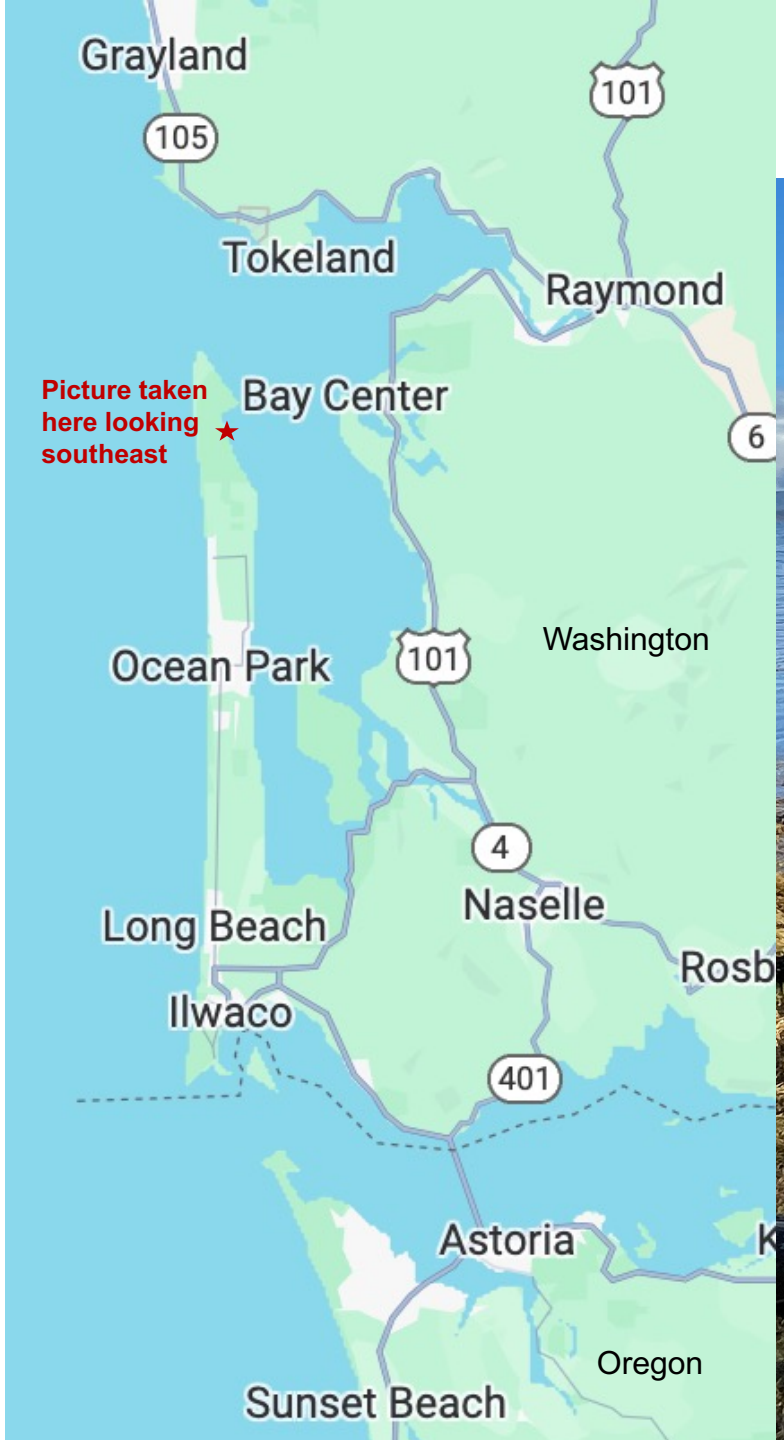
Wrapping-up OpenMP for CPUs and the Fundamental Design Patterns of Parallel Computing



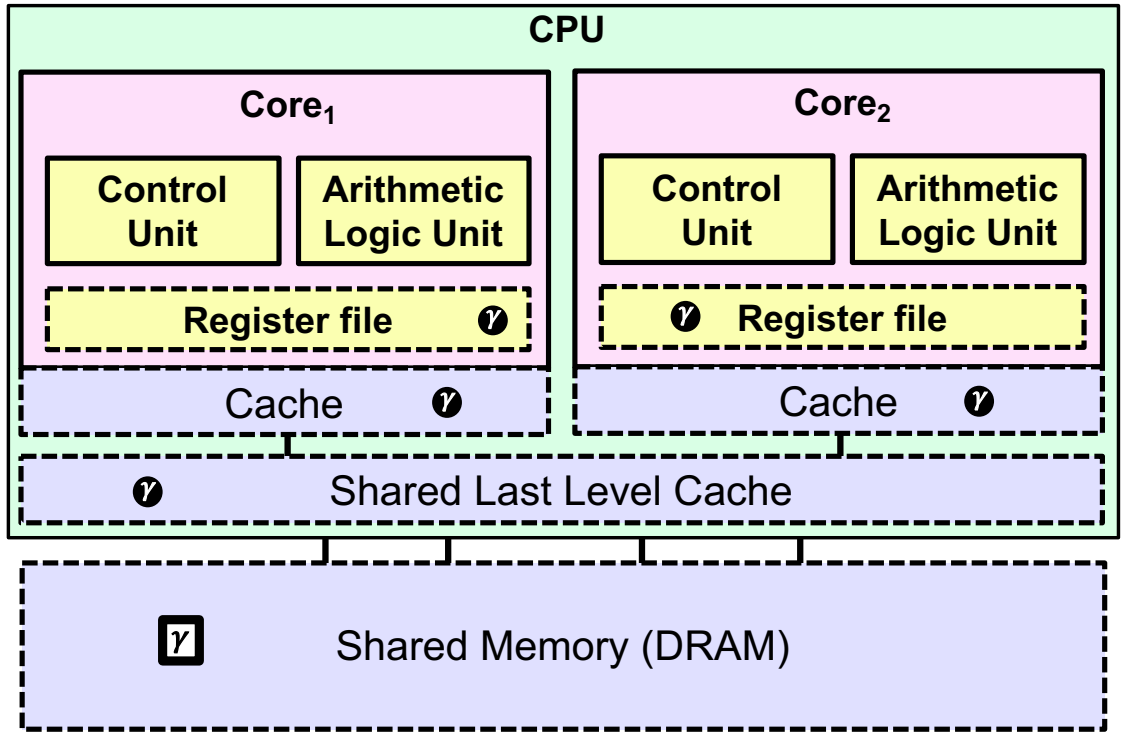
We covered two very important topics last time

