Concurrent programming languages

Week 2: Concurrent algorithms

Stefan Kuhn
Parallel algorithms

- “an algorithm is a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation” (wikipedia)
- Has sequence in it, thinking about algorithms is typically very much sequential
- In order to make use of parallel computers, algorithms need to be parallel
- Sometimes it is easy to use an algorithm in a parallel fashion, often it is not
- The implementation in a particular language is the next step
Parallel algorithms

- Example: Sieve of Eratosthenes
- Purpose: Find all prime numbers in range from 2 to n
- Idea: Starting with the lowest known prime number, cross out all multiples

Parallel algorithms

- Algorithm in Pseudo-Code:

```plaintext
algorithm Sieve of Eratosthenes is
    input: an integer n > 1.
    output: all prime numbers from 2 through n.
    let A be an array of Boolean values, indexed by integers 2 to n,
    initially all set to true.
    for i = 2, 3, 4, ..., not exceeding \( \sqrt{n} \) do
        if A[i] is true
            for j = i2, i2+i, i2+2i, i2+3i, ..., not exceeding n do
                set A[j] := false
    return all i such that A[i] is true.
```
Parallel algorithms

- Looking at the algorithm, we can see it is sequential
- The steps are written to be executed one after the other
- That is also true for the loops
- There may also be parts which are more complicated in an actual language (think about the array initialization)
Parallel algorithms

- Can this be converted into a parallel algorithm?
- Specifically, one which is more efficient than the sequential one
- In general, we have to do as many things in parallel as possible
- It depends on the hardware and the software what is possible
- Also, the pseudo code so far was not always explicit, even if it probably assumed sequential execution:
  let A be an array of Boolean values, indexed by integers 2 to n, initially all set to true.
- Is this talking about the implementation? About how this is done?
Parallel algorithms

• Let’s take an implementation (in C, [link](http://www.shodor.org/media/content/petascale/materials/UPModules/sieveOfEratosthenes/module_document_pdf.pdf))

```c
/* Declare variables */
int N = 16; /* The positive integer under which we are finding primes */
int sqrtN = 4; /* The square root of N, which is stored in a variable to avoid making excessive calls to sqrt(N) */
int c = 2; /* Used to check the next number to be circled */
int m = 3; /* Used to check the next number to be marked */
int *list; /* The list of numbers – if list[x] equals 1, then x is marked. If list[x] equals 0, then x is unmarked. */
char next_option = ‘ ‘; /* Used for parsing commandline arguments */
```
Parallel algorithms

/* Allocate memory for list */
list = (int*)malloc(N * sizeof(int));
/* Exit if malloc failed */
if(list == NULL) {
    fprintf(stderr, "Error: Failed to allocate memory for list.\n");
    exit(-1);
}
/* Run through each number in the list */
for(c = 2; c <= N-1; c++) {
    /* Set each number as unmarked */
    list[c] = 0;
}
/* Run through each number up through the square root of N */
for(c = 2; c <= sqrtN; c++) {
    /* If the number is unmarked */
    if(list[c] == 0) {
        /* Run through each number bigger than c */
        for(m = c+1; m <= N-1; m++) {
            /* If m is a multiple of c */
            if(m%c == 0) {
                /* Mark m */
                list[m] = 1;
            }
        }
    }
}
}//printing of numbers and deallocation of memory left out here
Parallel algorithms

- This is clearly sequential
- It is also clear what happens (assuming C is executed the same way on all machines)
Parallel algorithms

- What can we say about possible parallel actions here?
- Initialization, malloc, and the check for null seem hard to parallelize (ignoring memory models etc. for now)
- Initialization of the list to 0 – that could be done in parallel (not considering the question if it worth doing so for now)
- The main part is the marking of the numbers. An initial idea might be to have the marking of each number in parallel
- This is problematic, since we only now a new number to mark after we have done the marking with a number
Parallel algorithms

- Alternatively, we can try to split the data into chunks.
- The crossing out is done in parallel for each chunk with the same number.
- There would be some communication needed here to find the next number.
- Therefore, a common strategy is to have one process for the numbers from 2 to $\sqrt{N}$ (notice the algorithm only goes up to $\sqrt{N}$) and then a number of other processes for the rest.
- This means communication is reduced.
Parallel algorithms

• The algorithm, assuming **shared memory**, then is:

```
algorithm Sieve of Eratosthenes is
    input: an integer n > 1.
    output: all prime numbers from 2 through n.
    let A be an array of Boolean values, indexed by integers 2 to n,
    initially all set to true.
    for i = 2, 3, 4, ..., not exceeding \sqrt{n} do
        Thread 0:
            if A[i] is true
                Broadcast i to other threads
                for j = i^2, i^2+i, i^2+2i, i^2+3i, ..., not exceeding \sqrt{n} do
                    set A[j] := false
        Thread > 0:
            In the block of the process, cross out all multiples of i
    return all i such that A[i] is true.
```
Parallel algorithms

