPI, PVM, JXTA, MapReduce, CSP channels*				-
Hybrid System	-	OpenCL	-	-	Unified Parallel C, Titanium, Fortress, X10,Chapel

Threads vs. Tasks

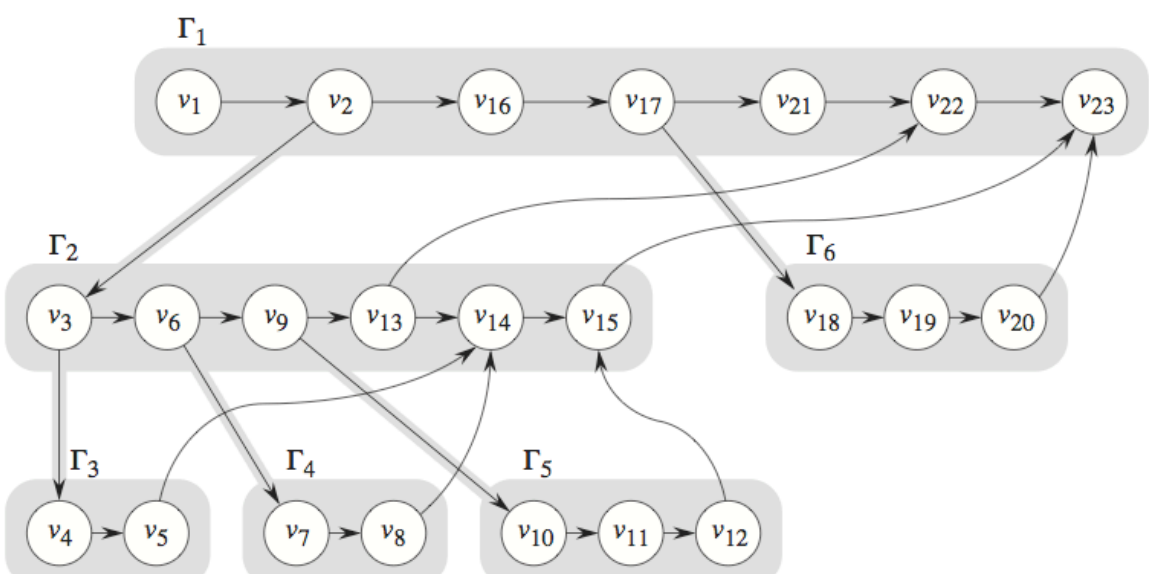
- **Process:** Address space, handles, code
- **Thread:** Stack, registers; preemptive; process address space
  - Scheduled by the operating system, can migrate between cores
- **Task:** Stack, registers; non-preemptive
  - Typically modeled as object (TBB, Java) or statement / lambda expression / anonymous function (OpenMP, MS TPL)
  - Scheduled by a user-mode library on a pool of threads
  - Task model replaces context switch with **yielding** approach
  - Typical scheduling policy for tasks is **central queue** or **work stealing**

Work Stealing

- *Blumofe, Robert D.; Leiserson, Charles E.: Scheduling Multithreaded Computations by Work Stealing. In: In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS. 1994), S. 356-368*
- Problem of scheduling scalable multithreading problems on processors
- Work sharing: When processors create new work, the scheduler migrates threads for balanced utilization
- Work stealing: Underutilized processor takes work from other processor
  - Intuitively, less thread migrations
  - Goes back to work stealing research in Multilisp (1984)
  - Communication reaches lower bound for parallel divide-and-conquer
  - Approach by Blumofe et al. relies on „fully strict multithreaded computation“

Computation Model

- Multithreaded computation is represented as directed acyclic graph (DAG), representing a set of of serial threads and their dependencies
  - Threads contain sequentially executed instructions - *continue edges*
  - Threads may create other threads - *spawn edges*
  - Threads may produce data that others rely upon, creating a producer -> consumer relationship - *join edges*



Strict Computation Concept

- Deterministic execution: Computation is independent from scheduling
- „Strict“ multithreaded computation: All join edges go to an ancestor of the thread, the only edge into a sub-tree is the spawn edge
- „Fully strict“ computation: All join edges go to the thread‘s parent
- „Busy-leaves“ property: In a strict computation, spawned threads can always be completed by a single processor
- „Busy-leaves“ scheduling algorithm: Global thread pool, used by all processors in the system
  - Spawned and stalled threads go (back) to the thread pool
  - On thread finalization, the living other children's are covered before the parent thread
  - Scaling problem through central resource contention