Memory Consistency Models (or just memory model for short)

- In a multithreaded programing model, multiple copies of a variable (such as γ) may be present in the memory hierarchy and they may ALL have different values.
- The memory model defines rules to specify which value of a variable (such as γ) will be accessed on a load.
- Two common options mentioned in the literature of concurrency theory:

1. Sequential Consistency

- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.
- **All threads see the same interleaved order of loads and stores**



2. Relaxed Consistency

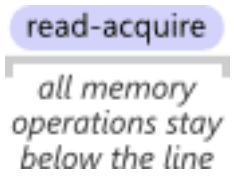
- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.
- **Threads may see different orders of loads and stores**

Most (if not all) multithreading programming models assume **relaxed consistency**. Maintaining sequential consistency across the full program-execution adds too much synchronization overhead.

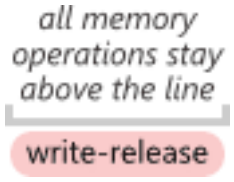
Memory Models: *Synchronized-with* and *Happens-before* relations

- Multithreaded execution … concurrency in action
  - Instructions between threads are unordered except when specific ordering constraints are imposed, i.e., **synchronization**.
  - Synchronization lets us force that some instructions **happens-before** other instructions

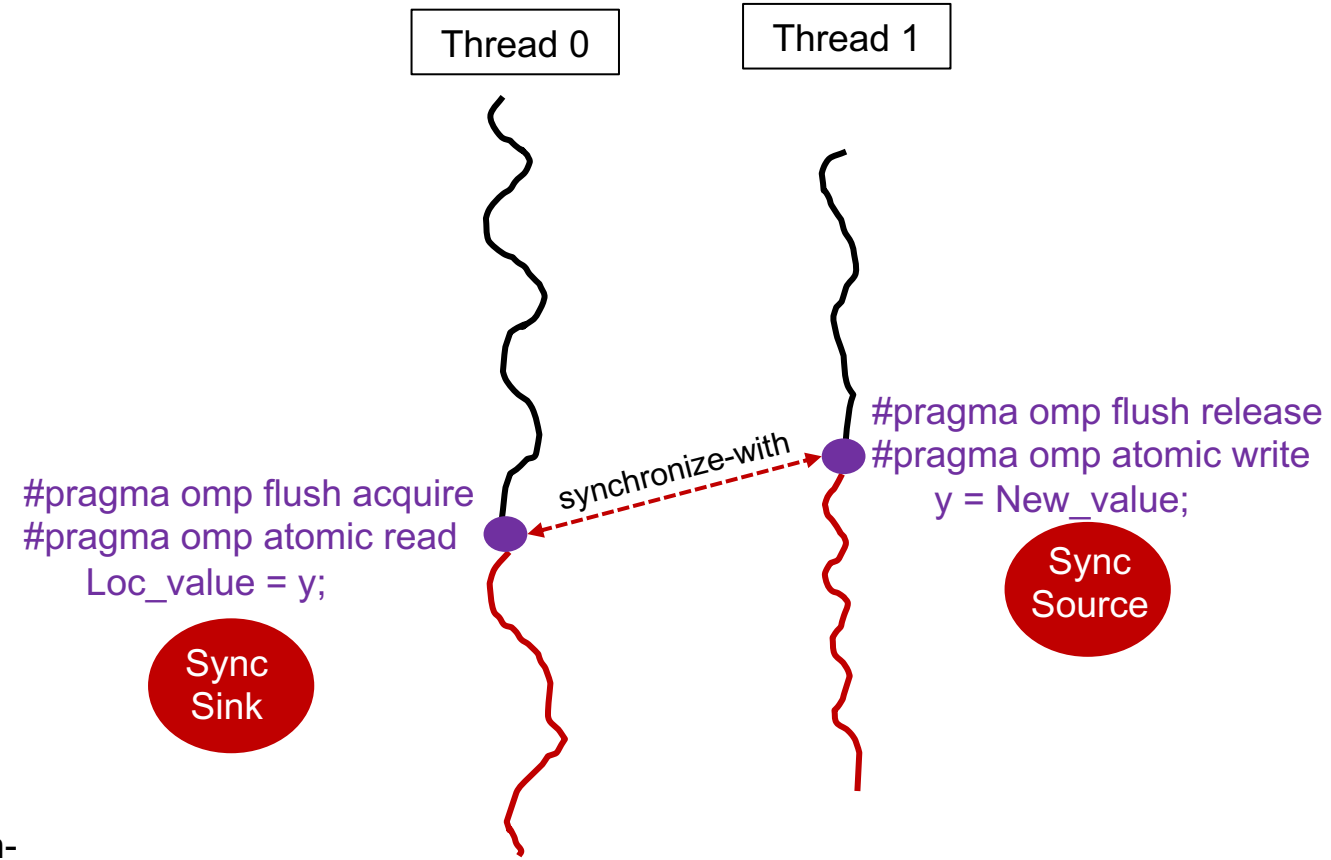
- Two parts to synchronization:
  - A **synchronize-with** relationship exists between a pair of **atomic** operations on a shared variable.
  - **Memory order**: a **flush** statement defines restrictions on loads/stores on either side of a synchronized-with operations.



  - Acquire: Reads/writes that follow the read-with-acquire cannot happen-before the read-with-acquire operation.



  - Release: Reads/Writes prior to the write-with-release must happen-before the write-with-release.

