

Parallelization of C programs through dependency analysis

Jos van Eijndhoven
jos@vectorfabrics.com

June 20, 2012

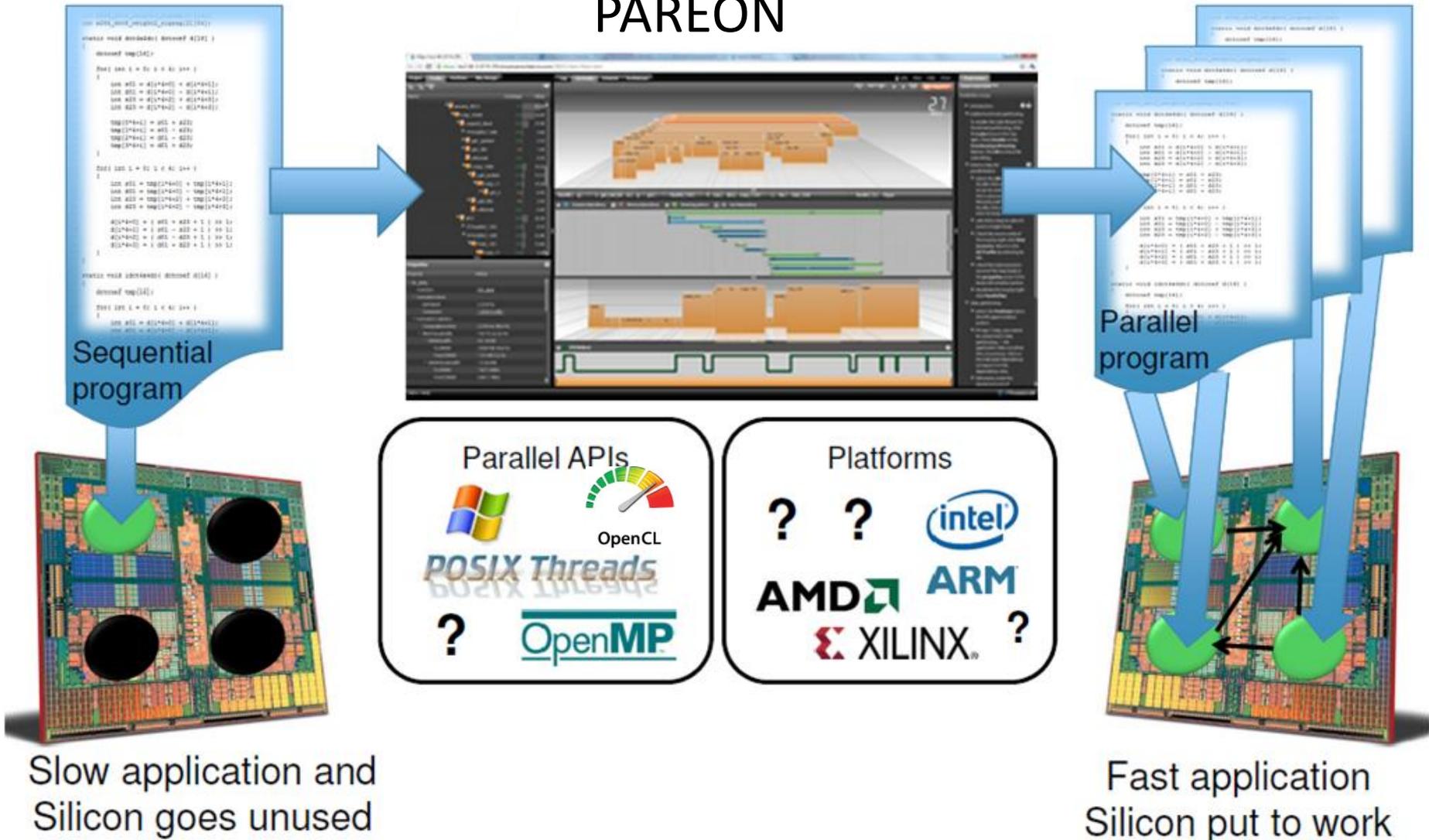
HPDC'12

Delft, The Netherlands



Discovering potential concurrency

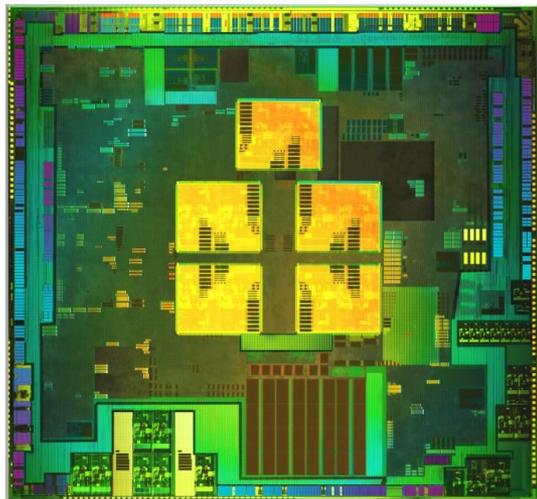
PAREON



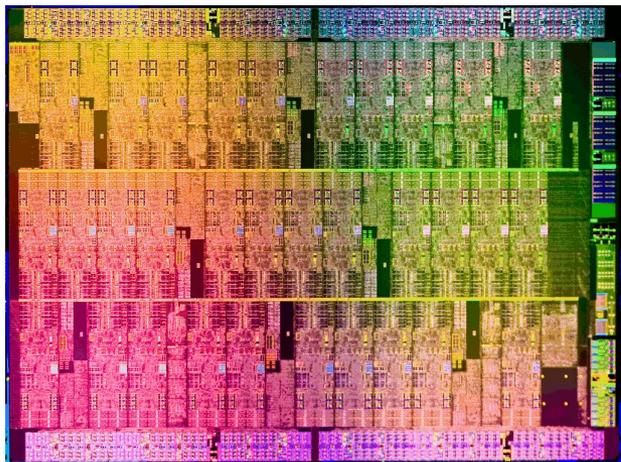
