Performance Tools for Task Parallel Programs

An Huynh

University of Tokyo

ADAC Workshop, Tokyo, Japan
February 15, 2018

About me

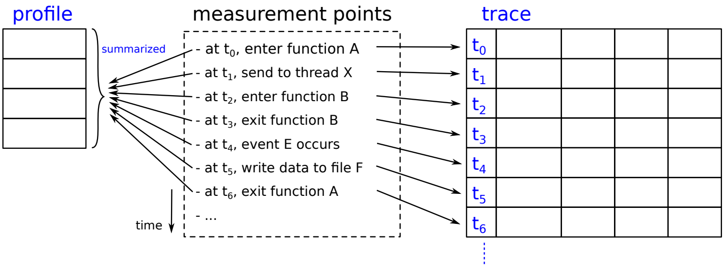
- I am a PhD candidate at the University of Tokyo (supervisor: Prof. Kenjiro Taura), expected to graduate in March 2018.
- Research: analyzing performance of task parallel programs
- Thesis title: "Analyzing Performance Differences of Task Parallel Runtime Systems based on Scheduling Delays"
- Today I'm going to introduce our performance toolset (DAGViz) from the perspective of performance tools for parallel programs.

Outline

- A light classification of common performance tools
- DAGViz
  - a task-centric performance tool for task parallel programs
- Related work
  - some similar approaches
- Conclusion

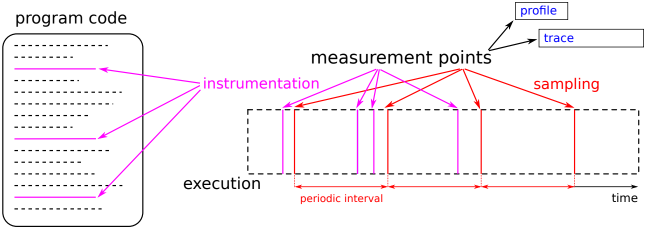
Profilers vs. Tracers

- profilers *summarize* information about events during a program run
- tracers *record* all occurrences of events with *timestamps*
- tracing vs. profiling
  - ✕ tracing consumes more memory
  - ○ a trace is exhaustive, can be used to reconstruct a profile
- most tools offer both profiling and tracing



Measurement approaches for collecting profile/trace data

- instrumentation: measurement probes are injected inside the program code by some method
  - e.g., directly in source, compiler injects, inject in binary, (instrumented library)
- sampling: program's execution is interrupted the from outside to collect samples
  - e.g., interval timer, hardware counter overflow, instruction-based sampling
- sampling vs. instrumentation
  - × sampling is less related to program source
  - ○ but it has an adjustable measurement resolution (by adjusting sampling frequency) useful for controlling overhead



A light classification of some performance tools

- most of tools produce both tracing and profiling data
- some tools use either only instrumentation (e.g., Score-P, Vampir, TAU), only sampling (e.g., HPCToolkit, perf), or both (e.g., gprof, Extrae, VTune)

	instrumentation	sampling	profiling	tracing
gprof	○	○	○	
Extrae/Paraver	○	○		○
VTune	○	○	○	○
HPCToolkit		○	○	○
perf		○	○	○
Score-P (Vampir,Scalasca,TAU)	○		○	○
...	...	...	...	...

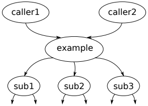
A light classification of some performance tools

- most of tools produce both tracing and profiling data
- some tools use either only instrumentation (e.g., Score-P, Vampir, TAU), only sampling (e.g., HPCToolkit, perf), or both (e.g., gprof, Extrae, VTune)
- two most common analyses are call path profiles and timelines visualizations of traces

	instrumentation	sampling	profiling	tracing
gprof	○	○	○	
Extrae/Paraver	○	○		○
VTune	○	○	○	○
HPCToolkit		○	○	○
perf		○	○	○
Score-P (Vampir,Scalasca,TAU)	○		○	○
...	...	...	...	...

