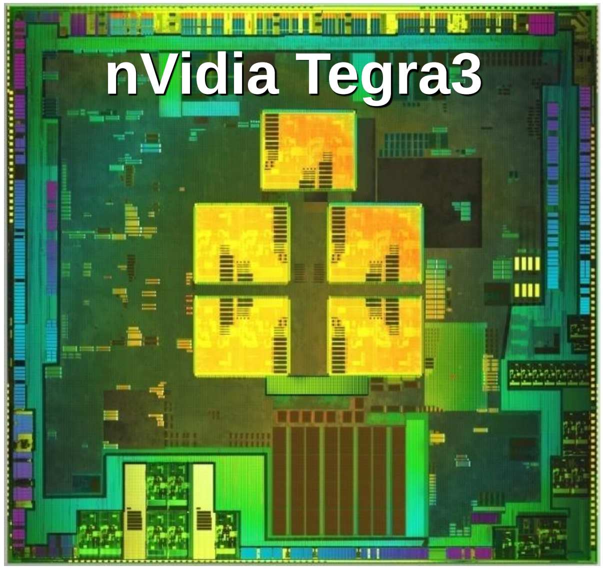
Application analysis for parallelization on multi-core devices

Jos van Eijndhoven
jos@vectorfabrics.com

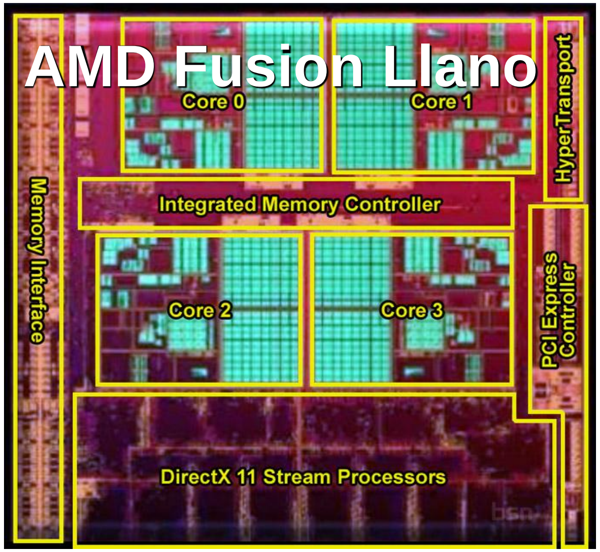
HiPEAC Computing Systems, Ghent, Belgium
Oct 16, 2012

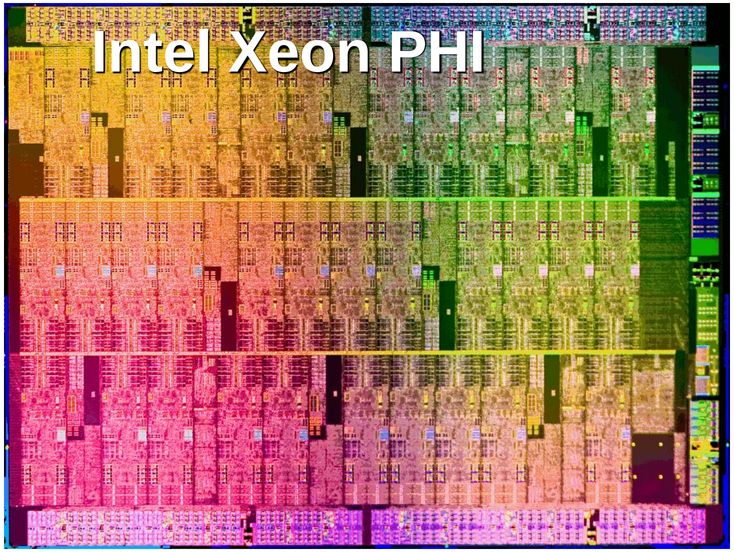
Vector Fabrics

Multi-core processors are here to stay



- To make use of growing transistor count
- To allow run-time trade-offs between performance and power





Multi-core in Mobile

- 2 cores:
  Assume the OS provides multiple processes and/or kernel threads for workload
- 4 cores (and beyond):
  Requires multi-threaded applications
  - To obtain sufficient concurrent workload
  - To obtain top user experience

Who makes such applications??

