

Parallel Programming Concepts

Multi-Tasking

Peter Tröger

Sources:

Clay Breshears: The Art of Concurrency

Blaise Barney: Introduction to Parallel Computing

OpenMP 3.0 Specification

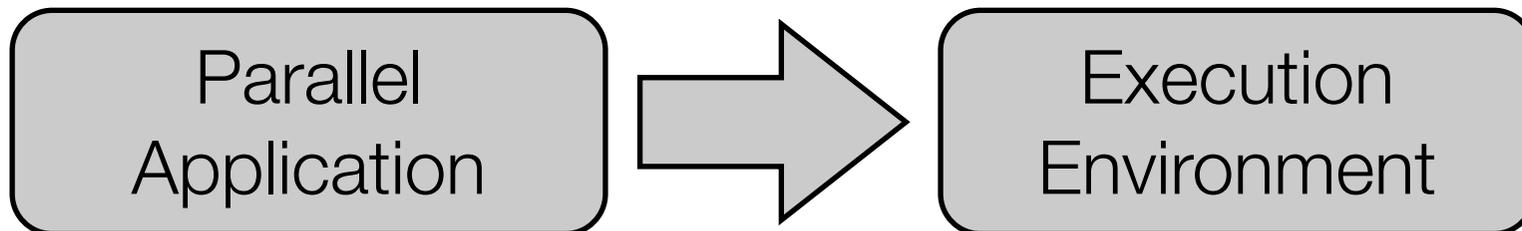
MPI2 Specification

Blaise Barney: OpenMP Tutorial, <https://computing.llnl.gov/tutorials/openMP/>

Parallel Programming

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

	Data Parallel / SIMD	Task Parallel / MIMD
Shared Memory (SM)	GPU, Cell, SSE, Vector processor ...	ManyCore/SMP system ...
Shared Nothing / Distributed Memory (DM)	processor-array systems systolic arrays Hadoop ...	cluster systems MPP systems ...



Multi-Tasking

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

Classical Add-Ons: Java

- Java supports concurrency on-top-of operating system threads for a while
- Functions bundled in `java.util.concurrent`
- Classical concurrency support, based on reentrant intrinsic object lock
 - `synchronized` methods: Allow only one thread in an objects' synchronized methods, based on intrinsic object lock
 - For static methods, locking based 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 void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
  
}
```

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 activities - *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
    }
}
```

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

Threads vs. Tasks

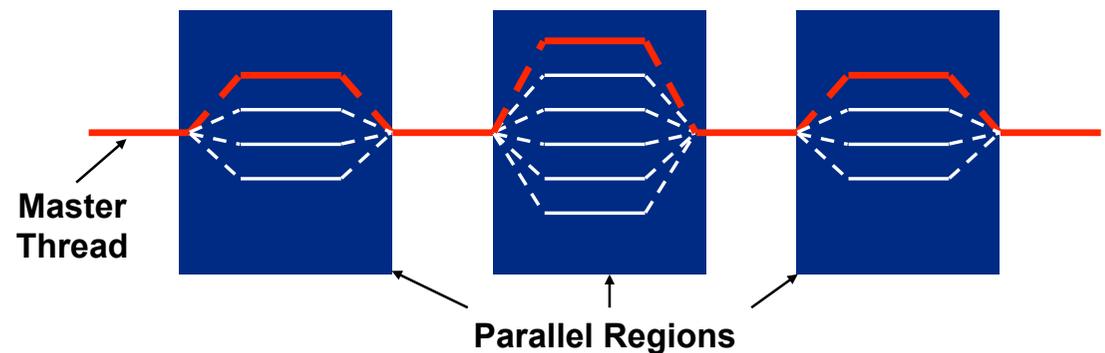
- **Process:** Address space, handles, code, set of threads
- **Thread:** control flow
 - Preemptive scheduling by the operating system
 - Can migrate between cores
- **Task:** control flow
 - Typically modeled as object (TBB, Java) or statement / lambda expression / anonymous function (OpenMP, MS TPL)
 - Cooperative scheduling by a user-mode library, mapping to thread pool
 - Task model replaces context switch with **yielding** approach
 - Typical scheduling policy for tasks is **central queue** or **work stealing**