Call path profiles

gprof [Graham et al. 2004] collects **instruction pointer** and **return address** → function & its calling parent

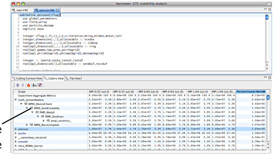



```
                        called/total      parents
index  %time    self descendents  called+self    name           index
                        called/total      children

                0.44        0.72      1/1           .__start [2]
[1]     99.1    0.44        0.72      1         .main [1]
                0.59        0.13   1024/1024        .fft [3]
                0.00        0.00    256/256         .cos [6]
                0.00        0.00    256/256         .sin [7]
                0.00        0.00      8/8           .gettimeofday [11]
                0.00        0.00      7/7           .printf [16]
                0.00        0.00      1/1           .atoi [31]
                0.00        0.00      1/1           .exit [33]

-----------------------------------------------

6.6s                                          <spontaneous>
[2]     99.1    0.00        1.16            .__start [2]
                0.44        0.72      1/1           .main [1]

-----------------------------------------------
```

text-based interface GUI

HPCToolkit [Adhianto et al. 2010] collects the **full function call path** by walking up the stack.

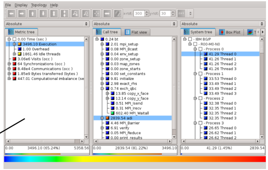
call path profile organized in tree



HPCToolkit's hpcviewer GUI

Score-P [Knupfer et al. 2012] organizes profile data in 3 dimensions: metrics–program–system (cube4 format).

three panes of three dimensions



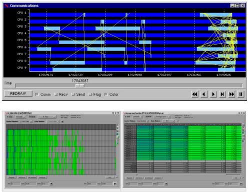
Score-P's CUBE GUI

→ help identify where in program code resources (e.g., execution time) are spent (function-centric)

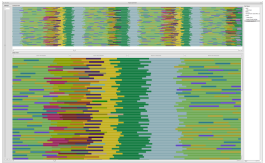
Timelines visualizations of traces

Many tools provide **timelines** visualizations (thread activities over time) of traces:

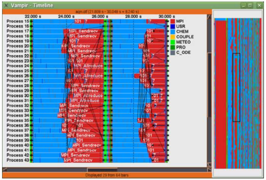
- e.g., Paraver [Llort et al. 2013], HPCToolkit [Adhianto et al. 2010], Vampir [Nagel et al. 1996], Jumpshot [Zaki et al. 1999], Jedule [Hunold et al. 2010], Aftermath [Drebes et al. 2014]



Paraver's GUI



HPCToolkit's hpctraceview GUI



Vampir's GUI

→ help pinpoint load imbalance among threads (thread-centric)

Task parallel programming models

Task parallel programming models expose a unified interface of logical tasks to programmers:

- ☺ arbitrarily nested hierarchical parallelism
- ☺ dynamic and automatic load balancing by (provably efficient) work stealing
- → a task-centric approach based on logical task structure is more meaningful

```
void quicksort(A,a,b,threshold) {
  if (b - a <= threshold) {
    simple_sort(A,a,b);
  } else {
    m = partition(A,a,b);
                quicksort(A,a,m,threshold) ;
    quicksort(A,m,b,threshold);

  }
}
```

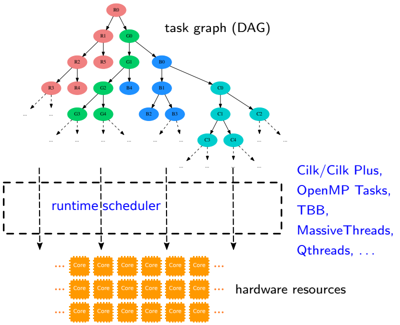
Task parallel programming models

Task parallel programming models expose a unified interface of logical tasks to programmers:

- ☺ arbitrarily nested hierarchical parallelism
- ☺ dynamic and automatic load balancing by (provably efficient) work stealing

→ a task-centric approach based on logical task structure is more meaningful

```
void quicksort(A,a,b,threshold) {
  if (b - a <= threshold) {
    simple_sort(A,a,b);
  } else {
    m = partition(A,a,b);
    create_task(quicksort(A,a,m,threshold));
    quicksort(A,m,b,threshold);
    wait_tasks;
  }
}
```

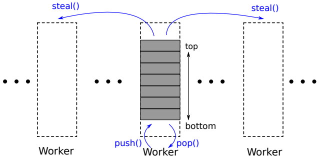


What is work stealing?

