Parallel Programming

Summary of the course in spring 2009 by Thomas Gross

Stefan Heule

2010-10-15

1 Java concepts

1.1 Access modifiers

In Java, there are various access modifiers which allow you to control who can access an attribute:

- (default): accessible in the current package
- public: accessible from everywhere
- private: accessible only from this class
- protected: accessible in the current package and all subclasses, regardless of their package

1.2 Interfaces, abstract classes and inheritance

In Java you have multiple ways to reuse your code:

1. **Interfaces**: With interfaces you have the possibility to define the skeleton of a class with method signatures and constant declarations, but nothing more. Your class may implement as many interfaces as you wish.
2. **Concrete class** with inheritance: You may inherit from at most one superclass, but now can reuse the method implementations (not just the declarations) of this class.
3. **Abstract classes**: An abstract class is like an ordinary class, which cannot be instantiated. Additionally, you do not need to provide an implementation for all methods, but can only give a method signature and therefore forcing your clients to implement such a method.

1.3 Exceptions

1.3.1 Introduction

Whenever an exceptional event occurs, we can throw an exception: throw(new Exception()). This exception then gets passed through the call stack, looking for an appropriate hander. To catch an exception and do some useful error handling, Java provides a catch clause: catch(ExceptionType name). Any exception that is of type ExceptionType or a descendent thereof will be caught and can be handled.

1.3.2 Exception types

There are two types of exceptions:

1. **Checked exceptions**: Exceptions a program wants to recover from, like bad user input (e.g. non-existing file). This kind of exceptions must be handled by the program.
2. **Unchecked exceptions**
    a. **Errors**: Exceptions of this type are beyond the program's control, like hard disk errors.
    b. **Runtime exceptions**: Violations of the program logic, essentially bugs, like null pointer access. The program could handle them, but it's better to fix the bug.

Often it is necessary to clean up at the end of a *try*-Block, regardless whether an exception was thrown. This is when *finally* comes in handy: It will always be executed at the end of the *try*-statement, even if an exception is thrown.

1.3.3 Exception handling and control flow

The control flow of a *try*-statement works as follows:

```
try {
        // code
} catch (Type1 e) {
        // handle exception
} catch (Type2 e) {
        // handle exception
} finally {
        // finally block
}
```

- The code in the try-statement is executed. If everything goes as expected, we directly proceed with the last step. Otherwise we throw an exception of a certain type and leave this block immediately.
- The *catch* clauses are inspected from first to last. The first that matches our exception type (that is the two types match directly), or the thrown exception is a subtype of the caught exception. If no catch clause matches, we proceed with the next step but will pass the exception on the caller of the current method.
- At the end, the finally block is executed.

As an important consequence of the way the catch clauses are processed, theses clauses should always be ordered from **most specific to most general**, otherwise an exception may be handled by a clause that works, but is not ideal.

1.3.4 Nested exceptions

In the catch and finally clauses it is possible to throw exceptions, too. Java specifies that always the **last unhandled exception thrown in a try block** will be the one passed on. For instance, suppose a try block that throws an exception A that can be handled in a catch block. In that catch block, a second exception B is raised. The control flow gets passed to the finally statement, where yet another exception C is thrown. Now, exception A is handled, B gets discarded and exception C will be passed to the caller for handling.

1.3.5 Announcing exceptions

To ensure that every checked exception has a matching handler, every method that can throw an exception must announce this behavior.

```
void someMethodWithAVeryLongNameToFillUnusedSpace() throws ExceptionType { /**/ }
```

2 Basic problems and concepts in parallel systems

2.1 Race conditions and stale values

A race condition is a flaw in a system whereby the output/result of an operation is unexpectedly and critically dependant on the sequence or timing of other events *[definition loosely taken from Wikipedia]*. For instance if two threads execute *x = x+1*, we have no way to know what will happen. It is even possible that for a very long time, nothing bad happens, and suddenly after a totally unrelated change in the software, updates to *x* are missed. Additionally, the compiler and various other things may influence the probability that a race occurs.

To prevent such behavior and force certain blocks of code to be executed atomically, we need synchronization.

2.2 Synchronization with locks

In Java, every object can serve as a lock, which is known as *intrinsic locks*. With locks we can enforce synchronization: The lock is acquired before a critical section and released afterward. Such a critical section will be executed mutually exclusive: only one thread at a time can be in a critical section guarded by the same lock.

