Parallelism

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Parallelism vs. Concurrency

Parallelism and concurrency sound like synonyms, but they are **not** directly related to each other at all.

Definition

Parallelism is the simultaneous execution of code on multiple processors or cores for the purposes of improved performance.

Example Applications: Simulations (weather, n-body, fluid flow), Stencil Convolutions, Neural Networks.

Definition

Concurrency is an abstraction for the programmer, allowing programs to be structured as multiple threads of control, called *processes*. These processes may communicate in various ways.

Example Applications: Servers, OS Kernels, GUI applications.

Parallelism vs. Concurrency

While any parallel algorithm ultimately must involve concurrency at the level of processor instructions, this is not necessarily the case at the language level.

- **Parallelism without Concurrency**: Parallel languages like TensorFlow, Accelerate (Haskell), Futhark, Parallel Fortran, CUDA.
- **Concurrency without Parallelism**: Any concurrent program running on a single core processor or a kernel without multiprocessing support.

While it is possible to describe parallel algorithms using concurrency abstractions, this is not usually desirable.

Stay away from Concurrency[1]

Concurrent programs are highly *non-deterministic*. The order in which each process takes steps is up to the scheduler of the operating system.
This necessitates a means of *synchronizing* processes so that they can produce predictable results. Failure to do so correctly results in *heisenbugs*.

The complexity of reasoning about concurrent systems grows (in the worst case) exponentially with the number of processes.
In many parallel algorithms, the number of processes is unbounded: reasoning about thousands of processes using concurrency models gets intractable very easily.

[1] for Parallelism

Parallelism Structure

There are two main types of parallelism:

1. **Data Parallelism** Also called *single instruction, multiple data* (SIMD). The same code is executed on multiple processing units, working on different parts of a large data set.
   Examples: GPUs
   Advantages: Simple, Scalable
   Disadvantages: Maybe limited applications.
2. **Control Parallelism** Also called *task parallelism* or *multiple instruction, multiple data* (MIMD). Different programs are executed on different processing units.
   Examples: Multicore CPUs
   Advantages: Flexible, Widely-supported
   Disadvantages: Less scalable, explicit synchronization.

SIMD and MIMD

- We can compile SIMD code to MIMD code straightforwardly, but typically synchronization code must be generated, which is hard to do optimally.
- We can compile MIMD code to SIMD code, by branching on the current process ID, however this leads to atrocious performance on SIMD hardware.

Determinism and Scalability

Ideally, we should be able to describe a parallel algorithm in our language, and expect deterministic results. That is, no matter how many processors or cores are used, we should get the same result.

Ideally, performance should improve linearly with the number of processing units we supply. This is called *scalability*.

Control parallelism is difficult to make deterministic[2], and impossible to make generally scalable. For that reason, we'll focus on data parallelism.

[2] Although possible, see Haskell's spark model.

Data Parallel Languages

Key Idea

We will introduce a *parallel array* type to MinHS, along with a series of collective operations to operate on arrays in parallel.

$$
\begin{array}{rcll}
\tau & ::= & \cdots & \\
 & | & [\tau] & \text{(parallel arrays)} \\
e & ::= & \cdots & \\
 & | & [e_1, e_2, \ldots, e_n] & \text{(constant arrays)} \\
 & | & e_1[e_2] & \text{(array indexing)} \\
 & | & \mathsf{len}(e) & \text{(length)} \\
 & | & [e_r \mid x \leftarrow e_1 \mid \cdots \mid x \leftarrow e_n] & \text{(zip comprehensions)} \\
 & | & [e_r \mid x \leftarrow e, e_g] & \text{(filter comprehensions)} \\
 & | & e_1 +\!\!+\ e_2 & \text{(concatenation)} \\
 & | & \mathsf{fold}_{\otimes}\ e & \text{(reduction)}
\end{array}
$$