- In this code, there is a single array for the numbers from 2 to n
- One thread (thread 0) does the number 2 to sqrt(n)
- Each prime number is broadcasted to all other threads
- After all threads are finished, only prime numbers should be left in the array
- Implementing this in a language supporting threads should not be too difficult
Parallel algorithms

- Here, for doing 2 to \( \sqrt{n} \), there is no speedup
- The speedup for the rest depends on the number of processors
- This is a bit like in Amdahl’s law
Parallel algorithms

- What is the situation like in a distributed memory setting, disregarding the actual hardware (cluster, type of network etc.)?
- We cannot use a single data structure
- Again, there are several options possible here
- A potential method is to have each process to the \(2\) to \(\sqrt{n}\) part and then cross out one segment of the remaining numbers
- Those part will then be combined to produce the result
Parallel algorithms

- An algorithm, assuming **distributed memory**, then is:

  algorithm Sieve of Eratosthenes is
  
  input: an integer \( n > 1 \), limits \( L \) and \( H \) for the chunk to deal with.
  output: all prime numbers from 2 through \( n \).
  let \( A \) be an array of Boolean values, indexed by integers 2 to \( \sqrt{n} \),
  initially all set to true.
  let \( B \) be an array of Boolean values, indexed by integers \( L+1 \) to \( H \),
  initially all set to true.

  All processes:
  
  for \( i = 2, 3, 4, ... \), not exceeding \( \sqrt{n} \) do
    if \( A[i] \) is true
      for \( j = i^2, i^2+i, i^2+2i, i^2+3i, ... \), not exceeding \( \sqrt{n} \) do
        set \( A[j] := false \)
      In \( B \), cross out all multiples of \( i \)
    Send \( B \) to process 0

  Process 0
  
  Combine all \( B \)s and return all \( i \) such that \( A[i] \) is true.
Parallel algorithms

- Doing the 2 to $\sqrt{n}$ part several times may sound inefficient, but it is all running parallel and saves on communication.
- There are several options how to split the number from $\sqrt{n} + 1$ to $n$.
- Equal chunks are a good choice, given all processes have equal computational power.
- We can divide in blocks or in “interleaving” chunks.
- We need to make sure we can handle cases where the split doesn’t “fit.”
Parallel algorithms

- A general design method for parallel algorithms was suggested by Ian Foster
- We will look at this as a general guidance on how to approach the problem

There are four steps in Foster’s Methodology:

- partitioning
- communication
- agglomeration
- mapping
Parallel algorithms

Partitioning:

- **Domain Decomposition:** Decompose the data into many small pieces to which parallel computations may be applied. Identify the largest and most frequently accessed data object, decomposing it into as many small, identical pieces as possible, and assign a primitive task to each piece.

- **Example:** We want to transform a 3D object. It might be a good plan to transform each pixel on its own (given this is how the algorithm works)

- This is ideal for data parallelism
Parallel algorithms

Partitioning:

• Functional Decomposition: Decompose the functions into separate tasks. These can then be applied to either some data or all data in parallel.

• Example: Transformation of an audio signal requires three different operations. They can be executed on separate hardware, each working on the same data.

• This is ideal for functional parallelism
Parallel algorithms

The partitioning should satisfy these criteria:

- The partition denes at least an order of magnitude more tasks than there are processors in your target computer. If not, you have little flexibility in subsequent design stages.
- Redundant computations and data storage are avoided as much as possible. If not, the resulting algorithm may not be able to deal with large problems.
- Primitive tasks are roughly the same size. If not, it may be hard to allocate to each processor equal amounts of work, and this will cause an overall decrease in performance.
- The number of tasks is an increasing function of the problem size. Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
- You have identified several alternative partitions. You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.
Parallel algorithms

Communication:

• Which communication is needed between tasks?
• Is it local (a few tasks) or global (all tasks)?
• Is there any pattern in it? E. g. tasks form a grid, and each cell communicates with its neighbours
Parallel algorithms

The communication should satisfy those criteria:

- All tasks perform about the same number of communication operations. Unbalanced communication requirements suggest a design that will not scale to a larger size instance of the problem.
- Each task should communicate only with a small number of neighbors. If each task must communicate with many other tasks, it will add too much overhead.
- The communication operations should be able to proceed concurrently. If not, your algorithm is likely to be inefficient and non-scalable to a larger problem instance.
- Tasks can perform their computations concurrently. If not, your algorithm is likely to be inefficient and non-scalable to a larger problem instance.
Parallel algorithms