In Java, locks are **reentrant**. That means, if a lock is acquired that is already held, the operations succeeds. This is necessary when overwriting methods where the overwritten method is called.

```
// synchronize just parts of a method with
// any object as lock
void someMethod() {
       /* normal code */
       synchronized (lock) {/* protected code */}
       /* normal code */
}

// synchronize whole method body with "this"
void synchronized someMethod() { /**/ }
```

Java also provides a very easy way to use the intrinsic locks of any object: *synchronized*. Any object then can serve as a lock. If a whole method body needs to be guarded with "this" as lock, then the whole method can be made *synchronized* as a shorthand notation.

2.3 Visibility

A consistent view of all data is only guaranteed **after synchronization**. Additionally, for fields it is possible to ensure visibility with the *volatile* keyword. This makes all updates visible globally; note that it does not guarantee atomicity, though. There even is no guarantee about the time when we see the changes, but once we see them, all prior changes to any variables will be visible as well.

2.4 Condition queue

In a parallel system it may be necessary to wait for a certain condition to become true. Instead of using a busy loop, which unnecessarily consumes resources, *sleep* may be used. But this approach brings some problems along:

```
Thread.sleep(upperBound)
```

- There is not enough control on how long a thread sleeps. It is possible to provide an upper bound to the sleeping time, but there is no guarantee, and a thread can even be interrupted earlier.

- If we want to wait for a certain condition to be met, we have no way to know for how long we have to wait.
- If we want to wait in a synchronized block, this can be difficult: If we don't release the lock before going to sleep, no other thread can make any progress and our condition will probably never change.

These problems are solved with a **condition queue**. Like every object can serve as a lock, every object can serve as a condition queue. The interface of such queues looks as follows:

- **wait()**: release the currently hold lock and suspend the current thread
- **notifyAll()**: inform all waiting threads that the object state has changed. As soon as the current thread has released the lock, one thread will wake up and reacquire the lock. If no thread is waiting, a *noop* is performed :)

Intrinsic queues are tightly coupled to intrinsic locks. It is only possible to call these queue methods if the lock is held. This makes sense, since it doesn't make any sense to wait for a condition to be met, if we cannot examine the object state (because we do not hold the lock). Equally one should only notify other threads if a change has been made, which is only possible, if the thread holds the lock.

2.5 Semaphore

A semaphore is an integer-valued object with two main operations:

- p(S): acquire. If S is greater than zero, decrement S by one or suspend the current thread
- v(S): release resource. If there are suspended threads, wake one of them up or else increment S by one.

Both these operations are atomic. The value of S can be initialized at the beginning and after that, the semaphore always fulfills the following invariants:

1. $S \geq 0$
2. $S = S_0 + \#V(S) - \#P(S)$

```
while (true) {
    // non-critical section
    p(S);
    //critical section
    v(S);
}
```

With semaphores it is very easy to get a working solution to the mutex problem, even for more than two threads; no deadlock, no starvation.

The fairness models do also apply to semaphores, and in java one can specify the fairness of the semaphore:

- **Blocked-set** semaphore: One (randomly picked) thread will be woken up.
- **Blocked-queue** semaphore: The suspended threads are kept in a FIFO-queue, and will be woken up in the order of suspension.
- **Busy-wait** semaphore: Value of S is tested in a busy wait-loop. Note that now only the if-statement is executed atomically, but multiple threads may execute their if-statement in interleaved manner. This version has several flaws, including the possibility of starvation.

2.6 Semaphores vs. monitor locks

Both concepts can be used to implement each other; they are in some sense equivalent. However, the use of synchronized methods to implement is significantly easier than the other way round.

```
class Semaphore {
    int s = 0;
    public Semaphore(int s0) { s = s0;}

    public synchronized void p() {
       while (s == 0) wait();
       s--;
    }

    public synchronized void v() {
       s++;
       notifiyAll();
    }
}
```

```
/* wait() is translated into */
C_S++;
S.v(); // release the lock
SCond.p();
~~S.p();~~
C_S--;

/* notifyAll() looks like */
if (C_S > 0) SCond.v();
else S.v();
```

To implement monitors based on semaphores, a little more thought is necessary: To control access to critical section, every lock needs a semaphore S. All synchronized blocks are then framed with *S.p()* at the beginning, and *S.v()* at the end. Another semaphore *SCond* is used to queue threads that released their lock because a condition is not met (*wait()*), and we must explicitly maintain a counter *C_S* to find out how many threads are waiting for a lock and then evaluating some condition (the threads sleeping after *wait()*). The methods *wait* and *notifyAll* are then translated into the code to the right.

