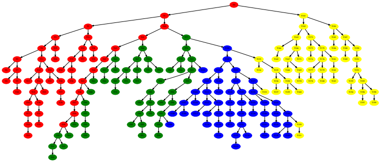
# A Quest for Unified, Global View Parallel Programming Models for Our Future

Kenjiro Taura

University of Tokyo

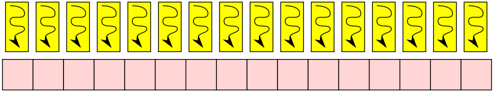


# Acknowledgements

- Jun Nakashima (MassiveThreads)
- Shigeki Akiyama, Wataru Endo (MassiveThreads/DM)
- An Huynh (DAGViz)
- Shintaro Iwasaki (Vectorization)

# What is task parallelism?

- like most CS terms, the definition is vague
- I don't consider contraposition *"data parallelism vs. task parallelism"* useful
    - imagine lots of tasks each working on a piece of data
    - is it data parallel or task parallel?

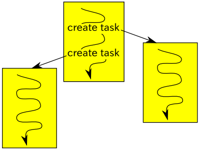


- let's instead ask:
    - what's useful from programmer's view point
    - what are useful distinctions to make from implementer's view point

# What is task parallelism?

A system supports *task parallelism* when:

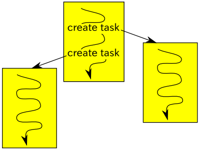
1. a logical unit of concurrency (that is, *a task*) can be created dynamically, at an arbitrary point of execution,



# What is task parallelism?

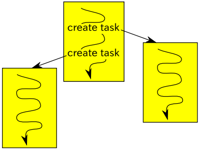
A system supports *task parallelism* when:

1. a logical unit of concurrency (that is, *a task*) can be created dynamically, at an arbitrary point of execution,
2. and cheaply;

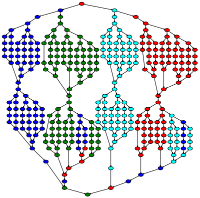


# What is task parallelism?

A system supports *task parallelism* when:

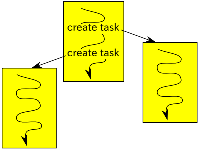


1. a logical unit of concurrency (that is, *a task*) can be created dynamically, at an arbitrary point of execution,
2. and cheaply;

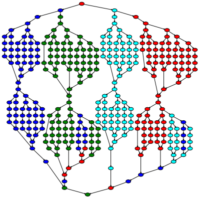


# What is task parallelism?

A system supports *task parallelism* when:

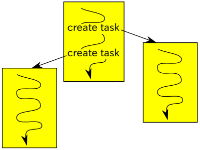


1. a logical unit of concurrency (that is, *a task*) can be created dynamically, at an arbitrary point of execution,
2. and cheaply;
3. and they are automatically mapped on hardware parallelism (cores, nodes, ...)

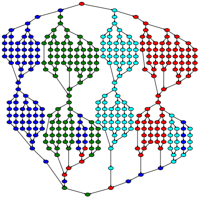


# What is task parallelism?

A system supports *task parallelism* when:



1. a logical unit of concurrency (that is, *a task*) can be created dynamically, at an arbitrary point of execution,
2. and cheaply;
3. and they are automatically mapped on hardware parallelism (cores, nodes, . . . )
4. and cheaply context-switched

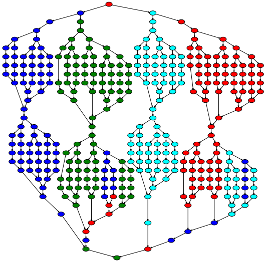


# What are they good for?

- generality: "creating tasks at arbitrary points" unifies many superficially different patterns
  - parallel nested loop, parallel recursions
  - they trivially compose
- programmability: cheap task creation + automatic load balancing allow straightforward, processor-oblivious decomposition of the work (*divide-and-conquer-until-trivial*)
- performance: dynamic scheduling is a basis for hiding latencies and tolerating noises

# Our goal

- programmers use tasks (+ higher-level syntax on top) as the unified means to express parallelism
- the system maps tasks to hardware parallelism
    - cores within a node
    - nodes
    - SIMD lanes within a core!



