

CODE OPTIMISATION

ON INTEL XEON (CO)PROCESSORS

André Pereira

ampereira@di.uminho.pt

Agenda

- * The Case Study
- * Identifying Inefficiencies
 - * Common code pitfalls
 - * Using compiler flags
- * Vectorisation
- * Shared Memory Parallelisation
- * Intel Xeon Phi Coprocessor

The Case Study

```
void matMult (float **a, float **b, float **c, int size) {  
    for (int i = 0; i < size; i++)  
        for (int j = 0; j < size; j++) {  
            c[i][j] = 0;  
            for (int k = 0; k < size; k++)  
                c[i][j] += a[i][k] * b[k][j];  
        }  
}
```

Common Code Inefficiencies

- * Avoidable memory accesses
- * False alias
- * Blocking/Tiling
- * Memory alignment - *to see later*
- * Row/Column major - *work assignment...*

Avoidable Memory Accesses

- ✱ Issue:

- ✱ The same element in a data structure (matrix c) is being accessed twice from memory per cycle iteration

- ✱ Solution:

- ✱ Use a temporary variable to store intermediate results

```
void matMult (float **a, float **b, float **c, int size) {  
    for (int i = 0; i < size; i++)  
        for (int j = 0; j < size; j++) {  
            c[i][j] = 0;  
            for (int k = 0; k < size; k++)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
        }  
}
```

False Alias

- ✱ Issue:

- ✱ a, b, or c may be pointing to overlapped memory blocks
- ✱ Compiler is very cautious using optimisations

- ✱ Solution:

- ✱ General C++ case prefer references over pointers!
- ✱ Give hints to the compiler (pragmas or restrict)

```
void matMult (float **a, float **b, float **c, int size) {  
    for (int i = 0; i < size; i++)  
        for (int j = 0; j < size; j++) {  
            c[i][j] = 0;  
            for (int k = 0; k < size; k++)  
                c[i][j] += a[i][k] * b[k][j];  
        }  
}
```

Memory Alignment

- * Issue:

- * Memory is not contiguously aligned
- * Alignment is not a multiple of 16 bytes

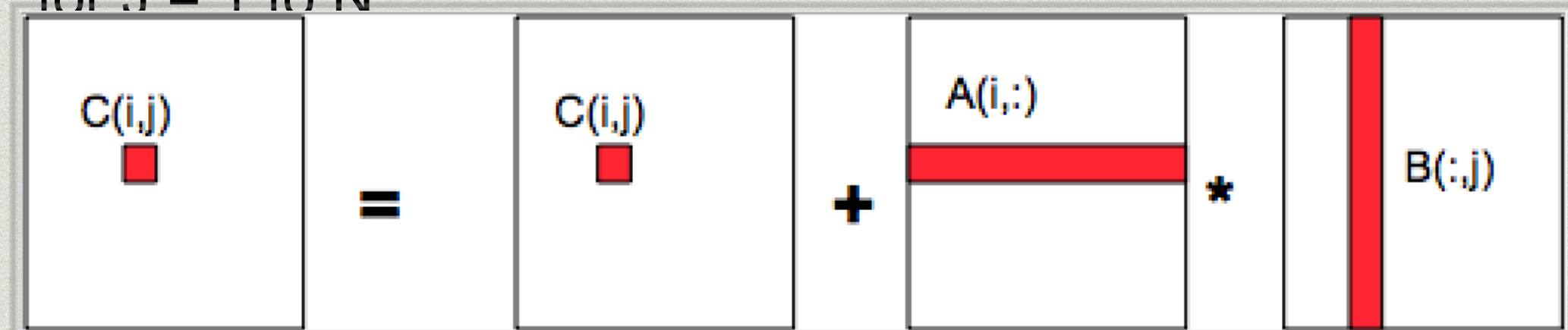
- * Solution:

