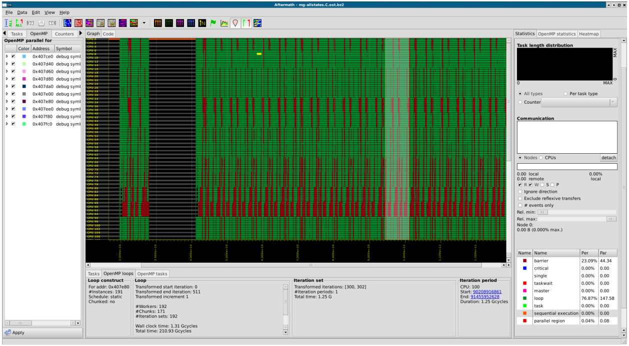
Cross-Layer Performance Analysis of Parallel Programs with Aftermath

Andi Drebes
Antoniu Pop

The University of Manchester
School of Computer Science
Advanced Processor Technologies
andi.drebes@manchester.ac.uk

Tutorial at HiPEAC 2017

Objectives of this Tutorial



- Present *Aftermath*: A tool for performance analysis of parallel programs
- Actually use Aftermath for performance analysis:
  - Loop-parallel OpenMP programs
  - Task-parallel OpenStream programs
- Get feedback from you

Outline

Morning: Intro + Analysis of OpenMP Programs [10:00 – 13:00]

- Overview of Aftermath
- Examples of OpenMP analyses
- Generating traces with Aftermath-OpenMP
- Hands-on session

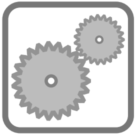
Lunch [13:00 – 14:00]

Afternoon: Task-Parallel Programs / NUMA [14:00 – 17:30]

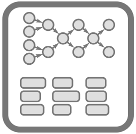
- Introduction to Data-Flow Tasks in OpenStream
- Visualization / analysis of task-based programs
- Performance analysis on NUMA platforms
- Hands-on session

Analysis of parallel Programs

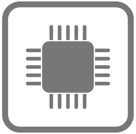
Run-time



OS

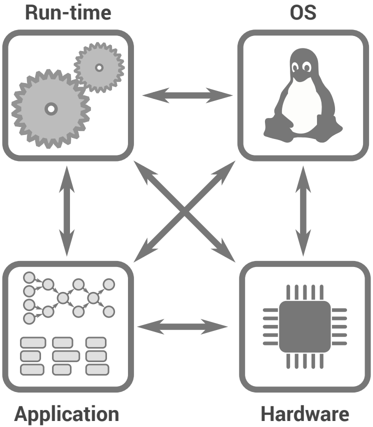


Application

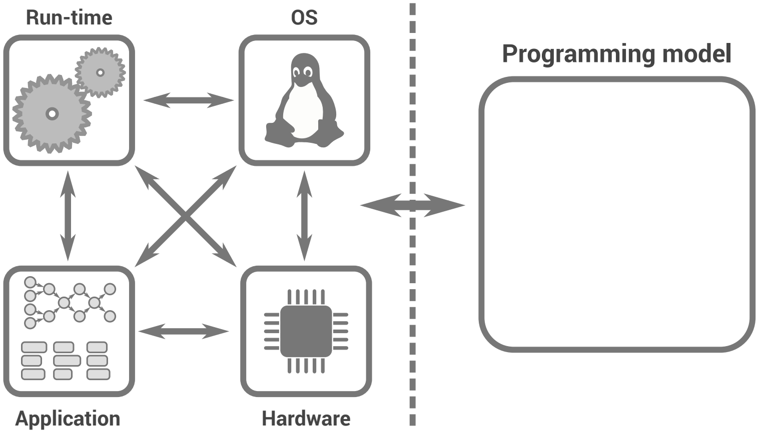


Hardware

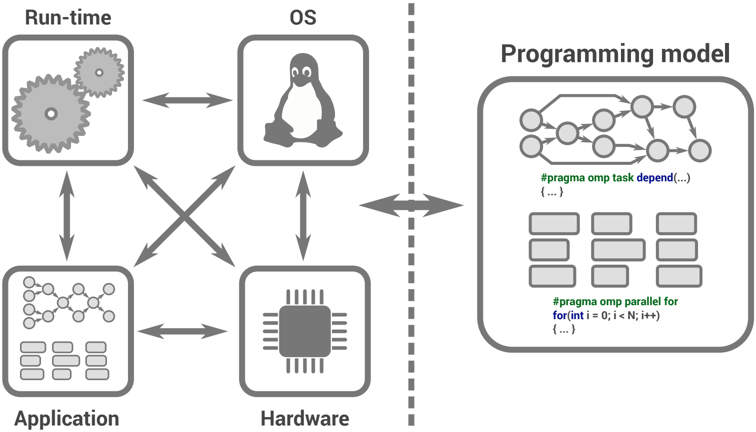
Analysis of parallel Programs



Analysis of parallel Programs



Analysis of parallel Programs



New Tools for Performance Analysis

Frequent topics in performance analysis:

- Amount of parallelism / load balacing
- Duration of execution phases
- Synchronization overhead (e.g., barriers)
- OpenMP loop schedules
- Data distribution on NUMA systems
- Relate hardware events to loops / tasks

New Tools for Performance Analysis

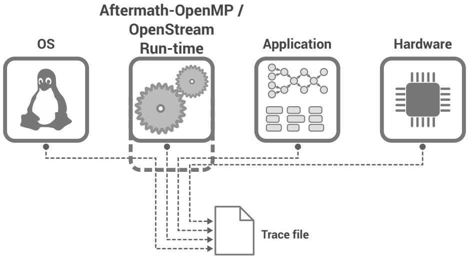
Frequent topics in performance analysis:

- Amount of parallelism / load balacing
- Duration of execution phases
- Synchronization overhead (e.g., barriers)
- OpenMP loop schedules
- Data distribution on NUMA systems
- Relate hardware events to loops / tasks

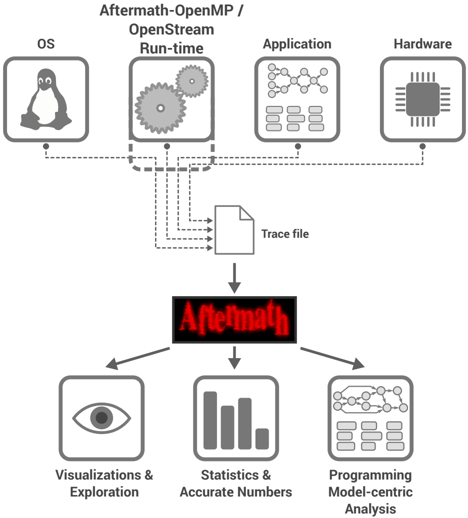
Our tools

- Aftermath: Graphical tool for performance analysis
- Aftermath-OpenMP. Instrumented LLVM/clang run-time
- OpenStream: Programming framework for data-flow tasks
- Libaftermath-trace: Low-level library for instrumenting run-time systems

Trace-based Analysis with Aftermath



Trace-based Analysis with Aftermath



Some History

2013–2016: OpenStream only

- Originally developed for OpenStream
- Focus on dependent tasks & NUMA
  - Analysis of data accesses by tasks
  - Analysis of data placement strategies by the run-time system

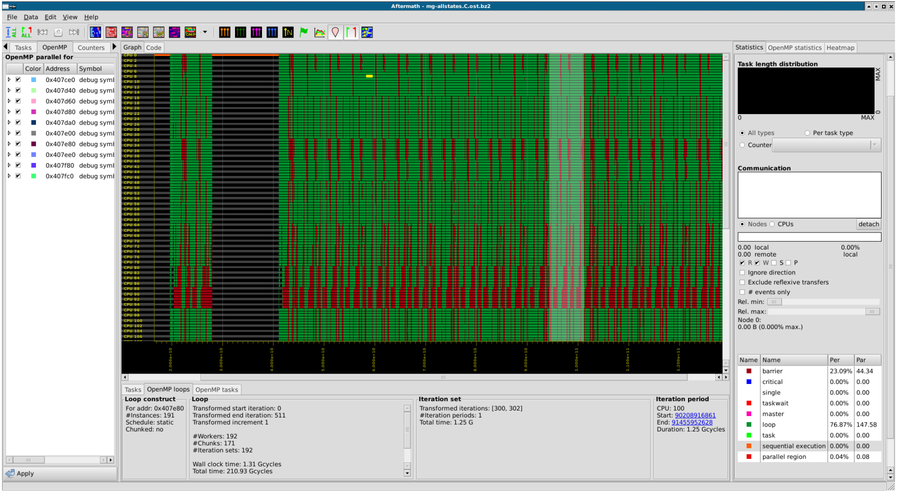
Since 2016: Loop-based OpenMP programs

- Specific views for OpenMP
- Instrumented version of the LLVM/Clang OpenMP run-time
- Library for the generation of trace files

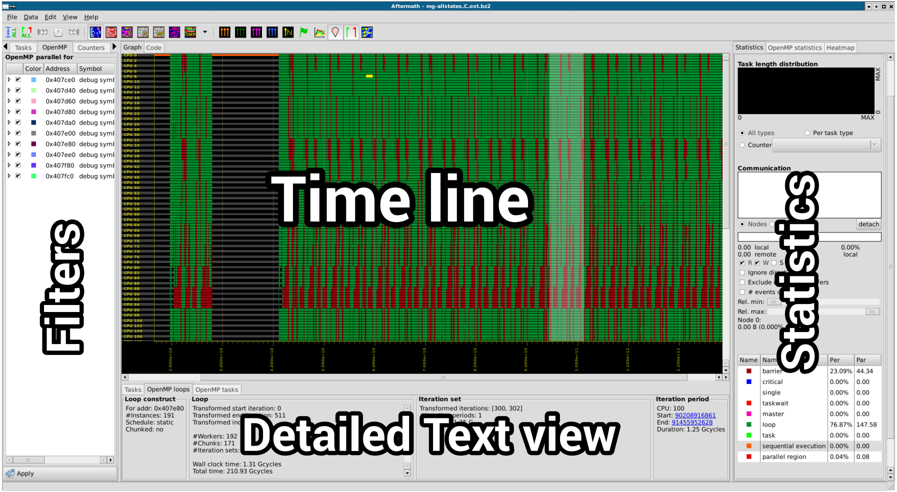
Near future: Dependent tasks in OpenMP

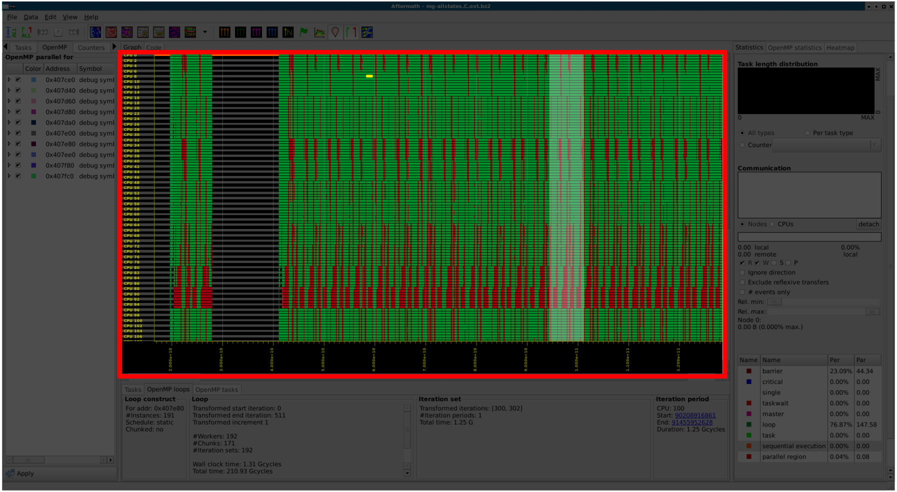
- Large overlap of analyses in OpenMP and OpenStream
- Instrumentation based on OMPT