# Rest of the talk

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Agenda

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Taxonomy

- library or frontend: *implemented with ordinary C/C++ compilers or does it heavily rely on a tailored frontend?*

# Taxonomy

- library or frontend: *implemented with ordinary C/C++ compilers or does it heavily rely on a tailored frontend?*
- tasks suspendable or atomic: *can tasks suspend/resume in the middle or do tasks always run to completion?*

# Taxonomy

- library or frontend: *implemented with ordinary C/C++ compilers or does it heavily rely on a tailored frontend?*
- tasks suspendable or atomic: *can tasks suspend/resume in the middle or do tasks always run to completion?*
- synchronization patterns arbitrary or pre-defined: *can tasks synchronize in an arbitrary topology or only in pre-defined synchronization patterns (e.g., bag-of-tasks, fork/join)?*

# Taxonomy

- library or frontend: *implemented with ordinary C/C++ compilers or does it heavily rely on a tailored frontend?*
- tasks suspendable or atomic: *can tasks suspend/resume in the middle or do tasks always run to completion?*
- synchronization patterns arbitrary or pre-defined: *can tasks synchronize in an arbitrary topology or only in pre-defined synchronization patterns (e.g., bag-of-tasks, fork/join)?*
- tasks untied or tied: *can tasks migrate after they started?*

# Instantiations

| | library /frontend | suspendable task | untied tasks | sync topology |
|---|---|---|---|---|
| OpenMP tasks | frontend | yes | yes | fork/join |
| TBB | library | yes | no | fork/join |
| Cilk | frontend | yes | yes | fork/join |
| Quark | library | no | no | arbitrary |
| Nanos++ | library | yes | yes | arbitrary |
| Qthreads | library | yes | yes | arbitrary |
| Argobots | library | yes | yes? | arbitrary |
| MassiveThreads | library | yes | yes | arbitrary |

# MassiveThreads

- `https://github.com/massivethreads/massivethreads`
- design philosophy: user-level threads (ULT) in *an ordinary thread API as you know it*
  - *tid* `= myth_create(`*f*`,` *arg*`)`
  - *tid* `= myth_join(`*arg*`)`
  - `myth_yield` to switch among threads (useful for latency hiding)
  - mutex and condition variables to build arbitrary synchronization patterns
- efficient work stealing scheduler (locally LIFO and child-first; steal oldest task first)
- an (experimental) customizable work stealing [Nakashima and Taura; ROSS 2013]

# User-facing APIs on MassiveThreads

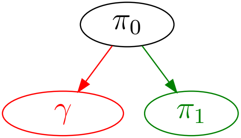
- TBB's task_group and parallel_for (but with untied work stealing scheduler)
- Chapel tasks on top of MassiveThreads (currently broken `orz`)
- SML# (Ueno @ Tohoku University) ongoing
- Tapas (Fukuda @ RIKEN), a domain specific language for particle simulation

```
quicksort(a, p, q) {
 if (q - p < th) {
   ...
 } else {
  mtbb::task_group tg;
  r = partition(a, p, q);
  tg.run([=]{ quicksort(a, p, r-1); });
  quicksort(a, r, q);
  tg.wait();
 }
}
```

TBB interface on MassiveThreads

# Important performance metrics

- low local creation/sync overhead
- low local context switches
- reasonably low load balancing (migration) overhead
- somewhat sequential scheduling order

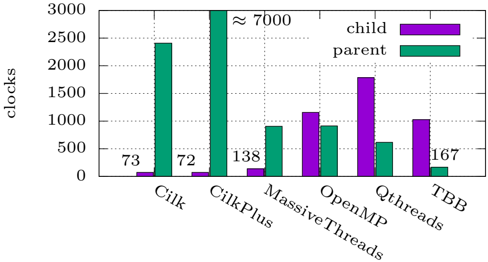


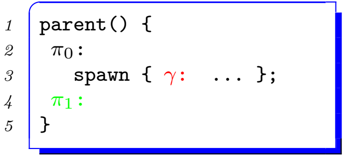
```
parent() {
 $\pi_0$:
   spawn { $\gamma$:  ... };
 $\pi_1$:
}
```

| op | measure what | time (cycles) |
|---|---|---|
| local create | $\pi_0 \to \gamma$ | $\approx$ 140 |
| work steal | $\pi_0 \to \pi_1$ | $\approx$ 900 |
| context switch | myth_yield | $\approx$ 80 |

