15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Parallel Models I **Date:** 15 January 2009
Scribe: Zizhuang Yang

2.1 PRAM (Processor-based RAM)

PRAM stands for **Parallel Random-Access Machine**, a model assumed for most parallel algorithms and a generalization of most sequential machines (which consist of a processor and attached memory of arbitrary size containing instructions - random access memory where reads and writes are unit time regardless of location). This model simply attaches multiple processors to a single chunk of memory. Whereas a single-processor model also assumes a single thread running at a time, the standard PRAM model involves processors working in sync, stepping on the same instruction with the program counter broadcast to every processor (also known as **SIMD** - **single instruction multiple data**). Each of the processors possess their own set of registers that they can manipulate independently - for example, an instruction to add to an element broadcast to processors might result in each processor referring to different register indices to obtain the element so directed.
Once again, recall that all instructions - including reads and writes - takes unit time. The memory is shared amongst all processors, making this a similar abstraction to a multi-core machine - all processors read from and write to the same memory, and communicate between each other using this.

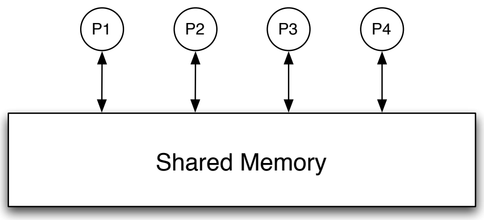

Figure 2.1.1: PRAM model

The PRAM model changes the nature of some operations. Given n processors, one could perform independent operations to n values in unit time, since each processor can perform one operation independently. Computations involving aggregation (eg. summing up a set of numbers) can also be sped up by dividing up the values to be aggregated into parts, computing the aggregate of parts recursively in parallel and then combining the results together.

While this results in highly parallel algorithms, the PRAM poses certain complications for such algorithms that have multiple processors simultaneously operating on the same memory location .
What if two processors attempt to read or write different data to the same memory location at the same time? How does a practical system deal with such contentions? To address thus issue, several constraints and semantic interpretations may be imposed on the PRAM model:

- **Exclusive Read** - at any given step, only 1 processor can read a location during the same step.
- **Concurrent Read** - no restriction on reading. While this model is usually easy to deal with, the presence of processors simultaneously writing to the same location asynchronously presents problems for programmers.
- **Exclusive Write** - at any given step, only 1 processor can write a location during the same step.
- **Concurrent Write** - no restriction on writing. Simultaneous writes can result in an arbitrary definition of behavior. Usually unsafe. Contention among multiple writers may be resolved in the following ways:
  - Arbitrary - some value is written, no deterministic way to tell which one.
  - Garbage - garbage (random values with little relevance to intended input) becomes written at the location. Still more powerful than exclusive write.
  - Priority - the processors are given priorities (ie. 1 to n), the value written by the highest priority processor is the result.
  - Combining - combine values (add, max, logical AND/OR)

Most work on parallel algorithms has been done on the PRAM model. However real machines usually differ from PRAM in that not all processors have the same unit time access to a shared memory. A parallel machine typically involves several processors and memory units connected by a network. Since the network can be of any shape, the memory access times differ between each processor-memory pair. However, the PRAM model a presents a simple abstract model to develop parallel algorithms that can then be ported to a particular network topology.

Advantages of PRAM: Simple

- abstracts away from network, including dealing with varieties of protocols.
- most ideas translate to other models.

Problems of PRAM: Programming

- unrealistic memory model - not always constant time for access. Practicality issue.
- synchronous; the lock-step nature removes much flexibility and restricts programming practices.
- fixed number of processors

2.2 DAG Model

While PRAM is a machine model (part of the reason why it got used, as it is easy to relate to), **DAG** (Directed Acyclic Graph) is a model used to analyse the cost of algorithm. A DAG, as it's name indicates, is a graph with directed edges with out cycles. As a further abstraction on algorithmic performance, a measurement from the DAG model also provides for a fair measurement of performance on the PRAM and even network-based models.

To model a parallel computation as a DAG:

- Represent small segments of sequential instructions as nodes of the graph. The weight of the node is the time cost for these instructions.
- Edges are dependencies between nodes. An edge from node A to node B indicates that the sequence of instructions in node A need to be executed those in node B.