Multi-Tasking

- Relevant issues: Task generation, execution synchronization, data access
 - Manual coordination in a sequential language (operating system threads, Java / .NET threads, ...)
-> „*explicit*“ *threading*
 - Using a framework for parallel tasks (OpenMP, OpenCL, Intel TBB, MS TPL, ...)
-> „*implicit*“ *threading*
- Concurrency problems remain
 - Critical section problem with shared variables in different tasks
 - Low-level synchronization primitives wrapped by „concurrent data structures“ in task framework
- Already covered: OpenCL

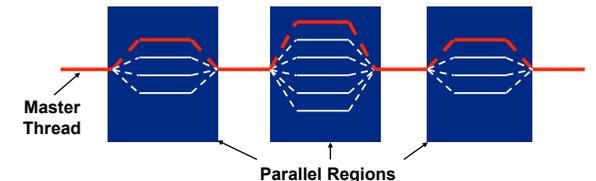
OpenMP

- Specification for C/C++ and Fortran language extension (currently v3.1)
 - 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`
 - Parallel region should have only one entry and one exit point
 - Implicit barrier at beginning and end of the block
- Thread pool for execution of parallel activities
- Idle worker threads may sleep or spin, depending on library configuration (performance issue in serial parts)



OpenMP Parallel Construct

- Encountering **thread** for the parallel region generates a set of implicit **tasks**
- Each resulting implicit **task** is assigned to a different **thread**
- Implementation may suspend **task execution** at a scheduling point

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 (I/O)
 - 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 (function partitioning)
- Executed in the context of the implicit task
- One task may execute more than one section - runtime decision

```
#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
 - Provides no automatic protection, just marking of variables for handling by runtime environment
- **Private variable:** Clone variable in each task, by default no initialization
 - Private by default: Local variables in functions called from parallel regions, loop iteration variables, automatic variables
 - Initialization with last value before region (`firstprivate`) possible

OpenMP Work Sharing with Loop Parallelization

- Loop construct: Parallel execution of iterations
- 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

```
#pragma omp parallel for
for(ii = 0; ii < n; ii++){
    value = some_complex_long_fuction(a[ii]);
    #pragma omp critical
    sum = sum + value;
}
```

```
#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 Consistency Model

- Thread's temporary view of memory is not required to be consistent with memory at all times (weak-ordering consistency)
 - Example: Keeping loop variable in a register for efficiency reasons
 - Compiler needs to be informed when consistent view is demanded
 - Implicit flush on different occasions, such as barrier region
 - In all other cases, shared variables must be flushed before reading

- Directive:

```
#pragma omp flush
```

```
                                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 Loop Parallelization Scheduling

- `schedule (static, [chunk])` - Contiguous ranges of iterations (chunks) are assigned to the threads
 - Low overhead, round robin assignment to free threads
 - Static scheduling for predictable and similar work per iteration
 - 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 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)

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        printf("Start: %d\n", omp_get_thread_num());
        #pragma omp single //nowait
        printf("Got it: %d\n", omp_get_thread_num());
        printf("Done: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

OpenMP Synchronization

- 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

