Toward HPC

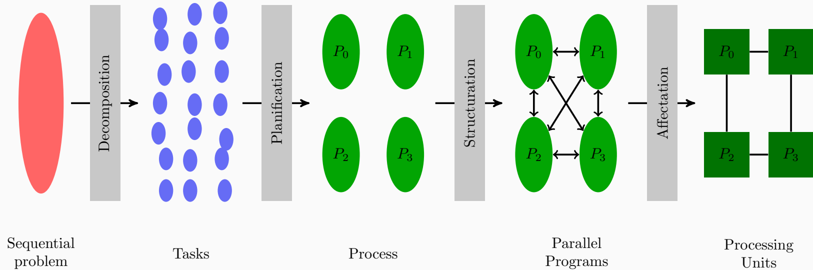
Chapter 5 – Patterns

M1 – MSIAM

March 15, 2017

So far . . .

Foster design



Serial control patterns

Definition

Structured serial programming is based on these patterns: sequence, selection, iteration, and recursion

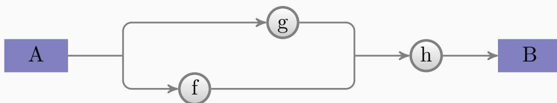
The nesting pattern can also be used to hierarchically compose these four patterns

Though you should be familiar with these, it's extra important to understand these patterns when parallelizing serial algorithms based on these patterns

Sequence

Ordered list of tasks that are executed in a specific order

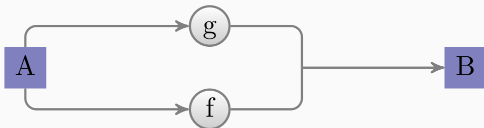
Assumption – program text ordering will be followed (obvious, but this will be important when parallelized)



Selection

Condition c is first evaluated. Either task a or b is executed depending on the true or false result of c.

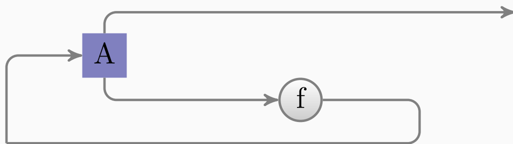
Assumptions – a and b are never executed before c, and only a or b is executed - never both



Iteration

Condition c is evaluated. If true, a is evaluated, and then c is evaluated again. This repeats until c is false.

Complication when parallelizing: potential for dependencies to exist between previous iterations



Recursion

Dynamic form of nesting allowing functions to call themselves

Tail recursion is a special recursion that can be converted into iteration – important for functional languages

Parallel control patterns

Definition

Parallel control patterns extend serial control patterns

Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns

Parallel control patterns: fork-join, map, stencil, reduction, scan, recurrence

Fork-Join

allows control flow to fork into multiple parallel flows, then rejoin later.

A ”join” is different than a ”barrier”

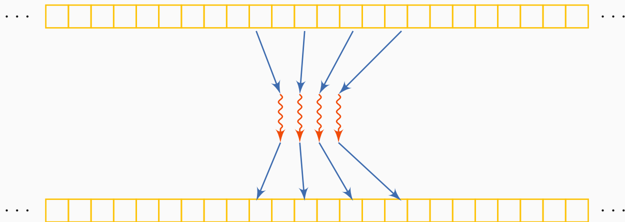
- Sync – only one thread continues
- Barrier – all threads continue

Map

Performs a function over every element of a collection

Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection

The replicated function is referred to as an “elemental function”

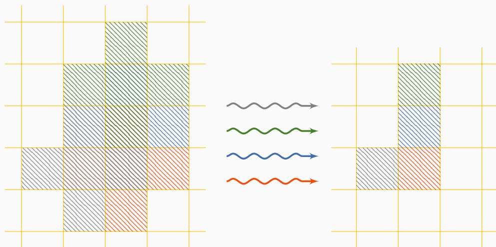


Stencil

Elemental function accesses a set of “neighbors”, stencil is a generalization of map