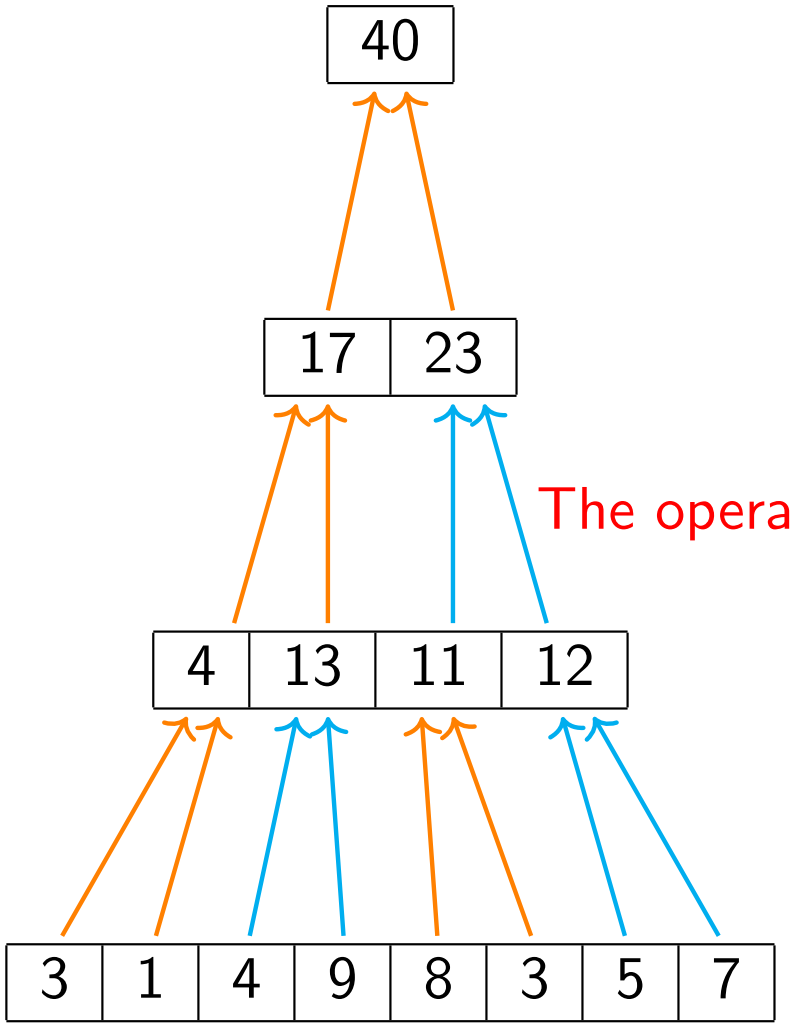
Operational Intuitions

Let $xs = [[1,2],[],[3,4,5,6]]$. Then:

- xs has type $[[\texttt{Int}]]$ – an array of arrays of integers.
- $xs[2]$ is standard array indexing, 0-based, so we get $[3,4,5,6]$.
- $\text{len}(xs)$ returns 3.
- $[x + y \mid x \leftarrow [1,2,3,4] \mid y \leftarrow [5,6,7,8]]$ evaluates to $[6,8,10,12]$ in parallel.
- $[1 + x \mid x \leftarrow [1,2,3,4], x \bmod 2 == 0]$ evaluates to $[3,5]$ in parallel.

Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



The operator must be associative for determinism!

Examples

Example (Vector Dot Product)

$$\textbf{recfun}\ \mathit{dotProduct} :: ([\texttt{Float}] \to [\texttt{Float}] \to \texttt{Float})\ \mathit{xs}\ \mathit{ys} = \\ \text{fold}_+\ [x + y \mid x \leftarrow \mathit{xs} \mid y \leftarrow \mathit{ys}]$$

Example (Sparse Matrix-Vector Multiplication)

$$\textbf{type}\ \mathit{SparseVector} = [\texttt{Int} \times \texttt{Float}] \\ \textbf{type}\ \mathit{SparseMatrix} = [\mathit{SparseVector}] \\ \textbf{recfun}\ \mathit{smvm} :: (\mathit{SparseMatrix} \to [\texttt{Float}] \to [\texttt{Float}])\ \mathit{sm}\ v = \\ [\text{fold}_+\ [\ v[\text{fst}\ \mathit{elem}] \times \text{snd}\ \mathit{elem} \mid \mathit{elem} \leftarrow \mathit{sv}] \mid \mathit{sv} \leftarrow \mathit{sm}]$$

Flat vs. Nested Data Parallelism

The second example, *smvm*, performed a nested parallel loop, i.e. a parallel computation inside another parallel computation. This is called *nested data parallelism*.

Most data-parallel architectures do not support such things directly. For example, GPU code is written as a single sequential loop body (called a *kernel*) that is run in parallel by the GPU driver.

There exists an algorithm to translate nested data-parallel programs to flat ones (*vectorisation*), but it does not always produce efficient code. In particular, it loops over the input array more often than necessary, and *fusion* optimisations are often required to consolidate multiple loops into one.

Another Example

Example (Parallel Quicksort)

$$
\begin{array}{l}
\textbf{recfun } \mathit{qsort} :: ([\mathtt{Int}] \to [\mathtt{Int}])\ \mathit{xs} = \\
\quad \textbf{if } \mathrm{len}\ \mathit{xs} \leq 1 \textbf{ then } \mathit{xs} \\
\quad \textbf{else let} \\
\qquad \begin{array}{lcl}
\mathit{p} & = & \mathit{xs}[0] \\
\mathit{ls} & = & [x \mid x \leftarrow \mathit{xs}, x < \mathit{p}] \\
\mathit{es} & = & [x \mid x \leftarrow \mathit{xs}, x == \mathit{p}] \\
\mathit{gs} & = & [x \mid x \leftarrow \mathit{xs}, x > \mathit{p}] \\
\mathit{results} & = & [\mathit{qsort}\ \mathit{section} \mid \mathit{section} \leftarrow [\mathit{ls}, \mathit{gs}]]
\end{array} \\
\quad \textbf{in } \mathit{results}[0] +\!\!+ \mathit{es} +\!\!+ \mathit{results}[1]
\end{array}
$$

Cost Models

A parallel algorithm gives rise two two time cost models:

- *Work cost* is the overall number of steps required to execute the program. Corresponds to time cost on a processor with just one core.
- *Depth cost* is the minimal number of parallel steps required to execute the program. Corresponds to the time cost on a processor with unlimited cores.