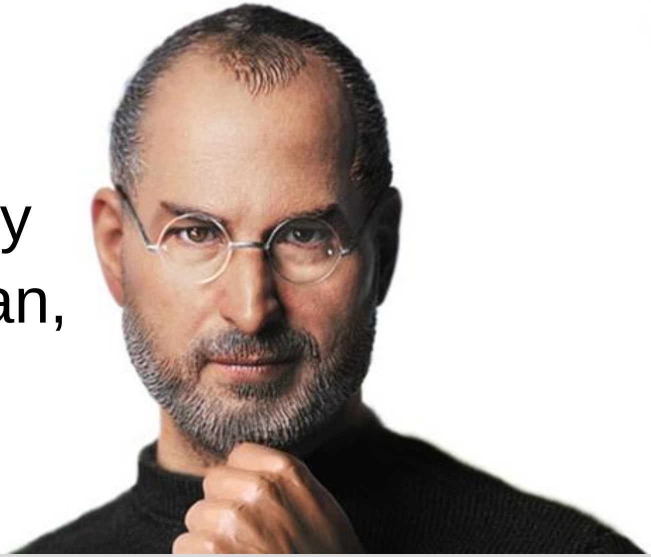
Creating parallel programs is hard…

Herb Sutter, chair of the ISO C++ standards committee, Microsoft:

“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren’t possible, and discovers that they didn’t actually understand it yet after all”

Steve Jobs, Apple:

“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two yeah; four not really; eight, forget it.”

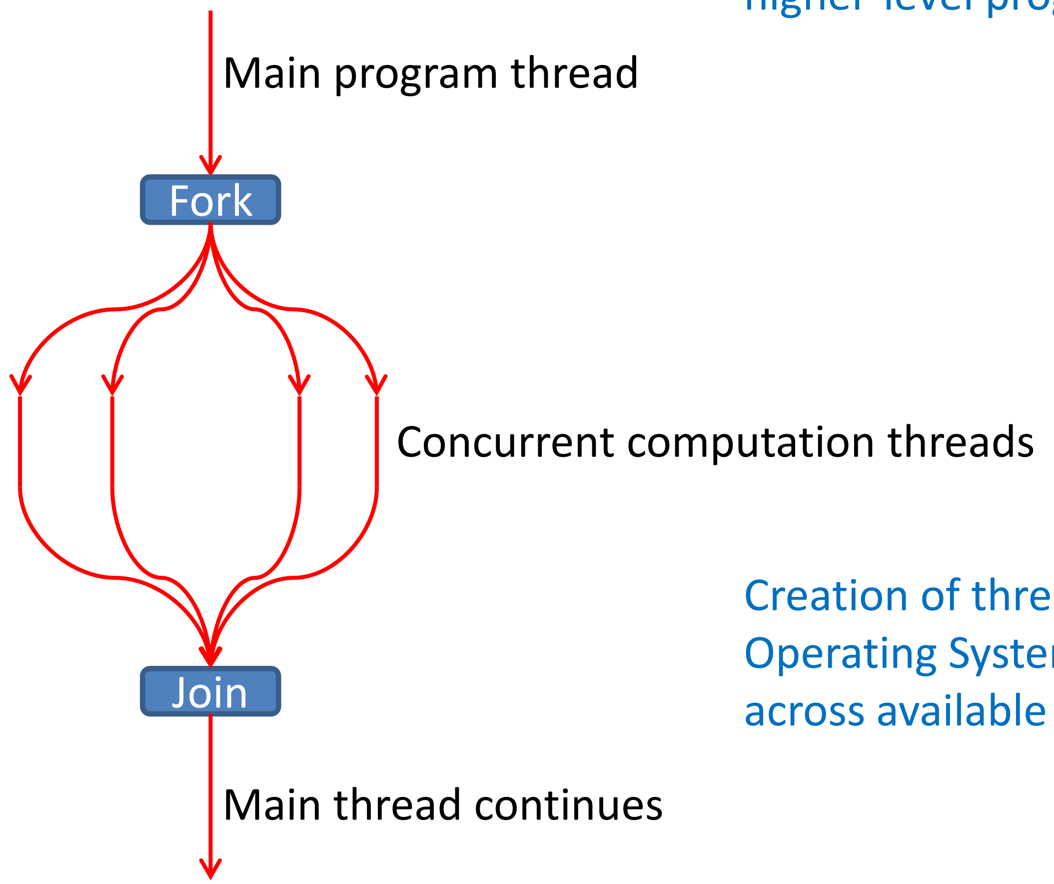


Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
  - Data-parallelization with reduction expressions
  - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Creating multi-threaded concurrency

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



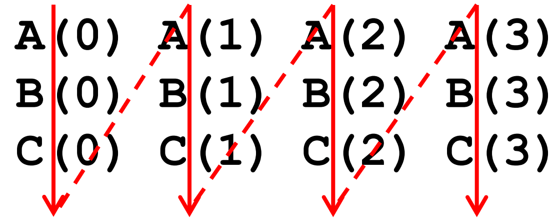
Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors!