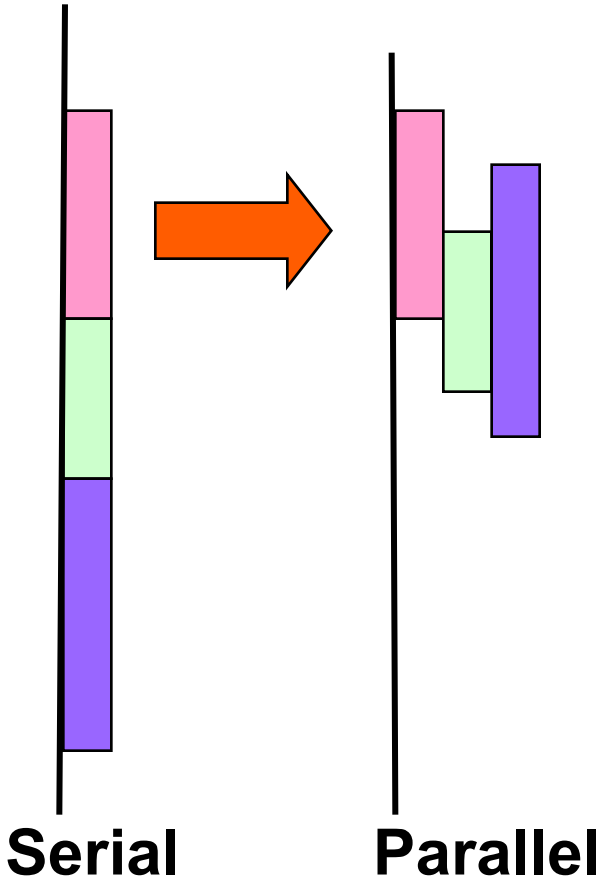


Black operations on Thread 1 happen-before Red operations on thread 0.

Be VERY careful with programming that depends on this sort of synchronization. It is easy to get it wrong and is some of the most error-prone programming we ever do.

Tasks for irregular Parallelism

- Tasks are independent units of work composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters a task construct may execute the task immediately or defer it for later execution.
  - The threads may defer execution until later
- Tasks are often nested: i.e., a task may itself generate tasks.
  - Tasks created from nested task constructs are called **descendent tasks**.
  - Tasks created in the same structured block are **sibling tasks**.
- When to tasks complete?
  - A taskwait causes prior sibling tasks to complete before subsequent task launch
  - All tasks (siblings and all descendants) complete at the next barrier



We have now covered the subset of OpenMP for multithreading that we call the Common Core

The OpenMP Common Core: Most CPU OpenMP programs only use these 21 items

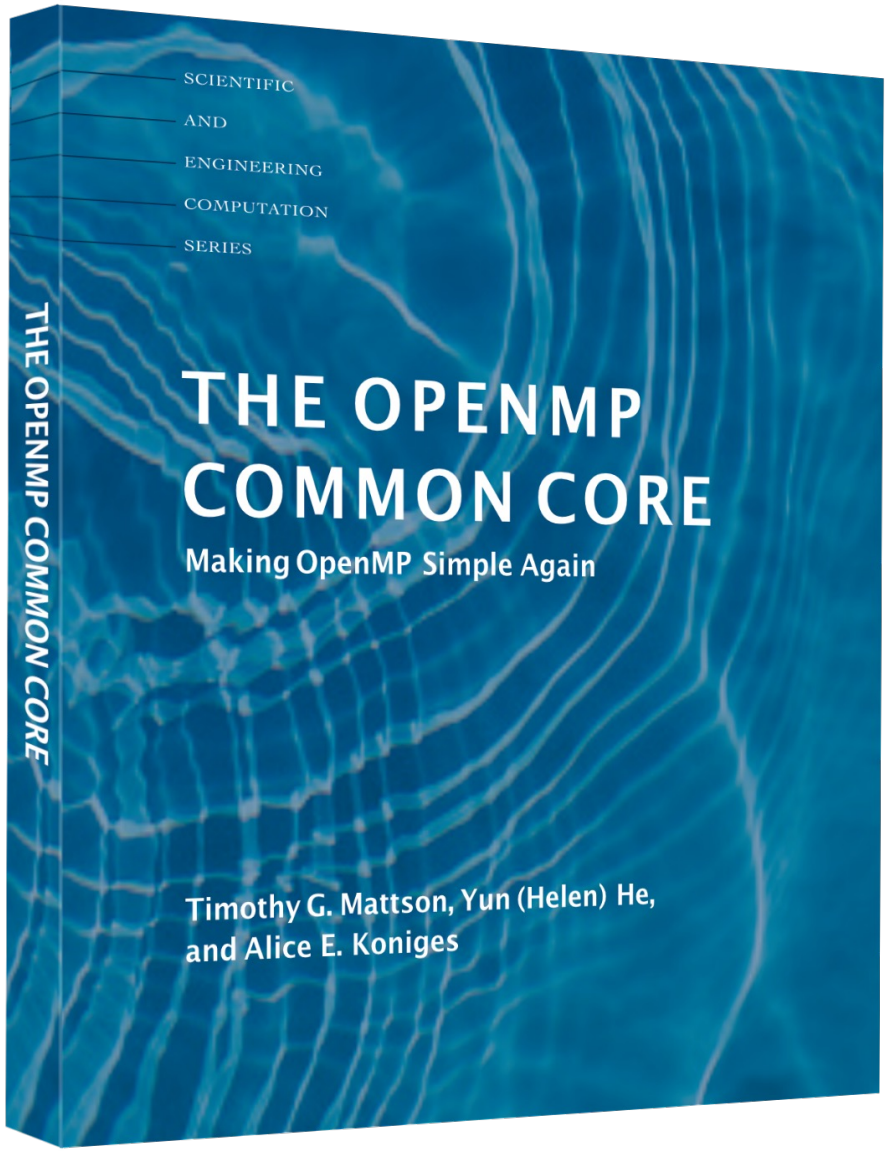
OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

To learn OpenMP:

- A great book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use

- It's geared towards people learning OpenMP, but as one commentator put it … **everyone at any skill level should read the memory model chapters**.