```
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;
}
```

OpenMP Synchronization

- **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
 - Supported associative operands:
`+`, `*`, `-`, `^`, bitwise AND, bitwise OR, logical AND, logical OR

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

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`

OpenMP Tasks

- Main change with OpenMP v3, 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` and `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)

Intel TBB

- *Task concept* - define what to run in parallel, instead of managing threads
- Portable C++ library, 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

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

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“

Randomized Work Stealing

- Lock-free ready dequeue per processor
 - Task are inserted on the bottom, and can be taken from both sides
 - Processor obtains local work by taking one task from the bottom
 - Migrated tasks are taken from the top
 - If no ready task is available on task 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 tasks which are enabled again by other tasks are placed on the bottom of the ready dequeue of their enabling processor
- Algorithm maintains the busy-leaves property - ready tasks are either executed or wait for a processor becoming free
- Supported in Microsoft TPL, Intel TBB, Java, Cilk, ...

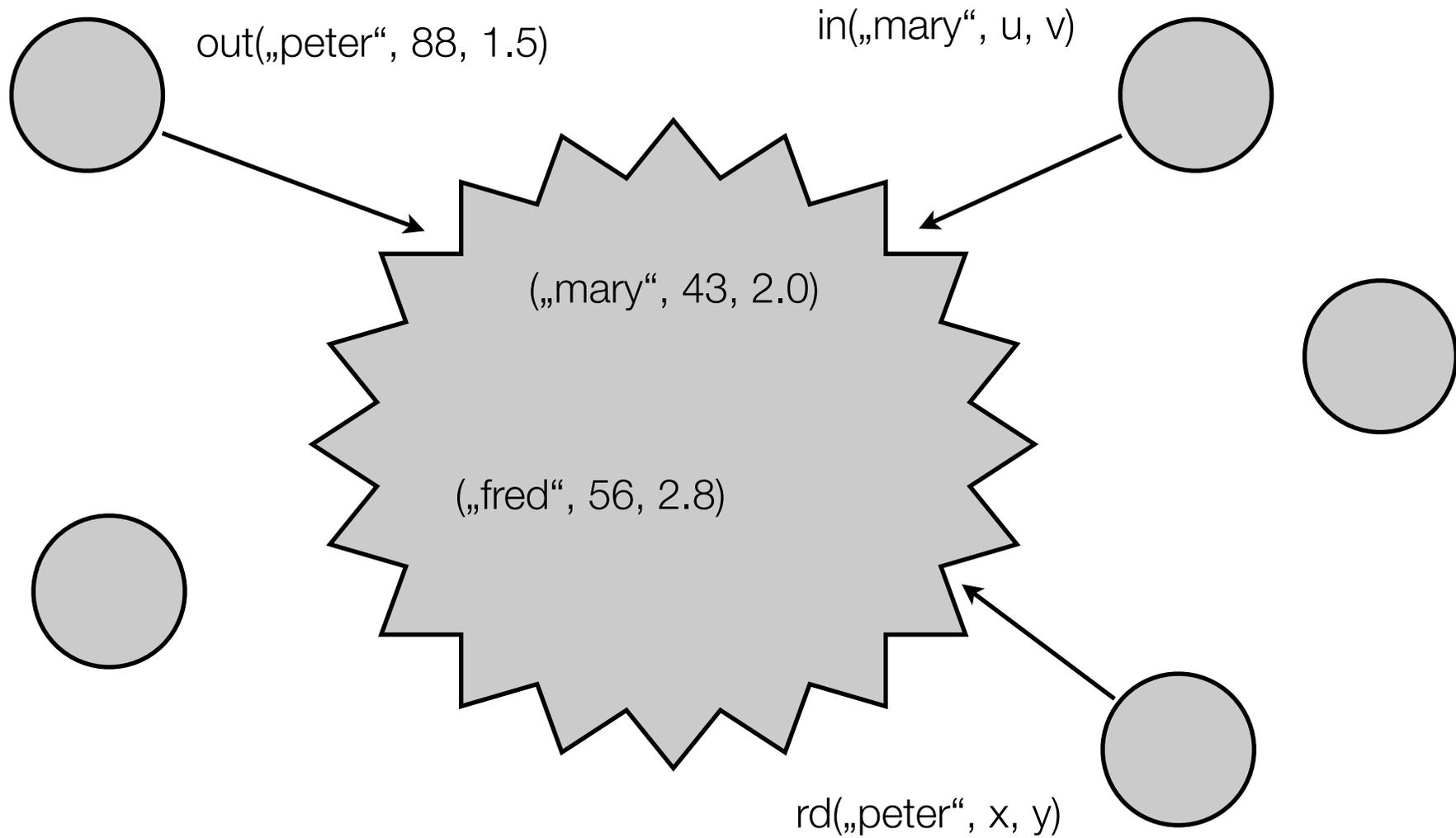
Apple Grand Central Dispatch

- 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

Linda Model

- Concurrent programming model, developed in Yale University research project
- Tuple-space concept
 - Abstraction of distributed shared memory
 - Set of programming language extensions for facilitating parallel programming
 - Tuple: Fixed fixed-length list containing elements of different type
 - Associative memory - tuples are accessed not by their address but rather by their content and type
 - Destructive (*in*) and nondestructive (*rd*) reads
 - Sequential programs embed insert/retrieve tuple operations
- Multiple implementations (LindaSpaces, GigaSpaces, IBM TSpaces, ...)

Tuple Spaces



```

procedure manager
begin
    count = 0
    until end-of-file do
        read datum from file
        OUT("datum",datum)
        count = count+1
    enddo
    best = 0.0
    for i = 1 to count
        IN("score",value)
        if value > best then best = value
    endfor
    for i = 1 to numworkers
        OUT("datum","stop")
    endfor
end

```

```

procedure worker
begin
    IN("datum",datum)
    until datum = "stop" do
        value = compare(datum,target)
        OUT("score",value)
        IN("datum",datum)
    enddo
end

```