HPC: Patterns

Parallel programming models

None of most used programming languages was designed for parallel programming

Evolution (compiler directives, libraries)

Portable parallel programming

- Portability is **impaired** when programming using **hardware mechanisms**
- **Nested parallelism** is important, **nearly impossible** to manage when implied by specific mechanisms like **threads**
  - Tasks have less overhead, better opportunities to allocate resources
  - Important to design libraries without exposing internal details
- Other mechanisms for parallelization exists and must be considered (**vectorization**)
  - Abstraction avoids dependencies on instruction set ...

Parallel programming models

Regular data parallelism (map)

- Array expressions (Directives for vector operations) (OpenMP)
- Elemental functions (OpenCL)

Other patterns (scan, reduce) can also be expressed directly using programming models

Conversion of serial programs

- **Refactoring**
  - Array-of-structures and structures-of-arrays (better for vectorization)
- **Compiler** technology support
  - Easy conversion of serial programs (OpenMP)
  - Pragma directives

Parallel programming models

Problem of composability

- **Two different programming models used simultaneously**
- Problem of **nesting**
  - Serial library called in parallel program was upgraded to parallel
- **Thread** oversubscription
  - OpenMP not optimal, threads created as part of the execution model

Portability

- OpenMP, OpenCL, and MPI are standards, many portable implementations are available
- Performance: abstract programming models are crucial
  - OpenMP/MPI offers strong performance portability
  - OpenCL is more tied to hardware, tuning can favour one hardware

Determinism

- The same results as serial not always needed, testing for correctness

Pattern-based programming

Patterns are “best practices” for solving specific problems.

Patterns can be used to organize your code, leading to algorithms that are **more scalable and maintainable**.

A **pattern** supports a particular algorithmic **structure** with an efficient implementation.

Good parallel programming models support **a set of useful parallel patterns** with low-overhead implementations.

Pattern-based programming

Focus on algorithm strategy patterns

- **Algorithm** skeletons
- **Not** design patterns (high level, abstract)
- **Not** implementation patterns (low-level, hardware specific)

Patterns

- Semantics
  - building blocks, arrangement of tasks, data dependencies
  - design phase
- Implementation
  - Granularity, good use of cache

Focus on **data parallelism to ensure scalability**

Pattern-based programming

Task

- Task is a unit of potentially parallel work
- Tasks are executed by scheduling on software threads
  - Usually cooperative: at predicted switch points
- Software threads are scheduled by OS onto hardware threads
  - Preemptive approach is most common (at any time)

Serial Patterns

Patterns are universal

- Can be applied to any programming system
- They lead to well-structured, maintainable, efficient code
- New patterns can be derived from existing patterns

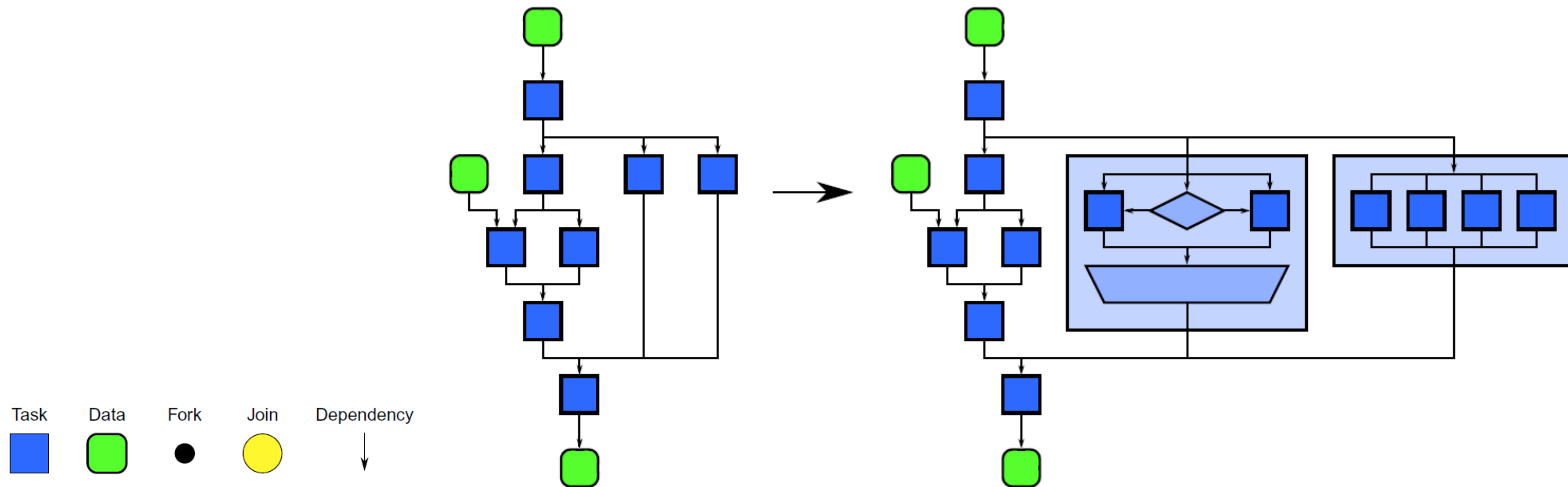
Serial patterns → parallel patterns

Nesting

Fundamental compositional pattern

Allows for hierarchical composition

Any task block in a pattern can be replaced with a pattern with the same input and output configuration and dependencies