Presentation index

- Multi-processors will further expand
- Partitioning workload across multiple cores:
 - Data partitioning
 - Functional partitioning
- Tooling for dependency analysis
- Example: LAMMPS
- Conclusion

Programming parallel computers



nVidia TEGRA 3



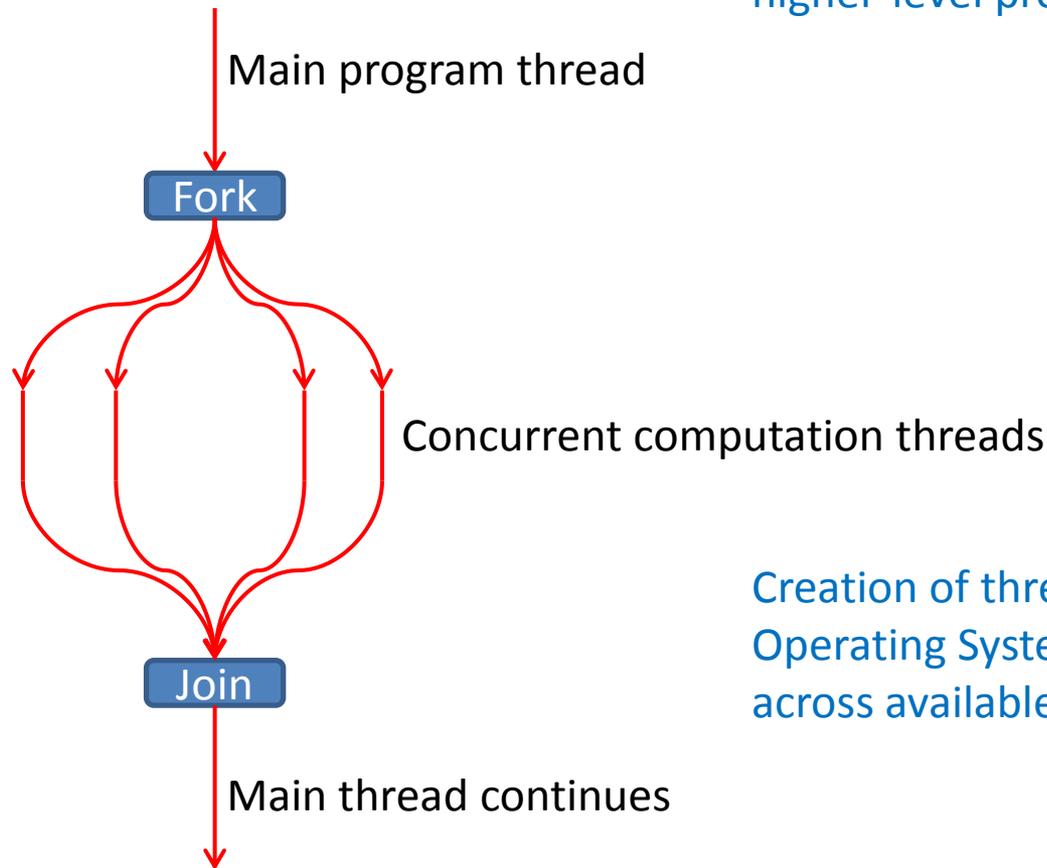
Intel Aubrey Isle (MIC)

