Writing Parallel Programs

Prof. Naga Kandasamy
ECE Department, Drexel University

The following material has been borrowed from: Culler, Singh, and Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1999.

Assume that the sequential algorithm to be made parallel is given to us, perhaps as a description or as a sequential program. In many cases, as in the case study discussed later, the best sequential algorithm for a problem lends itself easily to parallelization; in others, it may not afford enough parallelism, and a fundamentally different algorithm may be required. However, whatever the chosen underlying sequential algorithm, a significant process of creating a good parallel program is present in all cases, and we will discuss this process to program parallel machines effectively.

Steps in the Parallelization Process

Before detailing the steps in creating a parallel program, let us first define three important concepts: tasks, processes, and processors.

- *Task:* It is an arbitrarily defined piece of the work done by the program. It is the smallest unit of concurrency that the parallel program can exploit; that is, an individual task is executed by only one processor, and concurrency among processors is exploited only across tasks.
- *Process:* A process is an abstract entity that performs tasks. A parallel program is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program. Processes may need to communicate and synchronize with one another to perform their assigned tasks.
- *Processors:* Processes perform their assigned tasks by executing them on the physical processors in the parallel machine.

The job of creating a parallel program from a sequential one consists of four major steps, shown in Fig. 1. They are:

- Decomposition of the computation into tasks.
- Assignment of tasks to processes.
- Orchestration of needed data access, communication, and synchronization among processes.
- Mapping processes to processors.

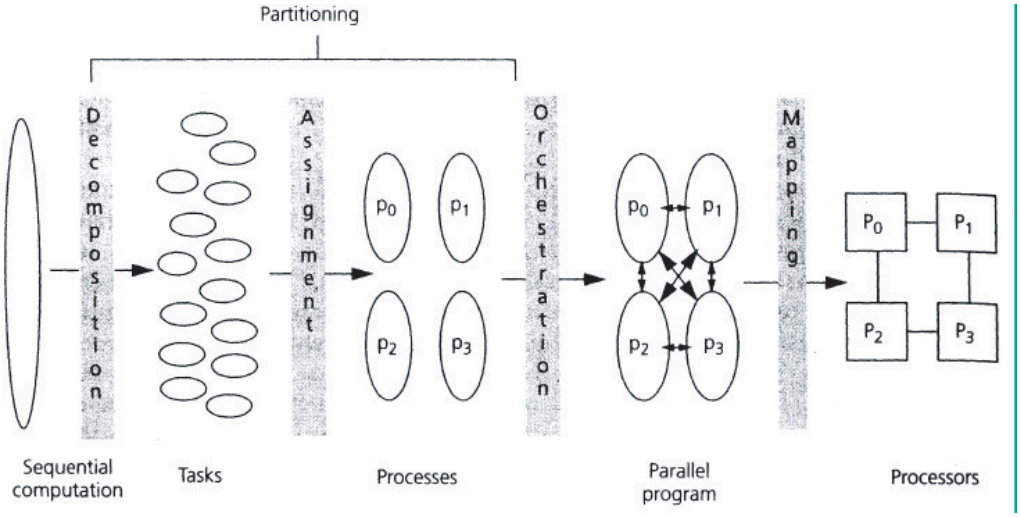


Figure 1: The various steps in the parallelization process and the relationships among tasks, processes, and processors.

Together, the decomposition and assignment steps are called partitioning, since they divide the work done by the program among the cooperating processes.

Decomposition. The computation is broken up into a collection of tasks. The major goal here is to expose enough parallelism or concurrency to keep the processes busy at all times, yet not so much that the overhead of managing the tasks becomes substantial compared to the useful work done. Limited concurrency is the most fundamental limitation on the speedup achievable through parallelism. It is not only the available concurrency in the underlying problem that matters but also how much of this concurrency is exposed or extracted in the decomposition. We know from Amdahl's Law that if some portions of a program's execution don't have as much concurrency as the number of processors used, then some processors will have to be idle for those portions and speedup will be suboptimal. As a simple example, consider what happens if a fraction s of a program's execution time on a uniprocessor is inherently sequential; that is, it cannot be parallelized. Even if the rest of the program is parallelized to run on a large number of processors in infinitesimal time, this sequential time will remain.