(Haswell i7-4500U (1.80GHz), GCC 4.9)

# Comparison to other systems





```
parent() {
 π0:
   spawn { γ:  ... };
 π1:
}
```

Summary:

- Cilk(Plus), known for its superb local creation performance, sacrifices work stealing performance
- TBB's local creation overhead is equally good, but it is "parent-first" and tasks are tied to a worker once started

# Further research agenda (1)

- task runtimes for ever larger scale systems is vital

# Further research agenda (1)

- task runtimes for ever larger scale systems is vital
- ⇒ “locality-/cache-/hierarchy-/topology-/whatever-aware” schedulers obviously important

# Further research agenda (1)

- task runtimes for ever larger scale systems is vital
- ⇒ "locality-/cache-/hierarchy-/topology-/whatever-aware" schedulers obviously important
- ⇒ hierarchical/customizable schedulers proposals

# Further research agenda (1)

- task runtimes for ever larger scale systems is vital
- ⇒ “locality-/cache-/hierarchy-/topology-/whatever-aware” schedulers obviously important
- ⇒ hierarchical/customizable schedulers proposals
- ⇒ yet, IMO, there are no clear demonstrations that clearly outperform simple greedy work stealing over many workloads

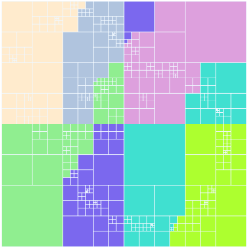
# Further research agenda (1)

- task runtimes for ever larger scale systems is vital
- ⇒ "locality-/cache-/hierarchy-/topology-/whatever-aware" schedulers obviously important
- ⇒ hierarchical/customizable schedulers proposals
- ⇒ yet, IMO, there are no clear demonstrations that clearly outperform simple greedy work stealing over many workloads
- the question, it seems, ultimately comes to this:

  *when no tasks exist near you but some may exist far from you, steal it or not (stay idle)?*

# Further research agenda (2)

- quantify the gap between hand-optimized decomposition vs. automatic decomposition (by work stealing); e.g.
    - Space-filling decomposition vs. work stealing
    - 2.5D matrix-multiply vs. work stealing
- both experimentally and theoretically



# Agenda

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Two facets of task parallelism in distributed memory settings

- a means to hide latency, for which we merely need a local user-level thread library supporting suspend/resume at arbitrary points
- a means to globally balance loads, for which we need a system specifically designed to migrate tasks across address spaces

MassiveThreads/DM is a system supporting

- distributed load balancing and latency hiding
- + global address space supporting migration and replication

# Tasks to hide latencies

The goal:

- individual tasks look like ordinary blocking access *(programmer-friendly)*
- hide latencies by creating lots of tasks

Ingredients for implementation:

- local tasking layer with *good context switch performance*
- message/RDMA layer with *good multithreaded performance*

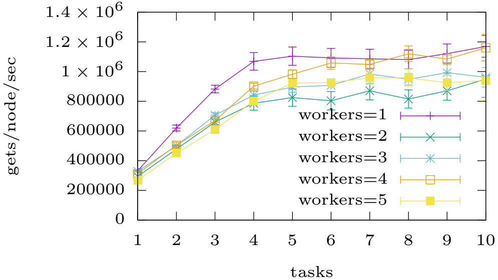
```
scan(global_array<T> a) {
  for (i = 0; i < n; i++) {
    .. = .. a[i] ..;
  }
}
```

```
scan(global_array<T> a) {
  pfor (i = 0; i < n; i++) {
    .. = .. a[i] ..;
  }
}
```

# Preliminary results

- context switch: we used MassiveThreads's `myth_yield` function to switch context upon blocking
- message/RDMA: we rolled our own thread-safe comm layer (on MPI, on IB verbs, and on Fujitsu Tofu RMA), partly because Fujitsu MPI lacks multithreading support

```
/* a[i] */
T get(address<T>) {
  issue non-blocking get(address);
  while (!the result available) {
    myth_yield();
  }
  return result;
}
```



