Vectorisation, Mapping and Reducing

Daniel Lawson — University of Bristol

Lecture 10.2 (v2.0.0)

How to write your lines faster



Vectorisation

Vectorised code is parallelised code.

Each operation for vectorised code is computable independently

The same operation is applied to each element (with different data)

CPU optimisation is possible and may be straightforward

- GPU acceleration is possible
- Vectorisations are always one dimensional representations

► A set of standardized elementwise computations is possible:

- addition, subtraction, multiplication, division
- other operations are possible, this becomes architecture dependent

Vectorisation of K-dimensional objects

 Matrices can be represented by standardized vectorisation procedures

$$\blacktriangleright A = \left[\begin{array}{ccc} a & b & c \\ d & e & f \end{array} \right]$$

- **Row major order**: vec(A) = (a, b, c, d, e, f)
- Column major order: vec(A) = (a, d, b, e, c, f)
- Matrix multiplication:
 - Is just sums of the correct components of the vectorised matrices
 - Choice of row vs column major order affects efficiency!
- Parallelization:
 - On a shared memory machine, the computations are distributed
 - Otherwise a memory distribution problem
- Efficient implementations for many common computations

Vectorisation and time complexity

- Assuming no parallelization:
- A for loop with N iterations is O(N)
- A vectorisation with N elements is O(N)
- But the vectorised code may still be orders of magnitude faster:
 - It often can be pushed into low-level code (C backend)
 - It can exploit CPU memory architecture: caching the correct content to avoid overhead
 - It can exploit CPU compute architecture: multiple registers in parallel

► Vectorisation also leads directly into parallel implementations:

- It emphasises dependencies,
- It encourages reordering of loops which can reduce time complexity.

Distributed computation

On a shared memory system, parallel computation is trivial:

- 1. Initialise a parallelisable step, i.e.
 - enumerate the computations to be performed.
- 2. Assign them to worker threads:
 - either evenly if compute resource is guaranteed and tasks take equal time, e.g. on a GPU,
 - or as a queue.
- 3. Action the computation,
- 4. Block, i.e. wait for all computations to complete.
- On completion, the results are in the same place in memory as if the computation was performed in series.

Accumulate/Reduce

Suppose that you wanted to compute the cumulative sum. Then the elements become **dependent** and you cannot use a purely independent vectorization.

▶ How can we combine results from N parallel computations?

- accumulate is a vectorisation of any (binary, i.e. pairwise) (associative and commutative) function returning a single value
- It may or may not provide access to intermediate function evaluations
- It is often called a Reduce operation
- It is a natively parallelisable way to view combining

Accumulate/Reduce computation graph



Computational graph properties:

Nodes \overline{n} internal to binary tree: $n(d) = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$ Depth \underline{d} : $\underline{d}(n) = \Theta(\log(n))$

Algorithm properties:

• Maximum compute could use $2^{d-1} = 2^{\log_2(n)} = n/2$ cores,

• Parallel maximum speedup: $\Theta(\log(n))$ due to depth,

• Simple blocking queue would reserve $n \log(n)/2$ processes,

Parallel efficiency cost: E = Θ(n/(n log(n))) if all memory operations are in place.

Map/Reduce parallel framework

For general purpose computation, the concepts of mapping and reducing enable efficient parallel code.

► This uses the concept of a key-value tuple.

- The data are mapped: each value is assigned one or more keys
- Data associated with each key is passed to a reducer
- The reducer completes the computation

More precisely,

- ▶ Map: $M(k_0, v_0) \rightarrow ((k_1, v_1), \cdots, (k_K, v_K))$ is a function taking an input key/value pair to a list of output key/value pairs
- ▶ **Reduce**: $R(k, (v_1, \dots, v_R)) \rightarrow (k, v)$ is a function taking an input key and list of values, to a single (list-valued) value.

Map/Reduce vector averaging example

• Let X be a vector of length N.

▶ Map: $(k_0, v) \to (k, \{w = 1, v = v\})$

• Assign each element a key $k \in [1, \cdots, K]$,

Assign a weight in the value,

The key acts as a fold of data.

Here, we are using the key as an arbitrary index, but this can be exploited.