Parallelization – two partitioning options

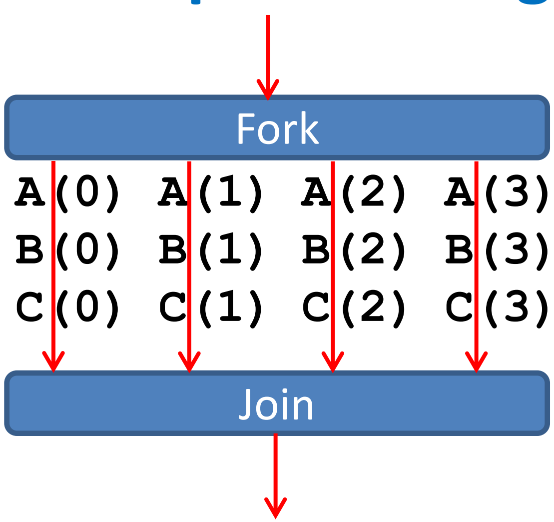
Source code:

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

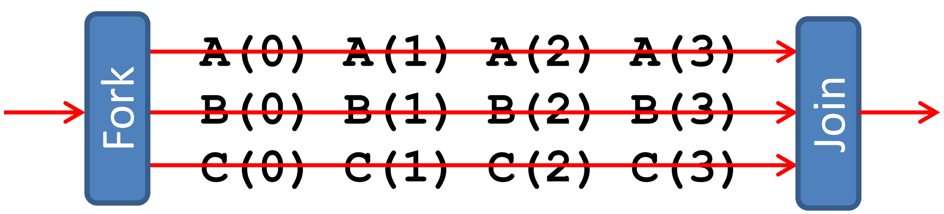
Sequential execution order:



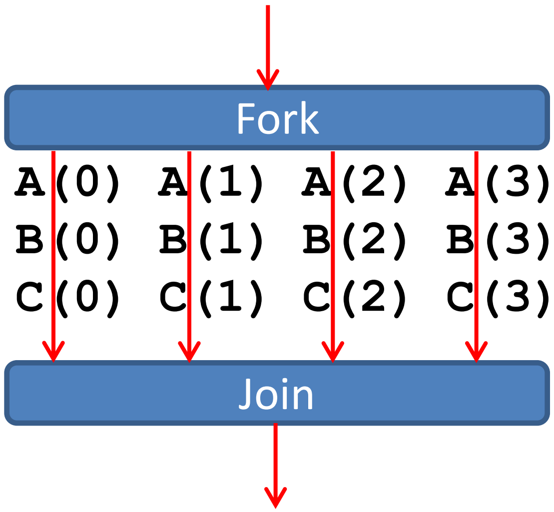
Data partitioning:



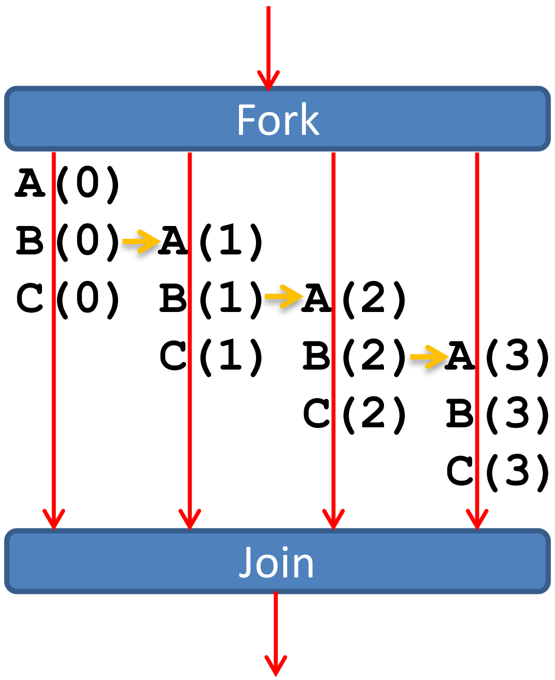
Task partitioning:



Issue: Data dependencies



Maybe, **B(i)** produces a value that is used by **A(i+1)**…



Adjust program source for parallelization:

- When feasible, remove inter-thread data dependencies
- Implement required data synchronization

Consciously choose task versus data partitioning, check dependency analysis!

Category 1: Data dependencies

Variable assigned in loop body, used in later iteration

```
// search linked-list for matching items
// save matches in 'found' array of pointers
for (p = head, n_found = 0; p; p = p->next)
  if (match_criterion(p))
    found[n_found++] = p;
```

Cannot (easily/trivially) spawn data-parrallel tasks!

- No direct parallel access to list members `*p`
- No direct way to assign index to matched item `n_found`
- Maybe more problems hidden in `match_criterion`

Category 2: Anti dependencies

Storage location used in loop body, shared over iterations

```
// convert table with floats to strings
char word[64];
for (i=0; i<N; i++)
{
  sprintf( word, "%g", table_float[i]);
  table_string[i] = strdup( word);
}
```

- Anti-dependencies are resolved by duplicating storage locations (thread-local storage)
- Need to make multiple copies of `word[]` space

Category 3: Control dependencies

Control flow can give order constraints that hinders parallelization:

```
// No creation of work beyond some point
for (i=0; i<N; i++)
{
  if (special_condition(i))
    break;
  table[i] = workload(i);
}
```

Since multiple threads proceed at non-determined mutual speed, above test risks violation in a data-parallel loop.