Nesting

Nesting is crucial for structured, modular code

Static

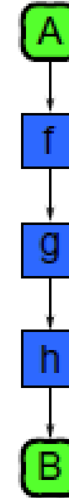
- Code structure
- Functional decomposition
- Serial program is composed of
  - sequence, selection, iteration, recursion
  - goto sentence violates nesting

Serial Control Flow Patterns

Sequence

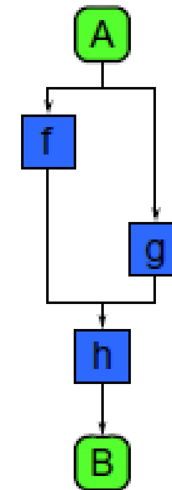
- No data dependencies
- Data dependencies restrict the order

```
1  T = f(A);
2  S = g(T);
3  B = h(S);
```



- Parallel generalization: superscalar sequence
  - removes code-text order constraint
  - Order tasks only by data dependencies

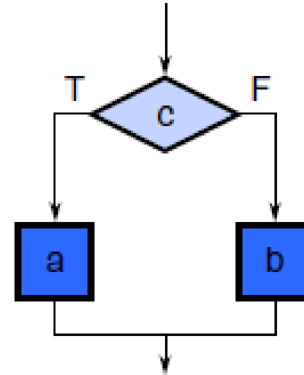
```
1  T = f(A);
2  S = g(A);
3  B = h(S,T);
```



Serial Control Flow Patterns

Selection

```
1  if (c) {
2    a;
3  } else {
4    b;
5  }
```

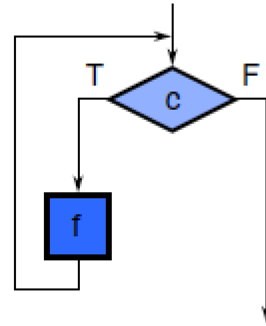


- Parallel generalization: speculative selection
  - a, b, c may be executed in parallel,
  - a or b is discarded when c is known

Serial Control Flow Patterns

Iteration

```
1  while (c) {
2    a;
3  }
```




```
1  for (i = 0; i < n; ++i) {
2    a;
3  }
```

```
1  i = 0;
2  while (i < n) {
3    a;
4    ++i;
5  }
```

- Countable iteration
  - Equivalence of for and while loops
- Parallel generalization
  - Task f may depend on previous invocations of itself (loop-carried dependencies)
  - The serial iteration pattern appears in several different parallel patterns
    - map, reduction, scan, recurrence, scatter, gather, pack
    - parallel patterns have a fixed number of invocations, known in advance

Serial Control Flow Patterns

Iteration

- Problem of hidden data dependencies
  - Parallelization capability depends on a[i], b[i], c[i], d[i]
  - Pointers
    - y can point to the same location as x

```
9    for (int i = 0; i < n; ++i)
10     x[a[i]] = x[b[i]] * x[c[i]] + x[d[i]];
11   }
```

```
10   for (int i = 0; i < n; ++i)
11     y[a[i]] = x[b[i]] * x[c[i]] + x[d[i]];
12   }
```

Serial Control Flow Patterns

Recursion

- dynamic nesting which allows functions to call themselves
- stack memory allocation

Serial data management patterns

Random read and write

- working with pointers, possible aliasing (two pointers refer to the same memory object)
- Aliasing can make vectorization and parallelization difficult
  - Undefined result
  - Extra object copies → slow
  - Burden put on programmer
- Working with array indices is safer and easier to transfer to another platform

Stack allocation

- Efficient as arbitrary amount of data can be allocated and preserves locality
- Each thread gets its own stack

Serial data management patterns

Heap allocation

- Slower then stack allocation
- Allocation scattered all over memory (matrix allocation)
  - More expensive due to memory hardware subsystem
- Implicitly sharing the data structure can lead to scalability problems
  - Better: maintain separate memory pool on each worker and avoid global locks
  - Get rid of false sharing

HPC: Parallel Patterns

Map

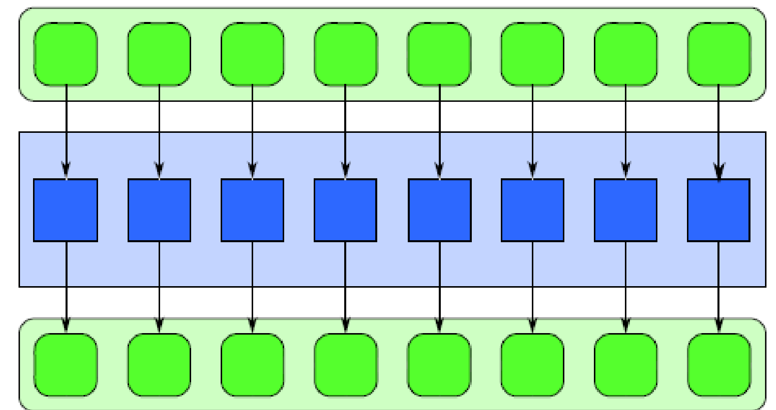
Replicates elemental function over every element of index set