Note that a call to *notifyAll* does also release the lock, so this method can only be called at the end of a synchronized block. In the code for *wait()* one may be tempted to acquire the lock by calling *S.p()* (crossed out in red), but that does not work! If the lock is released, another thread than the currently woken up thread will acquire the lock. Therefore the lock is directly transferred to the woken up thread, and this thread can directly proceed.

Important: Please note that this implementation presented in the lecture is complete and utter non-sense. A thread awakened with *notifyAll* does not have any privileges whatsoever and has to compete in the usual manner with any other threads that might be actively competing to synchronize on this object. Furthermore the given implementation is in no way equivalent to real Java locks.

2.7 Thread management

2.7.1 Thread states

A thread is at all times in one of the following states:

- `NEW`: The constructor has been finished, but no one called *start()* yet.
- `RUNNABLE`: Running or available to run.
- `TERMINATED`: Execution has been completed, either by normal exit or an uncaught exception.
- `BLOCKED`: This thread is waiting for a monitor lock.
- `WAITING`: The thread waits for a condition in an intrinsic queue.
- `TIMED_WAITING`: The thread has decided to sleep for a certain amount of time. This can happen when *sleep()* or *join()* is called.

3 Reasoning about parallel programs

3.1 Correctness

To reason about the correctness of a parallel program, we use two kinds of properties:

- **Safety properties**: These properties must always be true. For instance, mutual exclusion is such a property. A better definition may be: *Something bad never happens; that is, that the program never enters an unacceptable state.*
- **Liveness properties**: This kind of properties must eventually be true, but they make no statement about how long it will remain true. It may either now or at some future time be true. A better definition may be: *Something good eventually does happen; that is, that the program eventually enters a desirable state.*

Often the term **thread-safety** is used when talking about correctness in parallel systems: A thread-safe object ensures that the state of that object is always valid when being observed by other objects and classes, even when used concurrently by multiple threads.

3.2 Deadlock and Starvation

For a parallel system to be useful, some properties, or rather the absence of these, are very important. These properties especially are important for a mutex solution:

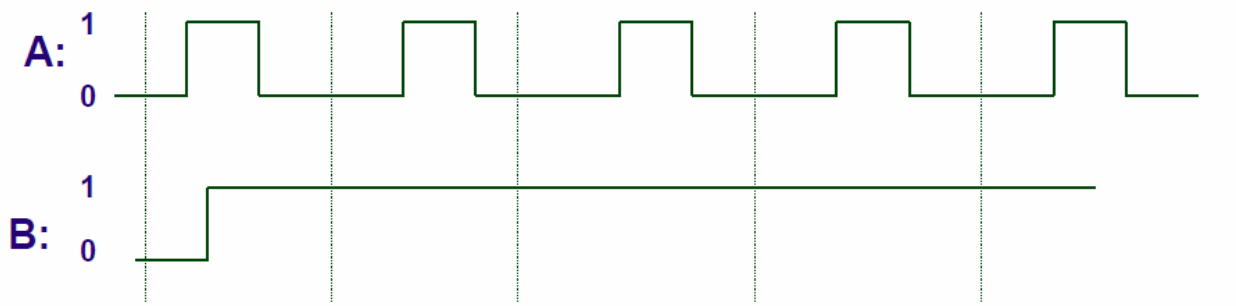
1. **Deadlock**: If no thread is able to make progress, we speak of deadlock. The absence of deadlock is clearly a safety property, but it does not imply that the thread actually makes progress. It just states, that the thread is not frozen. It very well be possible that only other threads make progress.
2. **Starvation**: If a thread wants to enter its critical section, eventually it will succeed. Note that the absence of starvation implies the absence of deadlock. In practice however, the absence of starvation is not always needed. In particular, if starvation is merely a theoretical issue, an algorithm that gives no guarantee about starvation may be used.
    a. **Livelock**: The livelock is a special case of starvation. The states of the processes involved in the livelock constantly change with regard to one another, none progressing.
3. **Undue delays**: If there is no contention, a thread that wishes to enter its critical section will succeed. The overhead should be small in the absence of contention.