Aftermath: Overview of the GUI

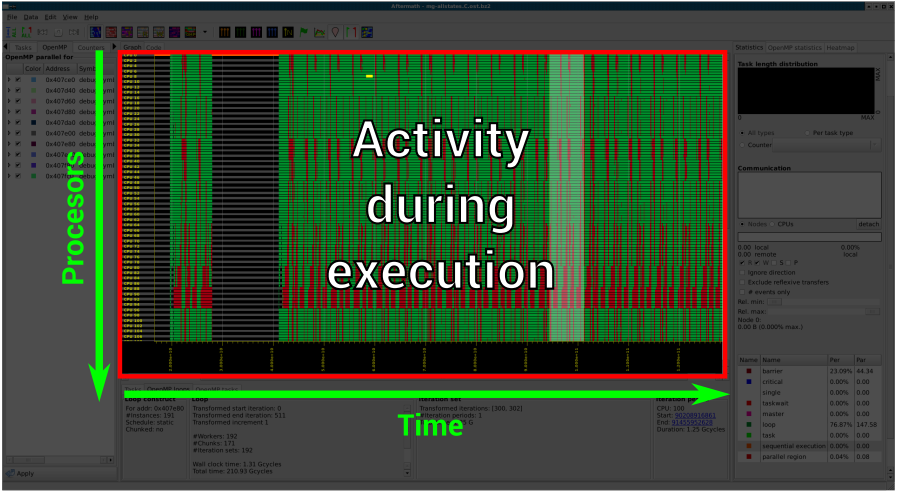


Aftermath: Overview of the GUI





Time line

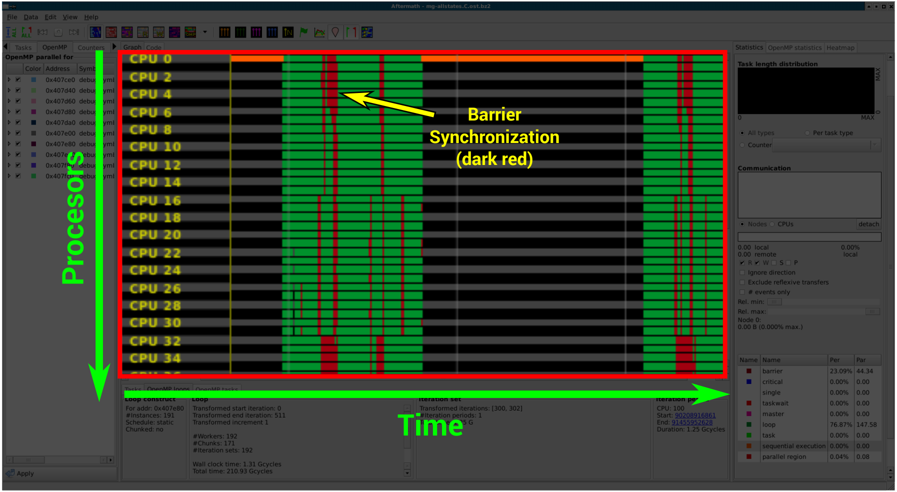


Time line

Aftermath: Overview of the GUI

Time line: Run-time states

Time line: Run-time states

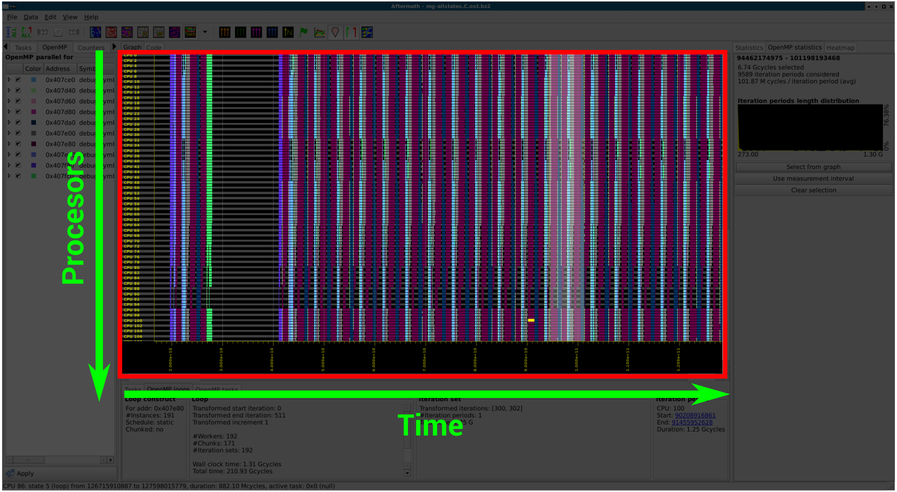


Time line: Run-time states

Aftermath: Overview of the GUI

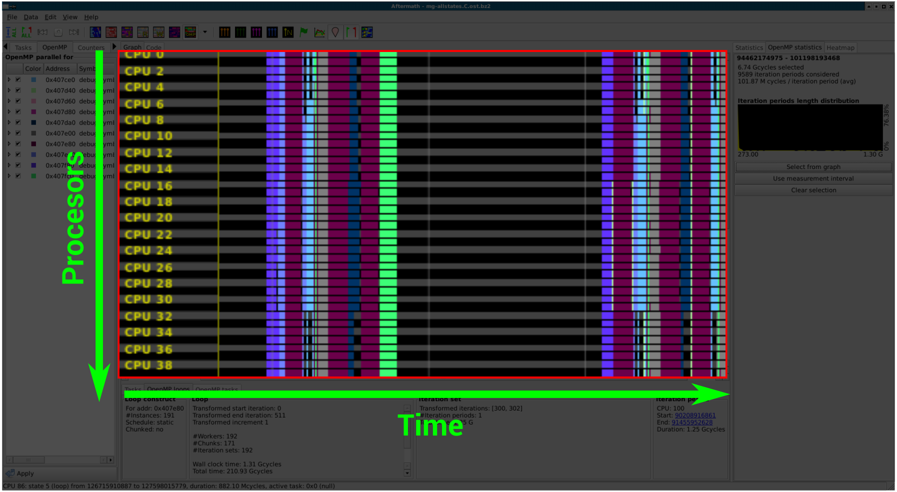
Time line: Run-time states

Aftermath: Overview of the GUI

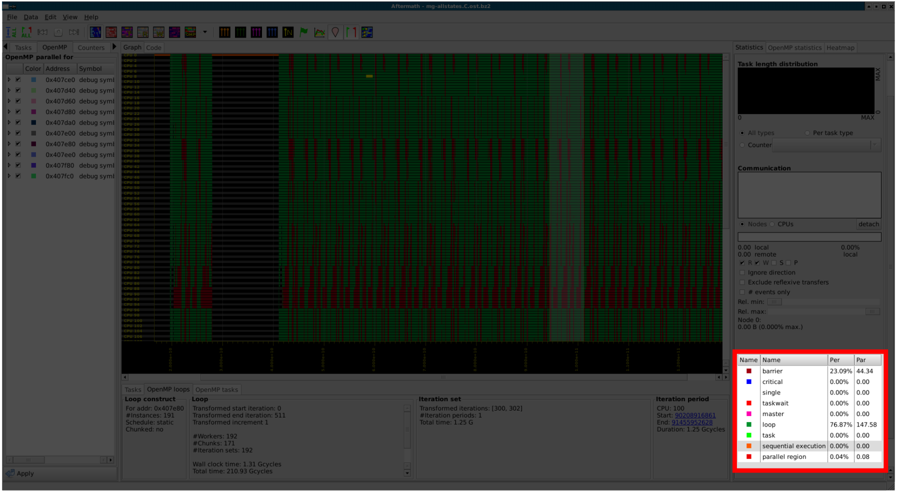


Time line: Loop constructs

Aftermath: Overview of the GUI

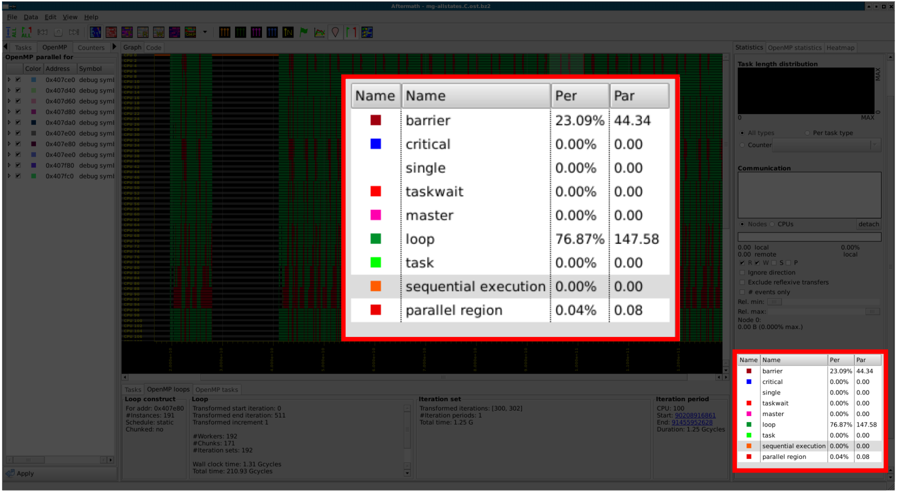


Time line: Loop constructs

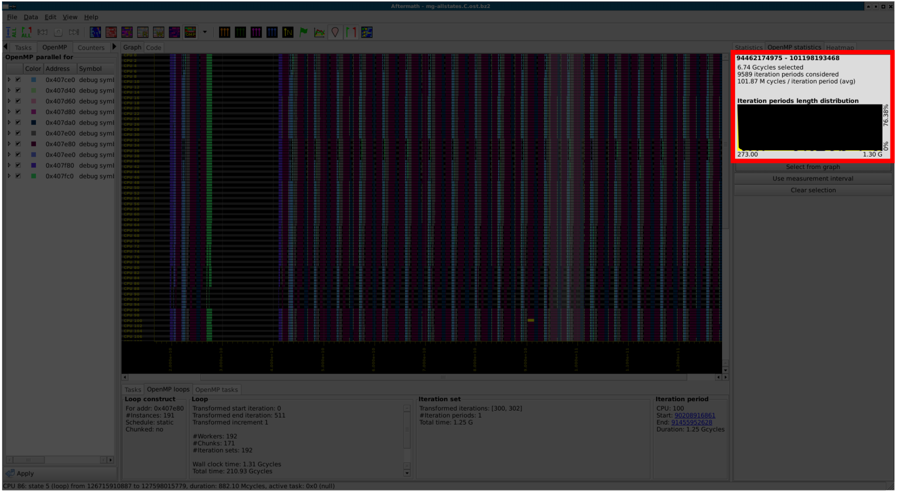


State statistics

Aftermath: Overview of the GUI

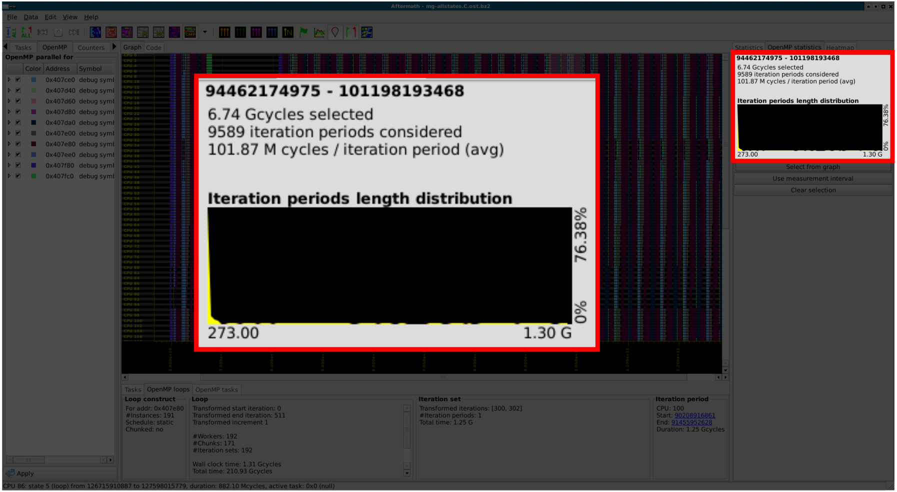


State statistics



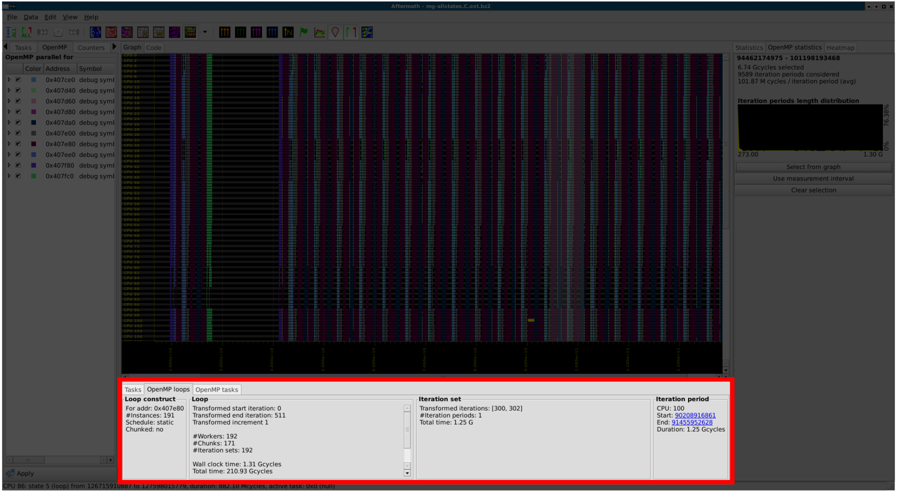
Histogram showing duration of iteration periods

Aftermath: Overview of the GUI

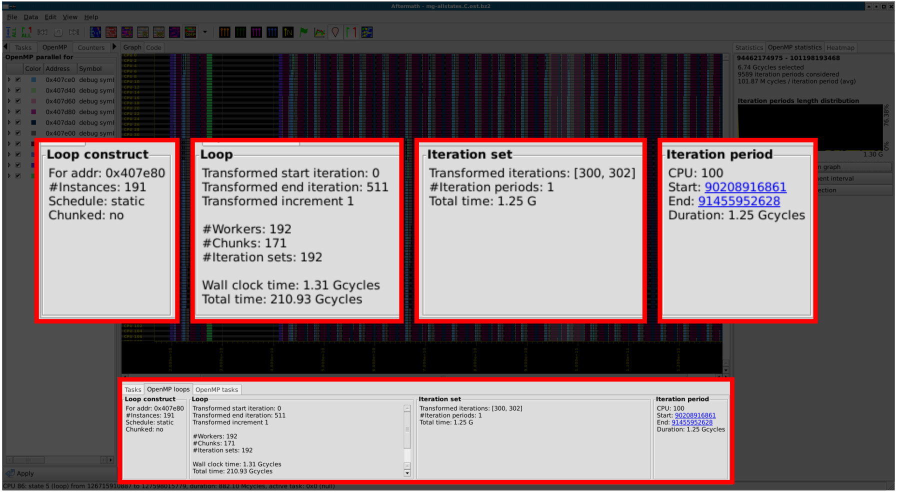


Histogram showing duration of iteration periods

Aftermath: Overview of the GUI

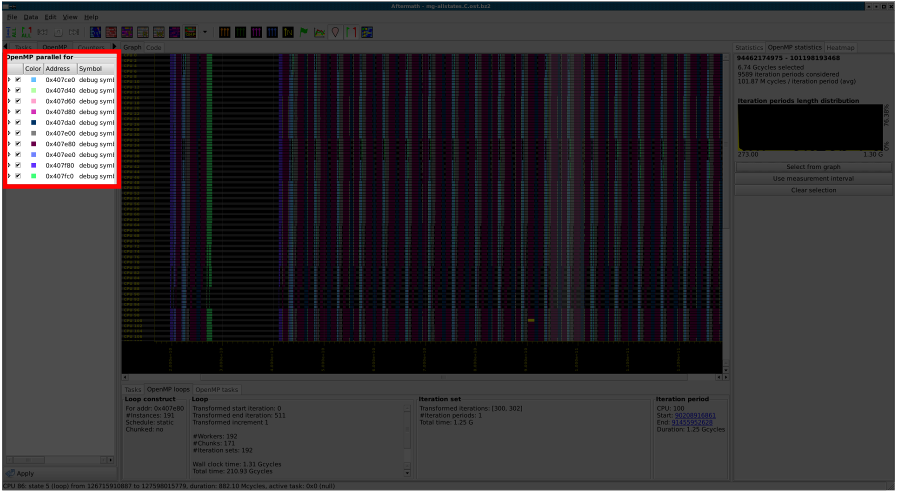


Detailed text view for parallel loops



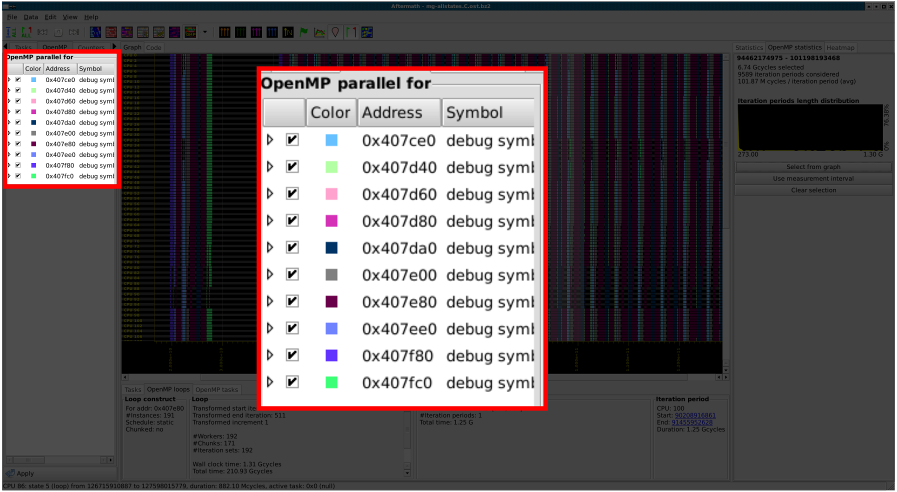
Detailed text view for parallel loops

Aftermath: Overview of the GUI



Filter for loop constructs