- * Guarantee the proper alignment of data structures manually

Blocking/Tiling

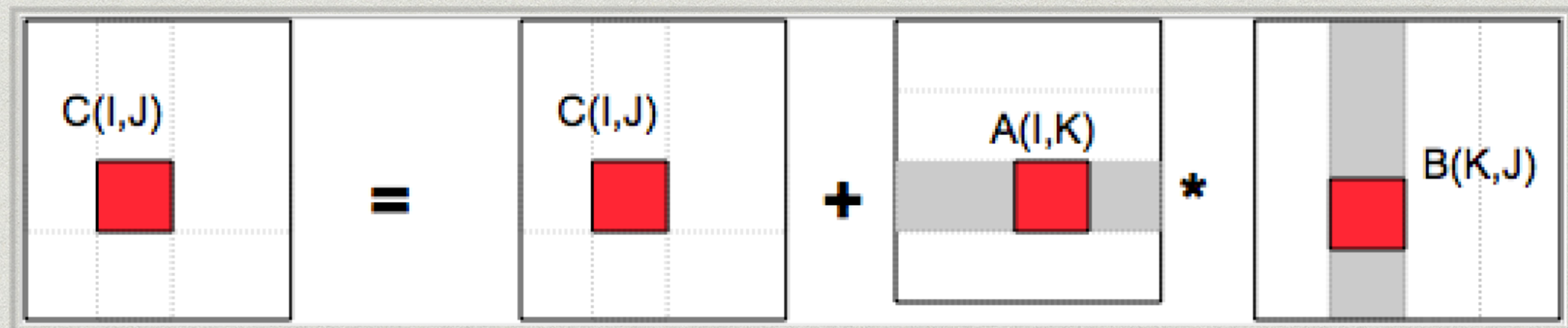
for $I = 1$ to N

for $J = 1$ to N



do a matrix multiply on blocks to compute block $C(I,J)$

{write block $C(I,J)$ back to slow memory}



Compiler Flags

- * Compilers provide flags to enable sets of optimisations
- * Both GNU and Intel compilers use the same syntax
- * “Free lunch” speedups!

Compiler Flags

- * -O1

- * Enable a small set of optimisations
- * Reduces both code size and execution time
- * Trade-off between performance and compilation time

- * -O2

- * Performs almost all supported optimisations
- * Generates faster code without compromising the space

- * -O3

- * Almost all available optimisations without considering any trade-off

Vectorisation

* How is it achieved? By hand?

```
void matrixAdd (void) {  
    __m256 ymm1, ymm2;  
  
    for (unsigned i = 0; i < SIZE; ++i) {  
        for (unsigned j = 0; j < VEC_SIZE; ++j) {  
            ymm1 = _mm256_load_ps(&m1[i][j * 8]);  
            ymm2 = _mm256_load_ps(&m2[i][j * 8]);  
  
            ymm1 = _mm256_add_ps(ymm1, ymm2);  
            _mm256_store_ps(&result[i][j * 8], ymm1);  
        }  
    }  
}
```

Vectorisation

- * Loop vectorisation is enabled by default with -O3 for the GNU compiler and -O2 for Intel compiler

```
src/matrix.cpp(102): (col. 3) remark: PERMUTED LOOP WAS VECTORIZED.  
src/matrix.cpp(102): (col. 3) remark: REMAINDER LOOP WAS VECTORIZED.  
src/matrix.cpp(105): (col. 4) remark: loop was not vectorized: not inner loop.  
src/matrix.cpp(101): (col. 2) remark: loop was not vectorized: not inner loop.
```

```
src/matrix.cpp(244): (col. 2) remark: loop was not vectorized: existence of vector dependence.  
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between line 24  
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between line 24  
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between line 24  
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between line 24
```

```
for (int k = 0; k < SIZE; k++)  
    c[i][j] += a[i][k] * b[k][j];  
}  
}
```

Vectorisation

- * Vectorisation report ICC: -qopt-report=X, where X
 - * 1 - Loops successfully vectorised
 - * 2 - Loops not vectorised (and the justification)
 - * 3 - Adds dependency information
 - * 4 - Non-vectorised loops report
 - * 5 - Non-vectorised loops report with dependency information

Help the compiler vectorise

- * Give information about loop dependencies
 - * `#pragma vector always`
 - * `#pragma ivdep`
- * Avoid nested loops
 - * or use `#pragma omp simd collapse(X)` - OpenMP 4.0
- * Data alignment and layout

Exercise 1

- * Perform basic code optimisations
- * Parallelise code execution for shared memory systems
- * <https://bitbucket.org/ampereira/matrix-optimization/downloads>

Shared Memory Parallelism

- * Several libraries to produce parallel code for shared memory environments
 - * **OpenMP**
 - * TBB
 - * CILK
 - * ...