3.3 Fairness

If multiple threads attempt to perform an action (e.g. acquiring a lock), different models of fairness may apply. If *n* threads compete for the same resource, we speak of **contention**. The fairness model now tells us who will be served first if there is contention:

- **Weak fairness**: If a thread continuously makes a request, eventually the request is granted.
- **Strong fairness**: If a thread makes a request infinitely often, eventually the request is granted.

To see the difference between weak and strong fairness, you may consider the following setup:



Threads A and B both want to access some resource, and they indicate this by setting a flag to 1 (the line in the picture above represents the value of this flag). With a weak fairness model, thread A is not guaranteed to be served. If you imagine a service provider checks at every vertical line, it may not “see” the request of the thread.

There are even more fairness models:

- **Linear waiting**: If a thread makes a request, the request is granted before any other thread is granted the request more than once.
- **FIFO waiting**: If a thread makes a request, the request is granted before a later request by another thread is granted.

4 Solutions to the mutual exclusion problem (critical section)

4.1 Ping pong, first attempt

There is a shared volatile variable *turn* that indicates, who can go into its critical section. Both threads get an ID, and if they want to enter their critical section, they wait as long as *turn* is not equal to their ID. Then, after they have executed their critical section, the thread changes *turn* so that the other thread can go on.

```
while (true) {
        // non-critical section
        while (turn != myID);
        //critical section
        turn = 1 - turn;
}
```

This attempt satisfies the mutual exclusion property, and there is no starvation (and therefore no deadlock). However, in the absence of contention, this attempt may fail.

4.2 Second attempt

The same idea is used, but now both threads do have an own volatile variable: *request0* and *request1*. *request* set to 1 indicates that this thread does not want to enter its critical section. The enter protocol works as follows:

- As long as the other thread wants to enter the critical section, we wait.
- Then we tell the world we want to be in the critical section by clearing our request flag.
- After executing we reset our flag to 1.

```
// code for thread with ID 0
while (true) {
        // non-critical section
        while (true)
                if (request1 == 1)
                        break;
        request0 = 0;
        //critical section
        request0 = 1;
}
```

This attempt does not even provide mutual exclusion, because the request flag is set too late. This directly leads to attempt three.

4.3 Third attempt

Same approach as above, but the flag *request* is set to 0 before the second while loop. Mutual exclusion is now guaranteed, but deadlock is possible.

4.4 Fourth attempt

The third attempt is extended further: In the inner while loop, when we see that the other thread wants to enter its critical section, we back off by setting our flag to 1, and right away again to 0. This way, no deadlock can occur, but a thread may starve. Fixing this last issue leads to Dekker's solution, the first known solution to the mutual exclusion problem in concurrent programming.

4.5 Dekker's solution

Dekker's solution combines the fourth attempt with attempt number 1, ping pong.

```
nextid = 1 - myid;
while (true) {
    mysignal.request();
    while (true) {
       if (othersignal.read() == 1) break;
       if (mysignal.getWinner() == nextid) {
          mysignal.free();
          while (true) {
             if (mysignal.getWinner() == myid)
             break;
          } // other thread won 1st round
          mysignal.request();
       }
       // critical section
       mysignal.free();
       mysignal.setWinner(nextid);
    } // end loop
}
```

5 CSP

CSP, *communicating sequential processes* is a model for synchronous communication with global channels. These global channels are the only state that is shared between threads and are used to transmit data. Because communication happens synchronously and without buffering, synchronization is implied. That means, in the CSP world, there is no need for locks or semaphores.