Map replaces iterations of independent loops

Elemental functions should not modify global data that other instances (iterations) depend on

Examples:

- gamma correction in images, color space conversions,
- Monte Carlo sampling, ray tracing



Map

Map applies an elemental function to every element of a collection of data in parallel

- Elemental functions should have no side effects
- No dependency among elements
- Can execute in any order

Embarrassingly parallel

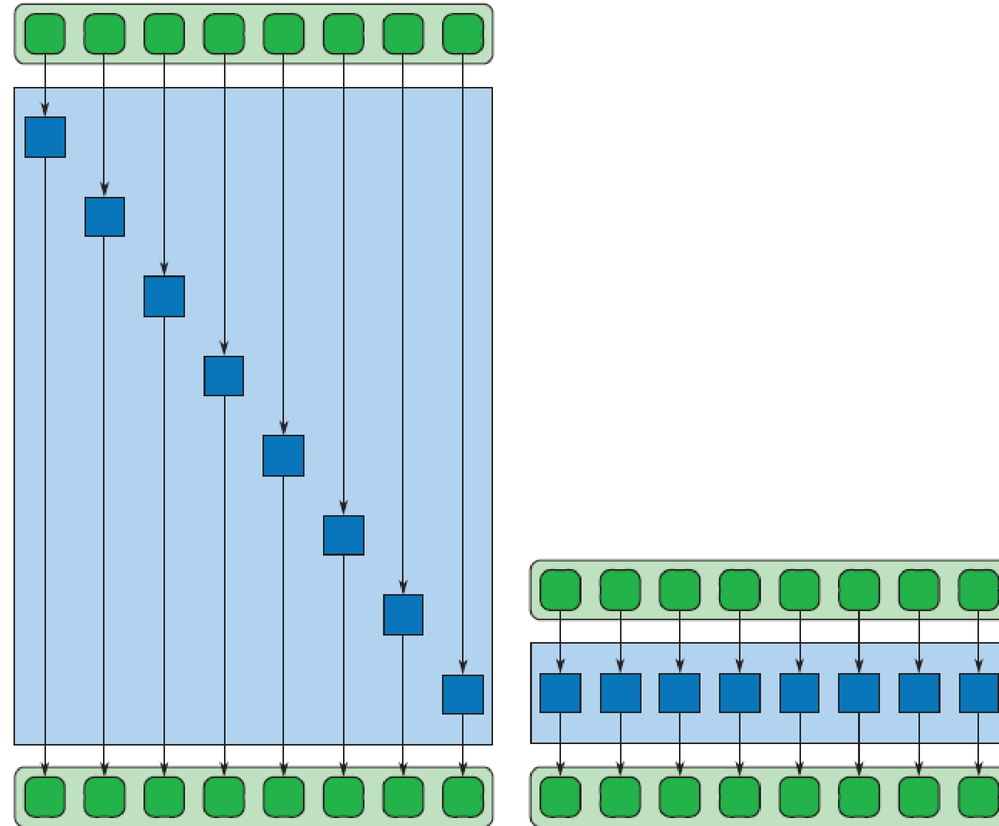
- One of the most efficient patterns
- If you have many problems to solve, parallel solution can be as simple as running problems in parallel

Typically combined with other patterns

- map does the basic computation, other patters follow
  - gather = serial random read + map, reduction, scan

Map

Serial and parallel execution of map pattern



Map

Scalable implementation of map
- A lot of care for best performance
- Threads
  - Mandatory parallelism
  - Separate thread for each element is not a good idea
- Tasks
  - Optional parallelism
  - Overhead and synchronization at the beginning and at the end when elemental functions vary in the amount of work

Map is a basis for vectorization and parallelization

Map-reduce
- Google’s big success

Map

Map is related to SIMD, SPMD, SIMT

- can be expressed as a sequence of vector operations

Parallel for construct in programming languages

- Map is parallelization of the serial iteration pattern where iterations are independent

If dependencies and side-effects are avoided, map is deterministic

saxpy

Scaled vector addition

- (**s**axpy - single precision)
- Elemental function
  - Uniform and varying parameters
- Serial implementation

$$\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y}.$$

$$f(t,p,q) = tp + q,$$

$$\forall i: \ y_i \leftarrow f(a, x_i, y_i).$$

```
7      for (size_t i = 0; i < n; ++i)
8          y[i] = a * x[i] + y[i];
9  }
```