Assignment. The mechanism by which tasks are distributed among processes must be specified. The primary performance goals here are to balance the workload among processes, to reduce the amount of interprocess communication, and to reduce the run-time overhead of managing the assignment. Balancing the workload is often referred to as *load balancing*. The workload to be balanced includes computation, input/output, and data access or communication; programs that are not balanced well among processes are said to be load imbalanced. Interprocess communication may be expensive, especially when the processes run on different processors, and complex assignments of tasks to processes may incur overhead at run time.

If the assignment is completely determined at the beginning of the program, or just after reading and analyzing the input, and does not change thereafter, it is called a *static* or *predetermined assignment*; if the assignment of work to processes is determined at run time as the program executes (perhaps to react to load imbalances), it is called a *dynamic assignment*.

Decomposition and assignment are the major algorithmic steps in parallelization, and as programmers, we usually focus on decomposition and assignment first.

Orchestration. To execute their assigned tasks, processes need mechanisms to name and access data, to exchange data (communicate) with other processes, and to synchronize with one another. Orchestration uses the available mechanisms to accomplish these goals correctly and efficiently. The major performance goals in orchestration are reducing the cost of the communication and synchronization as seen by the processors, preserving locality of data reference, scheduling tasks so that those on which many other tasks depend are completed early, and reducing the overhead of parallelism management. The job of the programming language is to provide the appropriate primitives with efficiencies that simplify successful orchestration.

Mapping. The cooperating processes resulting from the decomposition, assignment, and orchestration steps constitute a parallel program that must be mapped to the underlying processors for execution. The program may choose to control this mapping, but if not, the operating system will take care of it.

Parallelizing a Program: A Case Study

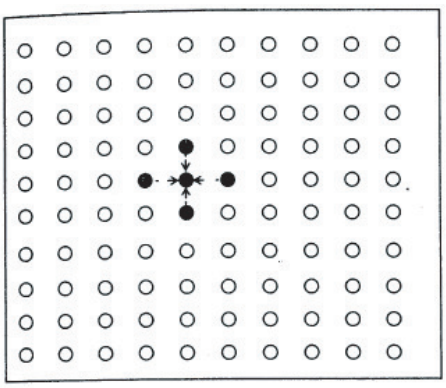
We will now discuss how to develop a parallel version of an equation solver that solves a simple partial differential equation on a grid using a finite differencing method. The solver operates on a regular, two-dimensional grid A of $(n+2)$-by-$(n+2)$ elements shown in Fig. 2. The border rows and columns of the grid contain boundary values that do not change, whereas the interior n-by-n points are updated by the solver, starting from their initial values. The computation proceeds over a number of iterations, where during each iteration, the solver operates on all the interior n-by-n points of the grid. For each point, it replaces its value with a weighted average of itself and its four nearest-neighbor points-above, below, left, and right, according to the update rule:

$$A[i,j] = 0.2 \times (A[i,j] + A[i-1,j] + A[i+1,j] + A[i,j+1] + A[i,j-1])$$

The updates are done *in place* in the grid, so the update computation for a point sees the new values of the points above and to the left of it and the old values of the points below and to its right. This form of update is called the *Gauss–Seidel* method. During each iteration, the solver also computes the average difference of an updated element from its previous value. If this average difference over all elements is smaller than a predefined "tolerance" parameter, the solution is said to have converged and the solver exits at the end of the iteration. Otherwise, it performs another iteration and tests for convergence again, and so on.

The pseudocode for a sequential implementation of the solver is shown in Algorithm 1. The main body of work to be done in each iteration is in the nested for loop in lines 6–12. Let us now attempt to parallelize the inner loops in the code listing. For programs that are structured in successive loops or loop nests, a simple way to identify concurrency is to start from the loop structure itself. We examine the individual loops or loop nests in the program one at a time, see if their iterations can be performed in parallel, and determine whether this exposes enough concurrency. We can then look for concurrency across loops.