Often combined with iteration – used with iterative solvers or to evolve a system through time

Boundary conditions must be handled carefully in the stencil pattern



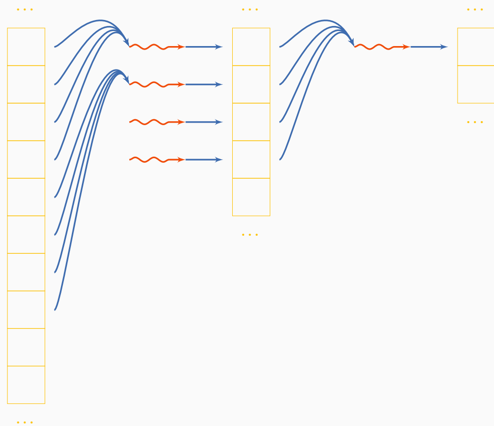
Réduction

Combines every element in a collection using an associative "combiner function"

Because of the associativity of the combiner function, different orderings of the reduction are possible

Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

Réduction



Scan

Computes all partial reduction of a collection

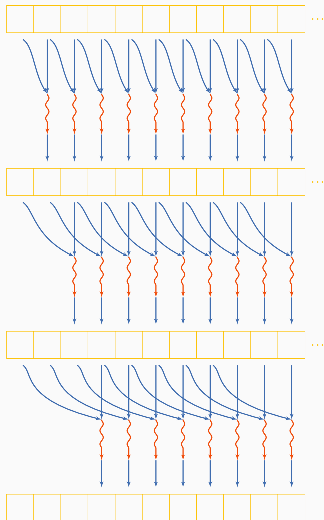
For every output in a collection, a reduction of the input up to that point is computed

If the function being used is associative, the scan can be parallelized

Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop

A parallel scan will require more operations than a serial version

Scan



Recurrence

More complex version of map, where the loop iterations can depend on one another

Similar to map, but elements can use outputs of adjacent elements as inputs

For a recurrence to be computable, there must be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs

Serial data management patterns

Definition

Serial programs can manage data in many ways

Data management deals with how data is allocated, shared, read, written, and copied

Serial Data Management Patterns: random read and write, stack allocation, heap allocation,objects

Random read/write

Memory locations indexed with addresses

Pointers are typically used to refer to memory addresses

Aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelized

Stack Allocation

Stack allocation is useful for dynamically allocating data in LIFO manner

Efficient – arbitrary amount of data can be allocated in constant time

Stack allocation also preserves locality

When parallelized, typically each thread will get its own stack so thread locality is preserved

Heap Allocation

Heap allocation is useful when data cannot be allocated in a LIFO fashion

But, heap allocation is slower and more complex than stack allocation

A parallelized heap allocator should be used when dynamically allocating memory in parallel

This type of allocator will keep separate pools for each parallel worker

Objects

Objects are language constructs to associate data with code to manipulate and manage that data

Objects can have member functions, and they also are considered members of a class of objects

Parallel programming models will generalize objects in various ways

Parallel data management patterns

Definition

To avoid things like race conditions, it is critically important to know when data is, and isn't, potentially shared by multiple parallel workers

Some parallel data management patterns help us with data locality

Parallel data management patterns: pack, pipeline, geometric decomposition, gather, and scatter

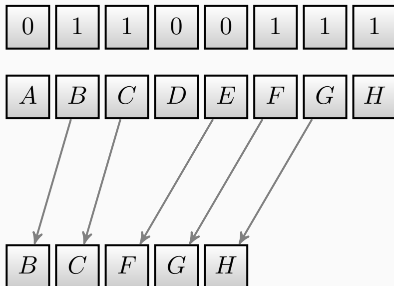
Pack

Pack is used eliminate unused space in a collection

Elements marked false are discarded, the remaining elements are placed in a contiguous sequence in the same order