▶ Reduce: $(k, \{v\}) \rightarrow (k, v)$

Count within each fold:

▶ Return
$$(k, v) = (k, \{w = \sum_{k=1}^{K} v_w, v = \sum_{k=1}^{K} v_v\})$$

▶ Postprocess: Return mean = $\frac{\sum_{k=1}^{n} v_{k,v}}{\sum_{k=1}^{K} v_{k,w}}$

Map/Reduce analysis

- Assume within-memory implementation
- ► Use p ≤ K parallel threads (assume an integer multiple for simplicity...)
- The map stage is entirely parallel for cost $\Theta(\lceil n/p \rceil)$
- There is a sort stage which would be handled by a set of K lists
 - ► Independently parallelised construction of the K lists for cost Θ([n/p])

In memory concatenation cost is negligible

- ► The **reduce** stage is parallel across $\Theta(\lceil K \lceil n/K \rceil/p \rceil) \approx \Theta(\lceil n/p \rceil)$ processes
- The postprocess stage is naively sequential with compute cost K
 - Total parallel time: $T_p = \Theta(\lceil n/p \rceil + \lceil n/p \rceil + \lceil n/p \rceil + K)$
 - Total sequential time: $T_s = \Theta(n)$
 - ▶ Total efficiency loss: $T_p/T_s \sim \Theta(1 + Kp)$

Map/Reduce reducer parallelisation

Practical concerns:

- Reducers don't automatically provide parallelism: we have to ask for it
- This is because the reducer is not assumed to be commutative
- But if the keys explicitly specify the desired folds, the reduce can be parallelised
- In Hadoop/Spark Map/Reduce, reduction is parallelised across keys

In python/local Map/Reduce, reduction parallelisation is manual

- We can also map the postprocess k-fold reduction sum. Using p₂ processes:
 - Reduce the postprocess time from K to $T'_p = \Theta(\lceil K/p_2 \rceil + \lceil K/p_2 \rceil + p_2)$
 - Minimized at $p_2 = \sqrt{K}$
 - So we should use $K = p^2$ keys, keeping $p_2 = p$.
 - ► Total parallel time: $T_p = \Theta(\lceil n/p \rceil + \lceil n/p \rceil + \lceil n/p \rceil + \sqrt{p})$

Map/Reduce Matrix Example



Map/Reduce Matrix Example

$$C = \begin{bmatrix} k & l \\ m & n \end{bmatrix} = AB = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} u & v \\ w & x \\ y & z \end{bmatrix}$$

• C has dimension $L \times L$, A has dimension $L \times K$

• where
$$k = au + bw + cy$$
, etc

- For a one-stage implementation, each of the four computations requires access to three elements from each array
- ▶ Represent the matrices in index form: (key, value) where key = (M, i, j) is the position (row and column) index and records the matrix type M ∈ [A, B, C].
- Computing (C, i, j) requires all elements of A from row i and all elements of B from row j
- There will be K = 3 such elements
- Required to compute L² entries of C

Map/Reduce Matrix Multiplication Algorithm

- Map: each element is mapped independently to a list of K elements:
- ► Map((M, i, j), v):
 - $\blacktriangleright \quad ((A,i,j),v) \to ((i,k),(A,j,v)) \quad \forall k = 1,\dots,K$
 - $\blacktriangleright ((B,i,j),v) \to ((k,j),(B,i,v)) \quad \forall k = 1,\dots,K$
 - Cost: 2K for each of L² independent entries
- Reduce: each key (i, j) is received 2K times, K from A and K from B.
- ▶ Reduce((i, j), (M, k, v)) :
 - $\blacktriangleright v_{i,j} = \sum_{k=1}^{K} v_{(A,k,v)} \overline{v_{(B,k,v)}}$
 - ▶ Return $((i, j), v_{i,j})$
 - Cost: K for each of L² independent entries

Cost:

- Parallel time $T_p = \Theta(\lceil L^2/p \rceil K)$
- Sequential time $T_s = \Theta(L^2 K)$
- ► Efficiency 1
- Despite inefficient duplication of data, which fast algorithms avoid!

Map/Reduce paradigm

- ► Map/Reduce is an essential tool in low-effort parallelism.
- The main computational advantage is that it is scalable: it can be parallelised across machines.
- So far we've described Map/Reduce as an in memory algorithm.
- In this case it naturally leads to fast analogues for a single computer:
 - We can imagine each reducer key being a memory location and the mappers are providing data fed to that location;
 - This is essentially how vectorised matrix computations are implemented efficiently.

Summary

- Vectorised code is efficiently computed
- Vectorised code is parallelisable with little effort
- Embarrassingly parallel algorithms are common
- Map/Reduce is a powerful paradigm for non-trivial parallelism and is the heart of massively parallel data processing
- Map/Reduce comes at an efficiency cost

References

- Chapter 27 of Cormen et al 2010 Introduction to Algorithms covers some of these concepts.
- Numpy vectorisation
- MapReduce algorithm for matrix multiplication
- Chrys Woods Parallel Python
- See the non-taught Block 12 content on Spark and HDFS if you want to learn about how this works on Distributed data.