Mandelbrot set

Problem

- Mandelbrot set (black) is the set of all points c in the complex plane that do not go to infinity with iterations
  - Divergence for large z
  - Compute the function up to some maximum value K
- Serial control flow in elemental functions

$$z_0 = 0,$$

$$z_{k-1} = z_k^2 + c,$$

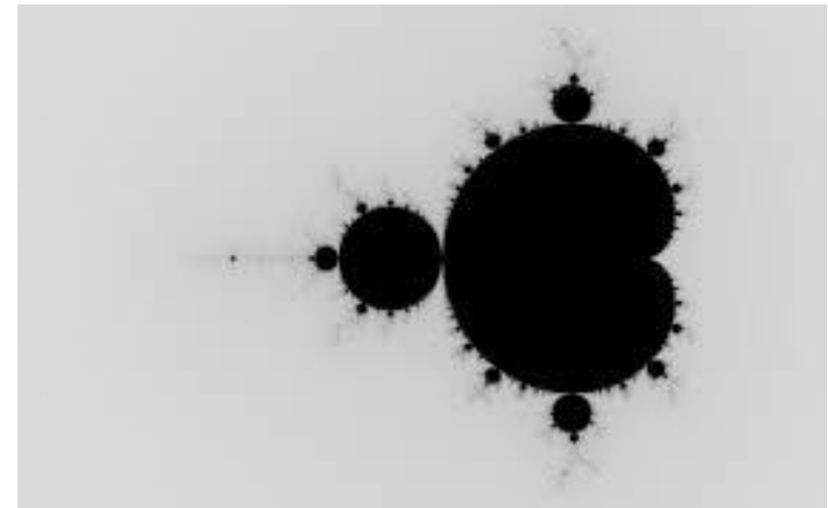
$$\text{count}(c) = \min_{0 \le k < K} (|z_k| \ge 2).$$

Problem

- Load imbalance

Implementation

- Opposed to saxpy it cannot be efficiently computed on SIMD machines
- best with SPMD or tiled SIMD



Mandelbrot set

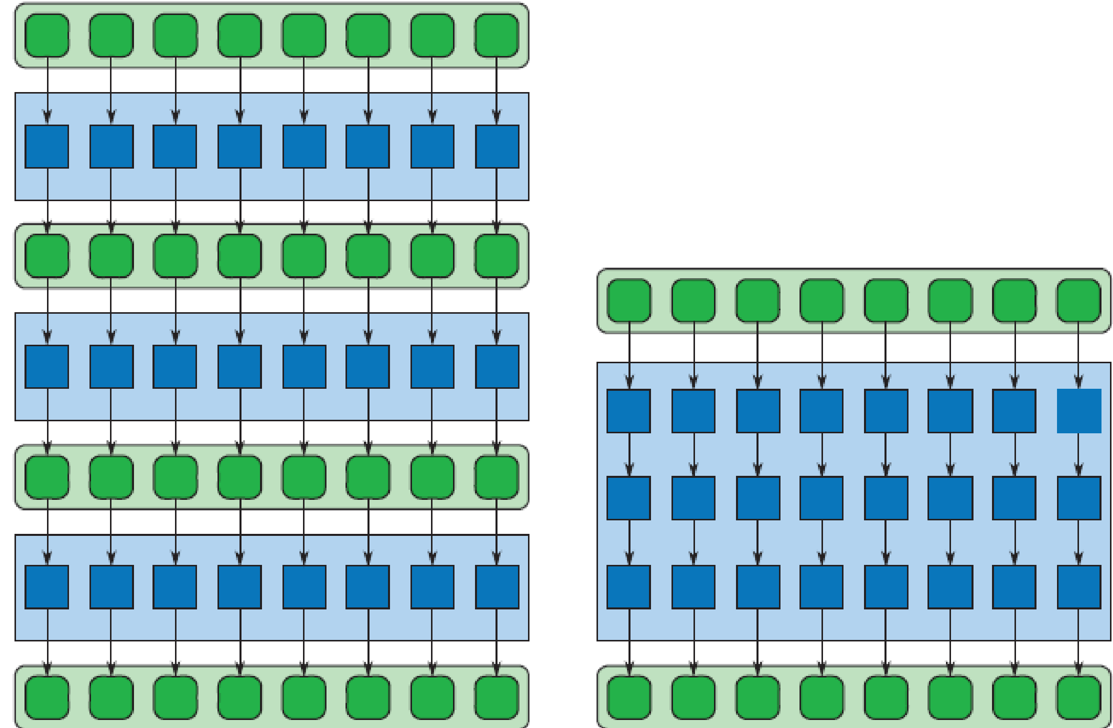
Serial

```
1  int mandel(
2     Complex c,
3     int depth
4  ) {
5     int count = 0;
6     Complex z = 0;
7     for (int k = 0; k < depth; k++) {
8        if (abs(z) >= 2.0) {
9           break;
10       }
11       z = z*z + c;
12       count++;
13    }
14    return count;
15 }
16
17 void serial_mandel(
18    int p[][],
19    int max_row,
20    int max_col,
21    int depth
22 ) {
23 for (int i = 0; i < max_row; ++i) {
24    for (int j = 0; j < max_col; ++j)
25       p[i][j] = mandel(Complex(scale(i), scale(j)),
26                        depth);
27 }
```