Each iteration of the outermost while loop, starting line 4, sweeps through the entire grid. These iterations clearly are not independent since data modified in one iteration is accessed in the next.



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

Figure 2: Nearest-neighbor update of a grid point in the simple equation solver. The black point is $A[i, j]$ in the two-dimensional array that represents the grid and is updated using itself and the four shaded points that are its nearest neighbors according to the equation to the right of the figure.

Code listing 1 Pseudocode describing the sequential equation solver.

```
 1: float *A; /* The n × n array containing the grid elements. */
 2: int i, j, done = 0;
 3: float diff = 0.0, temp;
 4: while (done == 0) do
 5:    diff = 0.0;
 6:    for i = 1 to n do
 7:       for j = 1 to n do
 8:          temp = A[i, j];
 9:          A[i, j] = 0.2 × (A[i, j] + A[i − 1, j] + A[i + 1, j] + A[i, j + 1] + A[i, j − 1]);
10:          diff += abs(A[i, j] - temp);
11:       end for
12:    end for
13:    /* Check for convergence here; TOL is a user-specified value. */
14:    if (diff/n^2 ≤ TOL) then
15:       done = 1;
16:    end if
17: end while
```

Consider the loop nest in lines 6–12. Each iteration of the inner j loop reads the grid point $A[i, j-1]$ that was written in the previous iteration. The iterations are therefore sequentially dependent, and we call this a sequential loop. The outer loop of this nest is also sequential, since the elements in row $i - 1$ were written in the previous $(i - 1)^{th}$ iteration of this loop. So this simple analysis of existing loops and their dependencies uncovers no concurrency in this example program.

Instead of relying on program structure to find concurrency, we can re-examine dependencies in the equation solver in terms of generation and usage of data values at the granularity of individual grid points. Since the computation proceeds from left to right and top to bottom in the grid, computing a particular grid point in the sequential program uses the updated values of the grid points directly above and to the left. This data dependence pattern is shown in Fig. 3.

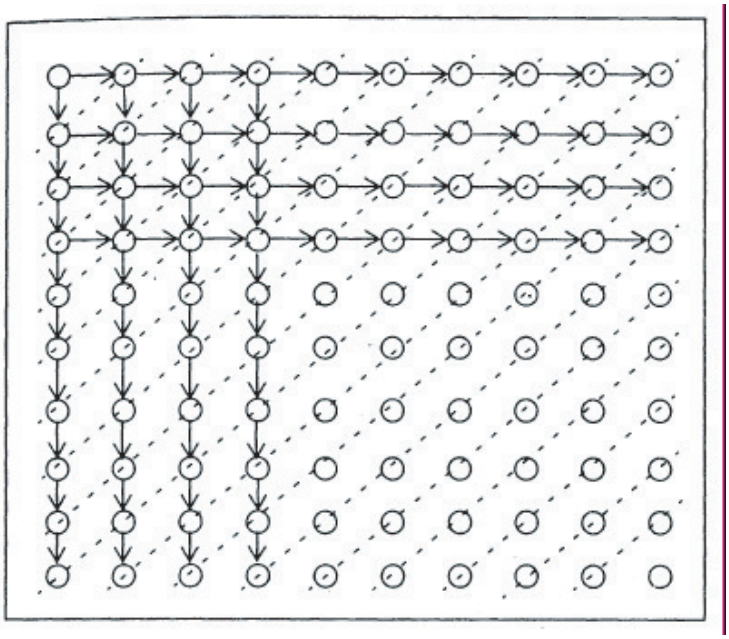


Figure 3: Data dependencies and concurrency in the Gauss-Seidel equation solver computation. The horizontal and vertical lines with arrows indicate dependencies; the anti-diagonal, dashed lines connect points with no dependencies among them that can be computed in parallel.