- Available from MIT Press



www.ompcore.com for code samples and the Fortran supplement

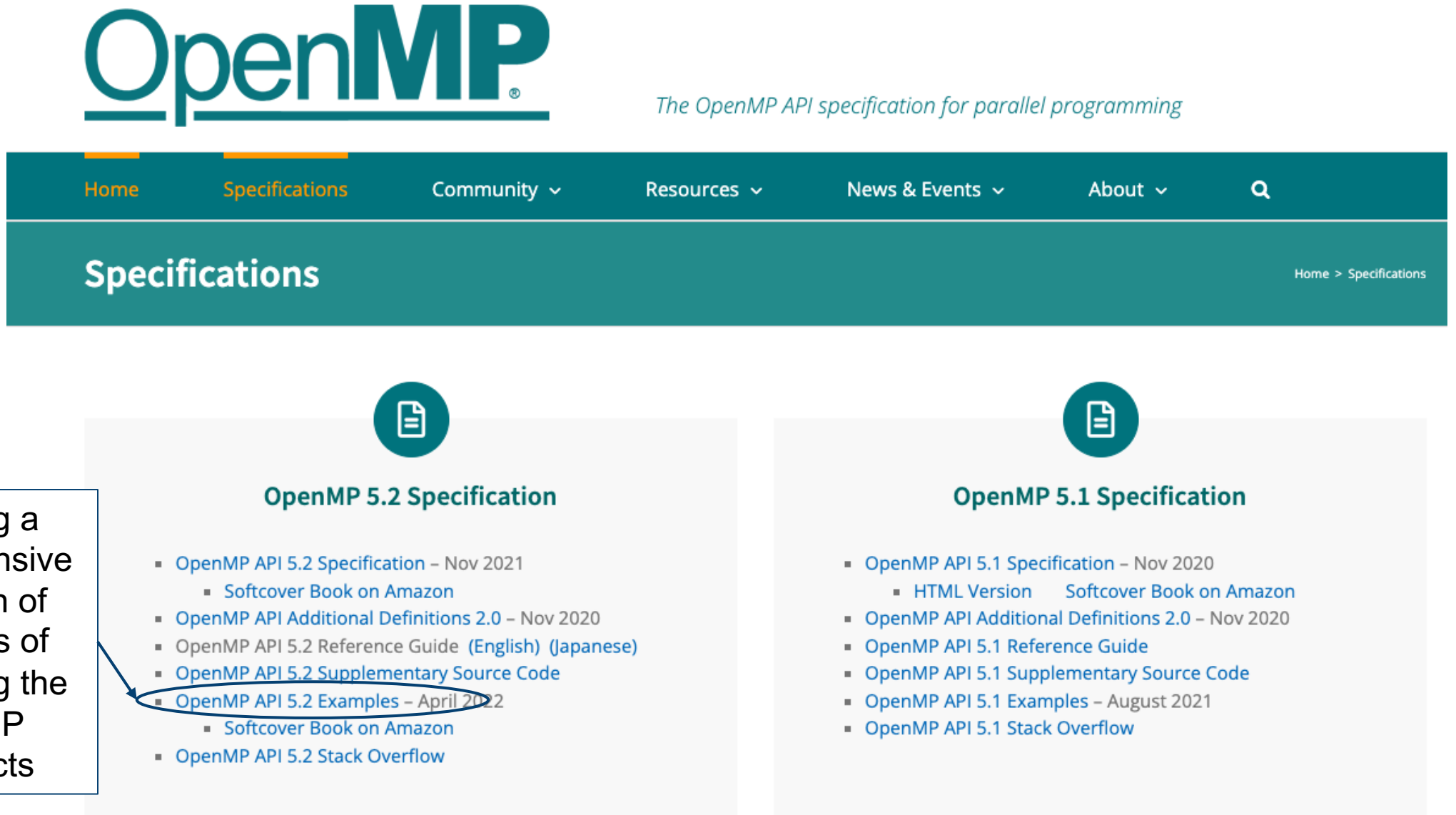
There is Much More to OpenMP than the Common Core

- Synchronization mechanisms
  - locks, synchronizing flushes and several forms of atomic
- Data environment
  - lastprivate, threadprivate, default(private|shared)
- Fine grained task control
  - dependencies, tied vs. untied tasks, task groups, task loops …
- Vectorization constructs
  - simd, uniform, simdlen, inbranch vs. nobranch, ….
- Map work onto an attached device (such as a GPU)
  - target, teams distribute parallel for, target data …
- … and much more. The OpenMP 6.0 specification is over 580 pages!!!

Don’t become overwhelmed. Master the common core and move on to other constructs when you encounter problems that require them.

Resources

- www.openmp.org has a wealth of helpful resources

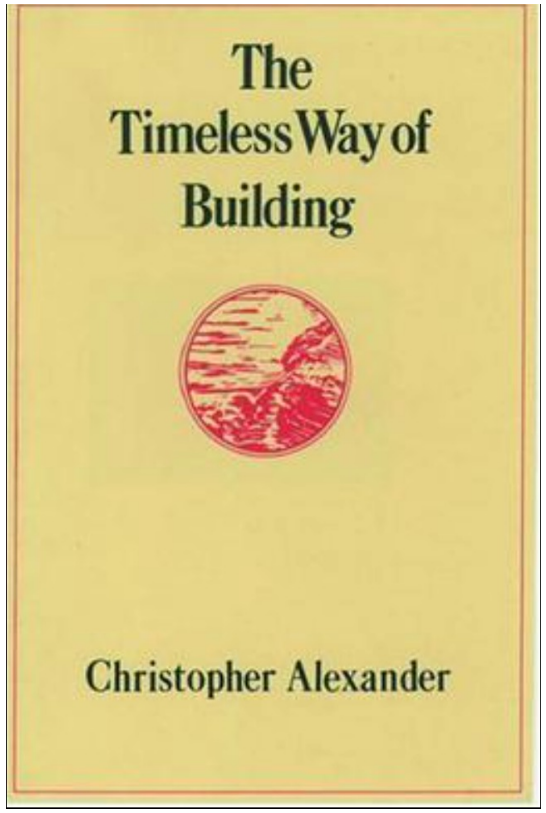


Including a comprehensive collection of examples of code using the OpenMP constructs