Aftermath: Overview of the GUI



Filter for loop constructs

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Loop construct

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Loop construct

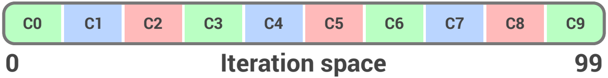
0 **Iteration space** **99**

Loop

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Loop construct

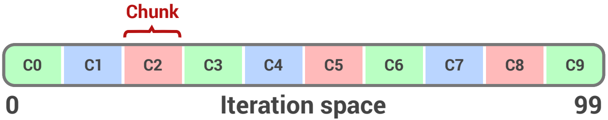


Loop

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Loop construct

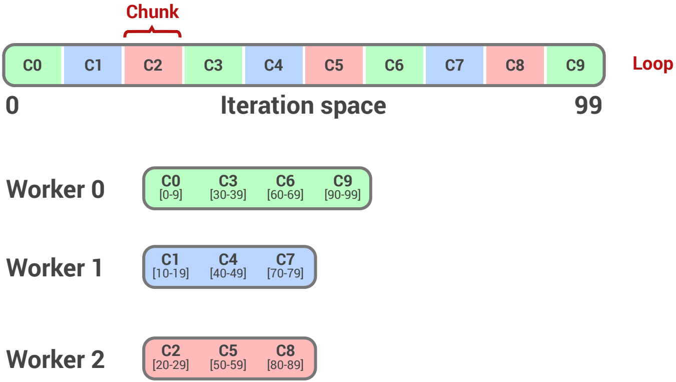


Loop

Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

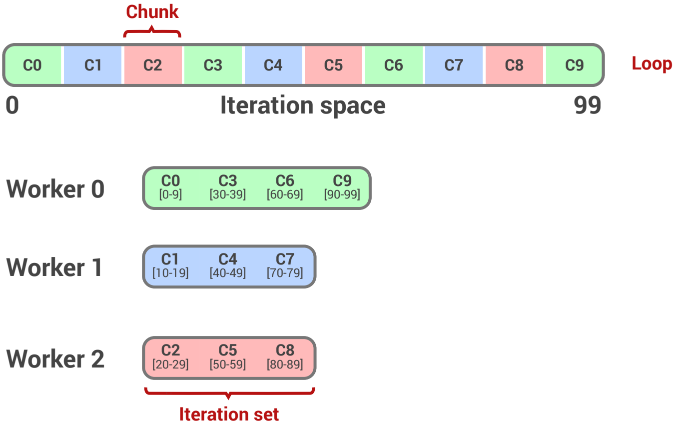
Loop construct



Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

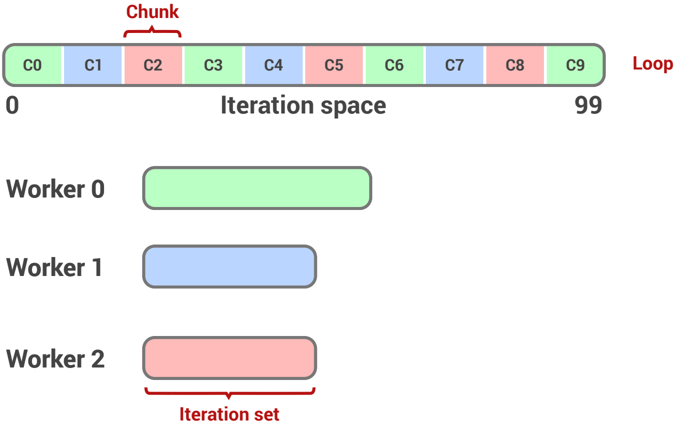
Loop construct



Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

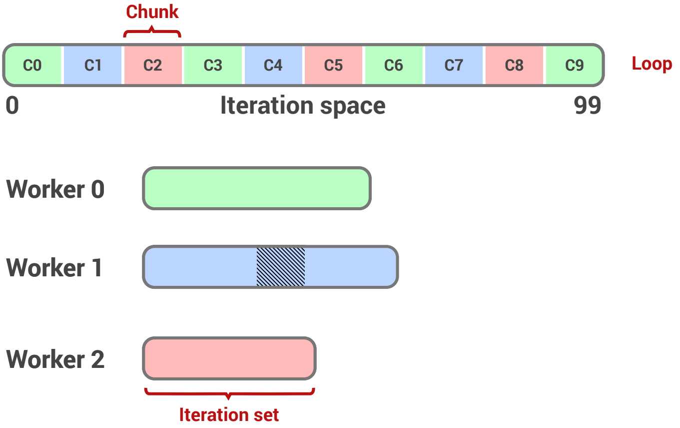
Loop construct



Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

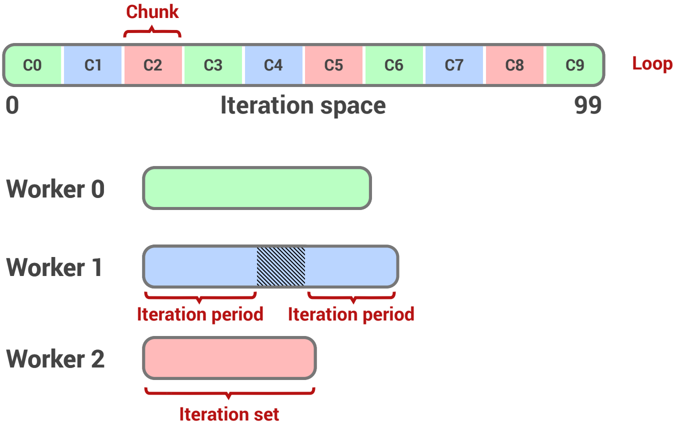
Loop construct



Terminology

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```

Loop construct



Example: Finding Prime Numbers in an Interval

```c
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(static) reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

Example: Finding Prime Numbers in an Interval