Note: C++ exceptions certainly belong to this category

Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
    - Data-parallelization with reduction expressions
    - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Can do: reduction data dependencies

- Reduction expressions: accumulate results of loop bodies with commutative operations
- Freedom of re-ordering allows to break sequential constraints

```
// conditionally accumulate results
int acc = 0;
for (i=0; i<N; i++)
{
  int result = some_work(i);
  if (some condition(i))
     acc += result;
}
...use of acc ...
```

- Commutative operations are basic math like +, *, &&, &, ||, but also more complex operations like ‘*add to set*’.
- Three(?) different methods to handle these ...

Three methods for reduction dependencies

- Create thread-local copies of the accumulator. Accumulate over local copy in each thread. Merge the partial accumulators after thread-join.
  Eg. created automatically by:
  ```
  #pragma omp parallel for reduction(...)
  ```
- Maintain single accumulator, synchronize updates through atomic operations. Eg. in C++11:
  ```
  atomic_add_fetch( &acc, result);
  ```
- Maintain single accumulator, synchronize updates through protection by acquiring and releasing semaphores.
  Eg. Used by C++ Intel TBB:
  ```
  concurrent_unordered_set<..> s;
  s.insert(...);
  ```

Example data partitioning

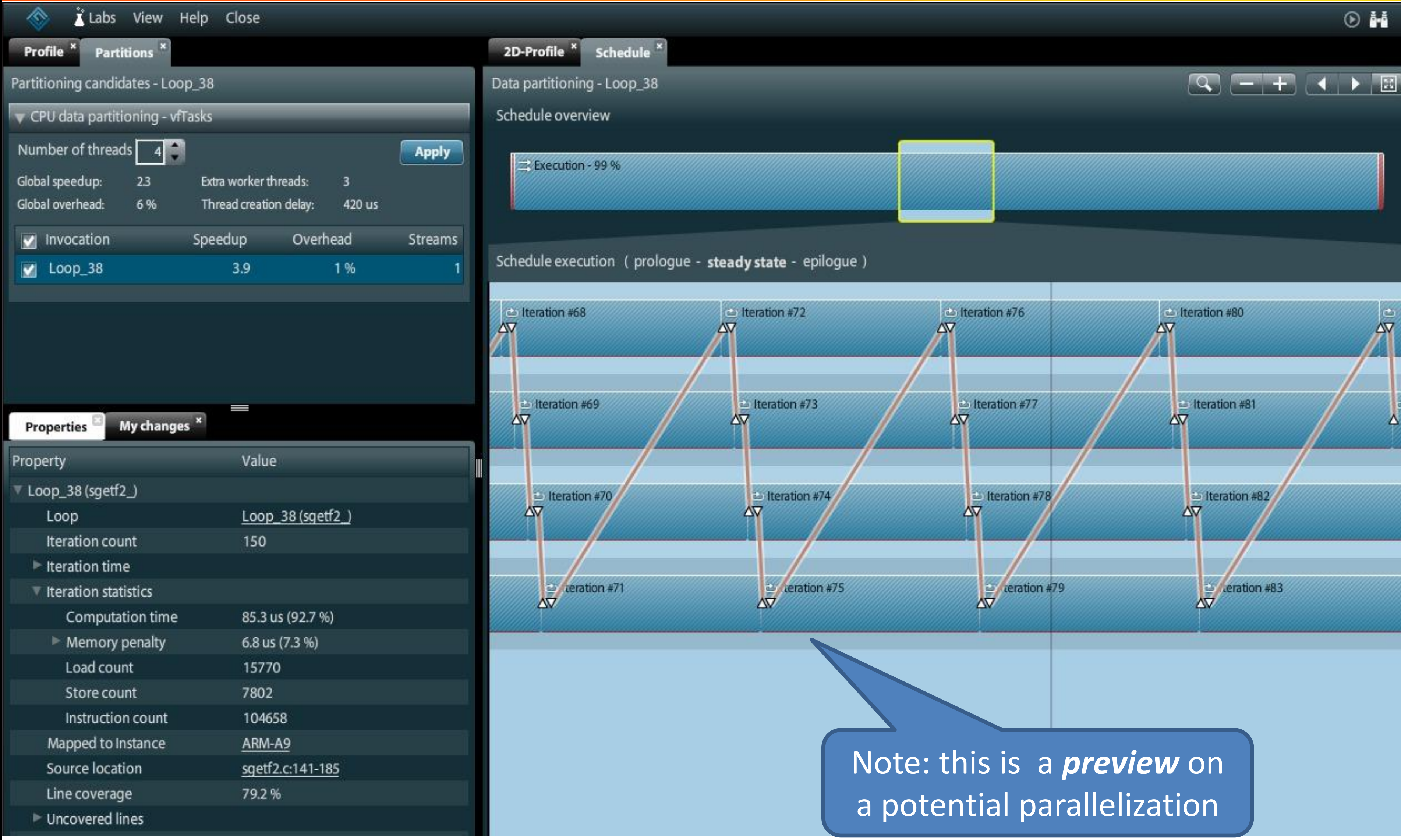
```
int sum = 0;
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- 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;
}
```