OpenMP

- * Easy to use, pragma-based
- * Implemented by default in both GNU and Intel compilers
 - * `-fopenmp` - gcc
 - * `-openmp` - icc
- * Add the `<omp.h>` header to the code and use the pragmas

(Very) Basic Pragmas

- * `#pragma omp parallel` options

`#pragma omp for/task`

- * Options:
 - * `num_threads`
 - * `schedule(X)`
 - * `private(X)`

Go In-depth

- * Several work scheduling heuristics available
- * Definition of the task grain
- * Data scoping
- * Reduction, synchronisation, nowait clauses...
- * Parallel tasks for irregular problems
- * Nested parallelism

PARALLELIZATION STRATEGIES

PCAM Methodology

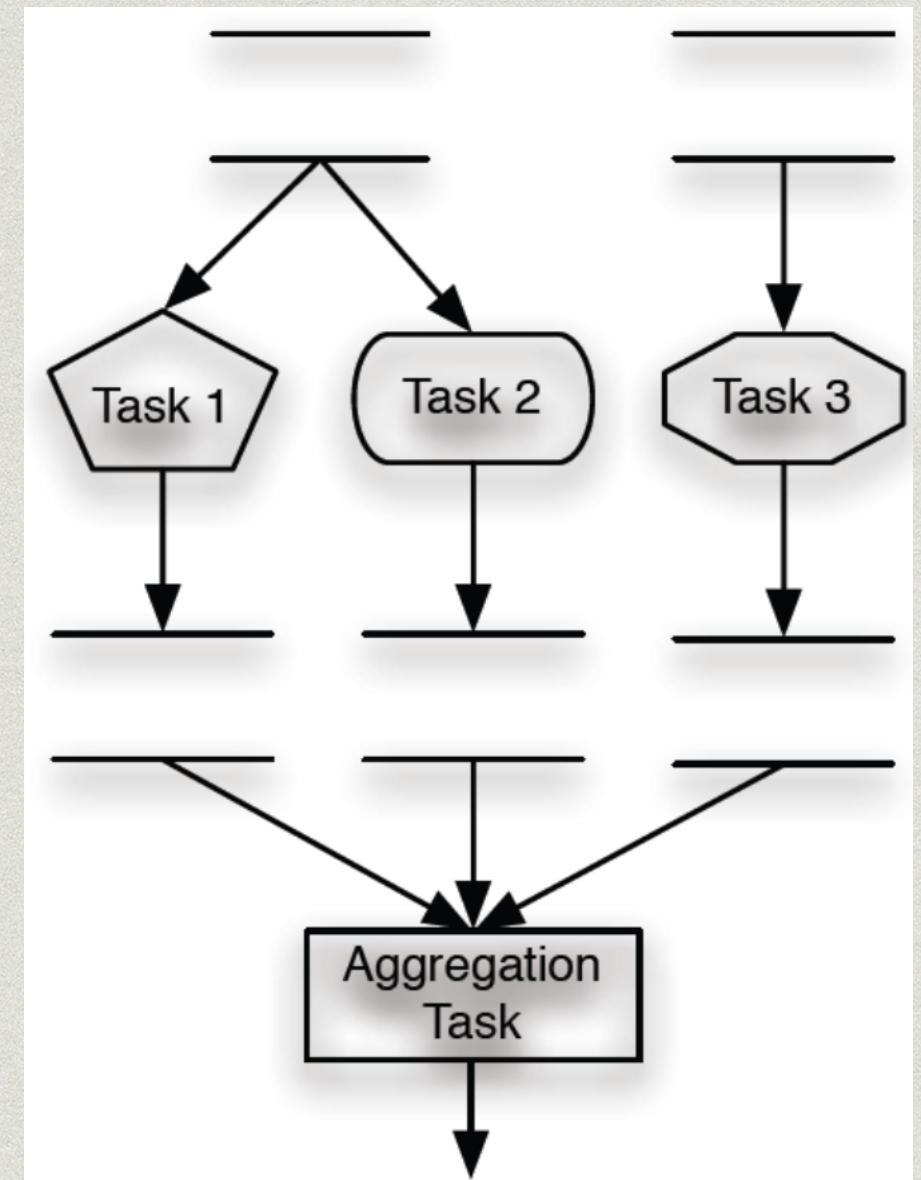
- * **P**artitioning
 - * Break computation into small pieces
- * **C**ommunication
 - * Identify communication among pieces
- * **A**gglomeration
 - * Group pieces to avoid communication
- * **M**apping
 - * Map the pieces to the computational devices