```
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(static) reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

```
int isprime_naive(int n)
{
  int j;

  if(n % 2 == 0 && n != 2)
    return 0;

  for(j = 3; j <= sqrt(n); j += 2)
    if(n % j == 0)
      return 0;

  return 1;
}
```

Example: Finding Prime Numbers in an Interval

```
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(static) reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

```
int isprime_naive(int n)
{
  int j;

  if(n % 2 == 0 && n != 2)
    return 0;

  for(j = 3; j <= sqrt(n); j += 2)
    if(n % j == 0)
      return 0;

  return 1;
}
```

Size of chunks?

Example: Finding Prime Numbers in an Interval

```
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(static) reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

```
int isprime_naive(int n)
{
  int j;

  if(n % 2 == 0 && n != 2)
    return 0;

  for(j = 3; j <= sqrt(n); j += 2)
    if(n % j == 0)
      return 0;

  return 1;
}
```

Size of chunks? Amount of work per chunk?

Example: Prime Numbers

```
# Go to folder with examples
$ cd /home/openstream/aftermath/doc/examples
```

Example: Prime Numbers

```
# Go to folder with examples
$ cd /home/openstream/aftermath/doc/examples

# Compile with clang
$ clang -fopenmp -g -o prime_naive prime_naive.c -lm
```

Example: Prime Numbers

```
# Go to folder with examples
$ cd /home/openstream/aftermath/doc/examples

# Compile with clang
$ clang -fopenmp -g -o prime_naive prime_naive.c -lm

# Execute using Aftermath-OpenMP
# -o prime_naive.ost:  write results to prime_naive.ost
# -f:  overwrite existing trace file
# Executable name and arguments after --
$ aftermath-openmp-trace -o prime_naive.ost -f -- ./prime_naive
```

Example: Prime Numbers

```
# Go to folder with examples
$ cd /home/openstream/aftermath/doc/examples

# Compile with clang
$ clang -fopenmp -g -o prime_naive prime_naive.c -lm

# Execute using Aftermath-OpenMP
# -o prime_naive.ost:  write results to prime_naive.ost
# -f:  overwrite existing trace file
# Executable name and arguments after --
$ aftermath-openmp-trace -o prime_naive.ost -f -- ./prime_naive
 There are 78498 prime numbers in the interval
```

Example: Prime Numbers

```
# Go to folder with examples
$ cd /home/openstream/aftermath/doc/examples

# Compile with clang
$ clang -fopenmp -g -o prime_naive prime_naive.c -lm

# Execute using Aftermath-OpenMP
# -o prime_naive.ost:  write results to prime_naive.ost
# -f:  overwrite existing trace file
# Executable name and arguments after --
$ aftermath-openmp-trace -o prime_naive.ost -f -- ./prime_naive
 There are 78498 prime numbers in the interval

# Open with aftermath
$ aftermath prime_naive.ost
```

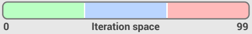
Demo: Static Schedule

OpenMP Schedules

OpenMP Schedules

Static

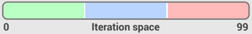
```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 100; i++)
{ ... }
```



OpenMP Schedules

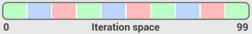
Static

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 100; i++)
{ ... }
```



Static + chunked

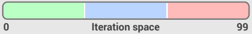
```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



OpenMP Schedules

Static

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 100; i++)
{ ... }
```

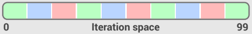


Dynamic

```
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < 100; i++)
{ ... }
```

Static + chunked

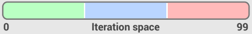
```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



OpenMP Schedules

Static

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 100; i++)
{ ... }
```

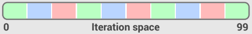


Dynamic

```
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < 100; i++)
{ ... }
```

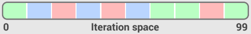
Static + chunked

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



Dynamic + chunked

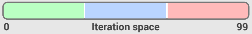
```
#pragma omp parallel for schedule(dynamic, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



OpenMP Schedules

Static

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 100; i++)
{ ... }
```

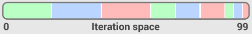


Dynamic

```
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < 100; i++)
{ ... }
```

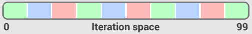
Guided

```
#pragma omp parallel for schedule(guided)
for(int i = 0; i < 100; i++)
{ ... }
```



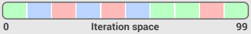
Static + chunked

```
#pragma omp parallel for schedule(static, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



Dynamic + chunked

```
#pragma omp parallel for schedule(dynamic, 10)
for(int i = 0; i < 100; i++)
{ ... }
```



Prime Numbers: Dynamic Schedule

```
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(dynamic, 1000) \
      reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

Demo: Dynamic Schedule

Prime Numbers: Guided Schedule

```c
int main(int argc, char** argv)
{
  int n = 1;

  #pragma omp parallel
  {
    #pragma omp for \
      schedule(guided) \
      reduction(+:n)
    for(int i = 3; i < 1000000; i += 2)
      n += isprime_naive(i);
  }

  printf("There are %d prime numbers"
         "in the interval\n", n);

  return 0;
}
```

Demo: Guided Schedule

NPB's MG benchmark

Benchmark: NPB MG

- NPB 2.3 C implementation from the Omni Compiler Project
- C input class (512×512 elements)

Test platform

- SGI UV 2000 (Xeon E5-4640)
- 192 cores (Hyperthreading disabled)
- 24 NUMA nodes, 756 GiB RAM
- LLVM/clang 3.8.0
- Aftermath-OpenMP for trace generation

Demo: NPB

Preparing the Hands-on Session

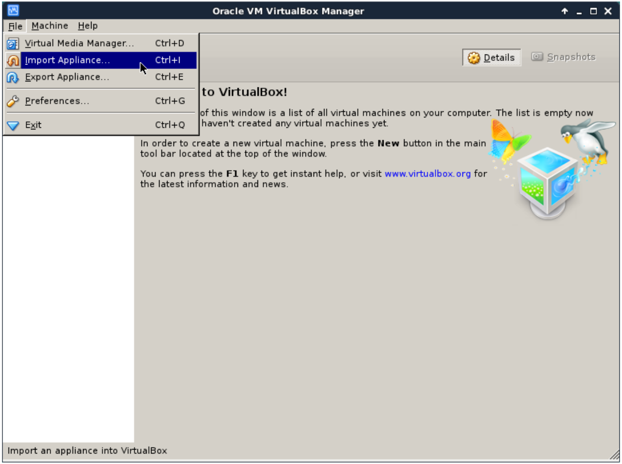
Virtual Machine Image

- Preinstalled Aftermath, Aftermath-OpenMP and OpenStream
- Example programs and traces
- Documentation and quick-start guide
- USB sticks with VM image available
- Available online:
  `https://www.aftermath-tracing.com/hipeac-2017-tutorial/`

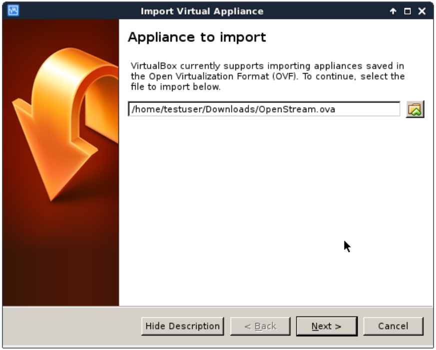
Handout

- Suggestions for exercises, with example commands
- Bibliography and online resources

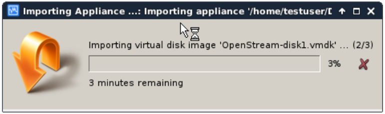
Virtual Machine Image



Virtual Machine Image

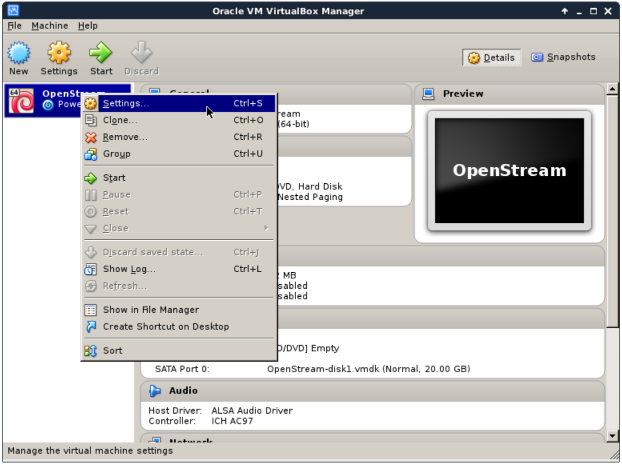


Virtual Machine Image

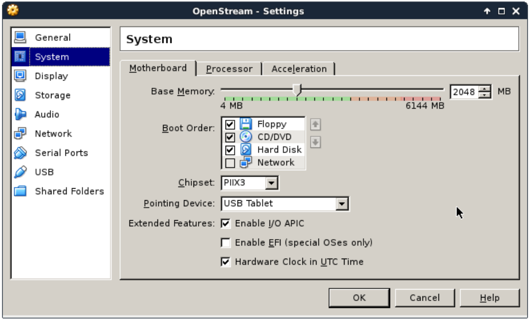


Virtual Machine Image

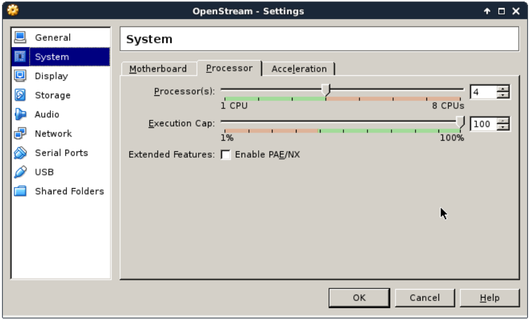
Virtual Machine Image



Virtual Machine Image



Virtual Machine Image



Hands-on Session

Outline

Morning: Intro + Analysis of OpenMP Programs [10:00 – 13:00]

- Overview of Aftermath
- Examples of OpenMP analyses
- Generating traces with Aftermath-OpenMP
- Hands-on session

Lunch [13:00 – 14:00]

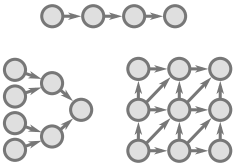
Afternoon: Task-Parallel Programs / NUMA [14:00 – 17:30]

- Introduction to Data-Flow Tasks in OpenStream
- Visualization / analysis of task-based programs
- Performance analysis on NUMA platforms
- Hands-on session

Task-parallel Programming

Principles of task-parallel programming

- Expose large amounts of parallelism
- Define small units of work: tasks
- Fine-grained synchronization
- Run-time manages execution



Frameworks for task-parallelism

- OpenMP (since 4.0: dependent tasks), OmpSs
- OpenStream
- Cilk, CnC, Habanero, X10

OpenStream

Concepts

- Extension to OpenMP for data-centric computing
  - **Streams**: Typed, unbounded channels for communication
  - **Tasks**: Short-lived entities accessing stream elements
- Tasks and streams are created *dynamically* at execution time

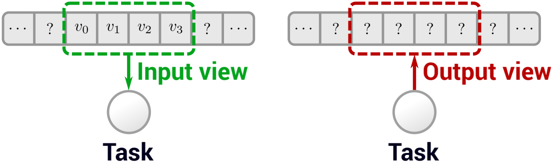
OpenStream

Concepts

- Extension to OpenMP for data-centric computing
  - **Streams**: Typed, unbounded channels for communication
  - **Tasks**: Short-lived entities accessing stream elements
- Tasks and streams are created *dynamically* at execution time

Stream accesses

- Stream elements accessed using **views**



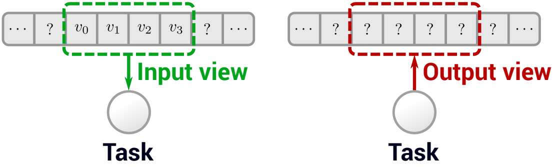
OpenStream

Concepts

- Extension to OpenMP for data-centric computing
  - **Streams**: Typed, unbounded channels for communication
  - **Tasks**: Short-lived entities accessing stream elements
- Tasks and streams are created *dynamically* at execution time

Stream accesses

- Stream elements accessed using **views**



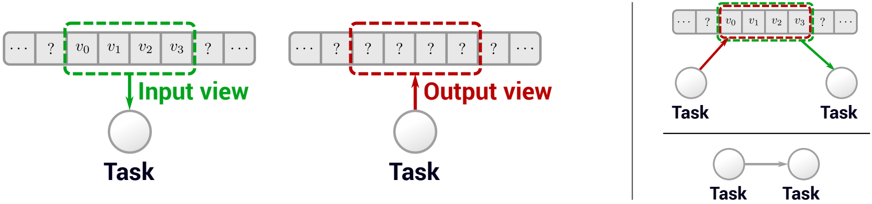
- Dependences between tasks result from stream accesses

Concepts

- Extension to OpenMP for data-centric computing
  - **Streams**: Typed, unbounded channels for communication
  - **Tasks**: Short-lived entities accessing stream elements
- Tasks and streams are created *dynamically* at execution time

Stream accesses

- Stream elements accessed using **views**



- Dependences between tasks result from stream accesses

OpenStream: Syntax

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));
```