- Multi-processor machines are all around us, ranging from mobile to super-computers.
- Multi-processor architectures are driven by technology factors: increased integration densities and power efficiency requirements.
- Parallel (multi-threaded) programs are required to exploit their capability.
- However, parallel programming is difficult and error-prone. Unfortunately, software productivity already is a major bottleneck.
- Distributed-memory architectures and heterogeneous processors (GP-GPUs) add further complications.

Application programmers need more help!

Creating multi-threaded concurrency

Basic fork-join pattern, created through different higher-level programming constructs



Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors.

Parallelization – two partitioning options

Source code:

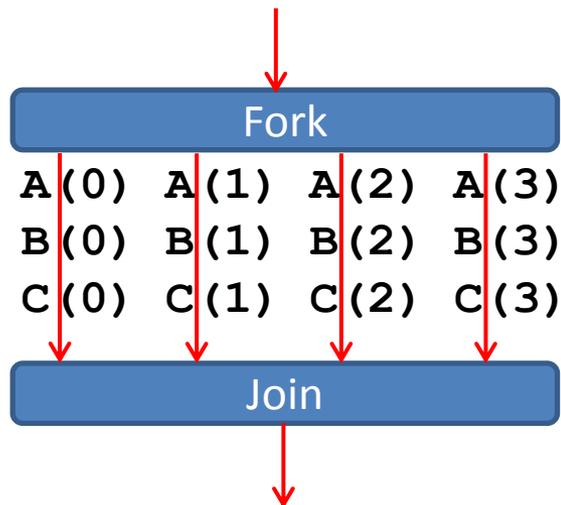
```
for (i=0; i<4; i++) {  
    A(i);  
    B(i);  
    C(i);  
}
```

Sequential execution order:

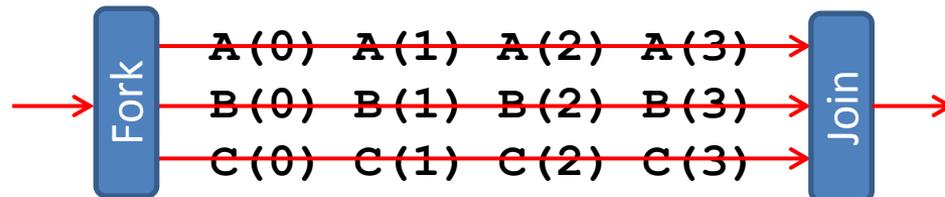
A(0)	A(1)	A(2)	A(3)
B(0)	B(1)	B(2)	B(3)
C(0)	C(1)	C(2)	C(3)

Vertical red arrows indicate sequential execution order: A(0) → B(0) → C(0), A(1) → B(1) → C(1), A(2) → B(2) → C(2), A(3) → B(3) → C(3).