Work stealing is a provably efficient **scheduling strategy** deployed in many parallel and distributed systems:

- each worker maintains a double-ended queue (deque) of ready tasks
- a worker pushes/pops tasks from the bottom end of its deque
- an idle worker becomes a thief and goes steal a task from another worker (victim)
- a thief steals tasks from the top end of the victim's deque

→ idle workers bear the overhead of distributing work



What is work stealing?

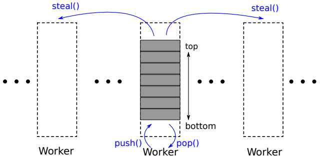
Work stealing is a provably efficient **scheduling strategy** deployed in many parallel and distributed systems:

- each worker maintains a double-ended queue (deque) of ready tasks
- a worker pushes/pops tasks from the bottom end of its deque
- an idle worker becomes a thief and goes steal a task from another worker (victim)
- a thief steals tasks from the top end of the victim's deque

→ idle workers bear the overhead of distributing work

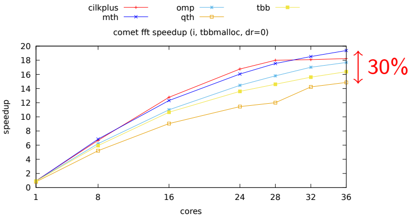
work stealing scheduler can perform within a factor of the optimal lower bound:

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$
- $T_P \leq c_1 T_1/P + c_\infty T_\infty$ [Blumofe et al. 1994]
- c_1: work overhead (e.g., push(), pop())
- c_∞: stealing overhead (e.g., steal())

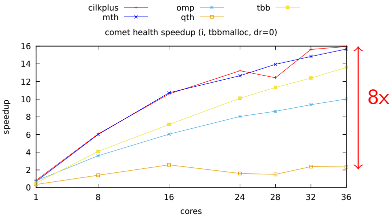


Scheduler implementation affects performance a lot

- almost all systems implement work stealing
- but there are still large performance differences among systems
- hence, a practical performance tool for evaluating task scheduler implementations is necessary



FFT



Health

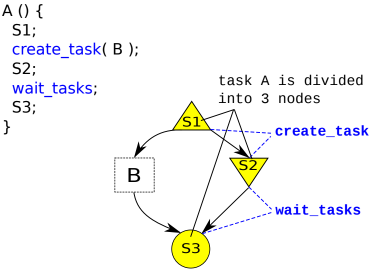
Computation DAG trace

Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

A task parallel program run can be modeled as a directed acyclic graph (computation DAG) in which

  - nodes: are serial code segments separated by task parallel primitives
  - edges: represent task parallel primitives

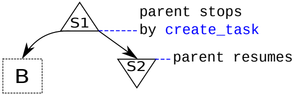


Computation DAG trace

Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created

```
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```

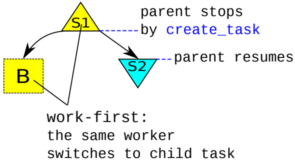


Computation DAG trace

Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created

```
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```

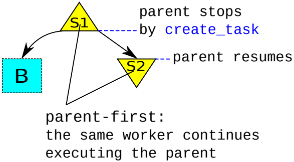


Computation DAG trace

Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created

```
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```

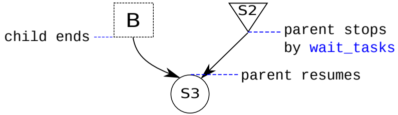


Computation DAG trace

Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

```
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```



Computation DAG trace

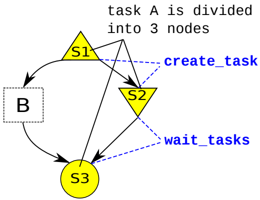
Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

A task parallel program run can be modeled as a directed acyclic graph (computation DAG) in which