Randomized Work Stealing

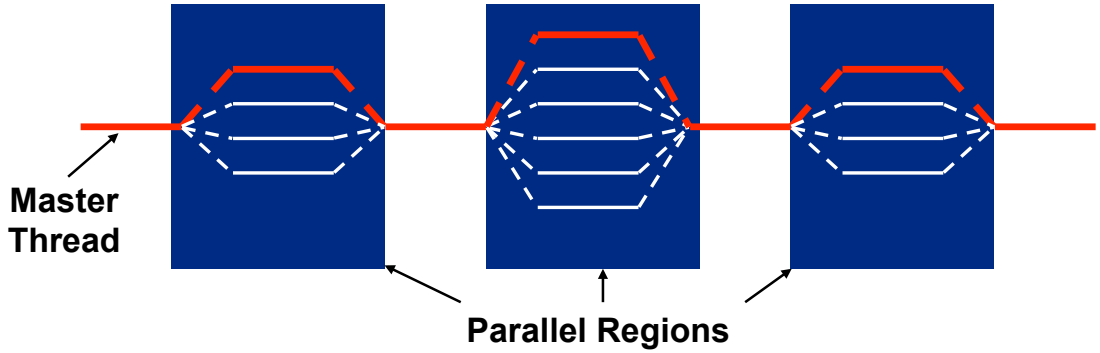
- Centralized thread pool is now distributed across the processors
- Ready dequeue: Lock-free private bottom end, and synchronized public end
  - Processor takes work from the bottom of its local queue
  - Spawned threads are placed on the bottom of the local dequeue
  - If no ready thread is available on thread stall / dead, the processor steals the top-most one from a randomly chosen processor
  - If no victim is available, processor continues to randomly search for one
  - Stalled threads which are enabled again are placed on the bottom of the ready dequeue of their enabling processor
- Algorithm maintains the busy-leaves property - ready threads are either executed or wait for a processor becoming free
- Supported in Microsoft TPL, Intel TBB, Java, Cilk, ...

Task-Parallel, Shared-Memory Programming

- Most common parallelization approach, often called „multi-threading“
- Relevant issues: Task generation, execution synchronization, data movement
  - Manual coordination in a sequential language (operating system threads, Java / .NET threads, ...) -> *„explicit“ threading*
  - Using a framework for parallel tasks (OpenMP, Intel TBB, MS TPL, ...) -> *„implicit“ threading*
  - Using a parallel programming language
- Common synchronization concepts: mutex, semaphore, conditions, barriers
    - **Volatile** keyword: Ensure that variable updates happen synchronously in memory (visible for all threads)
    - **Semaphore** protection (see earlier lectures, assignment 1)

OpenMP

- Specification for C/C++ and Fortran language extension (currently v3.0)
  - Portable shared memory thread programming
  - High-level abstraction of task- and loop parallelism
  - Derived from compiler-directed parallelization of serial language code (HPF), with support for incremental change of source code
- Programming model: Fork-Join-Parallelism
  - Master thread spawns group of threads for limited code region
    - PARALLEL directive
    - Barrier concept



OpenMP Pragmas

- `#pragma omp` *`construct`* `...` (include `omp.h`)
- OpenMP runtime library: query functions, runtime functions, lock functions
- Parallel region
  - OpenMP constructs are applied to dedicated code blocks, marked by `#pragma omp parallel`
  - Worker threads are created when the master thread enters this block
  - Should have only one entry and one exit
  - Implicit barrier at beginning and end of the block !
- Implementation maintains a thread pool for spawned tasks
- Idle worker threads may sleep or spin, depending on library configuration (performance issue in serial parts)

OpenMP Parallel Construct

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the parallel construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.7 on page 59).

OpenMP Configuration / Query Functions

- Environment variables
  - `OMP_NUM_THREADS`: number of threads during execution, upper limit for dynamic adjustment of threads
  - `OMP_SCHEDULE`: set schedule type and chunk size for parallelized loops of scheduling type `runtime`
- Query functions
  - `omp_get_num_threads`: Number of threads in the current parallel region
  - `omp_get_thread_num`: Current thread number in the team, master=0
  - `omp_get_num_procs`: Available number of processors
  - ...