The fact that reading a channel blocks if no data is available, **guarded commands** may be useful. Multiple guards with associated commands can form a select clause with the following properties:

- If only one guard evaluates to true, the corresponding block is executed.
- If multiple guards evaluate to true, one of the associated blocks is chosen randomly.
- It may be necessary to wait if no guard evaluates to true, but if all guards evaluate to false (e.g. channel closed), the command fails.

```
// abstract select clause
select {
    guard1 -> command1;
    guard2 -> command2;
}
```

5.1 Buffering in CSP

If buffering is needed in CSP, one has to implement its own buffer, because hidden buffering may be dangerous. The problem of hidden buffers is that they are bounded and the clients may not know enough about the size of these buffers. If they make wrong assumptions, the program may work on one system, but fail on another (with a different buffer size). Consider the code example to the right. Imagine a hidden buffer of size 128. Everything works, but if the buffer offers only space for 64 items, the producer will not be able to send *count* over channel 2.

```
// producer
while (count < 100) {
    put(ch1, b[count++]);
}
put(ch2, count);


// consumer
get(ch2, count);
while (j < count) {
    get(ch1, b[j++]);
}
```

Note: In my opinion this is not really a problem of hidden buffering but of this particular code: Sending the number of items this way is always wrong! It does not work at all without buffering, and even with buffering, it only works if the buffer is bigger than the total number of items. So either buffer the items locally in the producer, or use a second channel to signal that there is no more input (used with guards / select). But this is an example that you need to be careful when working with (hidden) buffers.

5.2 JCSP

In Java JCSP can be used for CSP style communication through channels. The units of computation are of type *CSProcess*, an interface of JCSP. The only thing in that interface is a public method *run*.

The communication happens through different kinds of channels:

- *One2OneChannel* and *One2AnyChannel*
- *Any2OneChannel* and *Any2AnyChannel*

Each channel has two "ends", *ChannelOutput* and *ChannelInput*. A program can write to the first type of end, while reading from the *ChannelInput*.

Starting the processes works with class *Parallel* that expects an array of processes as parameter. The method *run* then starts all the threads.

Choosing between input channels works with guards, as one would except from a CSP library: Any input channel can serve as guard, one just need to change its type to *AltingChannelInput*. As shown to the right, the *Alternative* class makes it possible to have a *select* clause, through the use of *switch*. There is also a special input channel *skip* that is always ready. To change the behavior of the *select*, one may use different select methods:

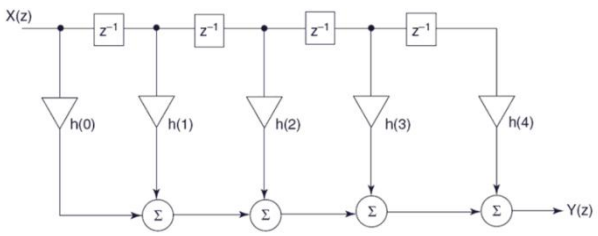
```
// a process
class MyWorker implements CSProcess {}

// a channel
One2OneChannel chan = Channel.one2one();

// reading and writing
ChannelInput in = chan.in();
ChannelOutput out = chan.out();
out.write(val);
val = in.read();

// starting processes
new Parallel
(
    new CSProcess[] {
        new MyWorker(),
        new MyWorker()
    }
).run();

// choosing between input channels
AltingChannelInput in, request;
Guard[] guards = {in, request, skip};
Alternative alt = new Alternative(guards);
switch (alt.select()) {
    case 0: .. break;
    case 1: .. break;
    case 2: .. break;
}
```

- select: works as introduced in CSP, that is, picking a channel at random.
- priSelect: priority select; the order matters.
- fairSelect: provides fairness, meaning that no channel is selected a second time before all other channels that are ready have been read at least once.

5.3 Examples of CSP

5.3.1 Finite impulse response filter

For a given stream of data, the computation of a finite impulse response filter may be done using the CSP model for communication. Each thread is responsible for the calculation with one weight and the data stream elements are then passed through the chain of threads.



6 Problem decomposition

When given a problem that should be solved using a parallel system, one has to think about how to decompose the problem. Two principal approaches exist:

- **Data partitioning**: Focus on data, each thread works on a subset of the data, often with identical instructions.
- **Computation / task partitioning**: Focus on computation, each thread gets a different task.

There is no general way to tell what is better. The decomposition depends heavily on the problem itself, but also the parallel system influences what is best.

6.1 Data partitioning

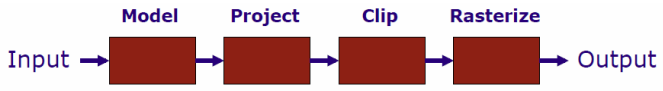
Data partitioning is rarely possible without communication, in general threads may need access to some shared date and/or produce data for other threads.