  - nodes: are serial code segments separated by task parallel primitives
  - edges: represent these task parallel primitives

```
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```



Computation DAG trace

Two basic operations:
create_task and wait_tasks

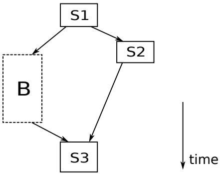
- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

A task parallel program run can be modeled as a directed acyclic graph (computation DAG) in which

  - nodes: are serial code segments separated by task parallel primitives
  - edges: represent these task parallel primitives

```
A () {
  S1;

  create_task(    B;    );

  S2;

  wait_tasks;

  S3;
}
```

recording timestamps



Computation DAG trace

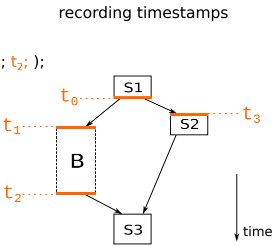
Two basic operations:
create_task and wait_tasks

- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

A task parallel program run can be modeled as a directed acyclic graph (computation DAG) in which

  - nodes: are serial code segments separated by task parallel primitives
  - edges: represent these task parallel primitives

```
A () {
  S1;
  t0;
  create_task( t1; B; t2; );
  t3;
  S2;

  wait_tasks;

  S3;
}
```



Computation DAG trace

Two basic operations:
create_task and wait_tasks

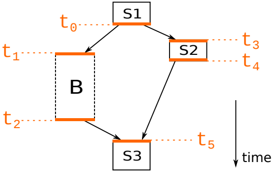
- At create_task, a new task is created
- At wait_tasks, the parent waits for children to complete

A task parallel program run can be modeled as a directed acyclic graph (computation DAG) in which

  - nodes: are serial code segments separated by task parallel primitives
  - edges: represent these task parallel primitives

```
A () {
  S1;
  t0;
  create_task( t1; B; t2; );
  t3;
  S2;
  t4;
  wait_tasks;
  t5;
  S3;
}
```

recording timestamps



Our performance toolset

Our performance toolset includes 3 parts:

- tpswitch: a portable wrapper around different task APIs
- DAG Recorder: a tracer that captures computation DAG
- DAGViz: a visualization and analysis tool for computation DAG

tpswitch

```
/* tpswitch.h */

/* To Cilk Plus */
#define create_task(st) cilk_spawn(st)
#define wait_tasks      cilk_sync

/* To OpenMP */
#define create_task(st) pragma_omp_task(,st)
#define wait_tasks      pragma_omp_taskwait

/* To TBB */
#define create_task(st) __tg__.run_([=]{st;})
#define wait_tasks      __tg__.wait_()
```

two generic primitives translate to equivalent ones in specific systems with measurement probes.

```
#include <tpswitch/tpswitch.h>

int fib( int n ) {
  if (n < 2) return n;
  int x, y;
  create_task({x = fib( n-1 );});
  y = fib( n-2 );
  wait_tasks();
  return x + y;
}
```

Cilk Plus

```
#include <cilk/cilk.h>

int fib( int n ) {
  if (n < 2) return n;
  int x, y;
  x = cilk_spawn fib( n-1 );
  y = fib( n-2 );
  cilk_sync;
  return x + y;
}
```

OpenMP

```
#include <omp.h>

int fib( int n ) {
  if (n < 2) return n;
  int x, y;
#pragma omp task
  { x = fib( n-1 ); }
  y = fib( n-2 );
#pragma omp taskwait
  return x + y;
}
```

TBB
(MassiveThreads, Qthreads)

```
#include <tbb/task_group.h>

int fib( int n ) {
  if (n < 2) return n;
  int x, y;
  tbb::task_group tg;
  tg.run([&]{x = fib( n-1 );});
  y = fib( n-2 );
  tg.wait();
  return x + y;
}
```

tpswitch

```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);


    create_task(   quicksort(A,a,m,threshold);    );


    quicksort(A,m,b,threshold);

    wait_tasks;

  }
}
```

tpswitch

To Cilk Plus

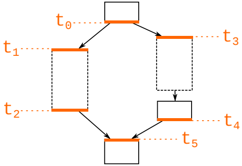
```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);


    cilk_spawn    quicksort(A,a,m,threshold);


    quicksort(A,m,b,threshold);

    cilk_sync;