Code fusion

Map of sequence can avoid intermediate memory operations

- Intermediate results are written in registers
- reduced memory bandwidth, cache and virtual memory problems
- Less synchronization at start/end of map



Cache fusion

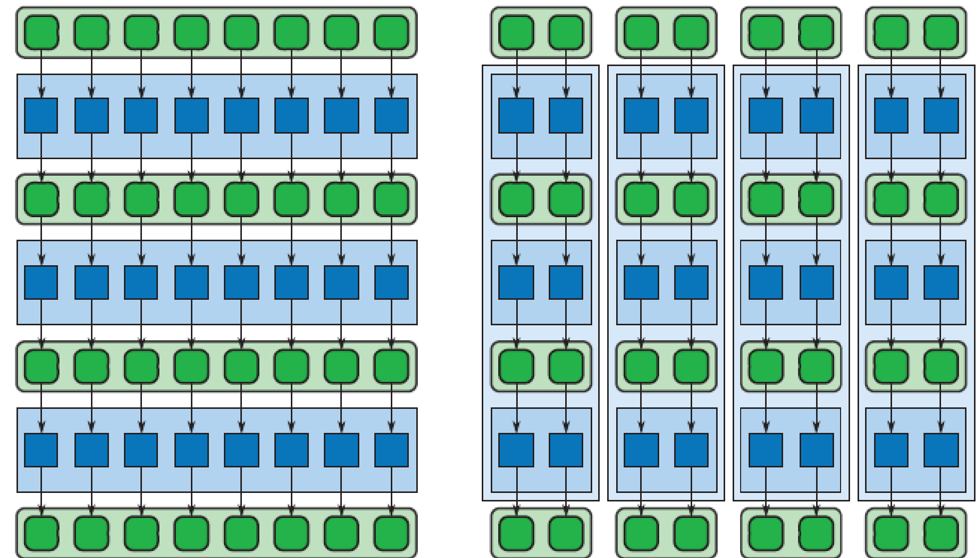
Maps broken to tiles

- each tile is executed sequentially on one core
- tile should fit in cache, avoid accessing main memory

Cache fusion on OpenMP

- Single parallel region
- Loop for each map has the same bounds and chunk size
- Each loop uses static scheduling

Can use both, code fusion is preferred



Related patterns

Stencil

- access to neighbors to get inputs of elemental function
- Important is reading of data
  - Data reuse
  - Hardware specific for good results (cache size, GPU memory)

Workpile

- Work grows as it is consumed by map
- Could be implemented in OpenMP and OpenCL using explicit work queues

Divide-and-conquer

- Recursive division into smaller parallel subproblems until base is reached, which can be solved sequentially
- Combination of partition and map patterns
- OpenMP: supported through tasking model
- OpenCL: no support for nested parallelism, extremely difficult with work queues

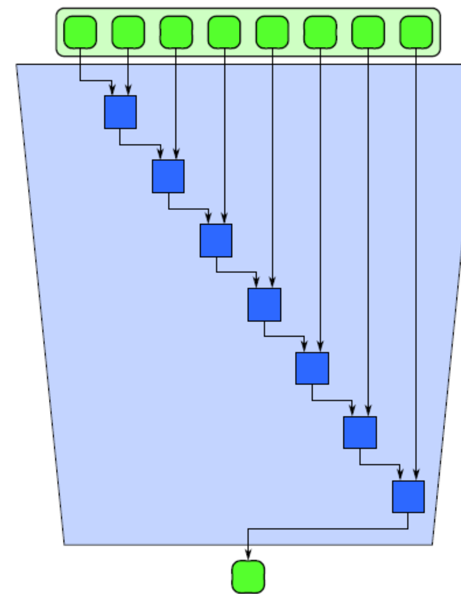
HPC: Reduction pattern

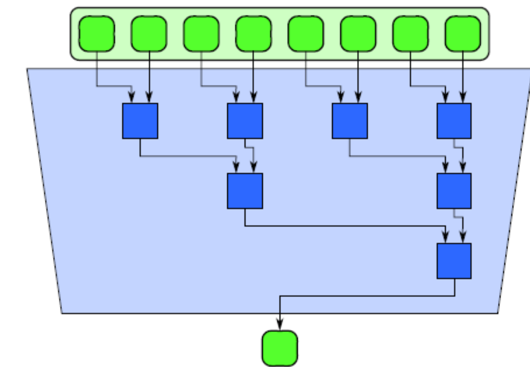
Reduce

A collective operation

Reduce pattern allows data to be combined

- Combiner function $f(a, b) = a \otimes b$
- Pairwise operation
- Associativity must hold
  - $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
  - Operands can be combined in any order
  - Floating point addition and multiplication are only partially associative
- Commutativity
  - $a \otimes b = b \otimes a$
  - Not required, but enables additional reorderings
- Identity
  - reduction of empty data collection is meaningful
  - Initial value of reduction