The major issues one needs to deal with concern identifying a good partitioning (all threads should take about the same amount of time) and handle the communication

6.2 Task partitioning

There are a number of possible structures on how to partition the computation:

- **Independent computations**: There are a number of completely different, independent tasks that can be carried out by different threads.
- **Pipelining**: Each thread performs one stage of the computation, for example some filter computation (see image).

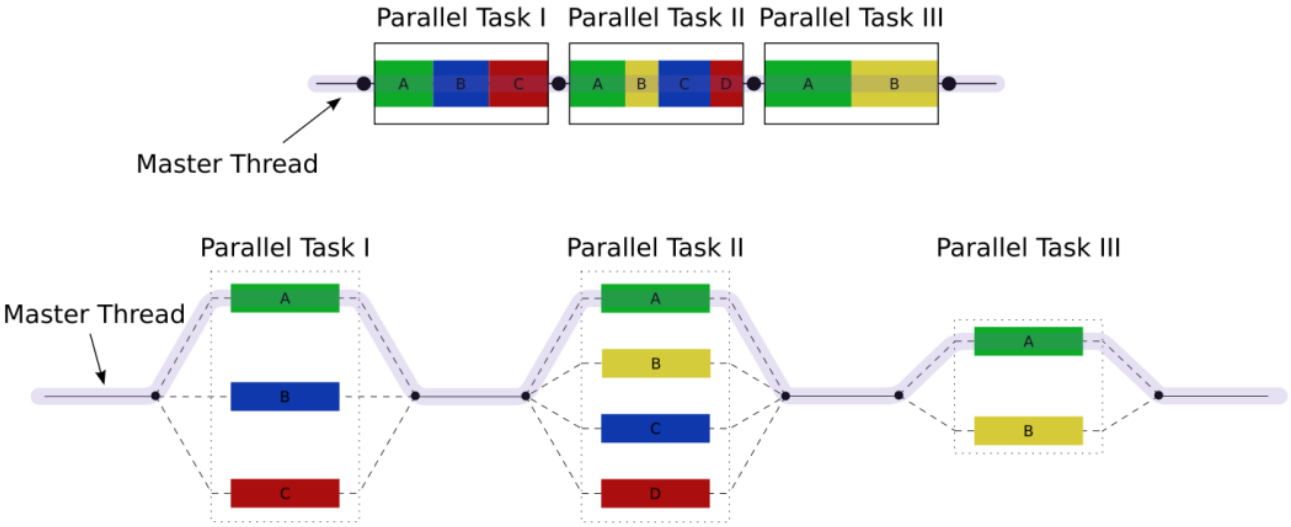


- **Task queue**: Various activities, usually of different execution time, are assigned to threads.

Parallel solution is easy to find	Difficultly to parallelize or impossible
Large data sets	Small data sets
No dependences (e.g. dense matrices)	Many (non-obvious or value-based) dependences (e.g. sparse matrices)
Computation time can be easily estimated and is a function of the size of input data.	Computation time difficult to estimate and/or dependant on the value of the input data.

7 Fork-Join model and OpenMP

In the fork-join programming model, one thread is called the master thread. At the begin of the pro-



gram, only this master thread is active and executes all sequential parts of the program. Whenever a task can be executed in parallel, the master thread forks, that is creates or awakens, additional threads. At *join*, the end of parallel sections, these extra threads are suspended or die.

This model is a special case of the general thread model, where at the begin only the main thread is active, which can spawn additional threads if needed. The general model however is much more flexible and has better support for task parallelism. On the other hand the fork/join model is more structured and therefore may be easier to optimize.

7.1 OpenMP

OpenMP is an industry standard based on the fork/join model. The user can mark the parallel section and add proper synchronization. The system then handles all the nasty details of the parallelism.

7.1.1 Parallel section

omp parallel starts a parallel region. In this region, the number of threads remains constant, unless the user explicitly changes it. If not stated otherwise, code in parallel regions is executed by all threads.

7.1.2 Basic parallel for loop

A *for*-loop with *omp for* in a parallel section (*omp parallel for* is also possible) is parallelized. The user has little control on how the loop iterations are mapped onto threads, and there are no constraints on order of execution.