Useful when used with map

Unpack is the inverse and is used to place elements back in their original locations



Pipeline

Connects tasks in a producer consumer manner

A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible

Pipelines are most useful when used with other patterns as they can multiply available parallelism

Geometric Decomposition

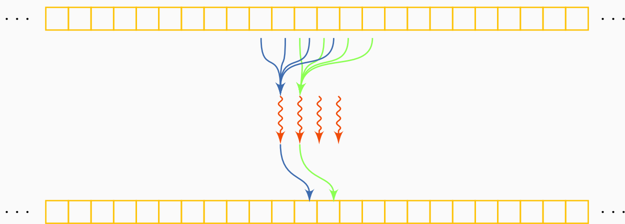
Arranges data into subcollections

Overlapping and non-overlapping decompositions are possible

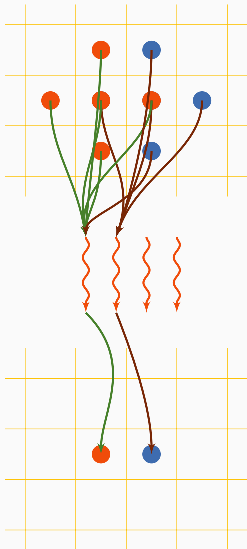
This pattern doesn't necessarily move data, it just gives us another view of it

Gather pattern

All the threads read data from specific and distincts places. Some operation is realized on the data. One thread write the result in a unique place.

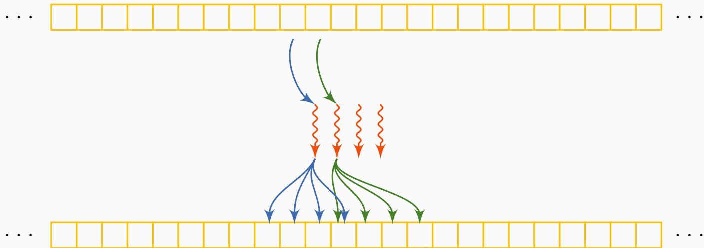


Gather pattern



Scatter pattern

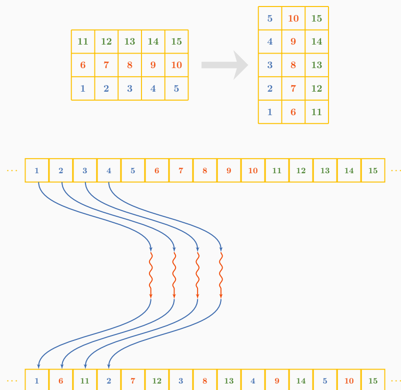
All the threads compute the memory space to which the output will be written.



Scatter pattern (2)

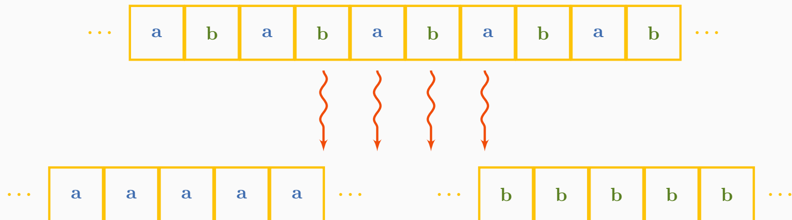
Transposition pattern (1)

All the threads read data in some array and rewrite it to some other part of the array. The position is well defined.



Transposition pattern (2)

We can also use the concept of transposition for an arrays of structures to build a structure of array.