Agglomeration:

- So far, we had many tasks and (in distributed memory) lots of communication
- The algorithm was not aimed for a particular system or computer
- In the agglomeration phase, two or more tasks are combined into larger tasks
- Aimed at reducing the number of tasks, and making them bigger, to improve performance and simplify code
- There is a tradeoff here
The agglomeration should satisfy those criteria (http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter03.pdf):

- Agglomeration should reduce communication costs by increasing locality.
- If agglomeration has replicated computation, the benefits of this replication should outweigh its costs, for a range of problem sizes and processor counts.
- If agglomeration replicates data, it should not compromise the scalability of the algorithm by restricting the range of problem sizes or processor counts that it can address.
- Agglomeration should produce tasks with similar computation and communication costs.
- The number of tasks can still scale with problem size.
- There is sufficient concurrency for current and future target computers.
- The number of tasks cannot be made any smaller without introducing load imbalances, increasing software engineering costs, or reducing scalability.
- The trade-off between the chosen agglomeration and the cost of modification to existing sequential code is reasonable.
Parallel algorithms

Mapping:

- Assigning tasks to processors
- May not be necessary if done automatically
- Should minimize execution time, which typically means to maximize
- If communication has different costs, frequently communicating tasks should have low communication costs
- Various load balancing algorithms can be used
Parallel algorithms

The mapping should satisfy those criteria:

- Designs based on one task per processor and multiple tasks per processor have been considered.
- Both static and dynamic allocation of tasks to processors have been considered.
- If a centralized load-balancing scheme is chosen, you have made sure that the manager will not become a bottleneck.
- If a dynamic load-balancing scheme is chosen, you have evaluated the relative costs of different strategies, and should account for the implementation costs in the analysis.

(http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter03.pdf)
Parallel algorithms

- Problems which can easily be parallelized are called \textit{embarrassingly parallel}.
- Typical examples are independent searches where simply each processor does some of the comparisons (typical in bioinformatics, algorithms like BLAST) or fractal calculations, where each pixel can be calculated independently from others (like the Mandelbrot set).
Parallel algorithms

- There are issues to avoid when writing concurrent programs
- These include starvation, race conditions, and deadlocks
- We will look at these whilst we discuss the different programming languages
Parallel algorithms

- What we said so far, begs the question if all problems can be solved faster in a parallel fashion
- The answer seems to depend on definition a lot
- All algorithms are probably parallelizable, but not all are well parallelizable
- Theoretically, the question is unsolved – there is complexity class NC, which are problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors
- The question if P=NC is unsolved
Parallel modelling

- CCS=Calculus of communicating systems
- By Robin Milner 199X
- There are competitors, like CSP (Hoare)
- Models components, their observational behaviour and how they interact
- Can model parallel/distributed systems
Parallel modelling

- Example: Our system is specified as $A := a.b.A$
- This does $a$ and $b$ in eternity
- For the implementation we have two components, one can do (only) $a$ and the other (only) $b$
- $B := a.B$  $C := b.C$
- If we put these in parallel in a system $(B \mid C)$ we get $a$ and $b$ in no defined order
Parallel modelling

- If we assume that there is no shared memory and components can exchange messages, we can write:

  - $B' := \overline{c.d.b}.B \quad C' := c.a.\overline{d}.C$
  - $A' := (B' \mid C')\{c,d\}$

  - These components communicate on $c$, then $a$ is done, then communication on $d$ is done, then $b$ happens and we start from the beginning

  - We could formally prove that our system implements the specification, i.e. that $A = A'$
Parallel modelling

- Communicating Sequential Processes (CSP) have served as inspiration for e.g. GO
- There are now many CSP implementations like pyCSP, js-csp, CSP for Java, …
- In CSP, there are names channels: \( A = \text{query!}i \) and \( B = \text{query?}i \) can communicate \( i \) over a channel query (there are other operators like parallel, choice, restriction etc. in CSP)
- There is CCS with value passing as well
Parallel modelling

- You also find a variety of diagram to illustrate various aspects of parallelism/distributed systems/..., e.g.

  The four basic task actions. In addition to reading and writing local memory, a task can send a message, receive a message, create new tasks (suspending until they terminate), and terminate. (https://www.mcs.anl.gov/~itf/dbpp/text/node9.html)

- This is all ad hoc and no standard exists
Summary

- Most algorithms are formulated in a sequential fashion.
- In order to transform them into concurrent algorithms, Foster’s method suggests the steps of partitioning, communication, agglomeration, and mapping.
- These are helpful steps, but not a mechanistic recipe.
- The question if all algorithms can be parallelized in a useful way is open.