Decomposition Patterns

- * Task parallelism
- * Divide-and-conquer decomposition
- * Geometric decomposition
- * Recursive data decomposition
- * Pipeline decomposition

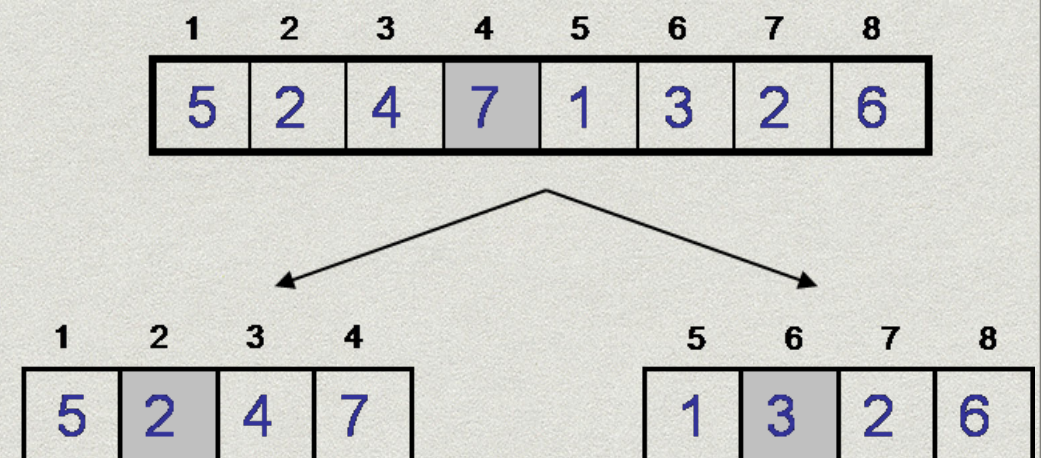
Task Parallelism

- * Divide the workload into tasks
- * Process independent tasks in parallel
- * Map tasks to processes/threads
- * Aggregate the results