The following two metrics indicate the computational requirements of the algorithm that the DAG models:

- Work - sum of node weights. This is total number of individual processor steps needed to run the entire computation.
- Depth - the weight of heaviest weight path in the DAG (weight of a path is the sum of node weights on the path). Since a path imposes a sequential ordering on the exexution of it's nodes, the depth of the heaviest path presents a lower bound on the minimum runtime of the algorithm.

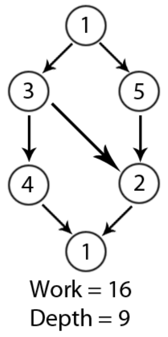

Figure 2.2.2: An example of a DAG

Depth and work are significant, as they provide an effective measurement to the performance of algorithms on the PRAM model. Placing them together allows an approximation of runtime on a fixed number of processors.

Nested Parallel Computation

An algorithm whose with an equivalent DAG that is **Series-parallel** is a nested parallel computation. A series-parallel DAG is a DAG recursively constructed out of individual nodes with the two operations: series composition and parallel composition which are illustrated below. The diagram also indicates the work and depth of a DAG as a funtion of the subgraphs it is composed of.

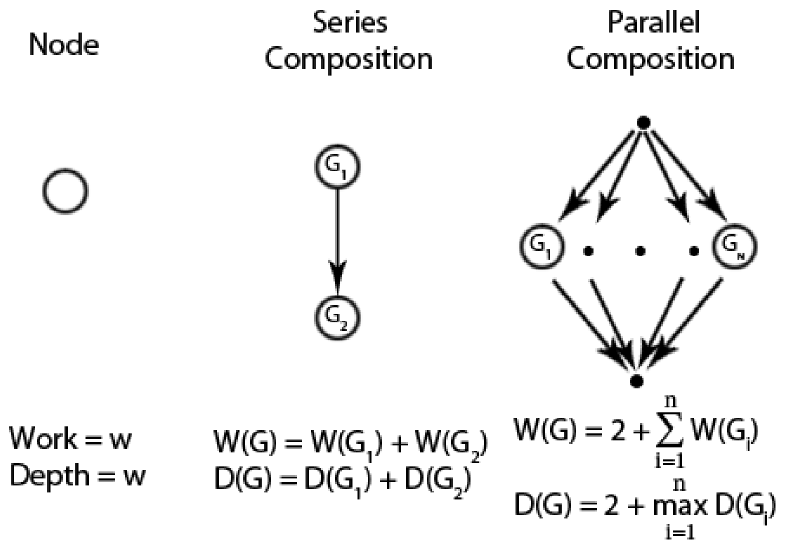

Figure 2.2.3: Series and parallel composition

2.3 Two examples

2.3.1 Summation

```
sum(A, i, n) =
  if (n = 1) then A[i]
  else sum(A, i, n/2) +
       sum(A, i+n/2, n-n/2)
```

Assume that there is no dependency between the recursve summation calls. They can thus be invoked in parallel.

$W(n) = 2W(\frac{n}{2}) + 1$ (1 is the constant work of the if), $W(1) = O(1)$. Thus, $W(n) = O(n)$.
$D(n) = D(\lceil \frac{n}{2} \rceil) + O(1) = O(\log n)$. (1 is the constant work of the addition).

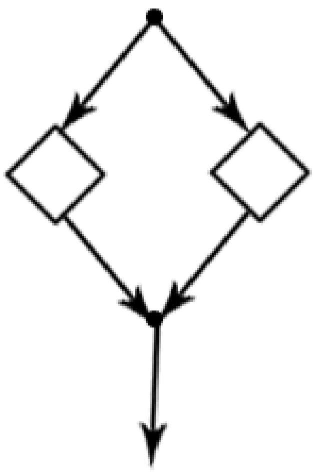

Figure 2.3.4: Parallel array summuation

2.3.2 Matrix multiplication

```
for i in [1 : n]
  for j in [1 : n]
    C_ij = sum_{k=1}^n A_ik * B_kj
```

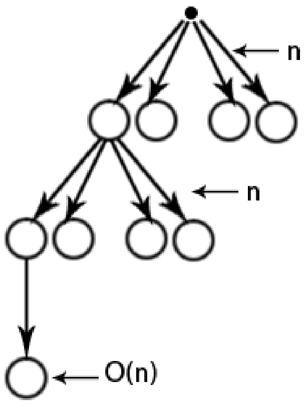

Figure 2.3.5: Parallel matrix multiplication

$W(n) = O(n^3)$ $D(n) = O(\log n)$