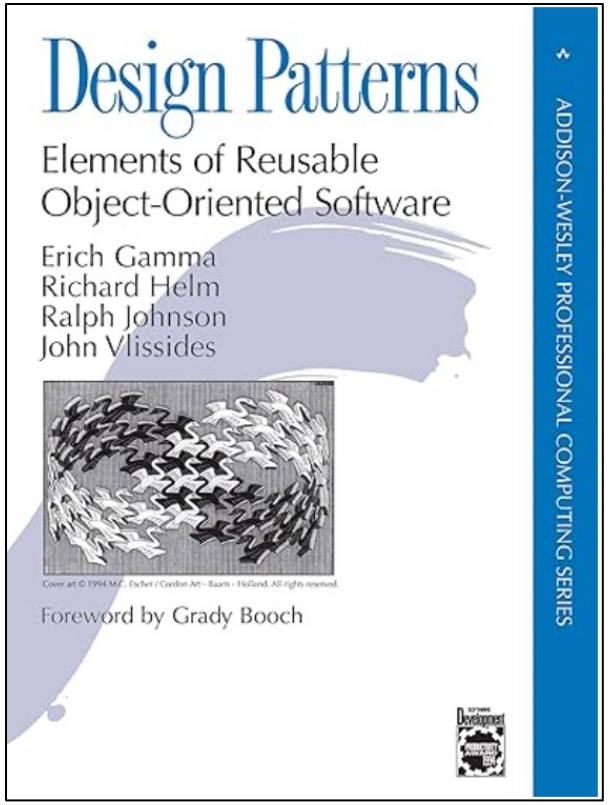
… and we covered many of the most important design patterns of parallel programming

Design Patterns: An Engineering discipline of design

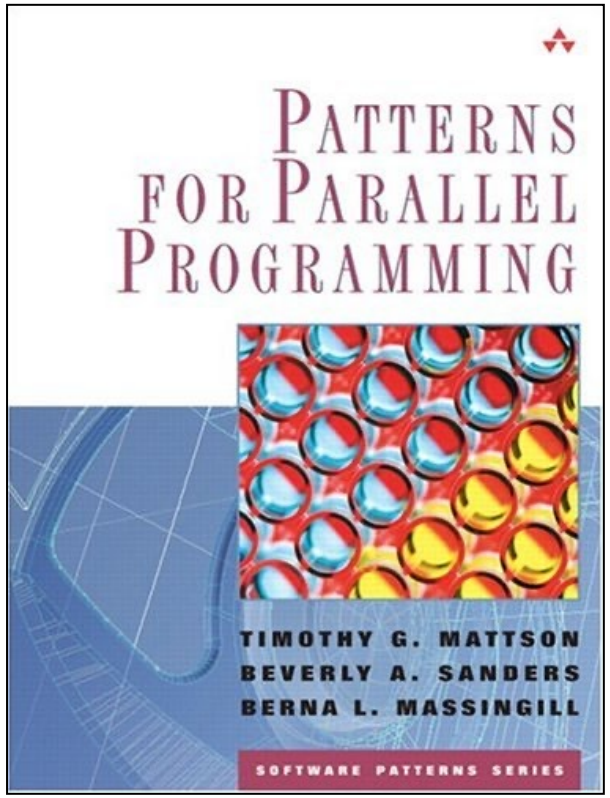
Design patterns encode the fundamental practice of how people think about classes of problems and their solutions.



The idea of design patterns comes from the architect Christopher Alexander in his 1979 book … where he presents design patterns as a way of capturing that essential ”quality without a name” experts see though often struggle to describe.



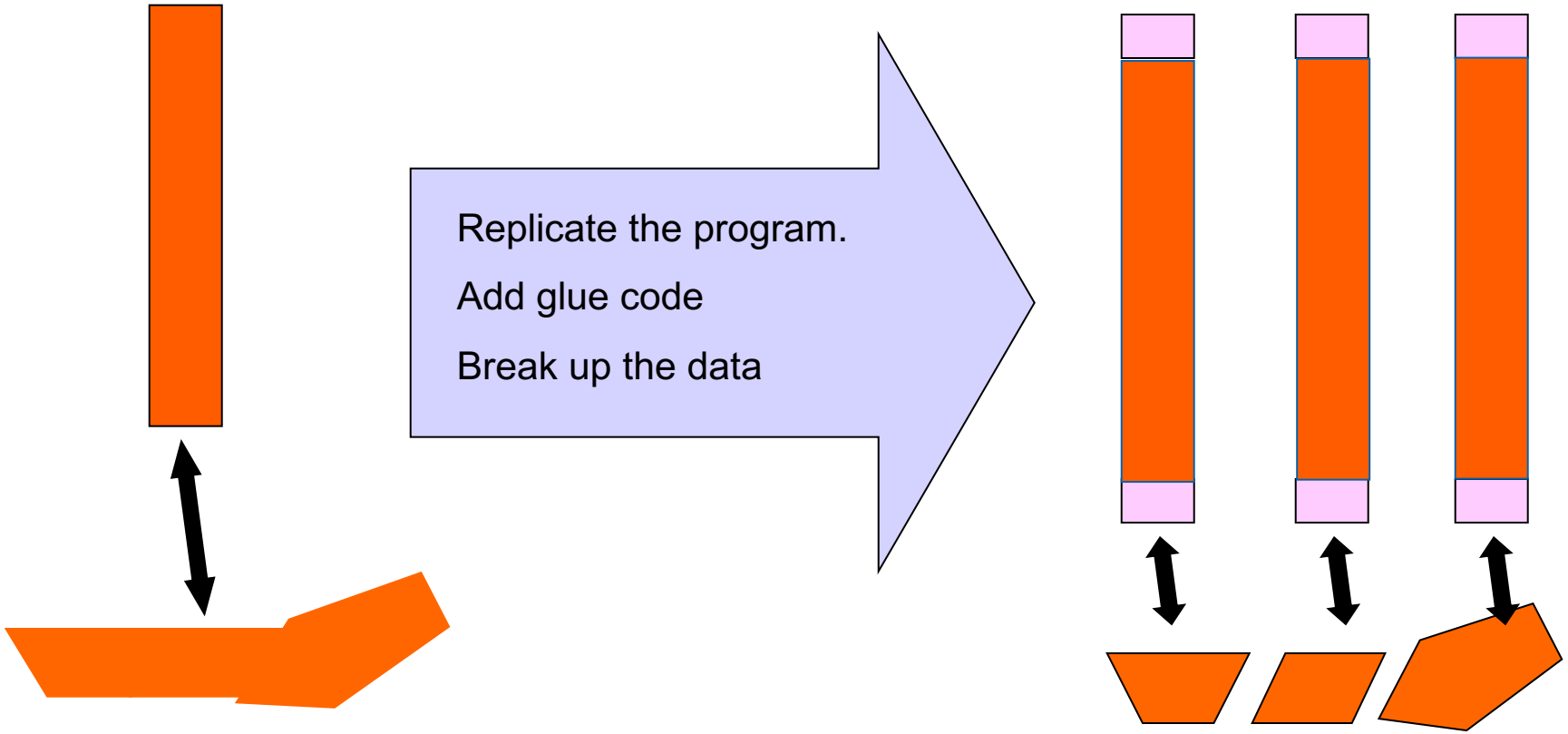
This book released in 1995 brought order to the chaos that characterized the early days of object oriented programming. They used design patterns to systematically organize best-practices in object oriented design. This is one of the most influential books in the history of computer science.