OpenStream: Syntax

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));

/* Declaration of a view providing access to 4 stream elements */
float in_view[4];
```

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));

/* Declaration of a view providing access to 4 stream elements */
float in_view[4];

/* Consumer with input view of 4 elements */
#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}
```

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));

/* Declaration of a view providing access to 4 stream elements */
float in_view[4];

/* Consumer with input view of 4 elements */
#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

/* Declaration of a view providing access to 2 stream elements */
float out_view[2];
```

OpenStream: Syntax

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));

/* Declaration of a view providing access to 4 stream elements */
float in_view[4];

/* Consumer with input view of 4 elements */
#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

/* Declaration of a view providing access to 2 stream elements */
float out_view[2];

/* Producers */
for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream program with one consumer and two producers

```
/* Stream of floating point elements */
float str __attribute__((stream));

/* Declaration of a view providing access to 4 stream elements */
float in_view[4];

/* Consumer with input view of 4 elements */
#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

/* Declaration of a view providing access to 2 stream elements */
float out_view[2];

/* Producers */
for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

How does the run-time match producers and consumers?

OpenStream: Matching of Producers and Consumers

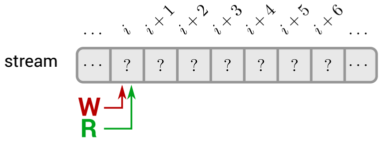
```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



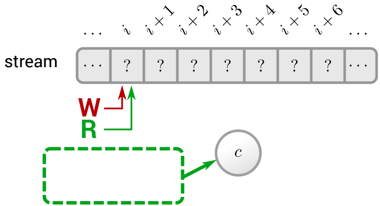

```
float str __attribute__((stream)); ◀◀◀
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



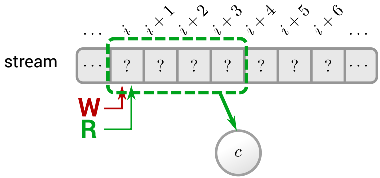

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4]) ◀◀◀
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers

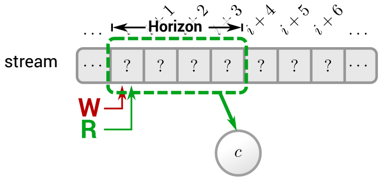



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])  ◀◀◀
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

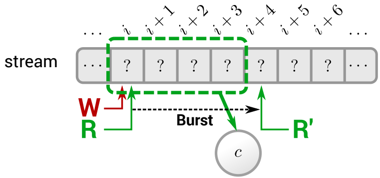



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4]) ◀◀◀
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```



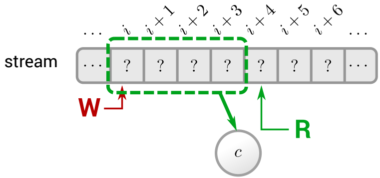

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4]) ◀◀◀
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers

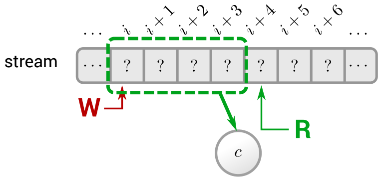



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4]) ◀◀◀
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

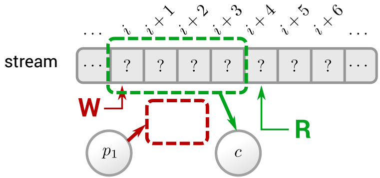



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) { ◀◀◀
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```



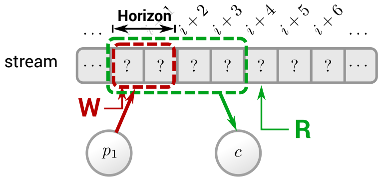

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



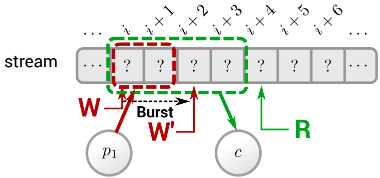

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



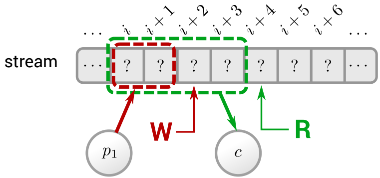

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2]) ◀◀◀
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers

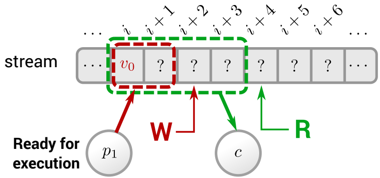



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2]) ◀◀◀
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```



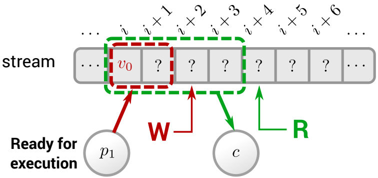

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



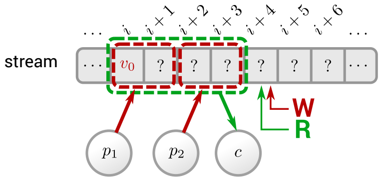

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) { ◀◀◀
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers



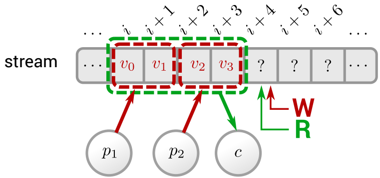

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

OpenStream: Matching of Producers and Consumers

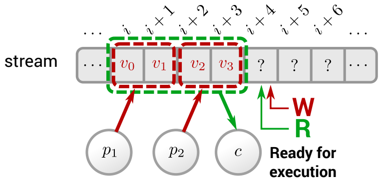



```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```



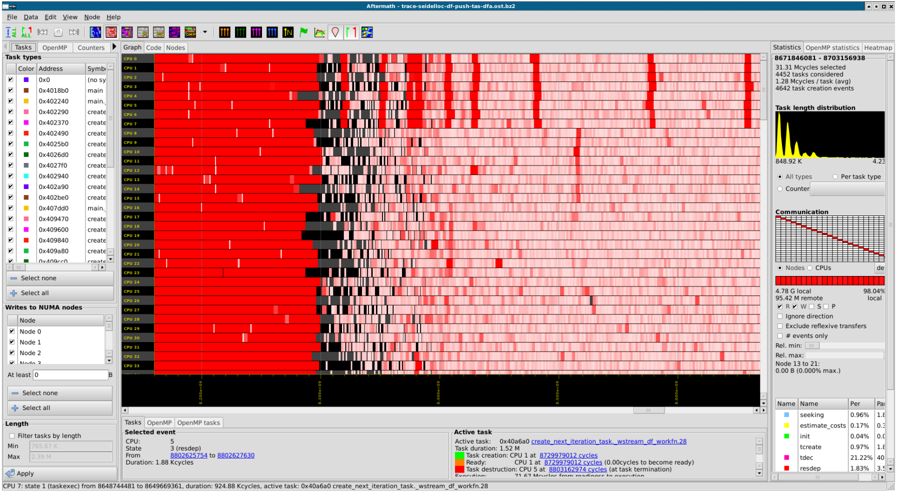

```
float str __attribute__((stream));
float in_view[4];

