Performance Tools for Task Parallel Programs

An Huynh

University of Tokyo

ADAC Workshop, Tokyo, Japan
February 15, 2018
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.
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**

![Diagram showing measurement points in a profile and corresponding trace events.

- at $t_0$, enter function A
- at $t_1$, send to thread X
- at $t_2$, enter function B
- at $t_3$, exit function B
- at $t_4$, event E occurs
- at $t_5$, write data to file F
- at $t_6$, exit function A
- ...]

- at $t_0$, enter function A
- at $t_1$, send to thread X
- at $t_2$, enter function B
- at $t_3$, exit function B
- at $t_4$, event E occurs
- at $t_5$, write data to file F
- at $t_6$, exit function A
- ...

- $t_0$
- $t_1$
- $t_2$
- $t_3$
- $t_4$
- $t_5$
- $t_6$

- ...
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 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)

<table>
<thead>
<tr>
<th></th>
<th>instrumentation</th>
<th>sampling</th>
<th>profiling</th>
<th>tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>gprof</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Extrae/Paraver</td>
<td>○</td>
<td>○</td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>VTune</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>HPCToolkit</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>perf</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Score-P</td>
<td>○</td>
<td></td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>(Vampir, Scalasca, TAU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Tool</th>
<th>instrumentation</th>
<th>sampling</th>
<th>profiling</th>
<th>tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>gprof</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>Extrae/Paraver</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>VTune</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>HPCToolkit</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>perf</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>Score-P (Vampir,Scalasca,TAU)</td>
<td>○</td>
<td></td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Call path profiles

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

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


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

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

```c
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 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

```c
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 steals 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

```
Worker Worker Worker
push() pop() steal() steal()
```
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())
- \( c_\infty \): 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
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
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
Two basic operations: 
create_task and wait_tasks

- At create_task, a new task is created

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

**parent stops by create_task**

**parent resumes**

**work-first:** the same worker switches to child task
Two basic operations: 
**create_task** and **wait_tasks**

- At **create_task**, a new task is created

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

A computation DAG trace 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

**parent stops by create_task**

**parent resumes**

**parent-first**: the same worker continues executing the parent
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
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

```plaintext
A () {
  S1;
  create_task( B );
  S2;
  wait_tasks;
  S3;
}
```
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
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.
Two basic operations: `create_task` and `waittasks`

- 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

```plaintext
A () {
    S1;
    t0;
    create_task( t1; B; t2; );
    t3;
    S2;
    t4;
    wait_tasks;
    t5;
    S3;
}
B
```
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
two generic primitives translate to equivalent ones in specific systems with measurement probes.
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;
    }
}
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;
    }
}

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 {t1; quicksort(A,a,m,threshold); t2;}
        t3;
        quicksort(A,m,b,threshold);
        t4;
        cilk_sync;
        t5;
    }
}
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;
    }
}
To OpenMP

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

```cpp
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();
    }
}
```
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;
    }
}

To TBB with DAG Recorder

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`

```plaintext
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

```c
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

```plaintext
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.

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

1. dag
   flattened to file

.flattened to file

.dag file
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.

```c
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.

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

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

+.dag file
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.

```c
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.

```plaintext
1 f(n) {
2   S1;
3   create_task(f(n-1));
4   f(n-2);
5   S2;
6   wait_tasks;
7   S3;
8 }
```

.dag file
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.

```c
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

```plaintext
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
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.
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 provides many kinds of visualizations of the DAG:

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

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

- Cilk, Qthreads, Pthreads

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;
}

[Wheeler and Thain 2010]
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 package)
Flow Graph Analyzer captures and visualizes task graph from program written with Flow Graph Interface of TBB 4.0.

- only TBB
- flat layout

An example program with Flow Graph Interface

```cpp
#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 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)

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)
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
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!