This book (released in 2004) brought the discipline of design patterns to parallel programming. The goal is to help people understand how to ”think parallel” with examples in OpenMP, MPI and Java

SPMD (Single Program Multiple Data) Design Pattern

- Run the same program on P processing elements where P can be arbitrarily large.



- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Example: SPMD Pi program with a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
      int i, id, nthrds;    double x, sum;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       if (id == 0)   nthreads = nthrds;
       for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
             x = (i+0.5)*step;
             sum += 4.0/(1.0+x*x);
        }
       #pragma omp critical
              pi += sum * step;
   }
}
```

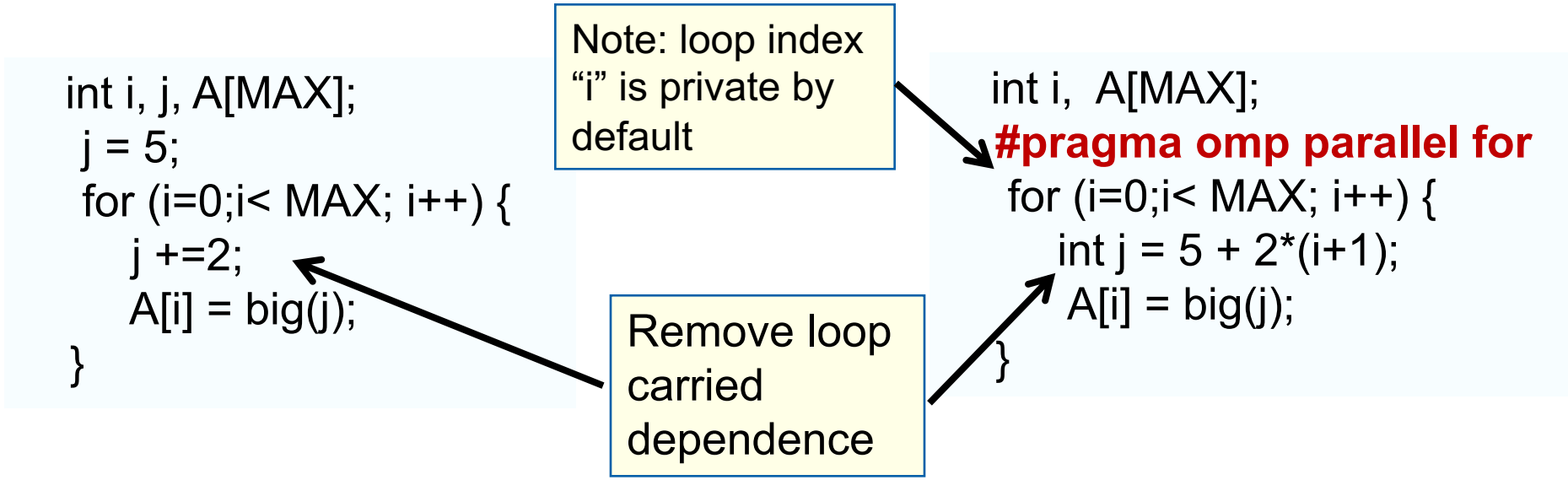
Use the **id** and **nthrds** to split up work between threads

Create a scalar local to each thread to accumulate partial sums.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Loop-level parallelism Design Pattern

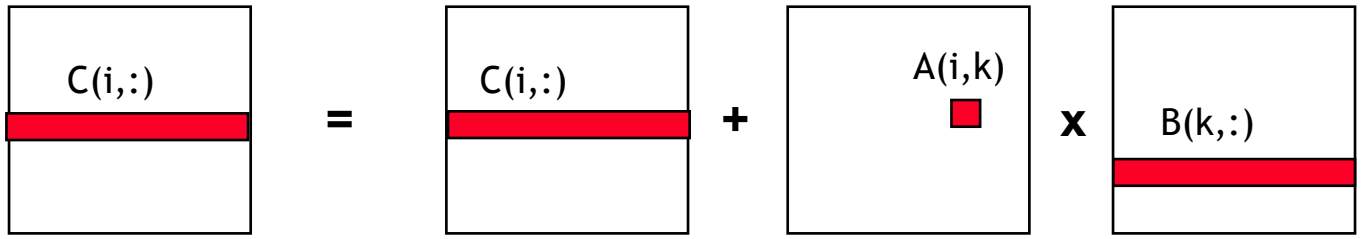
- Find compute intensive loops **with concurrency** that can be exploited in parallel
- **Expose the concurrency**, that is, make the loop iterations independent so they can run in parallel
- Modify loop body and scheduling of iterations to balance the load and optimize memory utilization.
  - This may require fusing loops to create enough work to compensate for loop control overhead.
- Insert appropriate constructs from a parallel programming application programming interface (such as OpenMP) so they run in parallel



Example: Parallel ikj Matrix Multiplication Algorithm

- We can reduce cache misses by swapping the j and k loops … so the innermost loop is over j and the access patterns across the C and B matrices cache friendly. We call this is ikj algorithm

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    #pragma omp parallel for private(k,j)
    for (i = 0; i < N; i++) {
      for (k = 0; k < N; k++) {
          for (j = 0; j < N; j++) {
              C[i*N+j] += A[i*N+k] * B[k*N+j];
          }
       }
    }
}
```