Parallelization Option 1. Returning to Fig. 3, we see that the elements along a given anti-diagonal (southwest to northeast) have no dependencies among them and can be computed in parallel, whereas the points in the next anti-diagonal depend on some points in the previous one. From this diagram, we can observe that of the $O(n^2)$ work involved in each sweep, there is an inherent concurrency proportional to n along anti-diagonals and a sequential dependence proportional to n along the diagonal. So, we can change the loop structure of the original sequential code where now, the first `for` loop (line 6) can be executed serially over anti-diagonals and the inner `for` loop can be executed in parallel over elements within an anti-diagonal.

Parallelization Option 2. Another approach to parallelizing sequential programs is to exploit knowledge of the problem beyond the dependencies in the sequential program itself. In the case of the solver, the order in which the grid points are updated in the sequential algorithm (left to right and top to bottom) is in fact not fundamental to the Gauss-Seidel solution method; it is simply one possible ordering that is convenient to program sequentially. Since the Gauss-Seidel method is not an exact solution method but rather iterates until convergence, we can update the grid points in a different order as long as we use updated values for grid points frequently enough.

One such ordering is called the *red-black ordering*, shown in Fig. 4, where the idea is to separate the grid points into alternating red points and black points as on a checkerboard so that no red point is adjacent to a black point or vice versa. Since each point reads only its four nearest neighbors, to compute a given red point, we do not need the updated value of any other red point; we need only the updated values of the black points above it and to its left and vice versa for computing black points. We can therefore divide each iteration into two phases, first computing all red points and then computing all black points. Within each phase no dependencies exist among grid points, so we can compute all $n^2/2$ red points in parallel, synchronize, and then compute all $n^2/2$ black

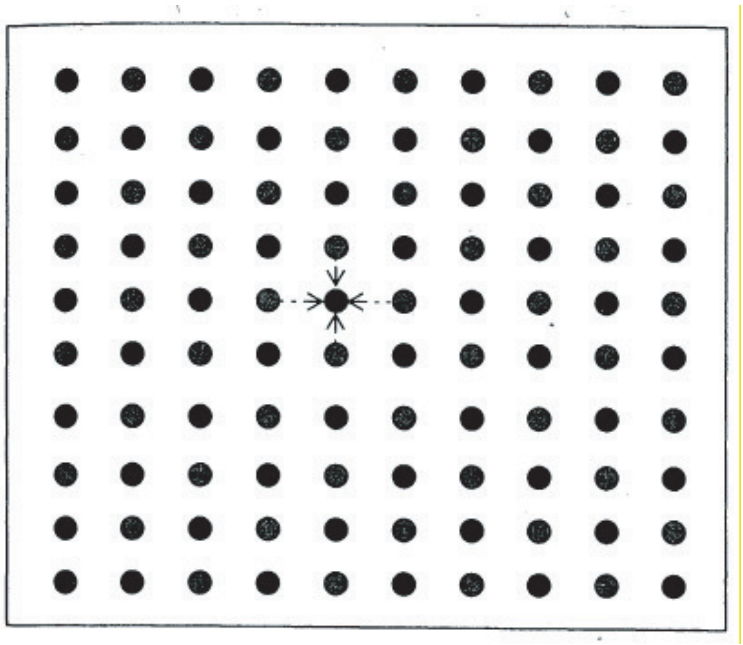


Figure 4: In red-black ordering, each iteration over the grid is broken up into two sub-operations: the first computes all the red points and the second all the black points. Since red points depend only on black points and vice versa, no dependencies occur within a sub-operation.

points in parallel.

Parallelization Option 3. Even if we don't use updated values from the current iteration for any grid points but always use the values as they were at the end of the previous iteration, the system will still converge, only much slower. This method is called Jacobi, rather than Gauss-Seidel. It simply ignores dependencies among grid points within an iteration. Global synchronization is used between iterations, but the loop structure for the program within an iteration is not changed from the top-to-bottom, left-to-right order. Instead, within an iteration a program simply updates the values of all its assigned grid points, accessing its nearest neighbors whether they have been updated in the current sweep by their assigned processes or not.

Both `for` loops in Fig. 1 are made parallel if the decomposition is into individual grid elements. Other than this change the code is the same as the sequential code. One can also decompose the grid into an equal number of rows, and during each iteration, a processor will perform the work needed to update all elements of its assigned rows.