Tiling

Use serial algorithm where possible

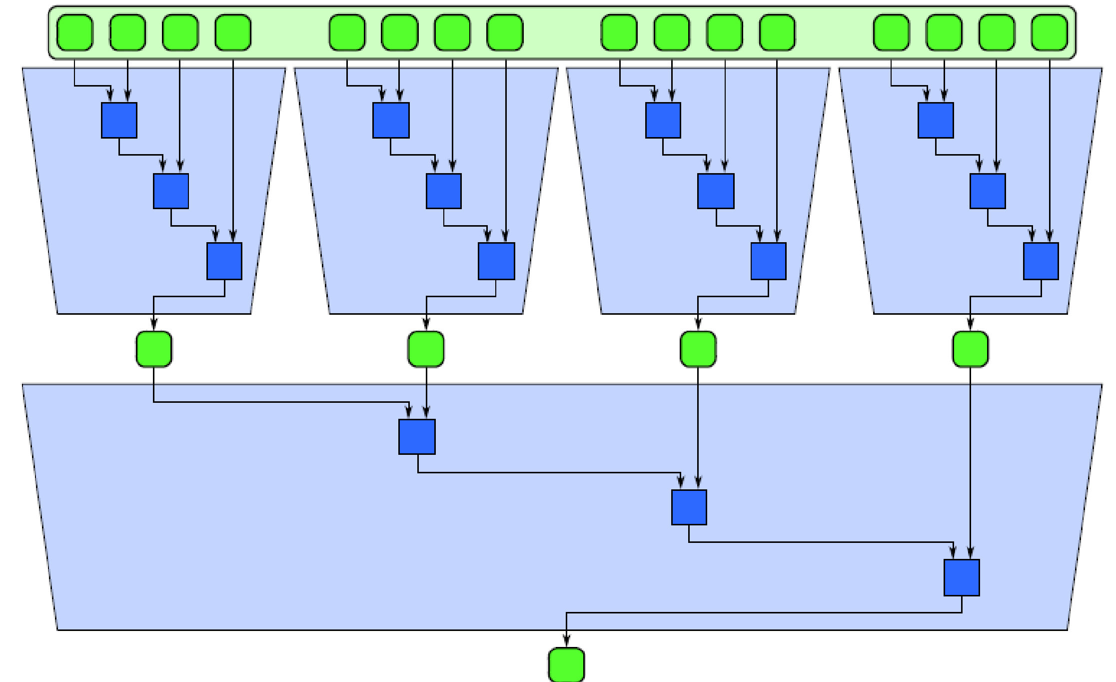
Do tree-like reduction to reduce communication costs

Process

- Break the work to tiles
- Operate on tiles separately
- Combine partial results from tiles

Serial and tree algorithms

- use the same number of application of the reduce function
- Serial algorithm requires less storage for intermediate results



Precision

Large summations

- Running out of bits to represent intermediate results
- Solution: use higher precision for reduction
- Example:
  - Summation of a large number of equal values
  - Expressed in single-precision arithmetic (precision to 6 or 7 decimal places)
  - Serial approach
    - At one point intermediate sum will be very large compared to operand
    - Adding will heave no effect on intermediate result
  - Approach with tree
    - Intermediate results will be of similar magnitude
    - Summation of operands with similar magnitude is more accurate

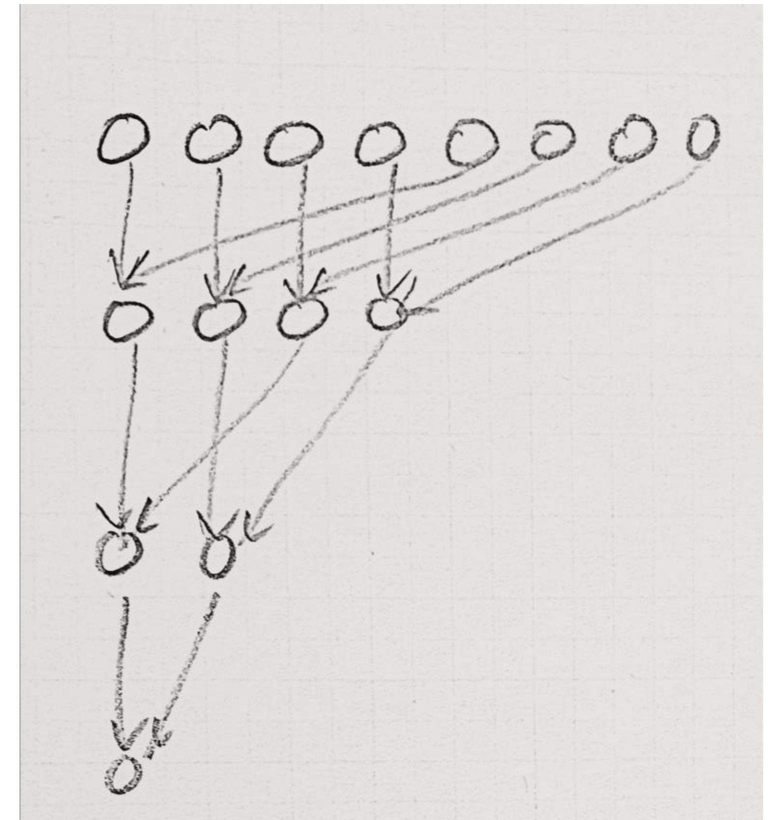
Theoretical considerations

Sequential reduction of n operands

- $n - 1$ reductions
- Each invocation of reduce function costs χ
- Total execution time $t_s(n) = \chi(n-1)$