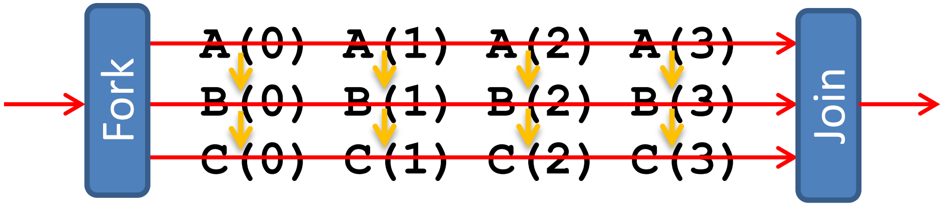
- Workload scales nicely across multiple cores
- Easy to write down ☺, but hard to grasp all consequences!
- Dangerous, might cause extremely hard-to-track bugs! ☹

PAREON: Parallelization Analysis

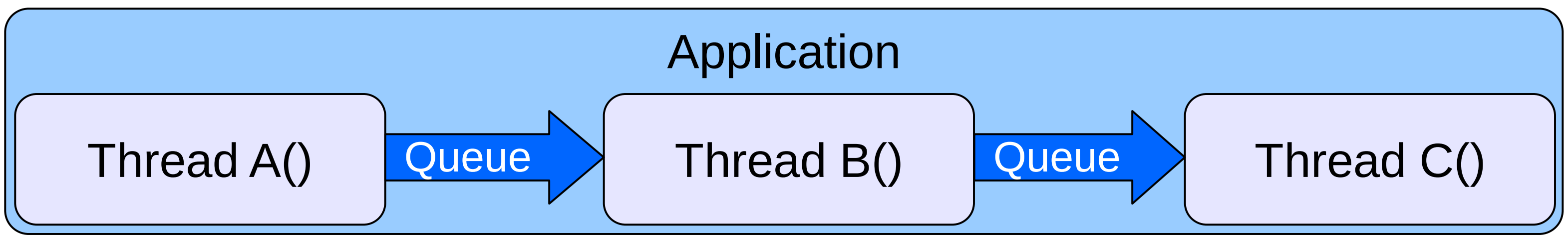


Pipelining: Data deps & functional partitioning

Functional partitioning with inter-thread dependencies:



Producer-Consumer pattern:



Queue implementation solves dependencies:

- **Synchronize Data dependencies**: Consumer thread waits for available data (stalls until queue is non-empty)
- **Solve Anti dependencies**: Producer thread creates next item in next memory location (prevents overwriting previous value)

Example functional partitioning

```
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];
}
```