  }
}
```

tpswitch

To Cilk Plus with DAG Recorder

```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);

    t0;
    cilk_spawn {t1;quicksort(A,a,m,threshold);t2;}
    t3;

    quicksort(A,m,b,threshold);
    t4;
    cilk_sync;
    t5;
  }
}
```

DAG captured by DAG Recorder



tpswitch

```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);


    create_task(   quicksort(A,a,m,threshold);    );


    quicksort(A,m,b,threshold);

    wait_tasks;

  }
}
```

tpswitch

To OpenMP

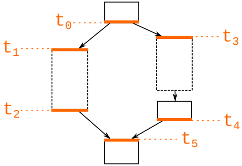
```
  void quicksort(A, a, b, threshold) {
    if (b - a <= threshold) {
      simple_sort(A, a, b);
    } else {
      m = partition(A, a, b);

#pragma omp task
                      quicksort(A,a,m,threshold);

#pragma omp task
      quicksort(A,m,b,threshold);

#pragma omp taskwait

    }
  }
```

tpswitch

To OpenMP with DAG Recorder

```
  void quicksort(A, a, b, threshold) {
    if (b - a <= threshold) {
      simple_sort(A, a, b);
    } else {
      m = partition(A, a, b);
      t0;
#pragma omp task
                    {t1;quicksort(A,a,m,threshold);t2;}
      t3;
#pragma omp task
      quicksort(A,m,b,threshold);
      t4;
#pragma omp taskwait
      t5;
    }
  }
```

DAG captured by DAG Recorder



tpswitch

```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);


    create_task(   quicksort(A,a,m,threshold);    );


    quicksort(A,m,b,threshold);

    wait_tasks;

  }
}
```

tpswitch

To TBB

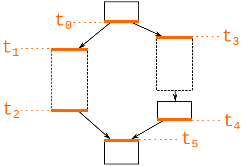
```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);
    tbb::task_group tg;

    tg.run([&]{     quicksort(A,a,m,threshold);     });


    quicksort(A,m,b,threshold);

    tg.wait();

  }
}
```

tpswitch

To TBB with DAG Recorder

```
void quicksort(A, a, b, threshold) {
  if (b - a <= threshold) {
    simple_sort(A, a, b);
  } else {
    m = partition(A, a, b);
    tbb::task_group tg;
    t0;
    tg.run([&]{ t1; quicksort(A,a,m,threshold); t2; });
    t3;

    quicksort(A,m,b,threshold);
    t4;
    tg.wait();
    t5;
  }
}
```

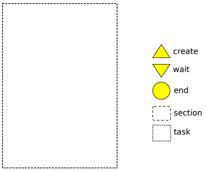
DAG captured by DAG Recorder



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task

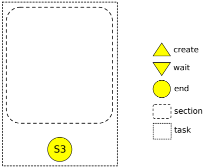
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task

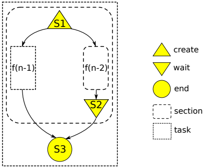
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task

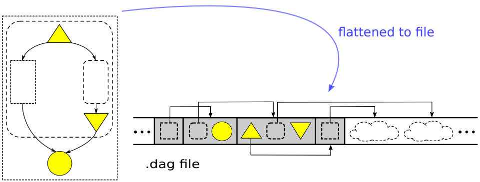
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

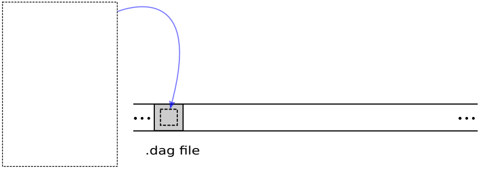
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

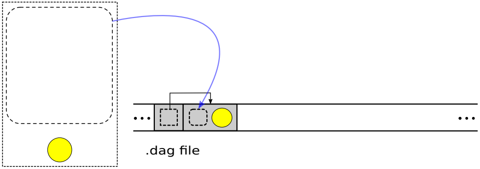
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

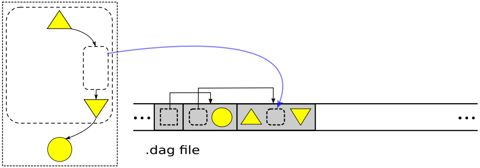
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

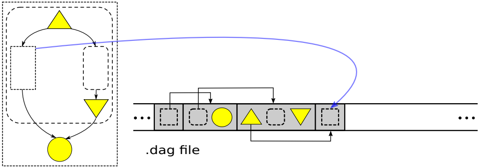
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

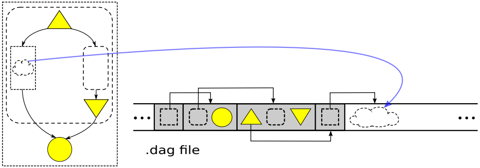
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

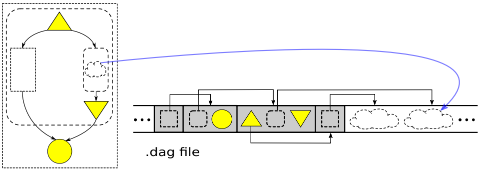
```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