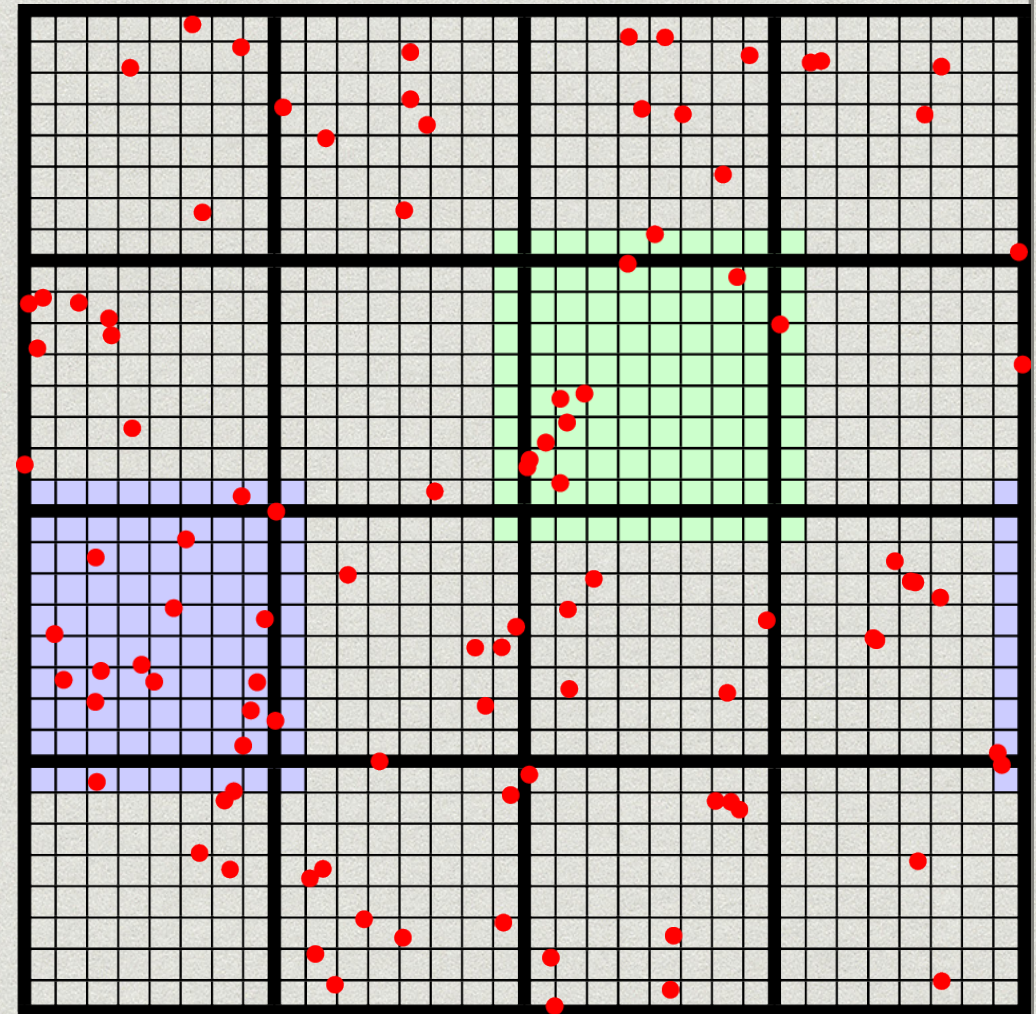
Divide-and-Conquer

- * Divide the algorithm into tasks
- * Tasks can be subdivided
- * Map groups of tasks to processes/threads



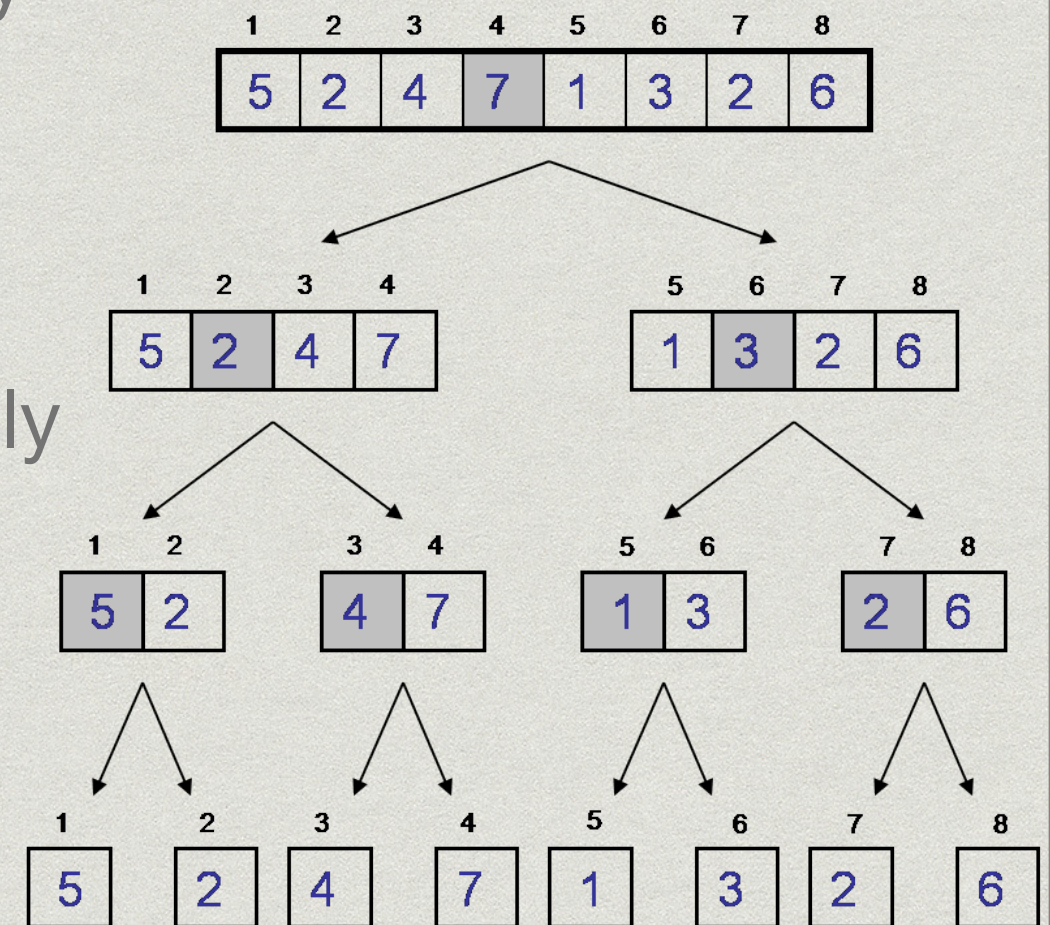
Geometric Decomposition

- * Divide the geometric space
- * Each sub-space is assigned to a process/thread
- * Communication may be required in the boundary regions