Data partitioning:



Functional partitioning:



Functional versus data partitioning

Data partitioning:

- Allows a high degree of parallelization for loops with high iteration count.
- Allows good distribution of workload across (homogeneous) processors.
- Loop-carried data dependencies can severely impact performance.

Functional partitioning:

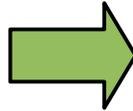
- Best for separation of workload across heterogeneous processors.
- Inter-function data dependencies typically converted into buffered streams.

Choice is directed by data dependency patterns.

Example functional partitioning

```
int A[N][M];
```

```
while (...)
{ produce_img();
  consume_img();
}
```



```
Thread1: while (...)
          produce_img();
```

```
Thread2: while (...)
          consume_img();
```

```
produce_img()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}
```

```
consume_img()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}
```

Synchronize thread progress:

- **True dependency:** consumer must wait for valid data
- **Anti dependency:** producer must wait with over-writing until after consumption

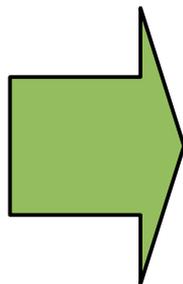
Function pipelining: synchronization

```
int A[N][M];
```

```
while (...)
{ produce_img();
  consume_img();
}
```

```
produce_img()
{ for (i ...)
  for (j ...)
    A[i][j] = ...
}
```

```
consume_img()
{ for (i ...)
  for (j ...)
    ... = A[i][j];
}
```



```
Channel ch;
```

```
Thread1: while (...)
  produce_img();
```

```
Thread2: while (...)
  consume_img();
```

```
produce_img()
{ for (i ...)
  for (j ...)
    write_int(ch, ...)
```

```
consume_img()
{ for (i ...)
  for (j ...)
    ... = read_int(ch);
```

Channel access functions
implement thread stall.

Pipeline dependency analysis

The screenshot displays a web application for pipeline dependency analysis. The browser address bar shows `https://jos.vectorfabrics.com/28000/html/Main.html`. The interface includes a top navigation bar with tabs for Project, Profile, Partitions, My changes, Log, 2D-Profile, Schedule, Architecture, and PGtrain.c. A left sidebar contains a 'Data partitioning candidates' section with a warning for 'Loop_36507' and a 'Functional partitioning - Loop_36507' table. The main area features a 3D visualization of a pipeline with colored blocks representing different stages. A bottom section shows a detailed dependency graph with nodes and edges, including a legend for Compute dependency, Memory dependency, Streaming pattern, and Anti-dependency. A right sidebar contains 'Cheat sheets' with an introduction and navigation instructions. A speech bubble points to the 3D visualization with the text: 'Potential pipelining showed in colors, with resulting Fifo's'. The status bar at the bottom indicates 'Status: ready' and 'vThreaded-x86'.

Data partitioning candidates

- Loop_36507 cannot be subjected to data partitioning: There are 1 loop-carried memory clusters that you have chosen not to ignore: (memory cluster 25).

Functional partitioning - Loop_36507

Partition id	Threads	Speedup	Streams	Apply
Partition 1	4	2.6	3	Apply
Partition 2	4	2.6	3	Apply
Partition 3	3	2.4	2	Apply
Partition 4	3	2.2	2	Apply
Partition 5	2	1.2	0	Apply

Properties

Function: SELECT_3_p_full_cal

Line coverage: 30.8%

Uncovered lines

Invocation time

- Estimated: 38.188 ns
- Constraint: <click to edit>

Invocation statistics

- Computation time: 28.812 ns (75.5%)
- Memory penalty: 9.375 ns (24.5%)
 - DRAM traffic: 0 B
 - DRAM bandwidth: 0 B
- Data cache statistics
 - Penalties
 - Level 1: 9.375 ns
 - Level 2: 0 ps

Status: ready

Potential pipelining showed in colors, with resulting Fifo's

Function pipelining: Channel APIs

Too many choices for channel-based communication:

- Standard Java util.concurrent queue classes
- Intel's TBB (C++) queues
- Linux 'pipes' and 'sockets'
- OpenCL channels
- OpenMAX IL for streaming media processing
- MPI message-passing channels
- . . .