OpenMP Work Sharing

- Possibilities for distribution of tasks across threads (‚work sharing‘)
  - `omp sections` - Define code blocks usable as tasks
  - `omp for` - Automatically divide a loop‘s iterations into tasks
    - Implicit barrier at the end
  - `omp task` - Explicitly define a task
  - `omp single / master` - Denotes a task to be executed only by first arriving thread resp. the master thread
    - Implicit barrier at the end, intended for non-thread-safe activities
  - Scheduling of tasks defined is handled by the OpenMP implementation
- Clause combinations possible: `#pragma omp parallel for`

OpenMP Work Sharing with Sections

- Explicit definition of code blocks as parallel tasks with section directive, executed in the context of the implicit task
- One thread can execute more than one section, if allowed

```
#pragma omp parallel
{
  #pragma omp sections [ clause [ clause ] ... ]
  {
    [#pragma omp section ]

      structured-block1

    [#pragma omp section ]

      structured-block2
```

OpenMP Data Scoping

- Shared memory programming model - communication through variables
- **Shared variable**: Name provides access to same memory in all tasks
  - Shared by default: global variables, static variables, variables with namespace scope, variables with file scope
  - `shared` clause can be added to any `omp` construct, defines a list of additionally shared variables
- **Private variable:** Clone variable in each task, from common initial value
  - Private by default: Local variables in functions called from parallel regions, loop iteration variables, automatic variables

OpenMP Consistency Model

- Thread's temporary view of memory is not required to be consistent with memory at all times - weak-ordering consistency
  - `#pragma omp flush`
  - Two flushes of the same variable are synchronous operations
  - No other restriction to re-ordering of memory operations - compiler issue
- Implicit flush on different occasions, such as barrier region

```
                    a = b = 0

thread 1                          thread 2

 b = 1                             a = 1
 flush(a,b)                        flush(a,b)
 if (a == 0) then                  if (b == 0) then
     critical section                  critical section
 end if                            end if
```

OpenMP Work Sharing with Loop Parallelization

- Loop construct: Execute iterations of the immediately following loop in parallel
- Iteration variable must be integer
- Mapping of threads to iterations is controlled by `schedule` clause
- Implications on exception handling, break-out calls and *continue* primitive

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic, chunk)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

OpenMP Loop Parallelization Scheduling

- `schedule (static, [chunk])` - Contiguous ranges of iterations (chunks) are assigned to the threads
  - Low overhead, round robin assignment to free threads
  - Increasing chunk size reduces overhead, improves cache hit rate
  - Decreasing chunk size allows finer balancing of work load
- `schedule (dynamic, [chunk])` - Threads grab iteration resp. chunk
  - Higher overhead, but good for unbalanced iteration work load
- `schedule (guided, [chunk])` - Dynamic schedule, shrinking ranges per step, starting with large block, until minimum chunk size is reached
  - Computations with increasing iteration length (e.g. prime sieve test)

OpenMP Loop Parallelization

- Recommended uses
  - Static scheduling for predictable and similar work per iteration
  - Dynamic / guided scheduling for unpredictable, highly variable work load
- *Auto scheduling mode*: Decision delegated to compiler or runtime system
- *Runtime scheduling mode*: Decision based on runtime variable
- `nowait` clause disables exit barrier
- Default for scheduling mode is implementation-dependent

```
#include <math.h>
void a92(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
#pragma omp parallel
   {
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
   for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
   }
}
```

OpenMP Tasks

- Main change with latest OpenMP 3.0,
  allows description of non-data driven parallelization strategy
  - Farmer / worker algorithms
  - Recursive algorithms
  - Unbounded loops
    (e.g. while loops)

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
       #pragma omp task    // p is firstprivate by default
           postorder_traverse(p->left);
    if (p->right)
       #pragma omp task   // p is firstprivate by default
           postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

- Definition of tasks as composition of code to execute, data environment, and control variables
- Implicit task generation with `parallel, for` constructs
- Explicit task generation with `sections` and `task` constructs

OpenMP Tasks

- Certain construct act as task scheduling point
  - When thread encounters this construct, it can switch to another task

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Example: Generating task will suspend on exit barrier
  - Executing thread can help processing the task pool, or run the spawned task directly (cache-friendly)

OpenMP Synchronization

- Synchronizing with task completion
  - Implicit barrier at the end of `single` block, removable by `nowait` clause
  - `#pragma omp barrier` (wait for all other threads in the team)
  - `#pragma omp taskwait` (wait for completion of created child tasks)
- Synchronizing variable access
  - `#pragma omp critical [name]`
    - Enclosed block executed by all threads, but restricted to one at a time
    - All unnamed directives map to the same unspecified name
- Alternative: `#pragma omp reduction (op: list)`
  - Execute parallel tasks based on private copies of `list`, perform reduction on results with `op` afterwards, without race conditions

OpenMP Synchronization / Reduction

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for
  for(int i = 0; i < N; i++) {
     #pragma omp critical
     sum += a[i] * b[i];
  }
  return sum;
}
```