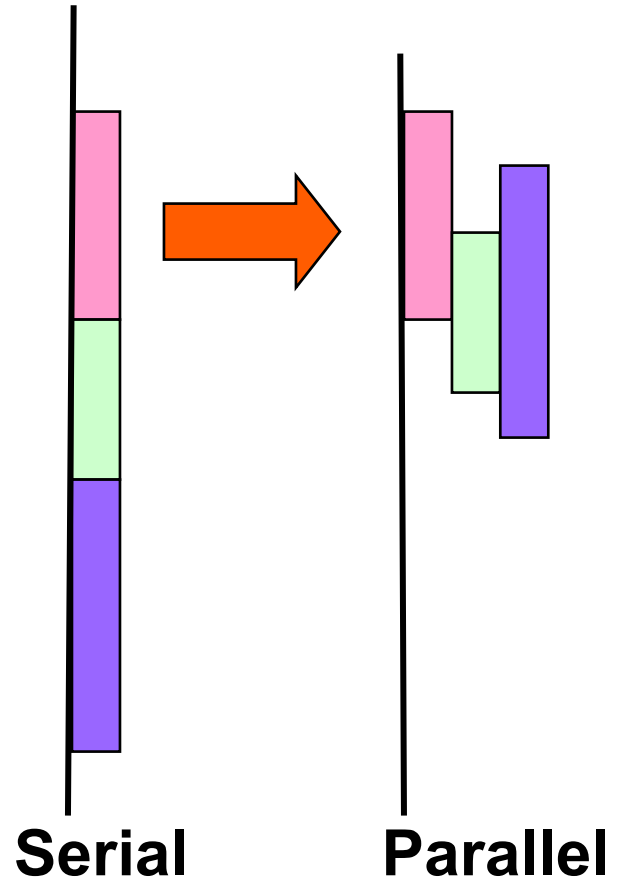
Scale a row of B by an element of A and add to a row of C

Task queue Design Pattern

- Used for irregular work that doesn't map onto basic for-loops. Also useful for work that can not be deterministically ordered for effective load balancing.

- Solution:
  - Fill a queue with a collection of tasks
  - A set of threads cooperatively execute the tasks until all tasks complete or an exit condition is met.
  - The queuing system is expected to provide the following:
    - The queue is a thread-safe, concurrent data structure designed to be manipulated by multiple threads with low overhead.
    - The queuing system tracks capacity of the queue and will limit creation of new tasks a so threads can drain the queue before more are added.
    - Quality queuing systems will distribute queues and use work-stealing to optimize scheduling of tasks

The key to the power of this pattern is that all this complexity is done by the system … transparent to the application programmer.



Parallel Linked List Traversal

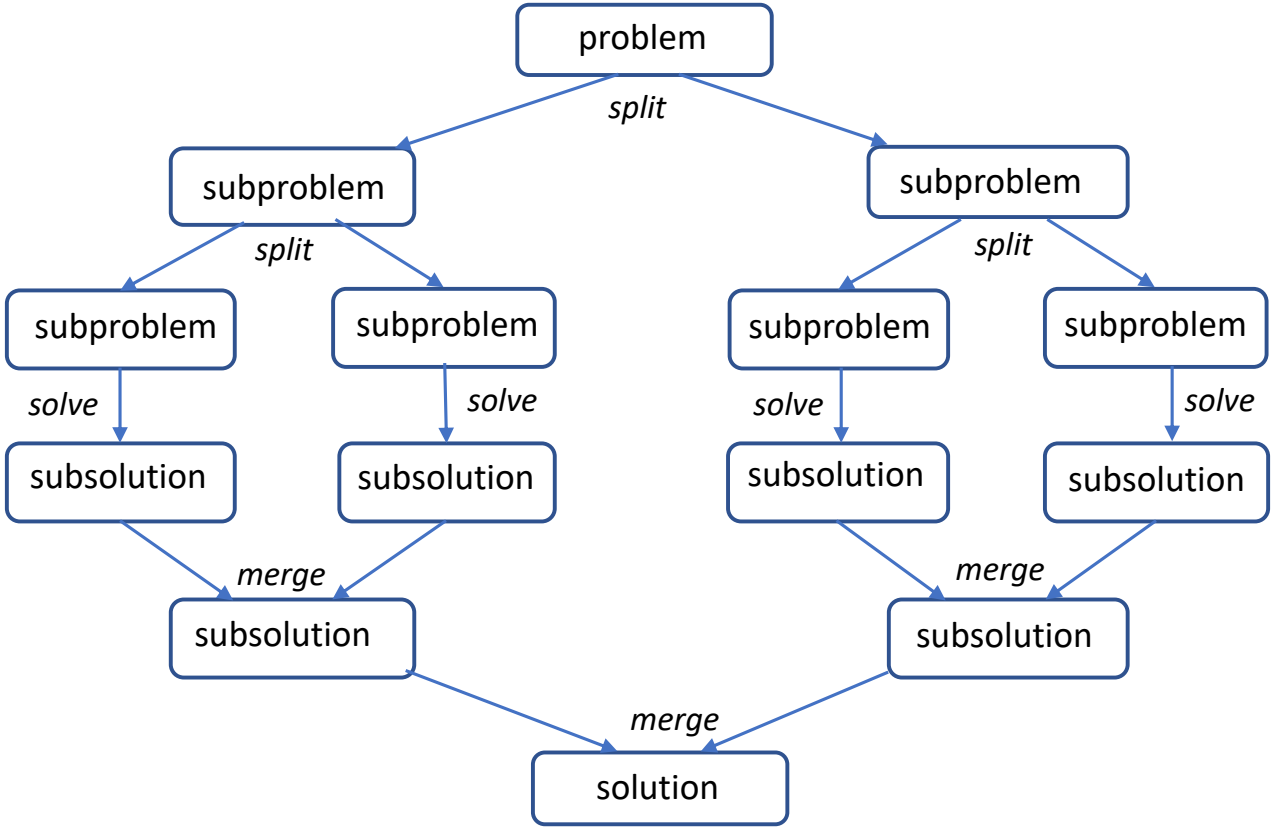
```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
                {
                  process (p);
                }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

makes a copy of `p` when the task is packaged

Divide and Conquer design pattern

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



- 3 opportunities for parallelism:
  - Do work as you split into sub-problems
  - Do work only at the leaves
  - Do work as you recombine

Program: OpenMP tasks