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

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

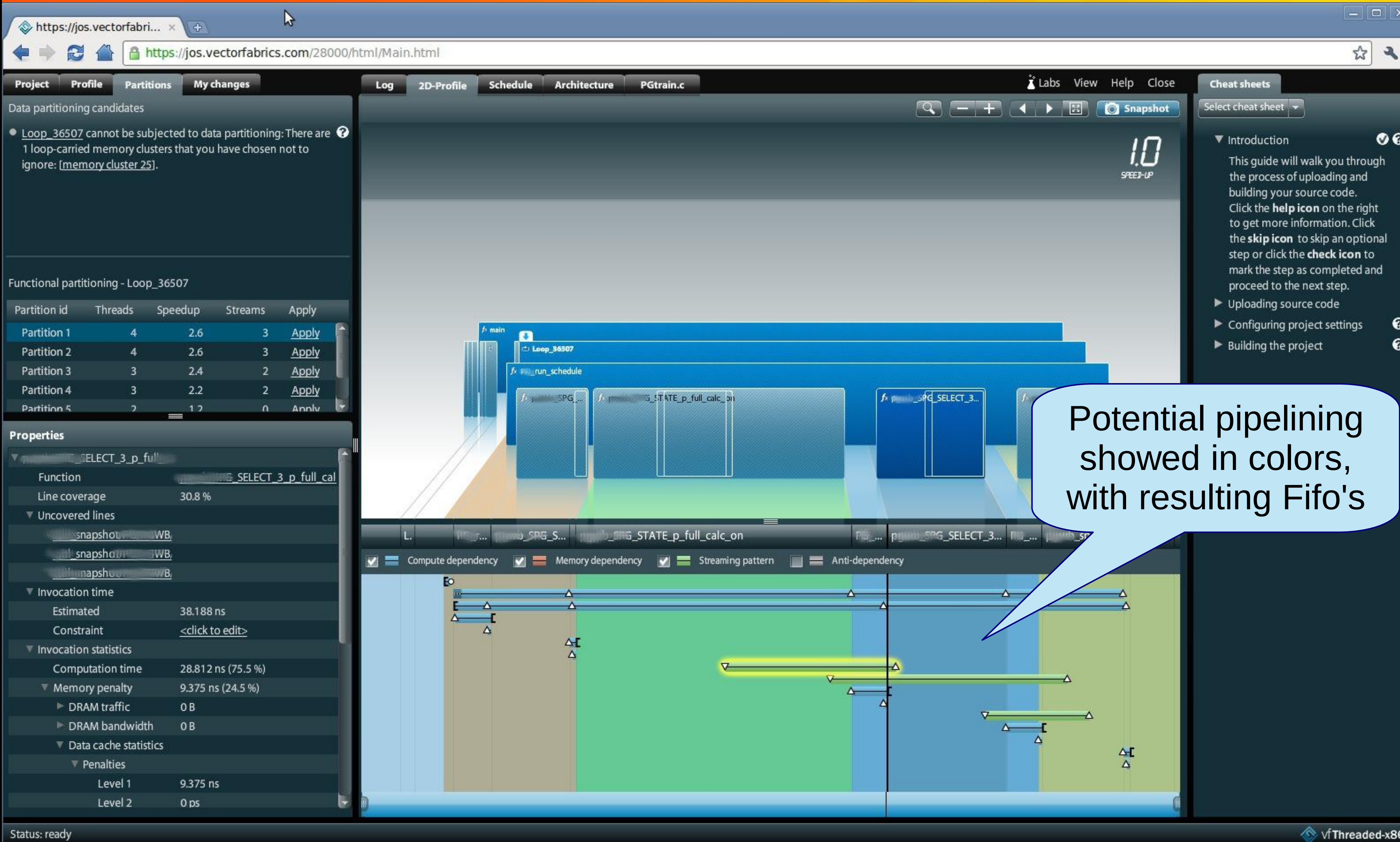
Function pipelining: synchronization

```
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];
}
```

```
Thread1: …

Thread2:…

concurrent_queue<int> qA;

produce_img()
{ for (i ...)
    for (j ...)
      qA.push(...)
}

consume_img()
{ for (i ...)
    for (j ...)
      qA.pop(&...);
}
```

Conversion to queues becomes more difficult when data items are not always assigned and referenced exactly once in order!

PAREON: Pipeline dependency analysis



Presentation index

- Introduction
- Dependencies that hinder multi-threading
- Parallelization with dependencies:
  - Data-parallelization with reduction expressions
  - Task-parallelization with streaming dependencies
- Tooling for parallelization of sequential C code
- Conclusion

Concurrent C/C++ programming: Pitfalls

Risc introduction of functional errors:

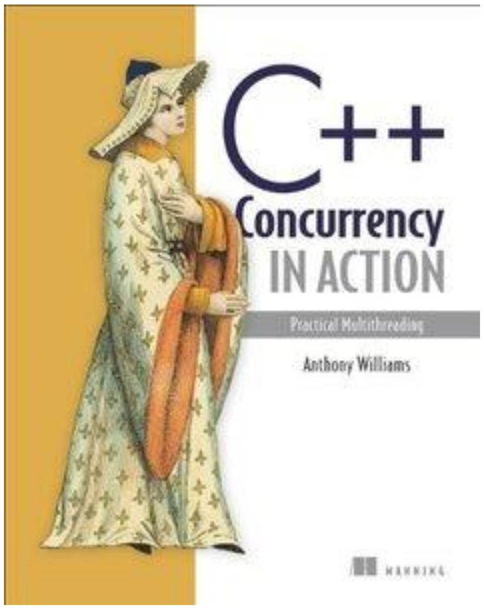
- Overlooking use of shared/global variables
  (deep down inside called functions, or inside 3rd party library)
- Overlooking exceptions that are raised and catched outside studied scope
- Incorrect use of semaphores: flawed protection, deadlocks

Unexpected performance issues:

- Underestimation of time spent in added multi-threading or synchronization code and libraries
- Underestimation of other penalties in OS and HW
  (inter-core cache penalties, context switches, clock-frequency reductions)

Parallel programming remains hard!

Concurrent programming remains hard



- C++11 standardizes valuable primitives
- Provides good insight in C++ concurrency
- Warns for many subtle problems
- From a research point-of-view, shows that C++ is not a nice language to design concurrency.