```
#pragma omp parallel for reduction(+:sum)
   for(i = 0; i < N; i++) {
     sum += a[i] * b[i];
   }
```

Supported associative operands:
+, *, -, ^(power), bitwise AND, bitwise OR, logical AND, logical OR

OpenMP Best Practices [Süß & Leopold]

- Typical correctness mistakes
  - Access to shared variables not protected
  - Use of locks / shared variables without `flush`
  - Declaring parallel loop variable as `shared`
- Typical performance mistakes
  - Use of `critical` when `atomic` would be sufficient
  - Too much work inside a `critical` section
  - Unnecessary `flush / critical`

Intel TBB

- *Task concept* - define what to run in parallel, instead of managing threads
- Portable C++ library, free and commercial toolkits for different operating systems
- Complements basic OpenMP features
  - Loop parallelization, parallel reduction, synchronization, explicit tasks
- High-level concurrent containers (hash map, queue, vector)
- High-level parallel operations (prefix scan, sorting, data-flow pipelining)
- Unfair scheduling approach, to favor threads having data in cache
- Supported for cache-aware memory allocation
- Comparable: Microsoft C++ Concurrency Runtime, additional agent support

Java Programming Language

- Supports concurrency on-top-of operating system threads
- Functions bundled in `java.util.concurrent`
- Classical concurrency support, based on reentract intrinsic object lock
  - `synchronized` methods: Allow only one thread in an objects‘ synchronized methods, based on intrinsic object lock
    - For static methods, works on class object
  - `synchronized` statements: Synchronize execution by intrinsic lock of the given object
  - `volatile` keyword: Indicate shared nature of variable - ensure atomic synchronized access, no thread-local caching
  - `wait` / `notify` semantics in `Object`

Java Examples

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Java wait / notify

- Each object can act as guard with `wait()` / `notify()` functions
  - Guard waiting must always be surrounded by explicit condition check

```
public synchronized guardedJoy() {
    //This guard only loops once for each special event, which may not
    //be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Java High-Level Concurrency

- Introduced with Java 5, summarized in `java.util.concurrent`
- `java.util.concurrent.locks`
- Separation of thread management and parallel tasks - *Executors*
  - `java.util.concurrent.Executor`
    - Implementing object provides `execute()` method, is able to execute submitted `Runnable` tasks
    - No assumption on where the task runs, might be even in the callers context, but typically in managed thread pool
    - `ThreadPoolExecutor` implementation provided by class library

Java High-Level Concurrency

- `java.util.concurrent.ExecutorService`
  - Supports also `Callable` objects as input, which can return a value
  - Additional `submit()` function, which returns a `Future` object on the result
    - `Future` object allows to wait on the result, or cancel execution
  - Methods for submitting large collections of `Callable`‘s
  - Methods for managing executor shutdown
- `java.util.concurrent.ScheduledExecutorService`
  - Additional methods to schedule tasks repeatedly
- Available thread pools from executor implementations:
  Single background thread, fixed size, unbound with automated reclamation

Java High-Level Concurrency

```
interface ArchiveSearcher { String search(String target); }
class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(final String target)
      throws InterruptedException {
    Future<String> future
      = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }});
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
  }
}
```

Java High-Level Concurrency

```
class NetworkService implements Runnable {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;

  public NetworkService(int port, int poolSize)
      throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void run() { // run the service
    try {
      for (;;) {
        pool.execute(new Handler(serverSocket.accept()));
      }
    } catch (IOException ex) {
      pool.shutdown();
    }
  }
}

class Handler implements Runnable {
  private final Socket socket;
  Handler(Socket socket) { this.socket = socket; }
  public void run() {
    // read and service request on socket
  }
}
```

Other Facts

- `BlockingQueue`: FIFO data structure, block / time out on full or empty queue
- `ConcurrentMap`: Map data structures (e.g. hash) with atomic functions
- Lock-free atomic variable support - `AtomicBoolean`, `AtomicLong`, ...
- JDK7: Fork/Join support with work stealing, scalable random numbers

Parallel Programming in .NET

- As Java, .NET CLR relies on native thread model
  - Synchronization and scheduling mapped to operating system concepts
- .NET 4 has variety of support libraries
  - *Task Parallel Library (TPL)* - Loop parallelization, task concept
  - Task factories, task schedulers
  - *Parallel LINQ (PLINQ)* - Implicit data parallelism through query language
  - Collection classes, synchronization support
  - Debugging and visualization support

Easy Mappings [Dig]