Other patterns

Superscalar Sequences: write a sequence of tasks, ordered only by dependencies

Futures: similar to fork-join, but tasks do not need to be nested hierarchically

Speculative Selection: general version of serial selection where the condition and both outcomes can all run in parallel

Workpile: general map pattern where each instance of elemental function can generate more instances, adding to the “pile” of work

Other patterns

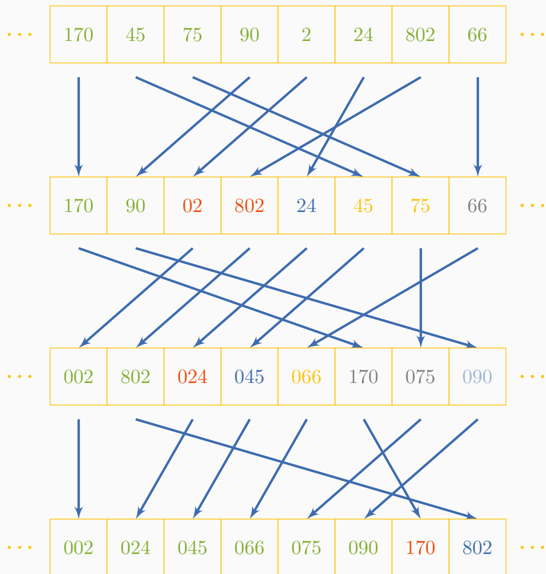
Search: finds some data in a collection that meets some criteria

Segmentation: operations on subdivided, nonoverlapping, non-uniformly sized partitions of 1D collections

Expand: a combination of pack and map

Category Reduction: Given a collection of elements each with a label, find all elements with same label and reduce them

Radix sort



Conclusion

Specific patterns

*

Performances

Measure of performances

- A sequential computation has 3 phases: setup, processing and completion.

$$T_{total} = T_{init} + T_{calcul} + T_{comp}$$

- If the computation is made using P units

$$T_{total}(P) = T_{init} + \frac{T_{calcul}}{P} + T_{compl}$$

Definition

We can define the speedup

$$S(P) = \frac{T_{total}}{T_{total}(P)}$$

and the efficiency

$$E(P) = \frac{S(P)}{P}$$

Amdhal's law (1)

- The part of the code that cannot be executed simultaneously represent a certain percentage of the total time

$$\gamma = \frac{T_{init} + T_{compl}}{T_{total}(1)}$$

Definition 1

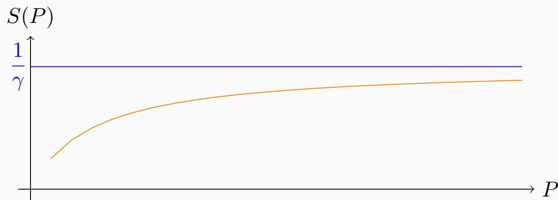
Using γ, we can write the Amdhal's law

$$S(P) = \frac{T_{total}(1)}{\left(\gamma + \frac{1-\gamma}{P}\right) T_{total}(1)} = \frac{1}{1 + \frac{1-\gamma}{P}}$$

Amdhal's law (2)

- The scalar part γ contains all the timing corresponding to thread managements, scalar computations, os operations.
- The parallel part $1 - \gamma$ is completely parallel
- The upper bound for the speedup is given by

$$S(P) < \frac{1}{\gamma}$$



- If the workload does not increase, it is unnecessary to increase the number of processor.

Gustafson's law

- γ is normalized with respect to the number of processing units.

$$\gamma(P) = \frac{T_{init} + T_{compl}}{T_{total}(P)}$$

> **Definition 2**
>
> Gustafson's law writes
>
> $$S(P) = P + (1 - P)\gamma(P)$$

- Using this law, we can study the impact of the number of processing unit on the computation.

Notes on performances (1)

- Quote only 32-bit performance results, not 64-bit results.
- Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
- Quietly employ assembly code and other low-level language constructs.
- Scale up the problem size with the number of processors, but omit any mention of this fact.
- Quote performance results projected to a full system.
- Compare your results against scalar, unoptimized code on Crays.

Notes on performances (2)

- When direct run time comparisons are required, compare with an old code on an obsolete system.
- If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.
- Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.
- Mutilate the algorithm used in the parallel implementation to match the architecture.
- Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.
- If all else fails, show pretty pictures and animated videos,