Very different queue implementations:

- Inter-thread, inside process memory context
- Inter-process, inside shared-memory system
- Inter-system, through device interfaces

NOTE: C++ STL queues are **NOT** thread-safe!

Example data partitioning

```
int sum = 0;
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Distribute the workload over multiple cores.
- Each core handles part of the loop index space.

```
int sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Workload scales nicely across multiple cores
- Easy to write down 😊, but hard to grasp all consequences!
- Highly dangerous, might cause extremely hard-to-track bugs! 😞

Application Analysis



Finding data dependencies

Vector Fabrics' approach:

- Compile program source code, compiler is adapted for instrumentation.
- Execute the instrumented program:
 - Traps all memory load/store operations:
match ld/st operations that address the same memory location
 - Relates ld/st operations with nested loop structure:
separate loop-carried dependencies from loop in-bound and loop out-bound dependencies
 - Builds an execution profile (call tree), across file boundaries
- Analyze loops with their data dependencies for parallelization patterns

Recognize parallelization patterns

Analyze loops with data dependencies for parallelization patterns:

- Reduction expressions
- Induction expressions
- Streaming dependencies, allowing data duplication and localization

Avoid considering 'false' memory dependencies:

- Local variables on stack, duplicated through thread local storage
- Re-use of memory locations through `malloc()` and `free()`.

Relate data dependencies and patterns to locations in C(++) source code for required code transformations.