	Java	TBB	TPL
Parallel For	ParallelArray	parallel_for	Parallel.For
Concurrent Collections	ConcurrentHashMap, ...	concurrent_hash_map, ...	
Atomic Classes	AtomicInteger, ...	atomic<T>	Interlocked
ForkJoin Task Parallelism	ForkJoinTask framework	task	Task, ReplicableTask

Apple Grand Central Dispatch

- Part of MacOS X operating system since 10.6
- Task parallelism concept for developer, execution in thread pools
  - Tasks can be functions or **blocks** (C / C++ / ObjectiveC extension)
  - Submitted to dispatch queues, executed in thread pool under control of the Mac OS X operating system
    - Main queue: Tasks execute serially on application‘s main thread
    - Concurrent queue: Tasks start executing in FIFO order, but might run concurrently
    - Serial queue: Tasks execute serially in FIFO order
- Dispatch groups for aggregate synchronization
- On events, dispatch sources can submit tasks to dispatch queues automatically

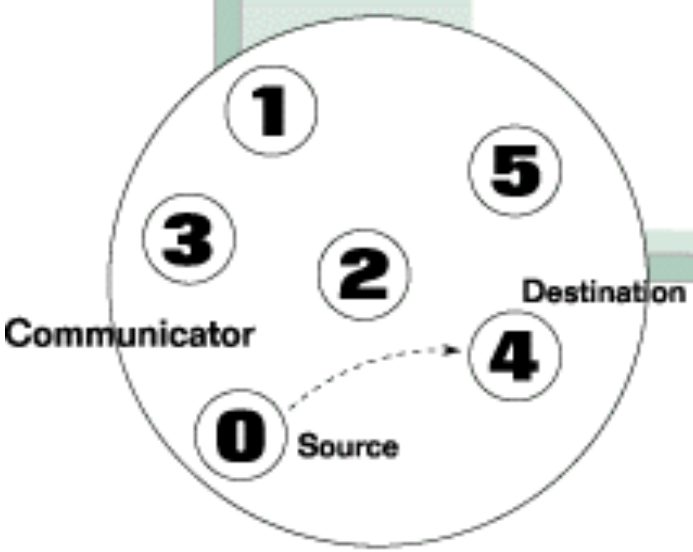
Message Passing Interface (MPI)

- Communication library for sequential programs
  - Definition of syntax and semantics for source code portability
  - Maintain implementation freedom on high-performance messaging hardware - shared memory, IP, Myrinet, propietary ...
  - MPI 1.0 (1994) and 2.0 (1997) standard, developed by MPI Forum
- Fixed number of processes, determined on startup
  - Point-to-point communication
  - Collective communication, for example group broadcast
- Focus on efficiency of communication and memory usage, not interoperability
- Fortran / C - Binding

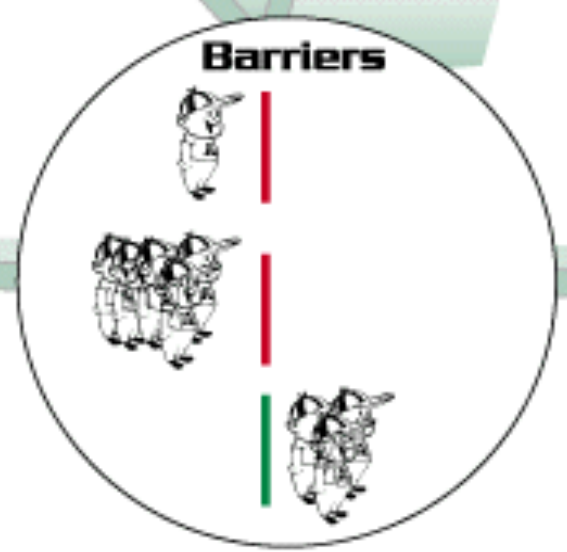
Machine-independent Message Passing Interface

Public: MPICH Chimp LAM-MPI MPI-FM MPIAP	**Tuned MPI Implementation**	**Vendor:** HP-MPI IBM-MPIF SGI-MPI
SunOS/Solaris Unicos ...	**Operating System**	AIX IRIX ...
SGI Intel Paragon Workstation Clusters ...	**Hardware**	Cray T3D/T3E IBM SP2 Fujitsu ...