# Taxonomy

- library or frontend
- tasks suspendable or atomic
- synchronization patterns arbitrary or pre-defined
- tasks untied or tied

- the main issue:
  *implementation complexity raises on*
  *distributed memory especially for untied tasks*

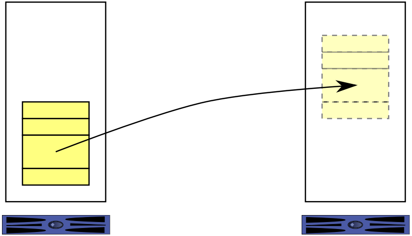
- that is, how to move tasks across address spaces?

# Instantiations

| | library /frontend | suspendable task | untied tasks | sync topology | scale |
|---|---|---|---|---|---|
| Distributed Cilk<br>[Blumofe et al. 96] | frontend | yes | yes | fork/join | 16 |
| Satin<br>[Neuwpoort et al. 01] | frontend | yes | no | fork/join | 256 |
| Tascell<br>[Hiraishi et al. 09] | frontend | yes | yes | fork/join | 128 |
| Scioto<br>[Dinan et al. 09] | library | no | no | BoT | 8192 |
| HotSLAW<br>[Min et al. 11] | library | yes | no | fork/join | 256 |
| X10/GLB<br>[Zhang et al. 13] | library | no | no | BoT | 16384 |
| Grappa<br>[Nelson et al. 15] | library | yes | no | fork/join | 4096 |
| MassiveThreads/DM<br>[Akiyama et al. 15] | library | yes | yes | fork/join | 4096 |

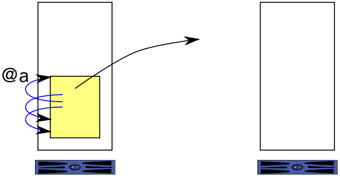
# MassiveThreads/DM

- *global (inter-node) work stealing* library
  - usable with ordinary C/C++ compilers
  - supports fork-join with untied tasks
- ⇒ *moves native threads* across nodes



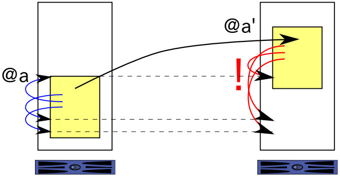
# Migrating native threads

- problem: the stack of native threads has pointers pointing to the inside
- migrating a thread to an arbitrary address breaks these pointers
- ⇒ upon migration, copy the stack to the same address (*iso-address* [Antoniu et al. 1999])



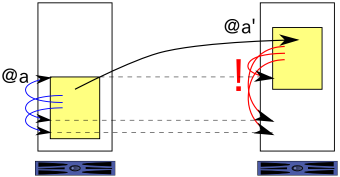
# Migrating native threads

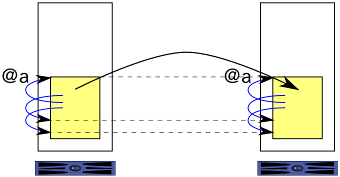
- problem: the stack of native threads has pointers pointing to the inside
- migrating a thread to an arbitrary address breaks these pointers
- ⇒ upon migration, copy the stack to the same address (*iso-address* [Antoniu et al. 1999])



# Migrating native threads

- problem: the stack of native threads has pointers pointing to the inside
- migrating a thread to an arbitrary address breaks these pointers
- ⇒ upon migration, copy the stack to the same address (*iso-address* [Antoniu et al. 1999])

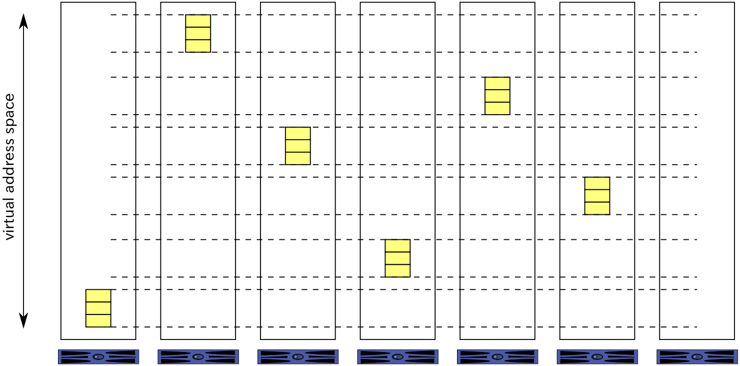




iso address

# Iso-address limits scalability

- for *each* thread, *all nodes* must reserve its address
- ⇒ a huge waste of *virtual* memory