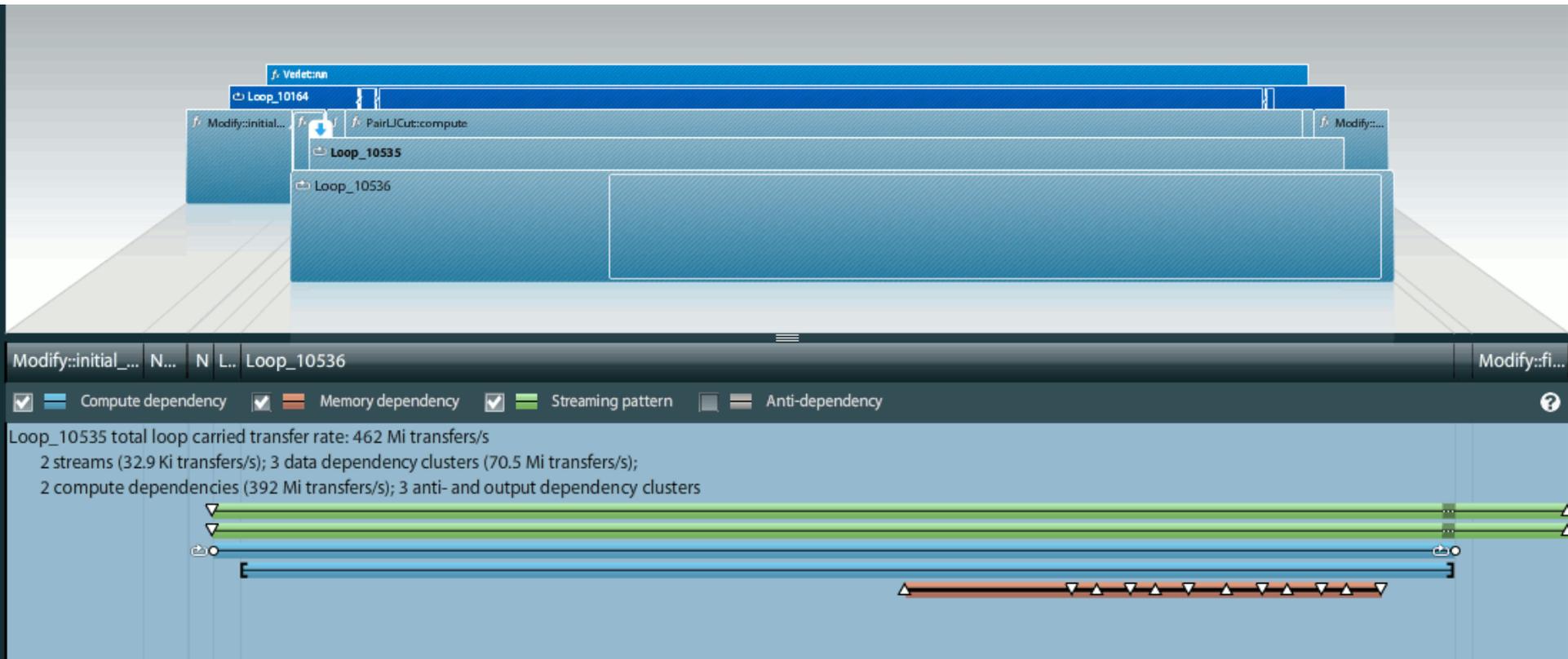
Example: LAMMPS molecular simulator

- Source code configured for sequential version.
- About 187Klines of C++ source code in 636 files.

Name	Coverage	Delay
entry	0	100.00
main	100	100.00
Input::file	80	100.00
Loop_3344	78	100.00
Input::execute_command	99	100.00
Run::command	32	99.78
Verlet::setup	79	0.50
Verlet::run	86	99.26
Loop_10164	84	99.26
Modify::initial_integrate	100	8.72
Neighbor::decide	67	2.78
Neighbor::build	40	1.56
PairLJCut::compute	100	78.95
Loop_10535	100	78.91
Loop_10536	100	77.07
Modify::post_force	100	0.99
Modify::final_integrate	100	6.15

Example: LAMMPS data dependencies

- Parallelization opportunity detected in loop over particles, inside loop over time steps
- Different dependency patterns shown in different colors



Parallel performance prediction

- Estimate overhead from thread fork/join and synchronization
- Estimate execution schedule with loop-carried dependencies

Speedup of this loop: 3.9x, overall speedup: 2.4x

Partitions Profile

Partitioning candidates - Loop_10535

CPU data partitioning - vFtasks

Number of threads: 4

Global speedup: 2.4 Extra worker threads: 3

Global overhead: 1% Thread creation delay: 33 us

Invocation	Speedup	Overhead	Streams
Loop_9720			
Loop_10535	3.9	2%	4

2D-Profile Architecture Help Schedule

Data partitioning - Loop_10535

Schedule overview

Execution - 98%

Schedule execution (prologue - steady state - epilogue)

Iteration #0, #4, #8, #12

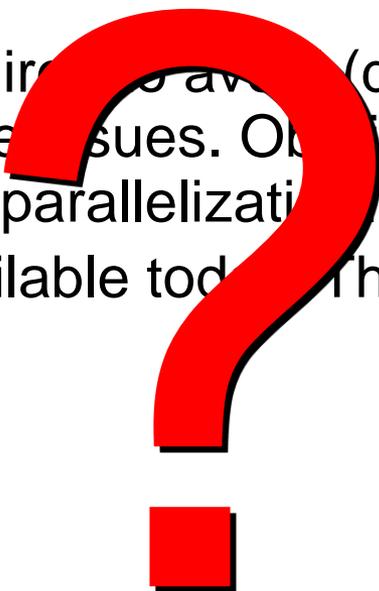
Iteration #1, #2, #3, #5, #6, #7, #9, #10, #11, #13

Conclusion

- **Today's gap:** multi-processor machines are everywhere, yet multi-threaded programming is difficult and error-prone.
- Proper tooling is required to avoid (data race-) errors and obtain insight in performance issues. Obtain such insight **before** spending time on re-coding for parallelization.
- Various tools are available today. They do support real-world application analysis.

Questions?

- **Today's gap:** multi-core CPUs and multi-processor machines are everywhere, yet multi-threaded programming is difficult and error-prone.
- Proper tooling is required to avoid (data race-) errors and obtain insight in performance issues. Obtain such insight **before** spending time on re-coding for parallelization.
- Various tools are available today. They do support real-world application analysis.



Thank you

Check www.vectorfabrics.com for a free trial of concurrency analysis