Point-to-point Communication



Collective Communication



Basic MPI

- Communicators (process group handle)
  - `MPI_COMM_SIZE (IN comm, OUT size),`
    `MPI_COMM_RANK (IN comm, OUT pid)`
  - Sequential process ID‘s, starting with zero
- `MPI_SEND (IN buf, IN count, IN datatype, IN destPid, IN msgTag, IN comm)`
  `MPI_RECV (IN buf, IN count, IN datatype, IN srcPid, IN msgTag, IN comm, OUT status)`
  - Source / destination identified by 3-tupel tag, source and comm
  - MPI_RECV can block, waiting for specific source
- Constants: `MPI_COMM_WORLD, MPI_ANY_SOURCE, MPI_ANY_DEST`
- Data types: `MPI_CHAR, MPI_INT, ..., MPI_BYTE, MPI_PACKED`

Communication Modes

- Blocking communication
  - Do not return until the message data and envelope have been stored away
  - *Standard*: MPI decides whether outgoing messages are buffered
  - *Buffered*: MPI_BSEND returns always immediately
    - Might be a problem when the internal send buffer is already filled
  - *Synchronous*: MPI_SSEND completes if the receiver started to receive
  - *Ready*: MPI_RSEND should be started only if the matching MPI_RECV is already available
    - Can omit a handshake-operation on some systems
- Blocking communication ensures that the data buffer can be re-used