DAG Recorder

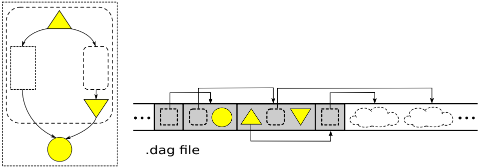
- DAG Recorder constructs the pointer-based hierarchical DAG in memory during the program run.
  - leaf nodes: create, wait, end
  - internal nodes: section (synchronization scope inside a task), task
- DAG Recorder flattens the DAG to file when the program finishes.

```
f(n) {
  S1;
  create_task( f(n-1) );
  f(n-2);
  S2;
  wait_tasks;
  S3;
}
```



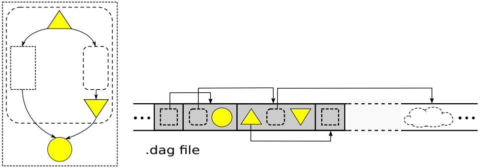
On-the-fly DAG contraction

- One challenge: storing every task in a fine-grained program consumes large memory
- Solution: collapse "uninteresting" subgraphs (e.g., executed solely by a single worker) into single nodes
  - still retain aggregate performance information of removed topology (e.g., total work, critical path)
  - memory overhead now scales with steals across workers rather than task creations



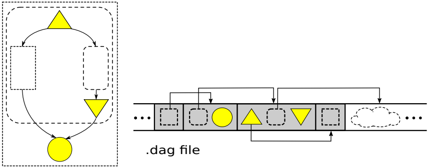
On-the-fly DAG contraction

- One challenge: storing every task in a fine-grained program consumes large memory
- Solution: collapse "uninteresting" subgraphs (e.g., executed solely by a single worker) into single nodes
  - still retain aggregate performance information of removed topology (e.g., total work, critical path)
  - memory overhead now scales with steals across workers rather than task creations



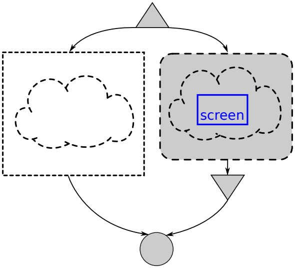
On-the-fly DAG contraction

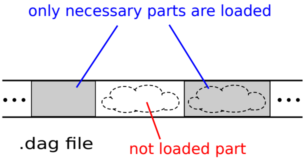
- One challenge: storing every task in a fine-grained program consumes large memory
- Solution: collapse "uninteresting" subgraphs (e.g., executed solely by a single worker) into single nodes
  - still retain aggregate performance information of removed topology (e.g., total work, critical path)
  - memory overhead now scales with steals across workers rather than task creations



DAGViz

- DAGViz reads DAG from file and re-constructs its hierarchical structure in memory to visualize
- One challenge: a (collapsed) DAG may still be very large, taking long time to load and render
- Solution: DAGViz deploys on-demand hierarchical expansion
  1. the DAG is expanded on demand in a top-down manner
  2. only expanded branch of the DAG is loaded and rendered