#pragma omp task input(str >> in_view[4])
{
  for(int i = 0; i < 4; i++)
    printf("Read %f\n", in_view[i]);
}

float out_view[2];

for(int i = 0; i < 2; i++) {
  #pragma omp task output(str << out_view[2])
  {
    for(int j = 0; j < 2; j++)
      out_view[j] = ...;
  }
}
```

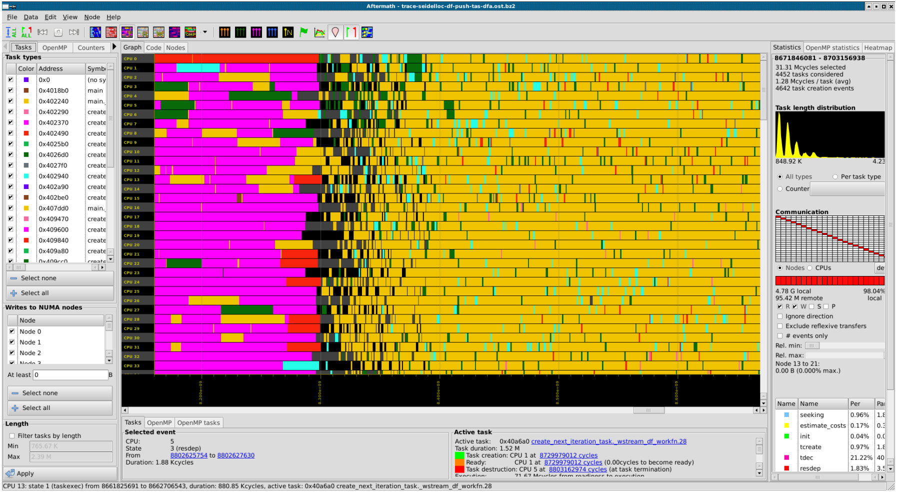
Visualization of per-task Performance Indicators



Heat map (duration of tasks)

Short (white) to long (red)

Visualization of per-task Performance Indicators



Type map

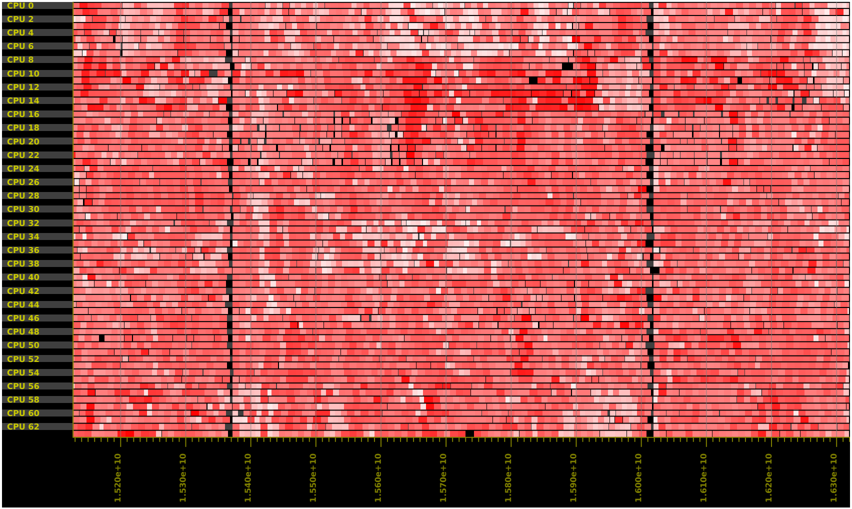
One color per task pragma

Visualization of per-task Performance Indicators

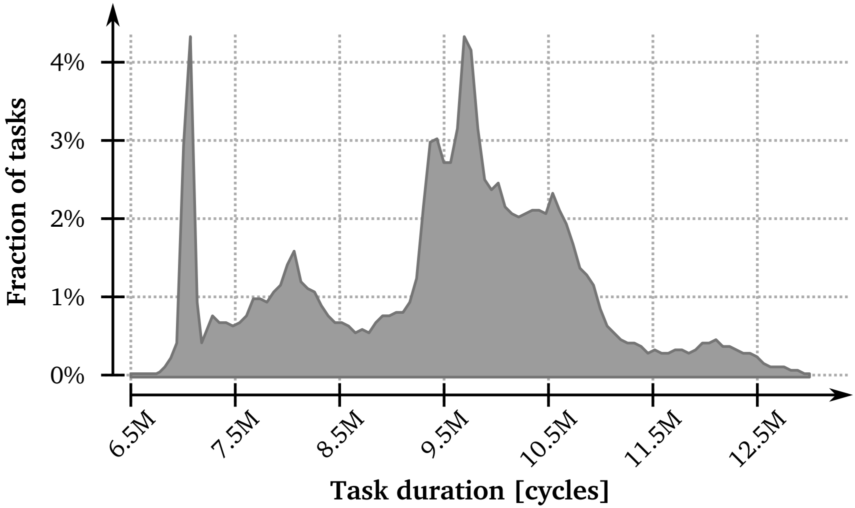
Heat map (duration of tasks)

Short (white) to long (red), overlay with branch misprediction rate

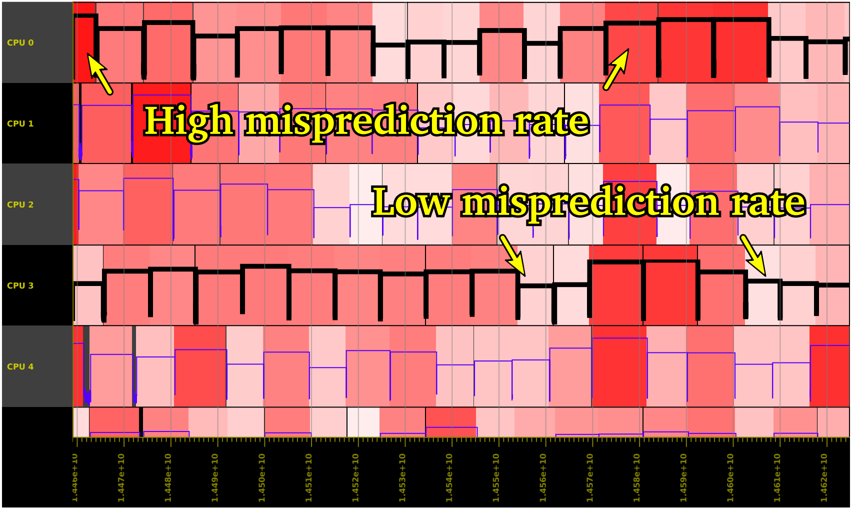
Finding Correlations



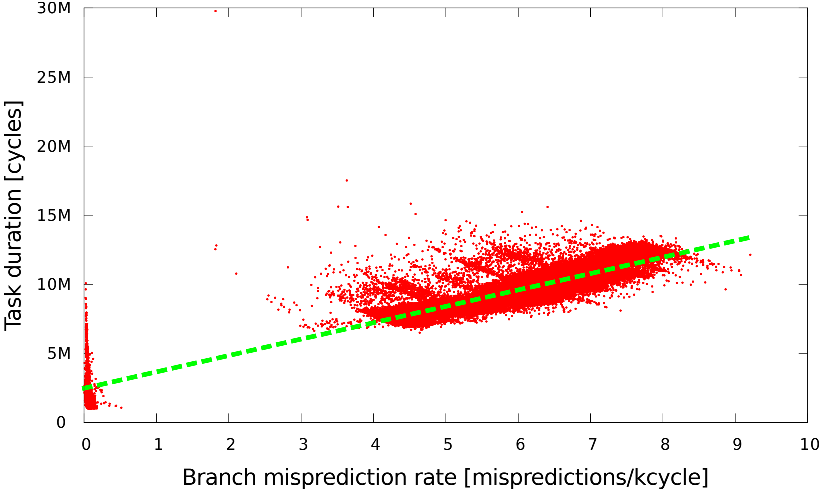
Finding Correlations



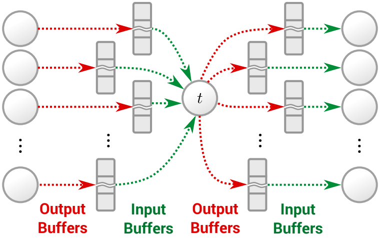
Finding Correlations



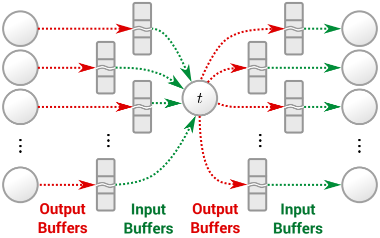
Finding Correlations



Task Buffers



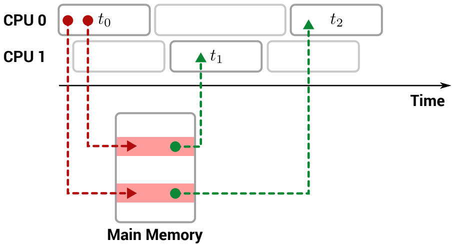
Task Buffers

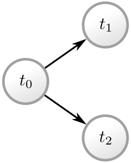


Buffers managed by run-time

- Amount of data accessed by a task known
- Access mode known
- Contiguous memory region for each dependence
- Aware of placement / control over placement
- Traces with rich information on dependences and memory accesses

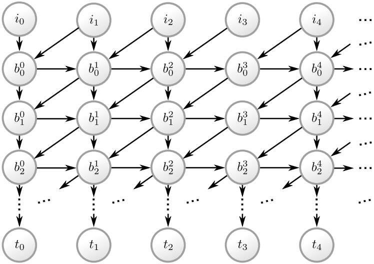
Reconstruction of the Task Graph



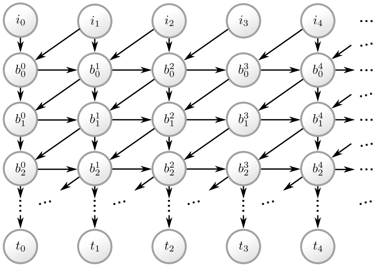


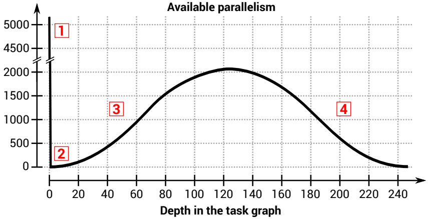
- During execution: only trace memory accesses for dependences
- When loading the trace: find reader and writers for each buffer
- Reconstruct task graph

Analysis of the Task Graph

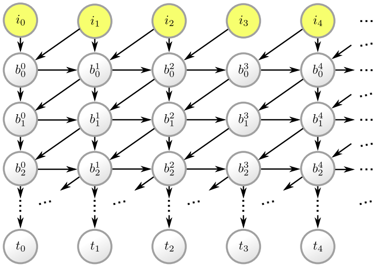


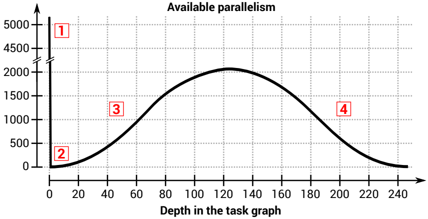
Analysis of the Task Graph



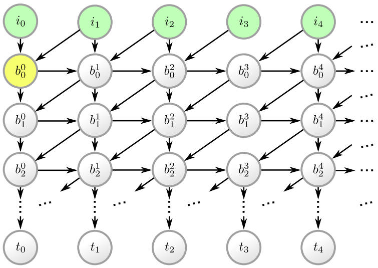


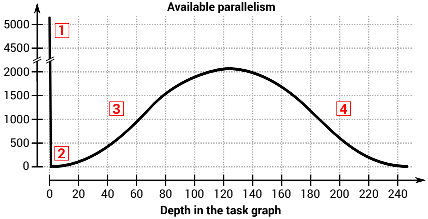
Analysis of the Task Graph



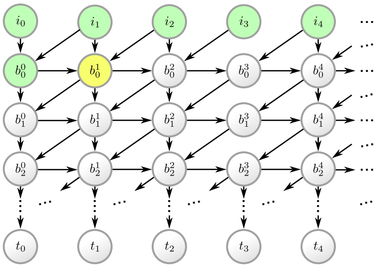


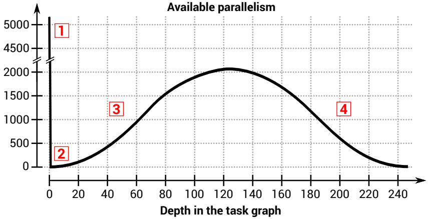
Analysis of the Task Graph



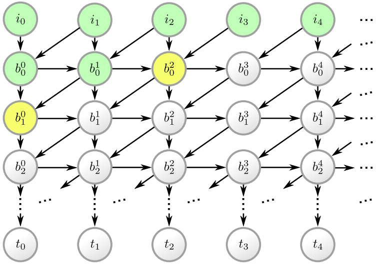


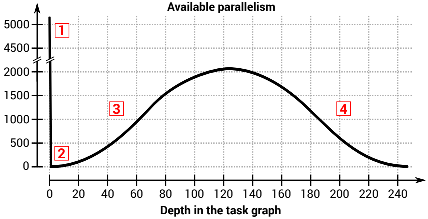