Development of parallel code

Guidelines:

- Base upon a sequential program:
  functional and performance reference
- Apply higher-level parallelization patterns and primitives:
  clear semantics, re-use code, reduce risk
- Use tooling for analysis and verification
  - Prevent introduction of hard-to-find bugs
  - Prevent recoding effort that does not perform

Managable development process!

PAREON step 1: Code instrumentation

Build application with compiler that inserts instrumentation:

- Creates instrumentation for run-time tracing of application activity (function entry/exit, loop entry/exit, ld/st addresses)
- To support run-time data-dependency analysis
- Also support code coverage analysis

PAREON 2: run-time dependency analysis

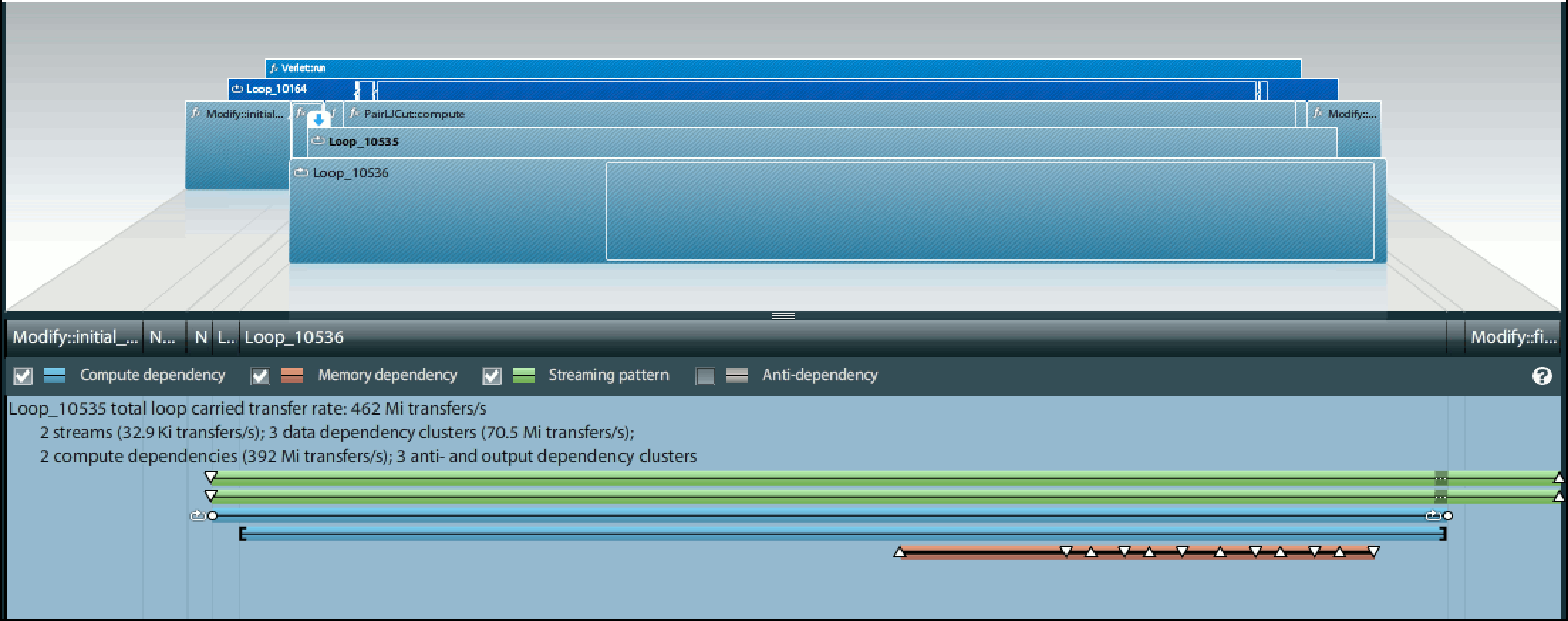
Execute instrumented program with test input data:

- Trace analysis detects dependencies between loads & stores at different program locations to same memory address.
- Differentiate loop-inbound, loop-carried and loop-outbound dependencies
- Relate with stack grow/shrink and heap malloc/free to break non-functional address re-use.
- Handle all scalar register-mapped data dependencies by static code analysis.