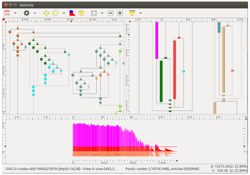


Demonstration

DAGViz's GUI and visualizations

DAGViz currently has two GUI versions based on two popular GUI toolkits:

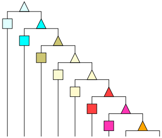
- C-based GTK+: GUI, rendering, and logics are written in C
- C++ and Python-based Qt5: GUI is written in Python, rendering is written in C++, logics are written in C



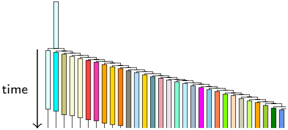
DAGViz's GUI

DAGViz provides many kinds of visualizations of the DAG:

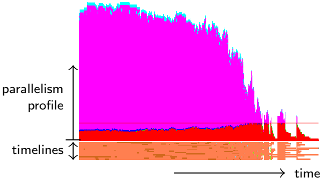
- basic DAG
- DAG with timing on vertical axis
- timelines together with parallelism profile



DAG



DAG with timing on y-axis



Case studies

We have found causes of performance bottlenecks in many cases:

- SparseLU
  - Cilk Plus, TBB have slow work stealing speed
  - Qthreads delays child tasks deliberately
- Alignment
  - OpenMP suffers from its size-limited task queue
- FFT
  - OpenMP suffers from its stack-overflow-avoiding measure
  - Qthreads delays child tasks deliberately
- Blackscholes
  - all systems suffer from Blackscholes' too small grain size
- Bodytrack
  - all systems suffer from Bodytrack's many long serial sections
- . . .

Related work

Some tools that visualize task graph (DAG) of task parallel programs are:

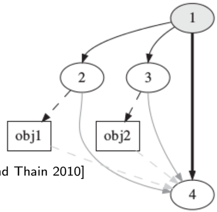
- ThreadScope [Wheeler and Thain 2010]: (Cilk, Qthreads, Pthreads) task graph with memory objects
- Temanejo [Brinkmann et al. 2011]: (OmpSs) task graph with dataflow dependencies
- Flow Graph Analyzer [Tovinkere and Voss 2014]: (TBB) task graph of TBB's flow graph interface
- Grain graph [Muddukrishna et al. 2016]: (OpenMP) task graph of tasks and loop chunks
- ...

ThreadScope

[Wheeler and Thain 2010]

ThreadScope uses Graphviz to visualize code regions and accessed memory objects.

- Cilk, Qthreads, Pthreads



[Wheeler and Thain 2010]

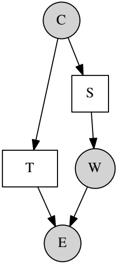
Graphviz [Gansner and North 2000] is a popular graph rendering engine:

- flatly renders all nodes & edges at once (flat layout)
- focuses on aesthetic aspects in layouts

→ easily gets slow with large graphs

DAGViz is scalable with hierarchical expansion

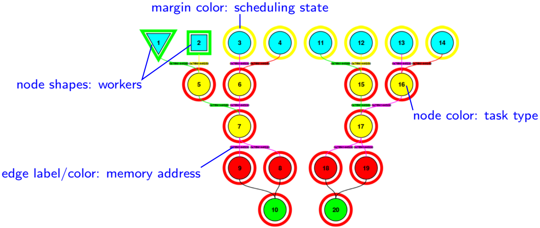
```
digraph G {
  /* nodes */
  C [style=filled,shape=circle];
  T [style=circle,shape=rectangle];
  S [style=circle,shape=square];
  W [style=filled,shape=circle];
  E [style=filled,shape=circle];

  /* edges */
  C->T;
  C->S;
  T->E;
  S->W;
  W->E;
}
```



Temanejo
[Brinkmann et al. 2011]

Temanejo interactively visualizes task graph with dataflow during a run of an OmpSs program

- OmpSs = OpenMP Tasks model + Mercurium compiler + Nanos++ runtime
- only OmpSs
- flat layout (NetworkX pakage)



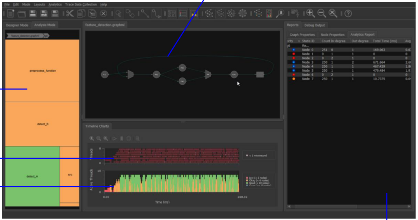
Temanejo's online visualization of task graph with data dependencies [Brinkmann et al. 2011]

Flow Graph Analyzer

[Tovinkere and Voss 2014]

Flow Graph Analyzer captures and visualizes task graph from program written with FLow Graph Interface of TBB 4.0.

