Contents

1 Schwartz algorithm

Consider a tree-operation (e.g. +-reduce) performed by P parallel processes on n data items, where $P < n$:

- The tree should connect the P processes, rather than the n data items
- i.e. each process should compute a local result on n/P data items, which are then combined by a P-leaf tree
- Rather than, e.g., creating a tree of n-1 processes each combining one pair of values

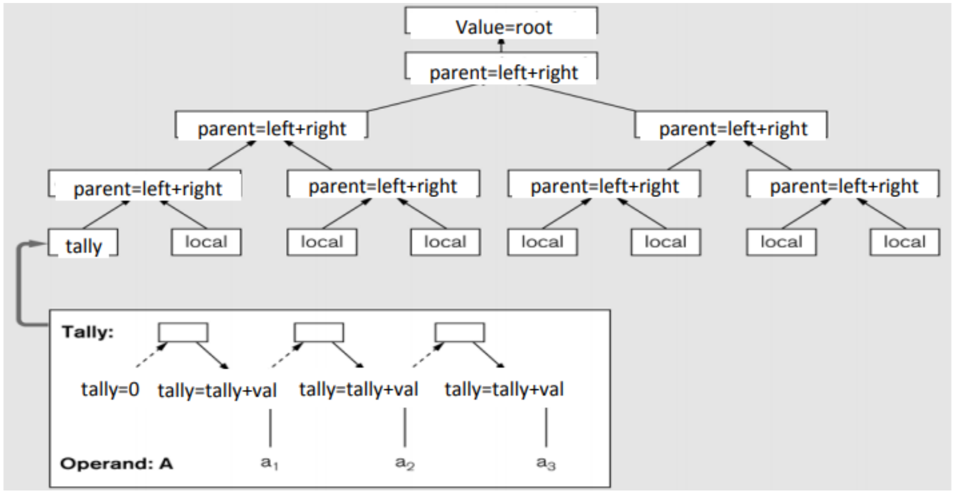
This will minimise **communication, coordination and overhead** (e.g. thread creation and multiplexing) but still have maximal parallelism if `P = number of processors`

1.1 Discussion

- Essentially this is an application of the locality rule to treebased algorithms
- It is a general strategy that is assumed to be used for, e.g., *reduce and scan* operations
- Typically the most important aspect is **having each process performed a balanced share** of the computation locally
- When P is relatively small (e.g. multicore computer) there will be **little observable difference** between sequential (critical section) and tree-based coordination between processes

2 Reduce

- Reduce combines a set of values to produce a single value E.g. count3s sum of counts across threads
- It is almost always necessary at some point to compare or combine values from different threads, e.g. to summarise the computation or control its execution
- Some computers even have hardware support for reduction, e.g. BlueGene/L
- Assumed to be implemented using a Schwartz-style algorithm, i.e.
  - Each thread combines a portion of the data to produce its own value
  - Values from each thread are then **combined in a tree** to produce the global value

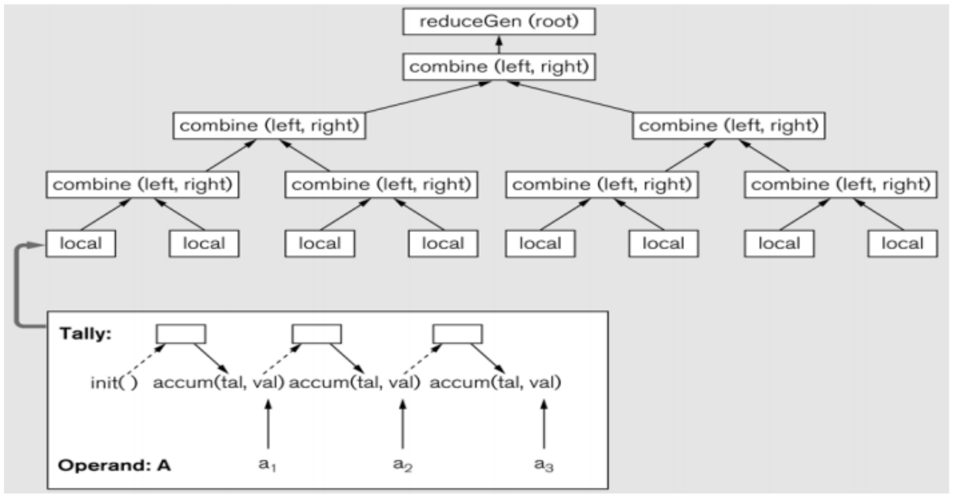


2.1 Generalised Reduce

- For simple reductions (as in OpenMP) there is a single summary value which is the same type as one data item
- E.g. +-reduce or max-reduce, reduces an array of floats (data items) to one float (summary value)
- Reduce can be generalised:
- The intermediate or tally value calculated by each thread can be a different type from the data items
- The global summary value can be a different type from the tally values
- They need not be single values, e.g. they could be compound values or array

