Designing Parallel Algorithms

As in the sequential case, there are no guaranteed rules or methodologies for designing good parallel algorithms and in the end there is no substitute for insight and intuition. That said, algorithm design skills can certainly be improved by exposure to the existing body of techniques, tricks and examples. While these are less developed in the parallel world, a number of useful ideas have emerged. The material is divided into *design techniques*, which provide us with a selection of solution strategies for whole problems, and *useful primitives*, each a simple operation with an efficient parallel implementation, which can serve as useful low-level building blocks (in a manner loosely analogous to library routines in the sequential world).

Design Techniques

We begin by reviewing two well known sequential design techniques which are often well suited to parallelization. We then move on to other approaches inspired directly by the requirement for parallel activity.

Divide & Conquer

This is probably the most widely known and understood sequential technique. A problem is solved by decomposition into a set of related sub-instances of the same problem which are solved recursively, followed by composition of the sub-solutions to solve the original problem. A test defines some simple examples as being directly soluble by some other method (often trivial). Two well-worn examples are quicksort and mergesort. The scope for parallelization is obvious but complications include the fact that work at and near the root of the implied process tree may become a sequentialized bottleneck, by the danger that (particularly in message passing models) communication of small problems and solutions may swamp the machine, and by the fact that the complexity of various sub-problems may vary widely, necessitating some form of (possibly expensive) dynamic load-balancing. The first of these concerns is sometimes addressed by parallelization of the decomposition and combination operations, leading to algorithms in which parallelism is (conceptually) nested.

Pipelining

Pipelining mimics the operation of an assembly line, in which some overall task (e.g. putting together a car) is decomposed into a *sequence* of sub-tasks. While the overall completion time of any one task is not improved (and may even be increased), a long sequence of tasks can be solved with significant speed-up by having one partially completed task undergoing processing at each sub-task site (or "stage"). A good balance between the execution times of stages is essential. Pipelining is widely used in hardware design to increase throughput in areas including instruction sequencing and vector processing, since if the sum of the hardware costs of the stages is comparable to that of a single general purpose process which can solve complete tasks, then the parallelism and speed-up come more or less for free.

Step by Step Parallelization

This approach should not be confused with attempts to automatically parallelize sequential programs - trying to unravel the dependencies between sequences of individual statements (or loop iterations) has proved successful in only a limited number of specialized (but important) situations. Instead, the algorithm designer can take a higher level view of the sequence of coarse steps performed by a sequential algorithm and try to parallelize each of these independently, keeping the higher level sequential control flow of the algorithm intact. For example, some algorithm might involve two stages (perhaps iterated) in which some value is first calculated from the data in an array (the maximum value satisfying some property, for example), and then all the entries in the array are updated in some way with respect to this new value. It will sometimes be possible to devise effective parallelizations of calculation and update steps, then to implement the whole algorithm as the original sequence of these. Equally, we will sometimes find phases which are hard (or impossible) to parallelize, or we may even find it hard to identify phases in any useful way. If sections of such algorithms are left to execute sequentially, then we must be aware of a simple result, known as "Amdahl's Law" which describes an important relationship between the fraction of code left sequential and the maximum overall speed-up which can be obtained.

> *If some fraction $0 \leq f \leq 1$ of a computation must be executed sequentially, then the speed up which can be obtained when the rest of the computation is executed in parallel, is bounded above by $\frac{1}{f}$ no matter how many processors are employed.*

Proof: Let the computation require time T when executed sequentially. The run time of a parallelized version will consist of a sequential component of duration fT and a parallel component of duration $\geq \frac{(1-f)T}{p}$. The maximum speed-up is then

$$
\begin{aligned}
& \frac{T}{fT+\frac{(1-f)T}{p}} \\
= \; & \frac{1}{f+\frac{(1-f)}{p}} \\
& \longrightarrow \frac{1}{f} \; as \; p \longrightarrow \infty
\end{aligned}
$$

●

For example, if f is 0.1 then we cannot achieve a speed-up of more than 10.

Useful Primitives

In tandem with the "top-down" design techniques discussed above, a number of useful primitive parallel building blocks have emerged. These provide pre-determined and efficient implementations of operations which are central to many algorithms.

Parallel Reduction and Parallel Prefix

We have already encountered an instance of parallel reduction in the guise of our summation algorithm. In general, given a sequence of values $x_1..x_n$ and some binary, associative operator $\oplus$ on these values, then the task of a reduction is to compute the value $x_1 \oplus x_2 \oplus ... \oplus x_n$. Associativity is essential for parallel implementation (and for the value described above to be well defined, of course). In a similar situation, the task of a prefix computation is to produce the sequence of values $x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, ..., x_1 \oplus x_2 ... \oplus x_n$.

As well as obvious, simple situations like finding maxima, summing etc, reductions can be used with more complex operators as the essence of more substantial algorithms. This should not be surprising, since a reduction can be viewed as a divide and conquer algorithm with a trivial sequence splitting divide step. Thus, mergesort can be expressed as a reduction in which the operator is “merge” (convince yourself that merge is associative). Similarly, prefix plays a role as a building block in many more complex algorithms. We now present one example.