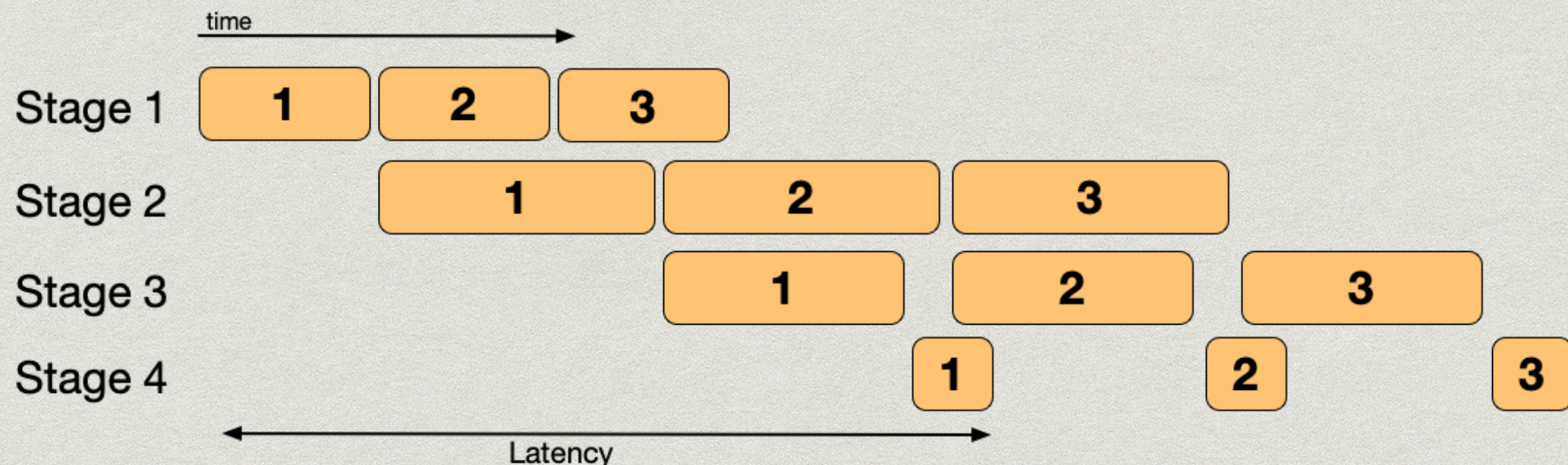
Recursive Decomposition

- * Data structures that cannot be easily partitioned
 - * Trees, lists, graphs, ...
- * Tasks **must** be subdivided recursively
- * Process the tree of tasks from the bottom
- * Map groups of tasks to processes/threads



Pipeline Decomposition

- * Sets of subsequent tasks (pipeline stages) applied to independent data
- * Data can be processed simultaneously by different pipeline stages
- * Assumes that the same stage cannot process multiple items simultaneously