7.1.2.1 Restrictions on for loops

Not all loops can be parallelized with OpenMP, there are certain restrictions:

- Init-expression must be *var = lower* where lower is a loop invariant.
- Simple increment, that is only constant increments or decrements are allowed.
- Loop continuation test must be one of *>,<,>=,<=*

7.1.2.2 *Private variables*

All variables are shared among all threads. If a variable should be local to each thread, this has to be specified with *private(list)*. Private variables are undefined at the beginning and end of the loop, so no thread sees the previously defined value for its private variables, nor can he assign a new value to the shared variable. If this behavior is not desired, *firstprivate* and *lastprivate* may come in handy: The first tells the compiler to inherit the value of a private variable from the shared variable on loop entry, while the second assures that the value of the variable of the sequentially last loop iteration is assigned to the shared variable.

7.1.3 single

The single pragma is used inside parallel regions to specify that following block should only be executed by one thread. The execution may not be done by the master thread.

7.1.4 Implicit barriers and nowait

After a parallel for loop or a single clause, there is an implicit barrier. That is, no thread can proceed until all other threads have finished their job. This barrier can be eliminated by the *nowait* clause.

```
area = 0.0;
//omp parallel private(tmp)
{
    tmp = 0.0;
    //omp for private (x)
    for (i = 0; i < n; i++) {
        x = (i + 0.5)/n;
        tmp += 4.0/(1.0 + x*x);
    }
    //omp critical
    area += tmp;
}
pi = area / n;
```

7.1.5 Number of threads

The number of threads is normally determined at startup, or the user selects a specific number of threads with *OMP.setNumThreads()*.

7.1.6 Critical sections

Critical sections are executed in a mutually exclusive manner and can be marked with the *critical* pragma.

7.1.7 Reductions

OpenMP provides a *reduction(operator:list)* clause that eliminates the need to create private variables and dividing a computation (into accumulations of local answers that contribute to a global result) by hand.

From the OpenMP specification:

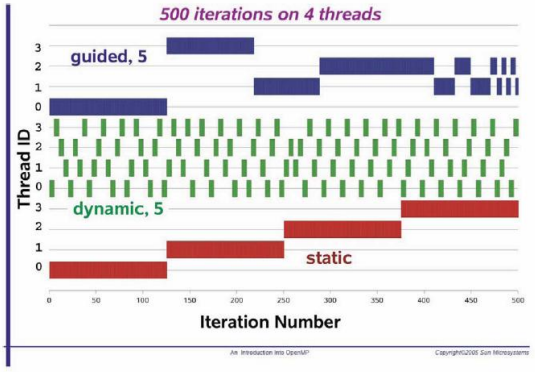
"The *reduction* clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator."

```
area = 0.0;
//omp parallel for private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

7.1.8 Scheduling

Scheduling controls how the iterations of a parallel loop are assigned to threads. The clause has the form *schedule(type [,chunk])*, where *type* is one of the following:

- **static**: The threads are statically assigned chunks of size *chunk*. The default value for *chunk* is *ceiling(N/p)* where *N* is the number of iterations and *p* the number of threads.
- **dynamic:** Threads are dynamically assigned chunks of size *chunk*, that is whenever a thread is ready to receive new work it is assigned the next pending chunk. Default value for *chunk* is 1.
- **guided**: Similar to *dynamic*, but the size of chunk decreases and is approximately
  (#remaining iterations) / (2 (#threads))
- **runtime**: Indicates that the schedule type and chunk are specified by the *jomp.schedule* system property. Only useful when debugging.

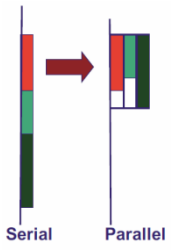


static	Predictable and similar work per iteration
dynamic	Unpredictable, highly variable work per iteration
guided	Special case of *dynamic* to reduce scheduling overhead

7.1.9 Task parallelism

Task parallelism can be expressed with OpenMP through *sections*.