Non-Overtaking Message Order

- „If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.“

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_BSEND (buf1, count, MPI_REAL, 1,
                tag, comm, ierr)
   CALL MPI_BSEND (buf2, count, MPI_REAL, 1,
                tag, comm, ierr)
ELSE   ! rank.EQ.1
   CALL MPI_RECV (buf1, count, MPI_REAL, 0,
                MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV (buf2, count, MPI_REAL, 0,
                tag, comm, status, ierr)
END IF
```

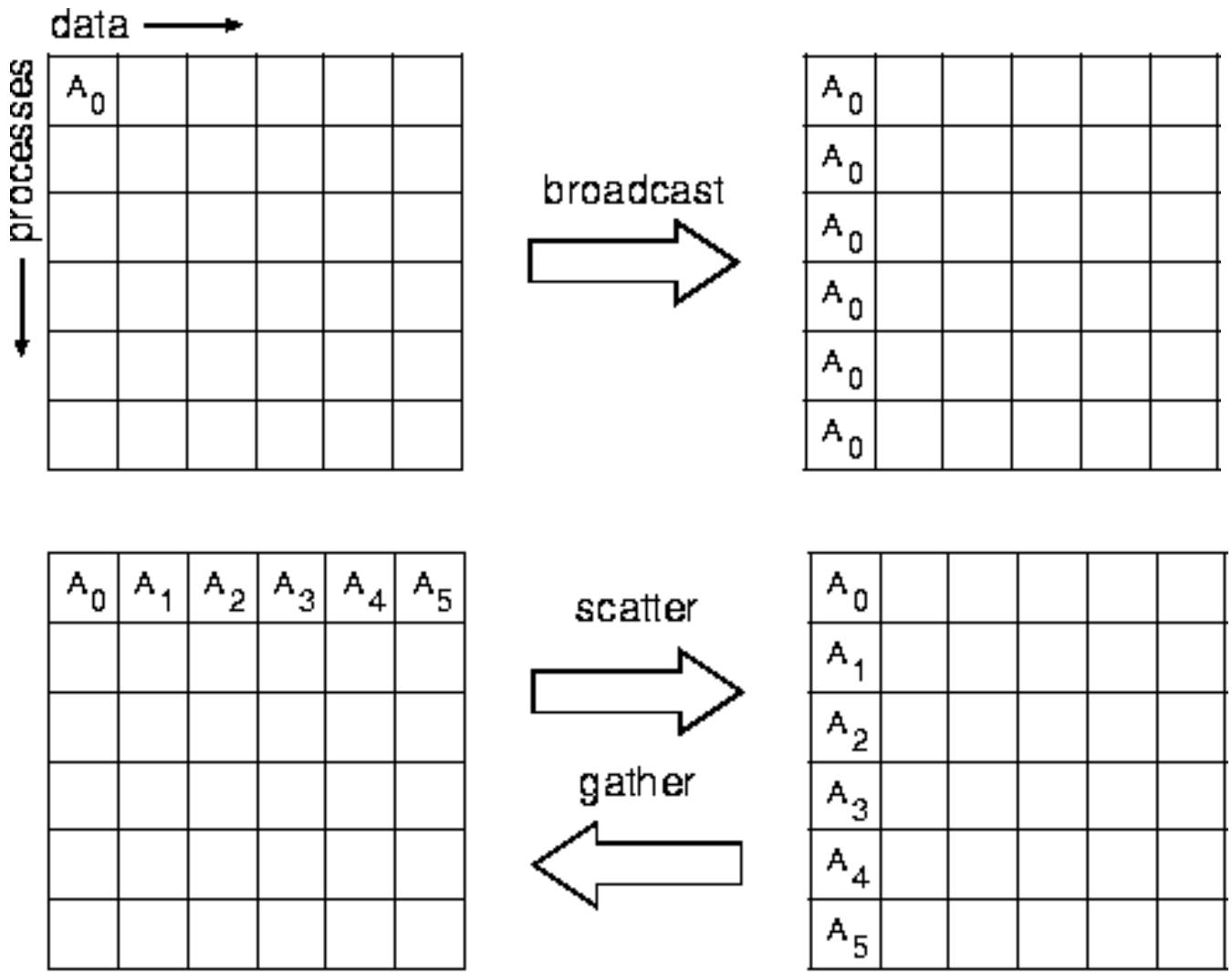
Non-Blocking Communication

- *Send/receive* start and *send/receive* completion call, with request handle
- Communication mode influences the behavior of the completion call
- Buffered non-blocking send operation leads to an immediate return of the completion call
- ‚Immediate send‘ calls
  - `MPI_ISEND, MPI_IBSEND, MPI_ISSEND, MPI_IRSEND`
- Completion calls
  - `MPI_WAIT, MPI_TEST, MPI_WAITANY, MPI_TESTANY, MPI_WAITSOME, ...`
- sending side: MPI_REQUEST_FREE

Collective Communication

- Global operations for a distributed application, could also be implemented manually
- `MPI_BARRIER (IN comm)`
  - returns only if the call is entered by all group members
- `MPI_BCAST (INOUT buffer, IN count, IN datatype, IN rootPid, IN comm)`
  - root process broadcasts to all group members, itself included
  - all group members use the same comm & root parameter
  - on return, all group processes have a copy of root‘s send buffer

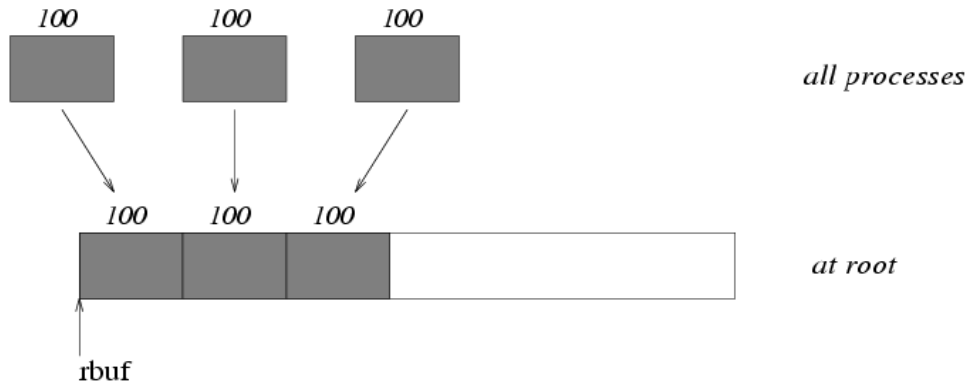
Collective Move Functions



Gather

- `MPI_GATHER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN comm )`
  - Each process sends its buffer to the root process (including the root process itself)
  - Incoming messages are stored in rank order
  - Receive buffer is ignored for all non-root processes
  - `MPI_GATHERV` allows varying count of data to be received from each process
  - No promise for synchronous behavior

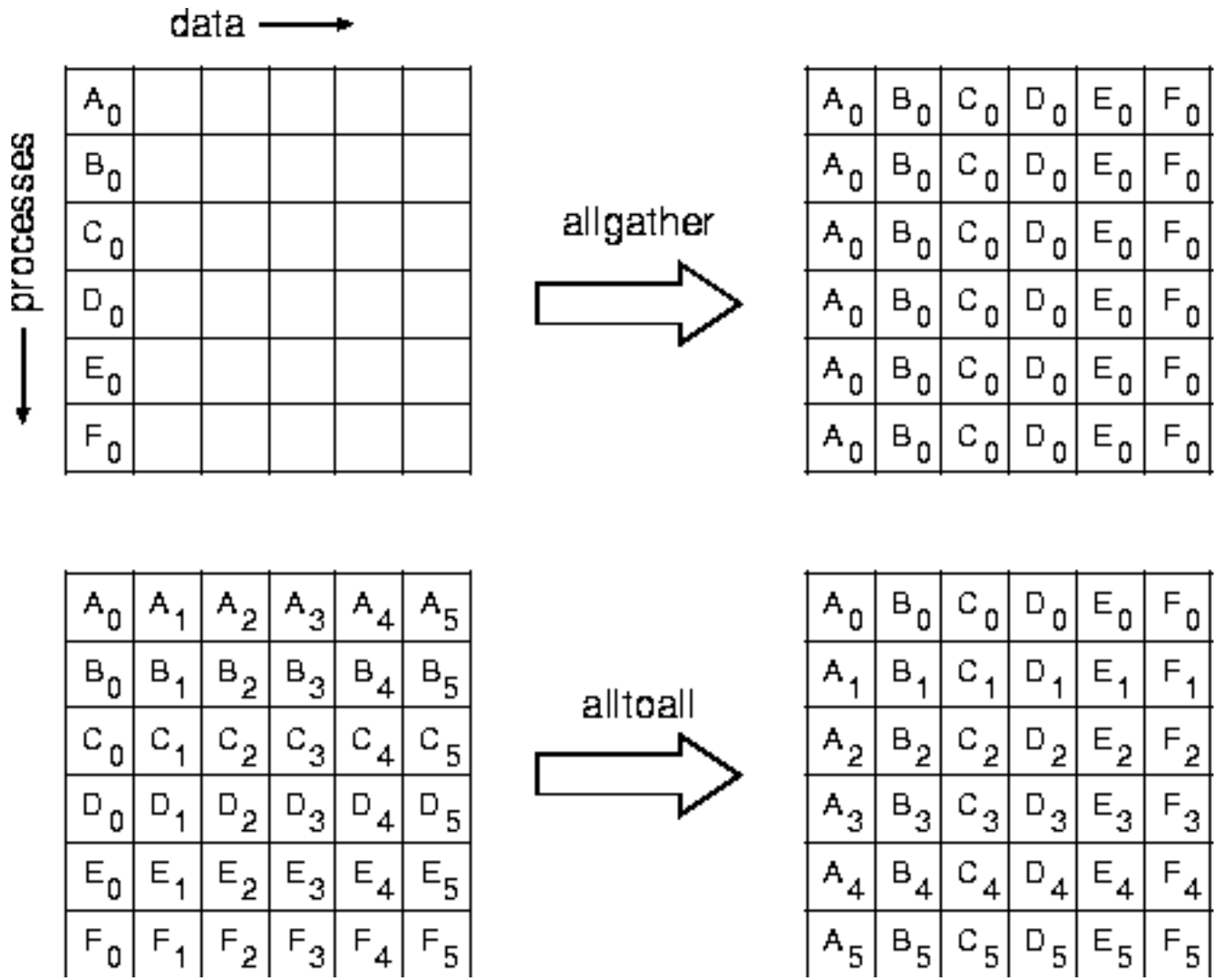
MPI Gather Example