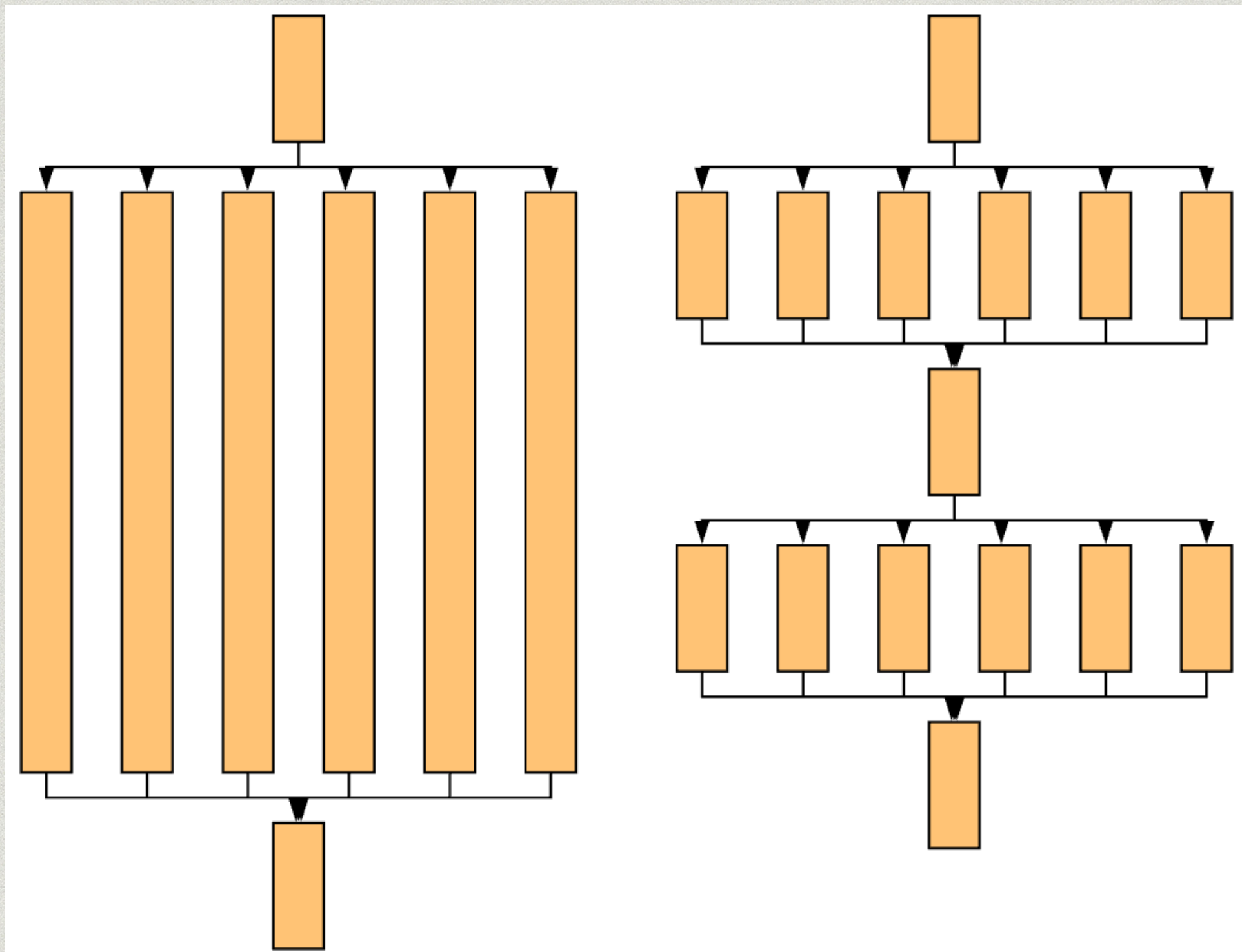


PARALLELIZATION IMPLEMENTATION

Program Structure

- * **Globally parallel, locally sequential** - multiple tasks performed simultaneously, each task sequential
 - * Single program, multiple data (SPMD)
 - * Multiple program, multiple data (MPMD)
 - * Master-worker
 - * Map-reduce
- * **Globally sequential, locally parallel** - sequential application with individual parallel sections
 - * Fork-join
 - * Loop parallelism

GPLS vs GSLP



GPLS

- * SPMD

- * Single executable
- * All devices compute the same code on different data

- * MPMD

- * Different executables for each device
- * Useful for heterogeneous nodes (x86 + ARM, PU + GPU)

- * Master-worker

- * Separates management and processing roles
- * Single master handles work to multiple workers using a strategy

- * Map-reduce

- * Maps independent data to different workers
- * Reduces (merges) the results of each worker