```
include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
    double x, sum = 0.0,sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
      }
    }
    else{
      iblk = Nfinish-Nstart;
      #pragma omp task shared(sum1)
          sum1 = pi_comp(Nstart,        Nfinish-iblk/2,step);
      #pragma omp task shared(sum2)
          sum2 = pi_comp(Nfinish-iblk/2, Nfinish,      step);
      #pragma omp taskwait
        sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
          sum =
              pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

Additional Design patterns (that we have not covered yet)

- Directed acyclic graph (DAG)
  - Create a collection of tasks connected by dependencies
- Actors
  - Create a set of distinct processes that wait on events to trigger a preset collection of actions.
  - Used in real time control applications and distributed systems.
- Geometric Decomposition*
  - Decompose a problem domain into chunks.
  - Communicate with neighboring agents to satisfy dependencies then update chunks
  - Used in linear algebra, partial differential equation solves, image processing and more
- Data Parallelism*
  - Define an index set. Map data and functions onto that index set. Do operations in parallel for each member of the index set.
  - Versions of this pattern for vector processors (single instruction multiple data) with lock-step execution or GPUs (single instruction multiple thread) where lock-step execution is relaxed

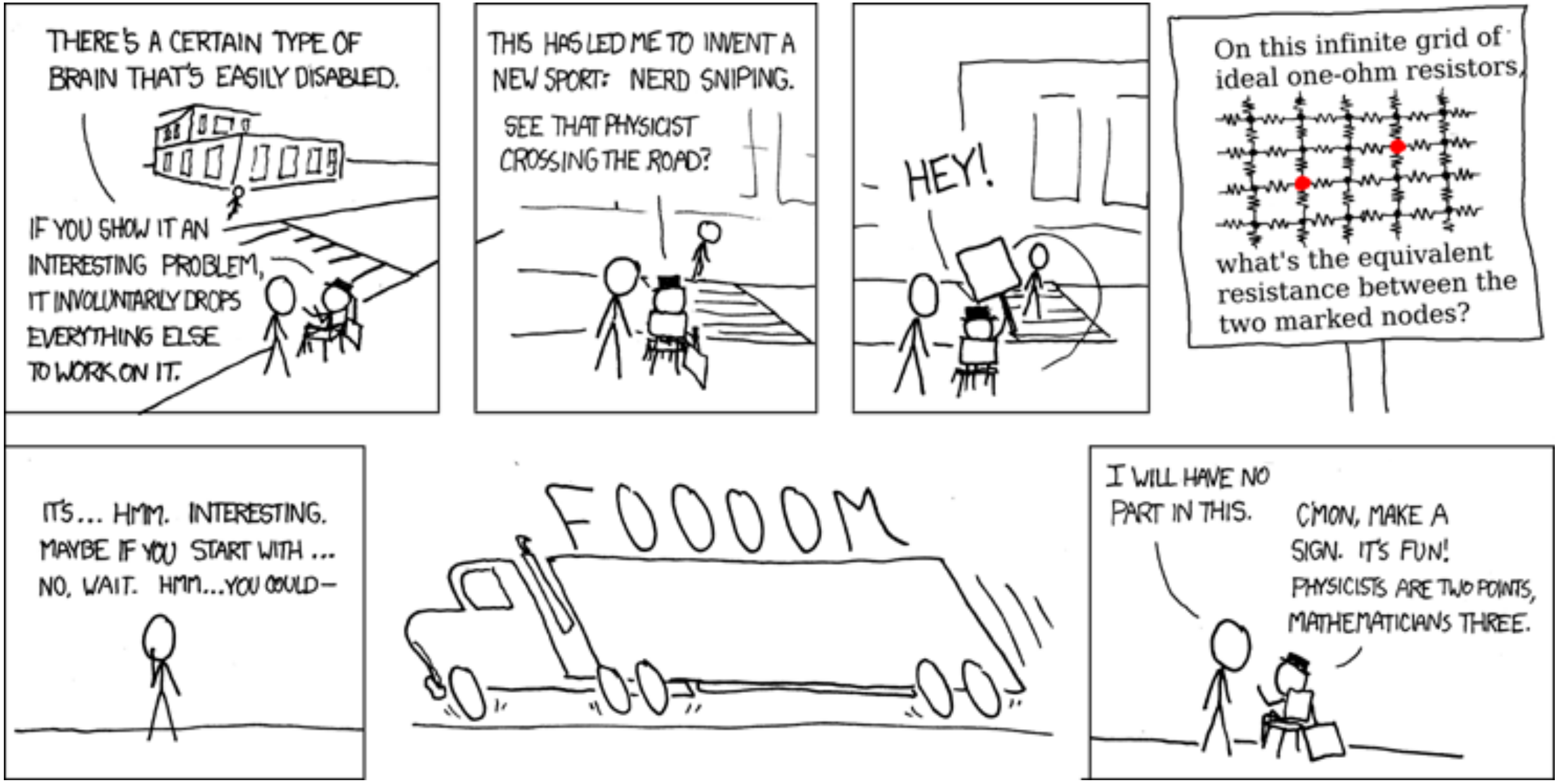
There are a few more less commonly used design patterns … but the 4 we've covered plus the four listed above represent the majority of parallel patterns used in scientific computing. 8 is not a huge number to master!!!

* We will cover these later

… So now for the pronouncement you've all been waiting for

An official proclamation!!!

By the Power Invested in me by the OpenMP Architecture Review Board, you are now OpenMP programmers with all the rights and privileges that come with that designation.



https://xkcd.com/356/