Analysis of the Task Graph



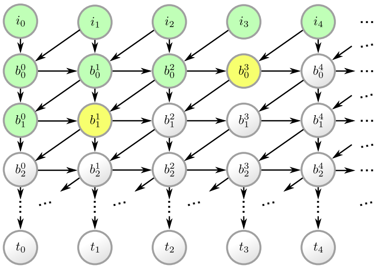


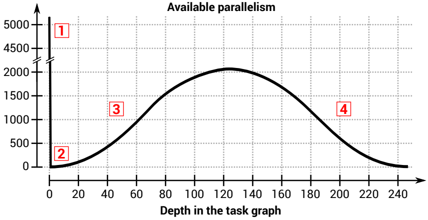
Analysis of the Task Graph



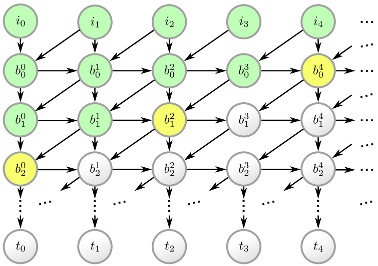


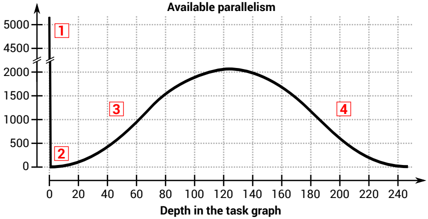
Analysis of the Task Graph



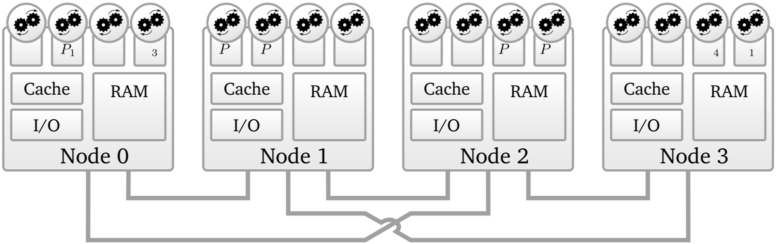


Analysis of the Task Graph

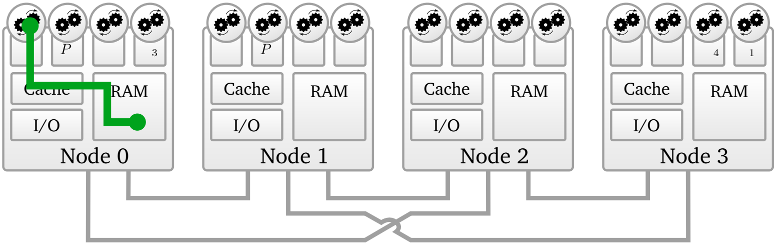




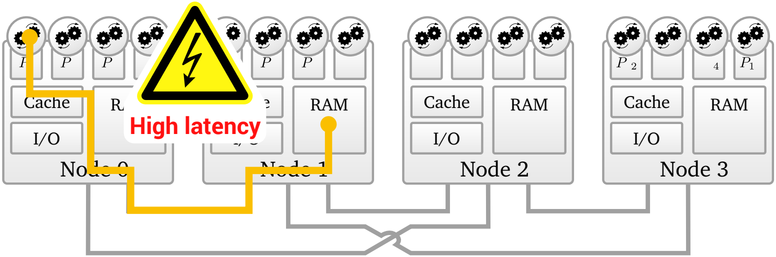
Non-Uniform Memory Access (NUMA)



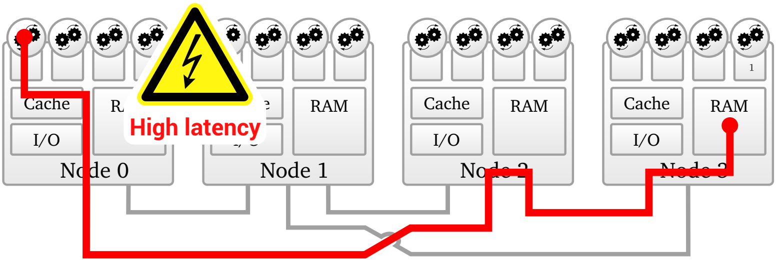
Non-Uniform Memory Access (NUMA)



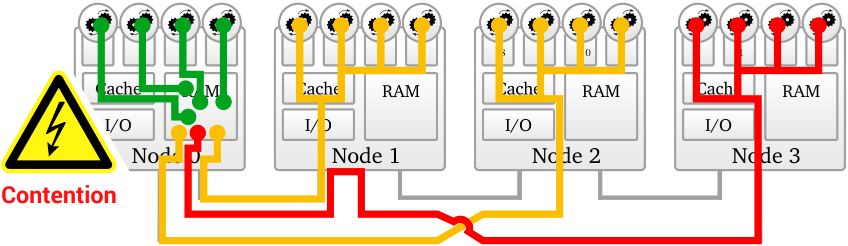
Non-Uniform Memory Access (NUMA)



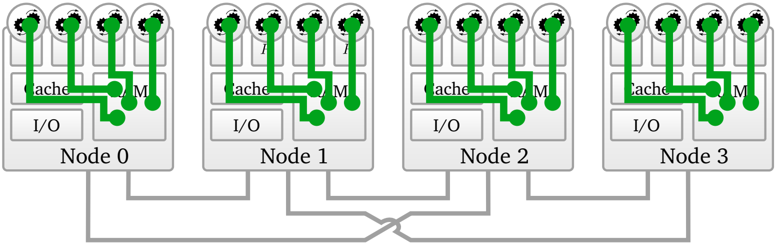
Non-Uniform Memory Access (NUMA)



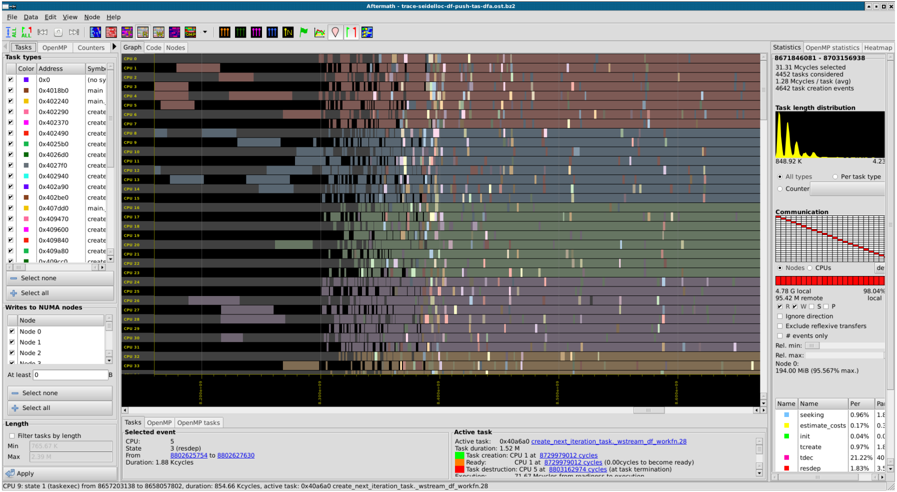
Non-Uniform Memory Access (NUMA)



Non-Uniform Memory Access (NUMA)



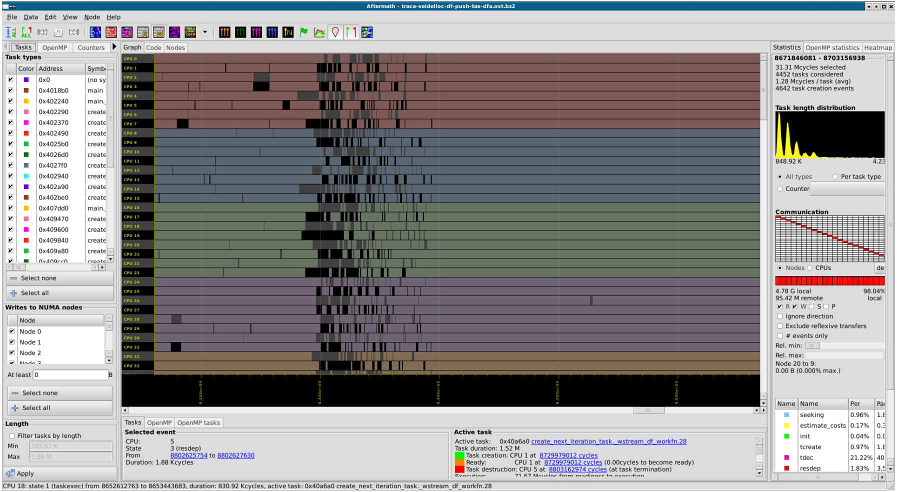
Visualization of Behavior wrt. NUMA



NUMA read map

NUMA nodes most data was read from, one color per NUMA node

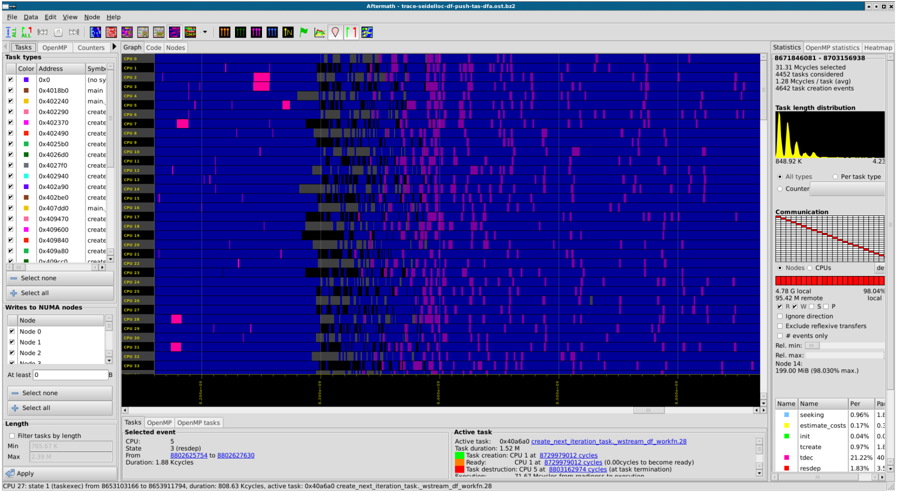
Visualization of Behavior wrt. NUMA



NUMA write map

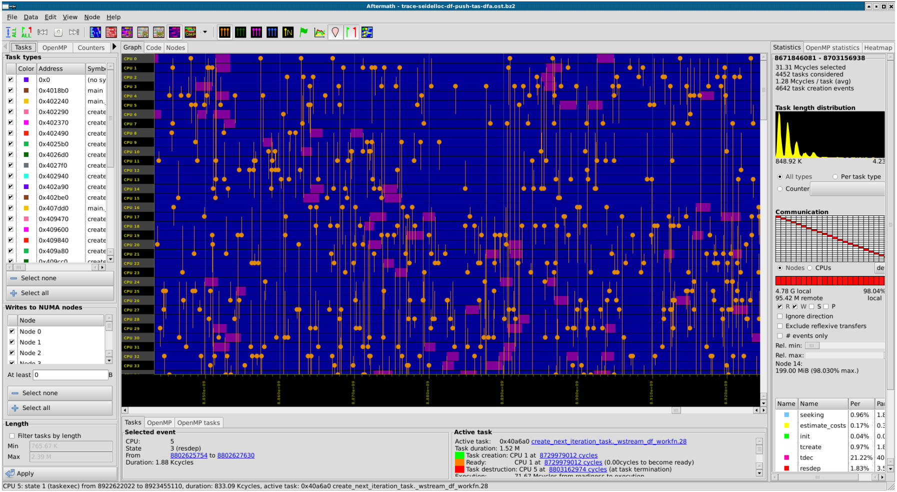
NUMA nodes most data was written to, one color per NUMA node

Visualization of Behavior wrt. NUMA



NUMA heat map

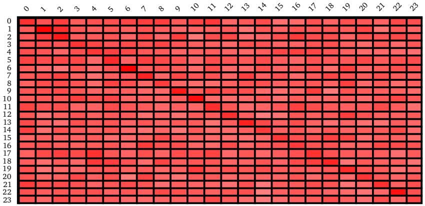
100% local (blue) to 100% remote (purple)



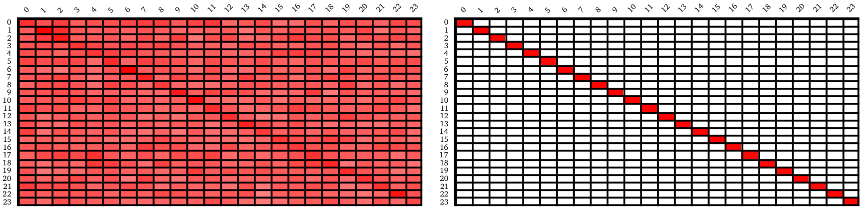
NUMA heat map

100% local (blue) to 100% remote (purple), overlay: work-stealing

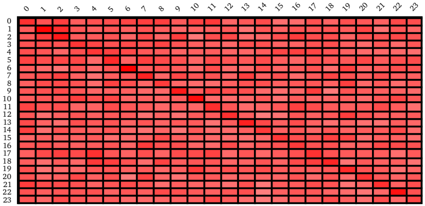
Visualization of Behavior wrt. NUMA

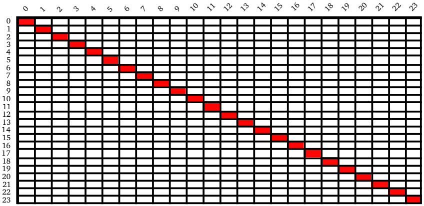


Visualization of Behavior wrt. NUMA

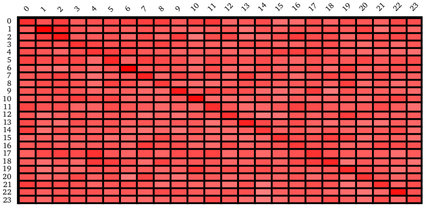


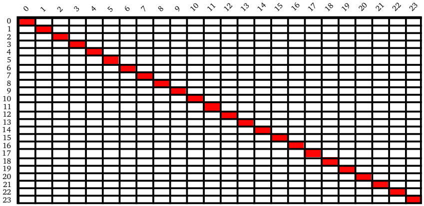
Visualization of Behavior wrt. NUMA

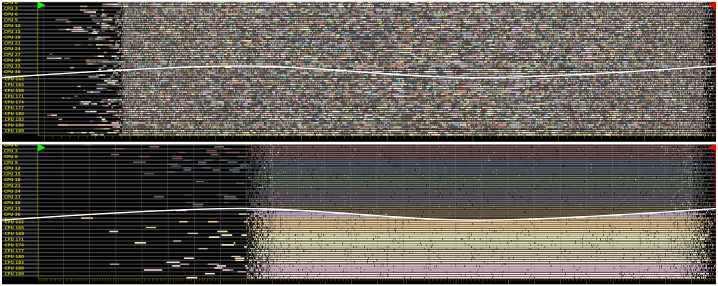




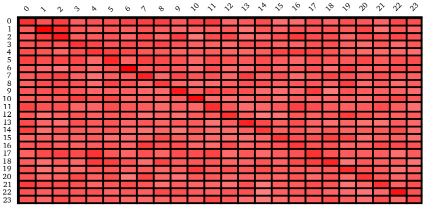
Visualization of Behavior wrt. NUMA

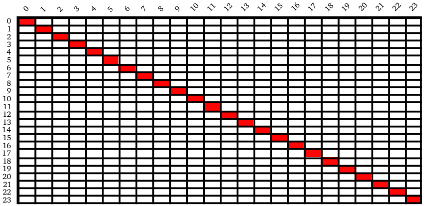


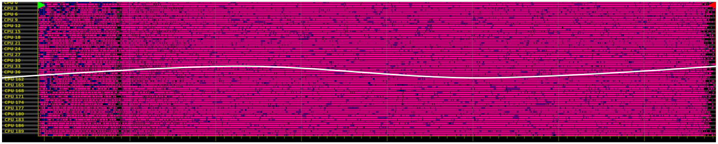




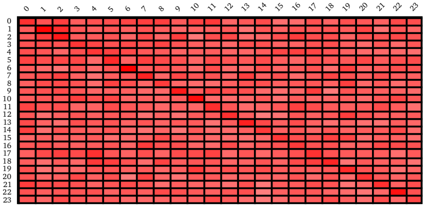
Visualization of Behavior wrt. NUMA

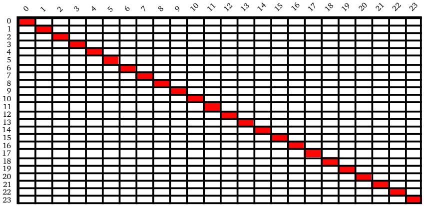


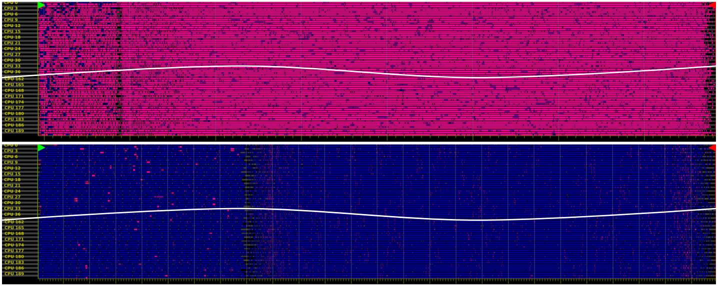




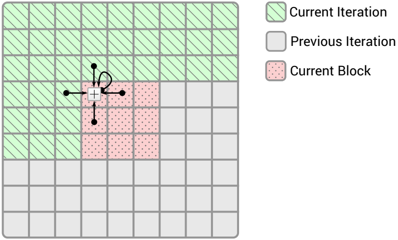
Visualization of Behavior wrt. NUMA