```
//omp parallel sections
{
    //omp section
    phase1();
    //omp section
    phase2();
}
```



8 Linearizability

A **linearizable object** is one all of whose possible executions are linearizable. Often it is not possible to talk about a linearization point of an operation, but only of the linearization point of an operation for a specific execution. This makes reasoning about parallel programs hard.

To make reasoning easier, method calls are split in two distinct events:

- **Method invocation**. Notation: "*Thread object.method(args)*", e.g. "*A q.enq(x)*"
- **Method response**. Notation "*Thread object: result*", e.g. "*A q: void*", or "*A q: empty()*". Note that the method is implicit.

8.1 History

A **history H** describes an execution; it is a sequence of invocations and responses. In such a history one may find method calls: An invocation and a response where both the thread name and object name agree.

8.1.1 Projections

For a history H there exist **object projections**, denoted *H|obj*. Such a history only contains invocations and responses of the object *obj*. Similarly with **thread projections**: *H|thread* is the history H without all invocations or responses that to not correspond to thread *thread*.

8.1.2 Complete, sequential and well-formed histories

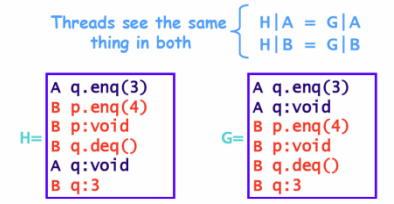
A history is **complete** if there are no pending invocations (invocations without matching response). *Complete(H)* is the history H without all pending invocations.

A history is called **sequential** is no method call is executed interleaved.

A history is **well-formed** if the per-thread projections are sequential.

8.1.3 Equivalent history

Two histories are called equivalent, if the per-thread projections of both histories are equivalent; all threads see the same things in both histories.



8.1.4 Precedence and linearizable histories

A method call **precedes** another if the response event of the first method call precedes the invocation event of the second call. If two method calls do not precede each other, they are said to **overlap**.

For two method calls m_1 and m_2 we say "$m_1 \rightarrow m_2$" if m_1 precedes m_2. The relation $\rightarrow$ is a partial order. If the history is sequential, $\rightarrow$ is a total order.

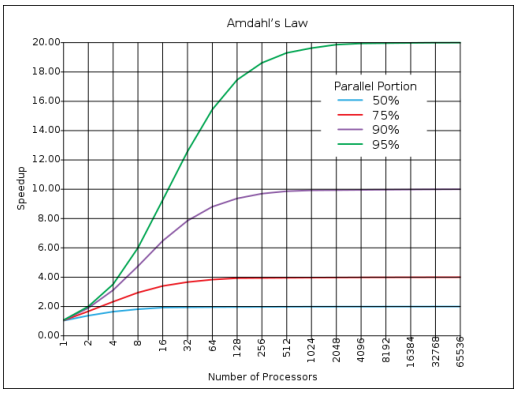
A history H is **linearizable** if it can be extended to G by appending zero or more responses to pending invocations and discarding any other pending invocations, so that G is equivalent to a legal sequential history S where $\rightarrow_G$ is part of $\rightarrow_S$.

9 Small topics

9.1 Amdahl's Law

Amdahl's Law is used to predict the theoretical maximum speedup using multiple processors when only a given part of the program can be parallelized. The speedup on *n* processing units, where *x* percent of the computation are sequential, is given by the following formula:

$$s = \frac{100}{x + \frac{100-x}{n}}$$



9.1.1 Efficiency

Efficiency is a measurement of the processor utilization and is given by the speedup divided by the number of processors: $e = \frac{s}{n}$

9.1.2 Amdahl's Law in practice

In practice, Amdahl's law is too optimistic, for various reasons:

- The overhead produced by the parallelism increases with the number of processors.
- In reality, the amount of work cannot be divided evenly among the processors. This creates waiting time for some processors, another form of overhead.

9.2 Locking data vs. locking code

Locks should be associated with data objects, and thus different data objects should have different locks. Suppose a lock is associated with a critical section of code instead of a data object:

- Mutual exclusion can be lost if same object manipulated by two different functions.
- Performance can be lost if two threads manipulate different objects attempt to execute the same function.

9.3 MapReduce

MapReduce is a parallel computing framework for a restricted parallel programming model. It is useful to distribute work to a farm / cluster of computing nodes. The user specifies what needs to be done for each data item (map) and how the results are combined (reduce). Libraries then can take care of everything else, like parallelization, fault tolerance, data distribution, load balancing, etc.

9.4 Why locking does not scale

In practice, locking does not scale well for a number of reasons:

- **Not robust**: If a thread holding the lock is delayed, no one else can make progress.
- **Relies on conventions**: These conventions often only exist in the programmers mind.
- **Hard to use** (conservative, deadlocks, lost wake-ups)
- **Not composable**: It is hard to combine two lock-based objects (e.g. two queue: how to wait for either queue to be non-emtpy?)