The *knapsack problem* gives us a set of n objects, each having a weight w_i and a value v_i, and a knapsack, with capacity c which we must fill with objects or fractions thereof, such that the total weight of objects used does not exceed the capacity, and so that the total value is maximised. We assume that weight and value are distributed proportionally with the division of an object.

The standard sequential algorithm proceeds as follows

```
sort items by decreasing profitability v/w;
usedweight = 0;  i = 0;
while (usedweight < c) {
   if (usedweight + (sorteditems[i].w) < c)
      include sorteditem[i];
   else
      include the fraction of sorteditem[i]
      which brings usedweight up to c;
   i=i+1;
}
```

We adopt a step-by-step parallelization strategy. First, independently in parallel we calculate profitabilities (constant time with n processors). Then we sort in parallel (of which more later). Next we compute a prefix with $+$ of the weights of the items in the sorted order. Finally, we observe that objects now fall into three categories, checkable in a single parallel step, which determine whether or not that object is part of the required collection. The cases are

1. $myprefix \leq c$, in which case the object is completely included,
2. $myprefix > c$, but left neighbours prefix is $< c$, in which case an easily determined fraction of the object is included,
3. all other objects not required.

Obviously the run time depends upon the performance of the various operations on the chosen architecture. For example it is possible for a CREW PRAM with n processors to sort n items in $\Theta(\log n)$ time and to execute a prefix with a constant time operation like $+$ in the same time. Since the other operations are clearly constant time on the CREW PRAM, we have a $\Theta(\log n)$ time parallel algorithm for this model.

Pointer Jumping

Pointer jumping is a PRAM technique for processing data stored in shared memory linked lists (as opposed to arrays). We assume that we know where the items are in memory (typically one per processor), but that we do not know their ordering (which is indicated by the "next" field of each item), except for the last item in the list whose "next" field is NULL. The key idea in pointer jumping is that we can quickly find our way from beginning to end of the list by skipping on from one item to the next field of its next item. If this happens concurrently across the list, we will jump from beginning to end in $\log n$ steps. To be interesting, we must also do some useful work along the way.

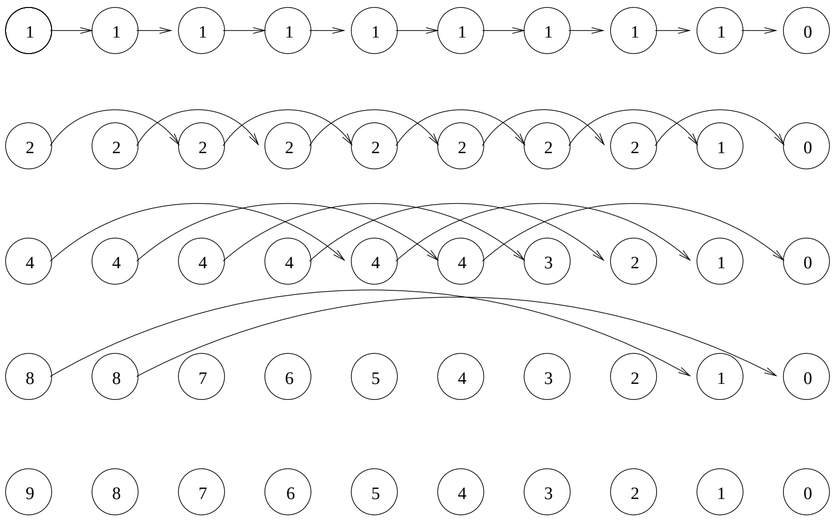
The simplest application of pointer jumping is in the implementation of an operation known as **list ranking** which requires us to compute, for each item in the list, its distance from the end of the list. We can obviously do this in n steps sequentially, but pointer jumping lets us complete the task in $\log n$ steps using n processors. Here is some pseudocode for the algorithm.

```
for all objects in parallel {
  this.jump = this.next;          // copy the list

  if (this.jump == NULL) then this.d = 0;
  else this.d = 1;                // initialise

  while (this.jump != NULL) {
    this.d +=  (this.jump).d;     // accumulate
    this.jump = (this.jump).jump; // move on
  }
}
```

We begin by making a copy of the "next" pointers which define the list, since the algorithm will destroy the copy it works with. The `d` field for each object ends up storing the required rank (distance from the end of the list). The following diagram illustrates the progress of the pointer jumps and the emerging `d` values.



A further application of pointer jumping lets us implement a **list prefix** algorithm for data stored in shared memory lists, rather than arrays. The structure is similar to that for list ranking, but notice that each operation is performed by a processor on the data field of the item which follows it. The prefix value is computed in the pf field.

```
for all objects in parallel {
  this.jump = this.next; this.pf   = this.x;

  while (this.jump != NULL) {
    this.jump.pf = Op(this.pf, this.jump.pf);
    this.jump = (this.jump).jump;
  }

}
```