Parallel reduction, $n = 2^k, k \in \mathbb{N}$

- Communication costs λ
- $n/2$ reductions in the first stage can go in parallel,
  $n/4$ in the second stage can go in parallel ...
  1 reduction in the last stage
- altogether we have $\log_2 n$ stages with $n - 1$ reductions
- Total execution time $t_p(n) = (\chi + \lambda)\log_2 n$



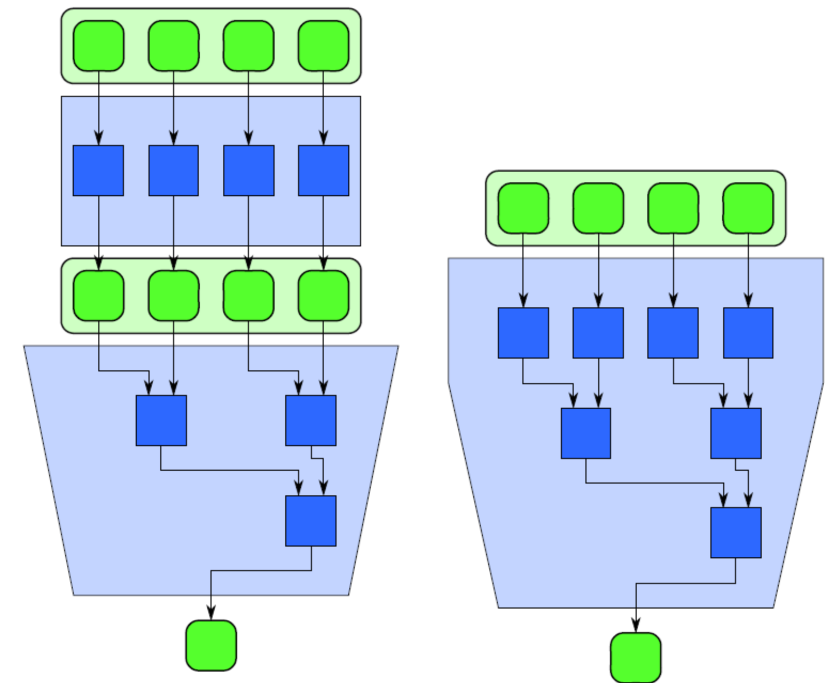
Fusing map and reduce

When map is feeding outputs directly into a reduction, the combination can be implemented more efficiently

No need for synchronization between map and reduce stages

No need to write intermediate results to memory or file

Map and reduce must be tiled in a same way



OpenMP continued

Distribution of work

parallel for

- most frequently used functionality of openmp
- Distributes iterations among threads
- For statement must be given in canonical shape

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    hardWork(i);
}
```

```
                                     { idx++           }
                                     { ++idx           }
                       { <  }        { idx--           }
                       { <= }        { --idx           }
for (idx=start; idx    { >= }  end;  { idx += inc      } )
                       { >  }        { idx -= inc      }
                                     { idx = idx + inc }
                                     { idx = inc + idx }
                                     { idx = idx - inc }
```

- parallel for collapse(num)
  - enables to parallelize num perfectly nested loops together

Distribution of work

- parallel for schedule(type, number)
  - useful when workload significantly differs from iteration to iteration
  - schedule clause
    - static: iterations are mapped to threads at compile time, default
    - dynamic: when thread finishes its work, it gets new *number* of iterations to compute
    - guided: each threads gets larger portion of iteration at the beginning, then the number is reduced
    - runtime: set though environment variables

```
Iteracija 1 koncana (1).
Iteracija 3 koncana (1).
Iteracija 5 koncana (1).
Iteracija 7 koncana (1).
Iteracija 9 koncana (1).
Iteracija 0 koncana (0).
Iteracija 2 koncana (0).
Iteracija 4 koncana (0).
Iteracija 6 koncana (0).
Iteracija 8 koncana (0).
```

static, 1

```
Iteracija 1 koncana (1).
Iteracija 2 koncana (1).
Iteracija 3 koncana (1).
Iteracija 4 koncana (1).
Iteracija 5 koncana (1).
Iteracija 6 koncana (1).
Iteracija 7 koncana (1).
Iteracija 8 koncana (1).
Iteracija 9 koncana (1).
Iteracija 0 koncana (0).
```

dynamic, 1

```
void main(void)
{
    int i;

    #pragma omp parallel for schedule(dynamic, 1)
    for(i=0; i<10; i++)
    {
        int id = omp_get_thread_num();
        if(i==0)
            Sleep(1000);
        printf("Iteracija %i koncana (%d).\n", i, id);
    }
}
```

Synchronization

Critical / atomic

- Only one thread enters critical section at once
- atomic
  - Applicable only to scalar variable assignment with operators ++, --, +=, -=, *=, /=, &=, |=, <<=, >>=
  - Faster than critical
- Example
  - Correct counting although slower than with reduction