# Is consuming a huge *virtual* memory really a problem?

- with high concurrency, it may indeed overflow *virtual* address space

$$\begin{array}{ccccccccc} \text{stack size} & \times & \text{tasks depth} & \times & \text{cores/node} & \times & \text{nodes} & & \\ 2^{14} & \times & 2^{13} & \times & 2^{8} & \times & 2^{13} & = & 2^{48} \end{array}$$

- more important, the luxury use of virtual memory *prohibits using RDMA for work stealing* (as RDMA memory must be pinned)
- ⇒ proposed UniAddress scheme [Akiyama et al. 2015]

# Further research agenda

- demonstrate global distributed load balancing with practical workloads with lots of shared data
- "locality-/hierarchy-. . . "awareness are even more important in this setting
- latency-hiding opportunity adds an extra dimension
- *steal or not, switch or not*

# Agenda

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

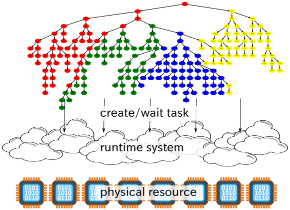
Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Analyzing task parallel programs

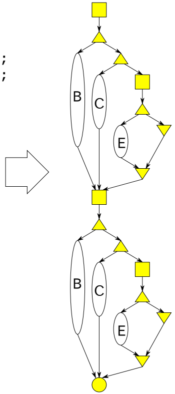
- task parallel systems are more "opaque" from users
    - task management, load balancing, scheduling
- they show performance differences and researchers want to precisely understand where they come from



# DAG Recorder and DAGViz

- DAG Recorder runs a task parallel program and extracts its DAG, augmented with timestamps, CPUs, etc.
- DAGViz is its visualizer

```
A() {
  for(i=0;i<2;i++) {
    mk_task_group;
    create_task(B());
    create_task(C());
    D();
    wait_tasks();
  }
}
D() {
  mk_task_group;
  create_task(E());
  F();
  wait_tasks();
}
```



■ begin_section

▲ create_task

▼ wait_tasks

● endtask

# Why record the DAG?

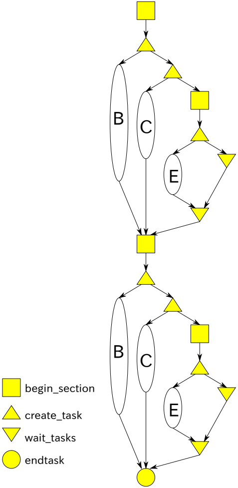
- DAG is a *logical* representation of the program execution independent from the runtime system
    - you can compare DAGs by two systems side by side
- DAG contains sufficient information to reconstruct many details
    - work and critical path (excluding overhead)
    - actual parallelism (running cores) along time
    - available parallelism (ready tasks) along time
    - how long each task was delayed by the scheduler

# DAGViz Demo

Seeing is believing.

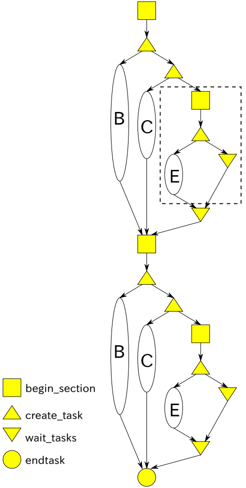
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
  1. its nodes are executed by a single worker,
  2. its span is smaller than a (configurable) threshold



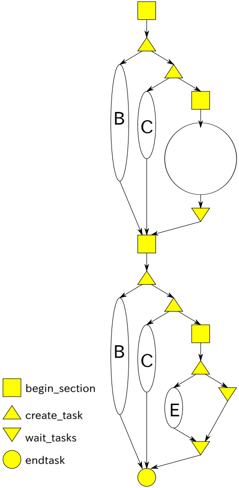
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\Longleftrightarrow$
    1. its nodes are executed by a single worker,
    2. its span is smaller than a (configurable) threshold