PAREON 3: find concurrency opportunities

GUI to browse loops with high workload and parallelization opportunities:

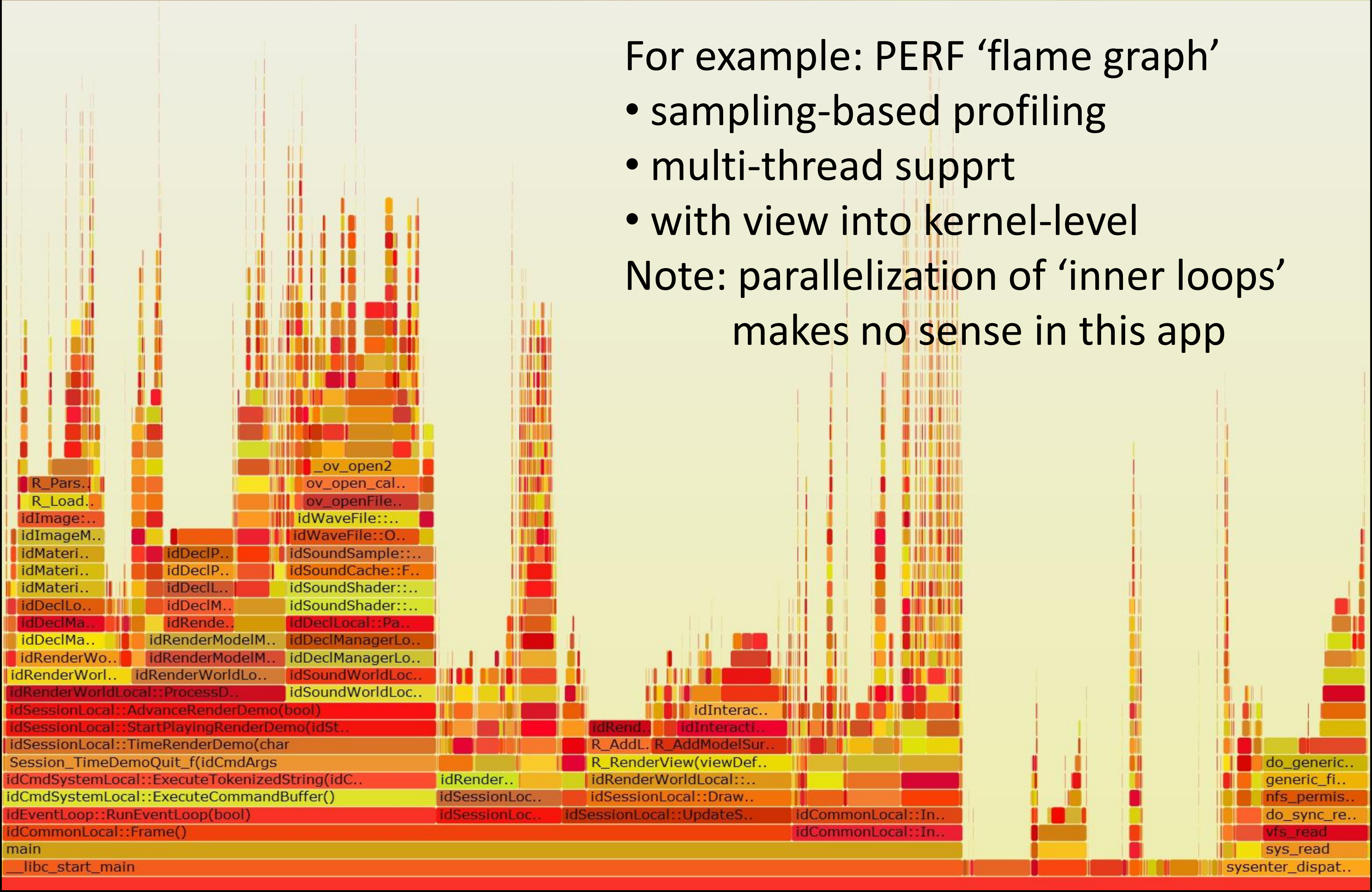
- Provide workload estimate and reachable speed-up
- Match detected dependencies with higher-level parallelization patterns for resolving (...)
- Prevent loop parallelization with unresolved dependencies



Performance Verification

For example: PERF ‘flame graph’
- sampling-based profiling
- multi-thread supprt
- with view into kernel-level

Note: parallelization of ‘inner loops’ makes no sense in this app



Conclusion

- Todays gap: multi-core CPUs are everywhere, yet multi-threaded programming remains hard (in C/C++):
  - Risc of creating hard-to-locate bugs regarding dynamic data races and semaphore issues
  - Obtained speedup is lower then expected
- A sequential functional reference implementation helps to set a baseline
- Proper tooling is needed to save on edit-verify development cycles

Conclusion

- Todays gap: multi-core CPUs are everywhere, yet multi-threaded programming remains h[illegible]C/C++):
  - Risc of creating h[illegible]-to-loca[illegible]bugs regarding dynamic data races and semaphore issues
  - Obtained speedup is lowe[illegible]en expected
- A sequential functional referen[illegible]mplementation helps to set a baseline
- Proper tooling is needed to[illegible]ave on edit-verify development cycles

Thank you

Check www.vectorfabrics.com for a free demo on concurrency analysis

Vector Fabrics