```
#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for
    for(i=0; i<N; i++)
        #pragma omp critical
        counter++;

    printf("Counter = %d\n", counter);
}
```

Barrier

- All processors must reach the barrier before any processor can proceed

Dot product

Problem

- Vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with n elements each
- Calculate $\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{n-1} a_i b_i$
- Combination of map and reduce
- Map: element-wise multiplications
- Reduce: summation of partial sums

OpenMP implementation for p tasks

- Clause reduction for limited number of operations
  - OpenMP 4.0 will probably have possibility to specify custom reduce function
- Manual tree-like reduction

Dot product

Sequential tiled map with sequential reduce of intermediate results

```c
// allocate and initialize vectors (sequential)
a = (double *)malloc(n*sizeof(double));
b = (double *)malloc(n*sizeof(double));
srand(time(NULL));
for(int i=0; i<n; i++)
{
    a[i] = 2.0*rand()/(double)RAND_MAX - 1.0;
    b[i] = 2.0*rand()/(double)RAND_MAX - 1.0;
}
```

```c
// parallel code - start
dt = omp_get_wtime();
omp_set_num_threads(p);

// tiled map with sequential reduction of intermediate results
sum = 0.0;
#pragma omp parallel private(id, istart, iend) firstprivate(sum)
{
    id = omp_get_thread_num();
    istart = id * n / (double)p;
    iend = (id + 1) * n / (double)p;
    for(int i = istart; i < iend; i++)
        sum += a[i] * b[i];
    #pragma omp atomic
    dotp += sum;
}

dt = omp_get_wtime() - dt;
// parallel code - end

printf("dotproduct: %lf\ntime: %lf\n\n", dotp, dt);
```

Dot product II

Map with step-by-step reduction

- does not use atomic directive

```
pdotp = (double *)malloc(p*sizeof(double));
```

```
// tiled map with sequential reduction
sum = 0.0;
#pragma omp parallel private(id, istart, iend) firstprivate(sum)
{
    id = omp_get_thread_num();
    istart = id * n / (double)p;
    iend = (id + 1) * n / (double)p;
    for(int i = istart; i < iend; i++)
        sum += a[i] * b[i];
    pdotp[id] = sum;
}
```

```
// tree-like reduction of intermediate results
p2 = exp2(floor(log2(p)));       // p2 = 2^k <= p
#pragma omp parallel private(i, id)
{
    id = omp_get_thread_num();
    if (p2 != p)                 // translate to power of 2
    {
        if (id >= p2)
            pdotp[id-p2] += pdotp[id];
        #pragma omp barrier
    }
    for(i = p2/2; i > 0; i /= 2)
    {
        if(id < i)
            pdotp[id] += pdotp[id+i];
        #pragma omp barrier
    }
}
dotp = pdotp[0];
```

Variables

Threads share global variables

Threads do not share

- Variables declared in parallel sections
- Variables of functions called from parallel sections

The scope of variable can be redeclared by clauses

- shared: default, no need to specify
- private: creates copies of global variable which are local for each thread
- firstprivate: like private, but also initializes local variables with current value of global variable
- lastprivate: when leaving parallel section it sets global variable to a value equal to the local variable of the thread which handled the last iteration
- threadprivate: values of local variables are kept in memory when parallel section is finished; useful when parallel sections are recreated

Variables

Example of using the private clause

```c
#include "omp.h"
#include <stdio.h>

void main(void)
{
    int i, j;

    #pragma omp parallel for private(j)
    for( i=0; i<2; i++)
        for(j=0; j<10; j++)
            printf("%d-%d\n", i, j);
}
```

- wrong result when private(j) is omitted →

```
1-0
1-1
1-2
1-3
0-0
0-5
1-4
1-7
1-8
1-9
0-6
```

- Correct result when using private(j) →

```
0-0
0-1
0-2
1-0
1-1
1-2
0-3
0-4
1-3
1-4
1-5
1-6
0-5
0-6
1-7
1-8
1-9
0-7
0-8
0-9
```

Variables

The scope of variable can be redeclared by clauses

- reduction(operator: variable)
  - When entering a parallel section local copies of a global variable are allocated and initialized
  - On parallel section exit all local copies are reduced and stored in a global variable
  - Reduction operators: +, *, &, |, ^, &&, ||

```
#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for reduction(+:counter)
    for(i=0; i<N; i++)
        counter++;

    printf("Counter = %d\n", counter);
}
```

Dot product III

Map and reduce using reduce clause

```
// parallel code - start
omp_set_num_threads(p);
dt = omp_get_wtime();

// map and reduce with OpenMP constructs
dotp = 0.0;
#pragma omp parallel for reduction(+:dotp)
for(int i=0; i<n; i++)
    dotp += a[i]*b[i];

dt = omp_get_wtime() - dt;
// parallel code - end

printf("dotproduct: %lf\ntime: %lf\n\n", dotp, dt);
```