2.2 Implementing Generalised Reduce

- `init()` initialise a threads tally value
- `accum(tally,val)` combine a single data value with a threads running tally
- `combine(left,right)` combine left and right tally values
- `reduceGen(root)` convert the root tally value to the global summary value



3 Scan

Parallel scan is another common parallel operation for calculating **local information** that depends on **non-local information**. Specifically that depends on information on the left of each value. Requires two passes:

- One upward pass, like reduce, calculating intermediate values
- One downward pass, combining and distributing intermediate values back down the tree to the individual threads
- In this case, the sum of all values to the left of the left-most child of a node

3.1 Key points

- Many parallel computations can be decomposed into blocks of independent parallel computation on local data interspersed with reduce and scan operations
- Reduce and scan have efficient parallel implementations that scale well to large number of processors
- Some thought may be required to see how a global operation can be solved as a generalised reduce or scan

4 Static work allocation

- Consider the common case where we have a 1- or 2-dimensional array of data to distribute between some set of processes for some parallel computation
- Our general approach is to **maximise locality**, so in most cases we will allocate contiguous portions of the data structure to each process
- The values operated on by each process will therefore be close to each other, and separate from values allocated to other processes
- For a 1-D array this will probably just be blocks on consecutive indices (as in the count3s examples)

4.1 Work allocation for 2D arrays

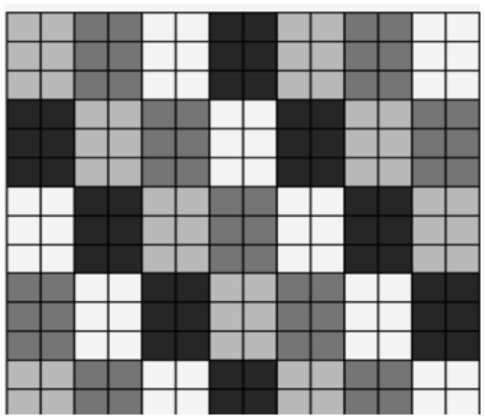
- Work allocation for 2D arrays, ﬀten each calculation will depend on **several neighbouring values** in the array E.g. calculating the average value of that item and its neighbours, and other stencil computations
- In this case it is usually best to allocate **square-like blocks** of values to each process More of the values needed are therefore local to the process.
- And for bigger arrays the difference grows as $O(n/2)$ vs $O(n)$

4.2 Overlap regions

- The block allocation still requires some **non-local references**
- It will usually be better to explicitly **fetch and cache** the required overlap regions first, and then perform the computation entirely locally
- Batching communication like this also tends to reduce overhead, esp. on disjoint memory machines
- If the computation is iterative then threads will need to synchronise at each step, e.g. with a barrier

4.3 Block cyclic allocations

- In some algorithms work is **not proportional** to the amount of data. E.g. some iterative algorithms require different numbers of iterations for different parts of the data
- In this case giving each process a single large contiguous block of data may result in **poor load balancing**
- It is better to allocate each process many **smaller blocks of data spread across the entire array**. Typically using a block-cyclic allocation



Each process is still allocated data in blocks, rather than rows, to reduce **non-local memory access**. But each process is allocated many different blocks, so that hopefully on average each process ends up with a comparable amount of work to do, even if some blocks take longer than others.

5 Dynamic work allocation

A static work allocation may not always be possible or efficient, e.g.

- Data references may be irregular, as in a graph or mesh
- The amount of work required for each data item may be very variable and unpredictable
- E.g. with an adaptive algorithm, that changes its behaviour depending on the results so far
- Work may continue to arrive while the program executes. e.g. an online server

5.1 Work queues

- The work queue itself is a shared data structure that holds the definitions of the currently unallocated tasks
- E.g. blocks of data to be processed
- The simplest possible work queue is a first-in first-out (FIFO) list of tasks
- The task description in the work queue should be as small as possible (to reduce overhead)
- Tasks should be of suitable granularity Too many small tasks will increase overhead; too few large tasks will result in idle processors
- For a large system multiple parallel work queues should be used to prevent the work queue becoming a serial bottleneck

Reference section

placeholder