SPMD

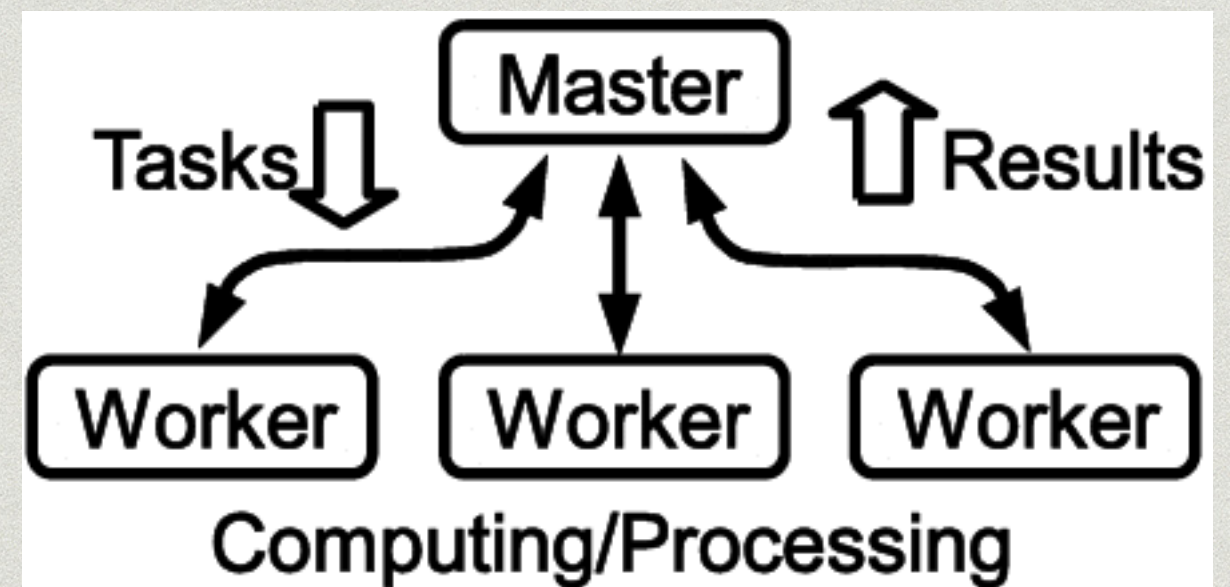
- * Single code base is easier to maintain
- * Typical structure
 - * Initialisation - thread pool and management
 - * Obtain unique thread identifier - scalar or vector id
 - * Run the code according to each id - workload distribution
 - * Shutdown - thread shutdown and result merge
- * Prone to algorithm logic errors due to data races

MPMD

- * Multiple code bases - harder to maintain
- * May require multiple different programming paradigms
- * Different device architectures adds another layer of irregularity
- * Should only be adopted due to apps RAM memory requirements or to use accelerators
- * Can be used in conjunction with SPMD strategies

Master-Worker

- * Clear definition of tasks for masters and workers
- * Masters
 - * Hand out work to workers - how to distribute it?
 - * Collect results from workers
 - * Interacts with files and with the users



Map-Reduce

- * A derivative of master-worker
 - * Popularised by the initial releases of the Google search engine
- * Multiple workers spawned to run the same code
 - * Can share intermediate results among them
 - * Results are merged by the master at the end
- * Master-worker usually implies a set of different tasks and/or different executables
 - * Map-reduce uses a single code base in a single executable

GSLP

- * Fork-join
 - * Creation of dynamic tasks (processes/threads) that must complete for the parent to continue
 - * Multiple instances of this behaviour per application
- * Loop parallelism
 - * Subset of fork-join
 - * Creates multiple processes/threads with the same code to process different blocks of independent data

Fork-Join

- * Used when the algorithm requires dynamic creation of tasks at runtime
 - * Parent task must wait for children tasks to finish to continue
 - * Tasks can be processed by pools of pre-generated processes/threads
- * Can lead to excessive amounts of idle times if not implemented properly

Loop Parallelism

- * The most simple and convenient method for most problems
 - * Loop iterations are divided into chunks
 - * Chunks are assigned to computing threads
 - * Static and dynamic strategies to assign chunks are often available in libraries - which should be used?
- * Is not efficient in distributed environments

Match Decomposition to Implementation

	Task Parallelism	Divide-Conquer	Geometric	Recursive	Pipeline
SPMD	✓	✓	✓	✓	✓
MPMD	✓	✓	✓	✓	✓
Master-Worker	✓	✓	✓	✓	
Map-Reduce		✓	✓	✓	
Fork-Join	✓	✓	✓	✓	✓
Parallel Loop		✓	✓		

LOAD BALANCING

References

- * An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, James Reinders, Intel
- * Intel® Xeon Phi™ Coprocessor High Performance Programming, Jim Jeffers, James Reinders, Elsevier Waltham (Mass.), 2013
- * Intel® 64 and IA-32 Architectures Software Developer's Manual