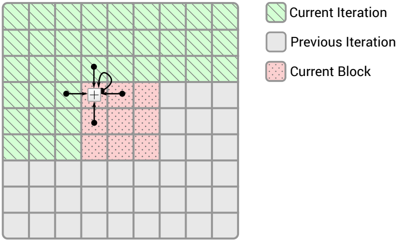




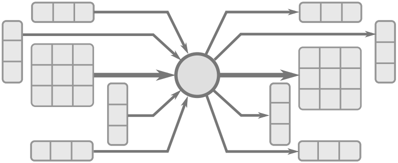
Demo: Seidel-2d



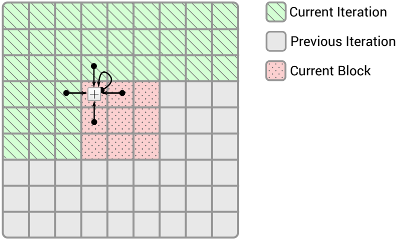
Demo: Seidel-2d



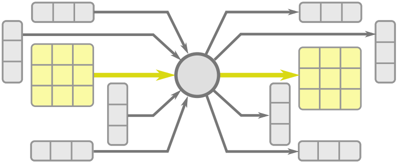
OpenStream implementation



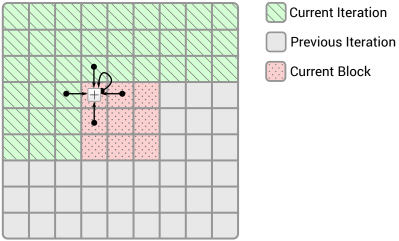
Demo: Seidel-2d



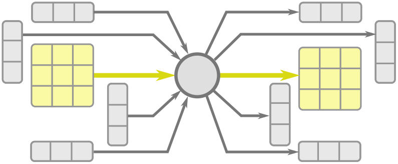
OpenStream implementation



Demo: Seidel-2d



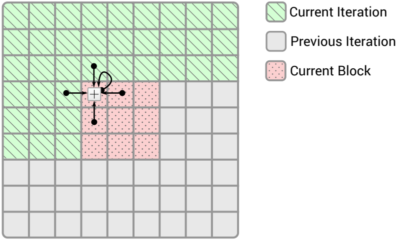
OpenStream implementation



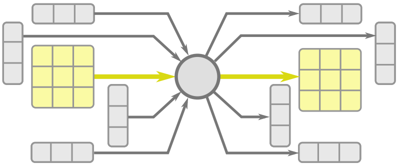
Benchmark parameters

- Double precision floats
- Matrix: $2^{14} \times 2^{14}$ (2 GiB)
- Block: $2^8 \times 2^8$ (512 KiB)
- 60 iterations
- 254,977 tasks

Demo: Seidel-2d



OpenStream implementation



Benchmark parameters

- Double precision floats
- Matrix: $2^{14} \times 2^{14}$ (2 GiB)
- Block: $2^{8} \times 2^{8}$ (512 KiB)
- 60 iterations
- 254,977 tasks

Test system

- SGI UV 2000
- 24×Intel Xeon E5-4640
- Hyperthreading disabled
- 192 cores
- 24 NUMA nodes, 756 GiB RAM

Hands-on Session

Summary & Conclusion

Aftermath

- Reactive graphical user interface for trace analysis
- Programming model-centric analysis: Loops and tasks

Summary & Conclusion

Aftermath

- Reactive graphical user interface for trace analysis
- Programming model-centric analysis: Loops and tasks

Aftermath-OpenMP

- Instrumented LLVM/clang OpenMP run-time
- Low tracing overhead

Summary & Conclusion

Aftermath

- Reactive graphical user interface for trace analysis
- Programming model-centric analysis: Loops and tasks

Aftermath-OpenMP

- Instrumented LLVM/clang OpenMP run-time
- Low tracing overhead

OpenStream

- Framework for task-parallel programming
- Data-flow tasks
- Built-in support for Aftermath

Summary & Conclusion

Aftermath

- Reactive graphical user interface for trace analysis
- Programming model-centric analysis: Loops and tasks

Aftermath-OpenMP

- Instrumented LLVM/clang OpenMP run-time
- Low tracing overhead

OpenStream

- Framework for task-parallel programming
- Data-flow tasks
- Built-in support for Aftermath

Source code / papers / documentation / mailing lists

- `https://www.aftermath-tracing.com`
- `http://www.openstream.info`