With these two measures, the cost for a given number of processing units can be derived.

Denotationally

Let's define two denotational cost models, $[\![e]\!]_{\mathcal{W}}$ for work cost, and $[\![e]\!]_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\begin{array}{lcl} [\![\mathsf{len}\ e]\!]_{\mathcal{W}} & = & [\![e]\!]_{\mathcal{W}} + 1 \\ [\![\mathsf{len}\ e]\!]_{\mathcal{D}} & = & [\![e]\!]_{\mathcal{D}} + 1 \end{array}$$

$$\begin{array}{lcl} [\![e_1[e_2]]\!]_{\mathcal{W}} & = & [\![e_1]\!]_{\mathcal{W}} + [\![e_2]\!]_{\mathcal{W}} + 1 \\ [\![e_1[e_2]]\!]_{\mathcal{D}} & = & [\![e_1]\!]_{\mathcal{D}} + [\![e_2]\!]_{\mathcal{D}} + 1 \end{array}$$

For parallel operations, we get different results for each measure:

$$\begin{array}{lcl} [\![[e_r \mid x_1 \leftarrow e_1 \mid \ldots \mid x_n \leftarrow e_n]]\!]_{\mathcal{W}} & = & [\![e_r]\!]_{\mathcal{W}} + \sum_i [\![e_i]\!]_{\mathcal{W}} \\ [\![[e_r \mid x_1 \leftarrow e_1 \mid \ldots \mid x_n \leftarrow e_n]]\!]_{\mathcal{D}} & = & [\![e_r]\!]_{\mathcal{D}} + \max_i [\![e_i]\!]_{\mathcal{D}} \end{array}$$

$$\begin{array}{lcl} [\![e_1 +\!\!+\ e_2]\!]_{\mathcal{W}} & = & [\![e_1]\!]_{\mathcal{W}} + [\![e_2]\!]_{\mathcal{W}} + \mathsf{length}(e_1) + \mathsf{length}(e_2) \\ [\![e_1 +\!\!+\ e_2]\!]_{\mathcal{D}} & = & [\![e_1]\!]_{\mathcal{D}} + [\![e_2]\!]_{\mathcal{D}} + 1 \end{array}$$

Fold

As fold involves building a tree, we have a logarithmic term to consider:

$$\begin{aligned} [\![\mathsf{fold}_{\otimes}\ e]\!]_{\mathcal{W}} &= [\![e]\!]_{\mathcal{W}} + \mathsf{length}(e) \\ [\![\mathsf{fold}_{\otimes}\ e]\!]_{\mathcal{D}} &= [\![e]\!]_{\mathcal{D}} + \log_2(\mathsf{length}(e)) \end{aligned}$$

Quicksort Work Complexity

$$
\begin{array}{lll}
\textbf{recfun}\ qsort :: ([\texttt{Int}] \to [\texttt{Int}])\ xs = & & \\
\quad \textbf{if}\ \text{len}\ xs \leq 1\ \textbf{then}\ xs & & \\
\quad \textbf{else let} & & \\
\qquad p = xs[0] & & 1 \\
\qquad ls = [x \mid x \leftarrow xs, x < p] & & n \\
\qquad es = [x \mid x \leftarrow xs, x == p] & & n \\
\qquad gs = [x \mid x \leftarrow xs, x > p] & & n \\
\qquad results = [qsort\ section \mid section \leftarrow [ls, gs]] & & 2 \times W\left(\frac{n}{2}\right) \\
\quad \textbf{in}\ results[0] +\!\!+\ es +\!\!+\ results[1] & & \frac{3n}{2}
\end{array}
$$

Let $W(n)$ be $[\![qsort\ xs]\!]_{\mathcal{W}}$ for an array xs that is bisected by the pivot on each recursion (average case). Then:

$$
\begin{aligned}
W(1) &= 1 \\
W(n) &= C_1 + C_2 n + W\left(\frac{n}{2}\right)
\end{aligned}
$$

So $W(n) \in \mathcal{O}(n\ \log n)$ (by the Master Theorem)

Quicksort Depth Complexity

recfun *qsort* :: ([Int] → [Int]) *xs* =
  if len *xs* ≤ 1 **then** *xs*
  else let

p	$=$	$xs[0]$	1
ls	$=$	$[x \mid x \leftarrow xs, x < p]$	1
es	$=$	$[x \mid x \leftarrow xs, x == p]$	1
gs	$=$	$[x \mid x \leftarrow xs, x > p]$	1
$results$	$=$	$[qsort\ section \mid section \leftarrow [ls, gs]]$	$D\left(\frac{n}{2}\right)$

  in $results[0] +\!\!+ es +\!\!+ results[1]$ 2

Let $D(n)$ be $[\![qsort\ xs]\!]_{\mathcal{D}}$ for an array xs that is bisected by the pivot on each recursion (average case). Then:

$$\begin{aligned} D(1) &= 1 \\ D(n) &= C + D\left(\tfrac{n}{2}\right) \end{aligned}$$

So $D(n) \in \mathcal{O}(\log n)$ (by the Master Theorem)