```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
... [compute sendarray]
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
  MPI_Comm_size( comm, &gsize);
  rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather ( sendarray, 100, MPI_INT, rbuf, 100,
             MPI_INT, root, comm );
```

Scatter

- `MPI_SCATTER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN comm )`
  - Sliced buffer of root process is send to all other processes (including the root process itself)
  - Send buffer is ignored for all non-root processes
  - `MPI_SCATTERV` allows varying count of data to be send to each process


data
processes
allgather
alltoall

What Else

- `MPI_SENDRCV` (useful for RPC semantic)
- Global reduction operators
- Complex data types
- Packing / Unpacking (`sprintf / sscanf`)
- Group / Communicator Management
- Virtual Topology Description
- Error Handling
- Profiling Interface

MPICH library

- Development of the MPICH group at Argonne National Laboratory (Globus)
- Portable, free reference implementation
- Drivers for shared memory systems (ch_shmem), Workstation networks (ch_p4) , NT networks (ch_nt) and Globus 2 (ch_globus2)
- Driver implements `MPIRUN` (fork, SSH, MPD, GRAM)
- Supports multiprotocol communication (with vendor MPI and TCP) for intra-/intermachine messaging
- MPICH2 (MPI 2.0) is available, GT4-enabled version in development
- MPICH-G2 is based on Globus NEXUS / XIO library
- Debugging and tracing support

Examples for Parallel Programming Support

	Task-Parallel Programming Model	Data-Parallel Programming Model	Actor Programming Model	Functional Programming Model	PGAS / DSM Programming Model
Shared Memory System	OpenMP, Threading Libs, Linda, Ada, Cilk	OpenMP, PLINQ, HPF	Scala, Erlang	Lisp, Clojure, Haskell, Scala, Erlang	-
Distributed Memory System	*Socket communication, MPI, PVM, JXTA, MapReduce, CSP channels*				-
Hybrid System	-	OpenCL	-	-	Unified Parallel C, Titanium, Fortress, X10,Chapel