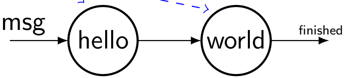
- only TBB
- flat layout



Flow Graph Analyzer's GUI [Tovinkere and Voss 2014]

An example program
with Flow Graph Interface

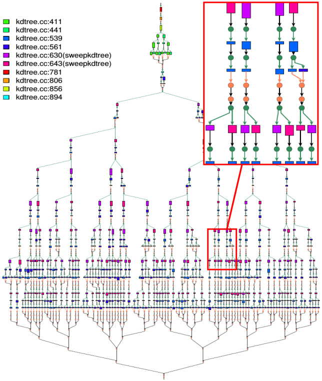
```
#include "tbb/flow_graph.h"
#include <iostream>

using namespace std;
using namespace tbb::flow;

int main() {
    graph g;
    continue_node< continue_msg> hello( g,
      []( const continue_msg &) {
          cout << "Hello";
      }
    );
    continue_node< continue_msg> world( g,
     []( const continue_msg &) {
          cout << " World\n";
      }
    );
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}
```



Grain graph
[Muddukrishna et al. 2016]

Grain graph captures and visualizes a graph of execution intervals of tasks and loop chunks (grains) from a run of an OpenMP program.

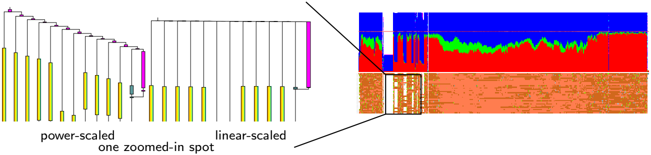
- only OpenMP
- flat layout
- non-interactive visualization (igraph package)



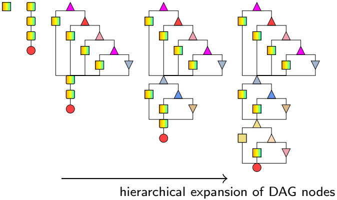
kdtree's grain graph [Muddukrishna et al. 2016]

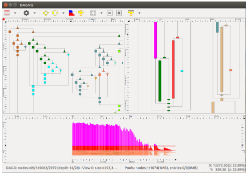
Publications

- A. Huynh, K. Taura, "**Delay Spotter: A Tool for Spotting Scheduler-Caused Delays in Task Parallel Runtime Systems**", IEEE International Conference on Cluster Computing (CLUSTER '17)



- A. Huynh, D. Thain, M. Pericas, K. Taura, "**DAGViz: A DAG Visualization Tool for Analyzing Task-Parallel Program Traces**", International Workshop on Visual Performance Analysis, held in conjunction with SC15 (VPA '15)





DAGViz's GUI

Conclusion

- DAGViz–a task-centric performance toolset for task parallel programs and schedulers:
  - ☺ logical task structure
  - ☺ scalable measurement (with DAG contraction)
  - ☺ scalable rendering (with on-demand hierarchical expansion)
- With a distinct focus on task schedulers, we hope DAGViz toolset to be a good addition to the existing large set of parallel performance tools.
- Future work:
  - to extend to distributed-memory systems
  - to analyze task locality with computation DAG

Conclusion

- DAGViz–a task-centric performance toolset for task parallel programs and schedulers:
  - ☺ logical task structure
  - ☺ scalable measurement (with DAG contraction)
  - ☺ scalable rendering (with on-demand hierarchical expansion)
- With a distinct focus on task schedulers, we hope DAGViz toolset to be a good addition to the existing large set of parallel performance tools.
- Future work:
  - to extend to distributed-memory systems
  - to analyze task locality with computation DAG

Thank you for listening!