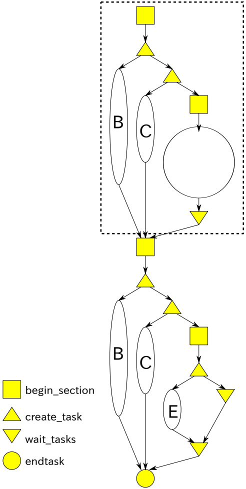
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
  1. its nodes are executed by a single worker,
  2. its span is smaller than a (configurable) threshold



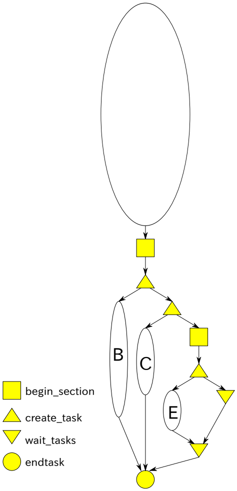
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
    1. its nodes are executed by a single worker,
    2. its span is smaller than a (configurable) threshold



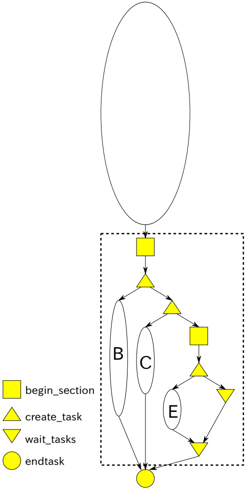
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
    1. its nodes are executed by a single worker,
    2. its span is smaller than a (configurable) threshold



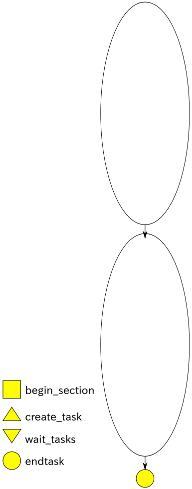
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
  1. its nodes are executed by a single worker,
  2. its span is smaller than a (configurable) threshold



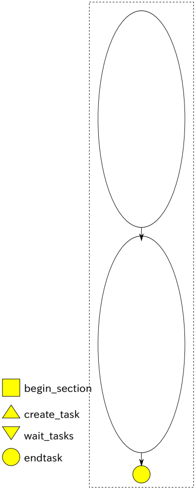
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
  1. its nodes are executed by a single worker,
  2. its span is smaller than a (configurable) threshold



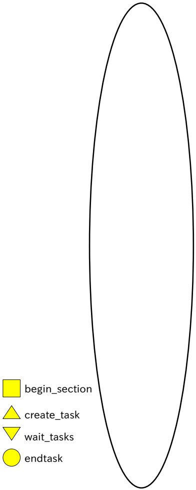
# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
  1. its nodes are executed by a single worker,
  2. its span is smaller than a (configurable) threshold



# Challenge : reducing space requirement

- literally recording all subgraphs is prohibitive
- *collapse "uninteresting" subgraphs into single nodes*
- current criteria: we collapse a subgraph $\iff$
    1. its nodes are executed by a single worker,
    2. its span is smaller than a (configurable) threshold



# Ongoing work

- hoping to use this tool to automate discovery of issues in runtime systems
    - scheduler delays along a critical path
    - work time inflation
- shed light on “steal or not” trade-offs

# Agenda

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Motivation

- task parallelism is a friend of divide-and-conquer algorithms
- divide-and-conquer makes coding “trivial,” by dividing until the problem becomes trivial
    - matrix multiply, matrix factorization, triangular solve, FFT, sorting, . . .
- in reality, the programmer has to *optimize leaves manually*
- why? because we lack good compilers

# The power of divide-and-conquer

```
/* quick sort */
quicksort(a, p, q) {
 if ($q - p < 2$) {
   return;
 } else {
   ...
 }
}
```

```
/* $C$ += $AB$ */
mm($A$, $B$, $C$) {
  if ($|A| = 1$ && $|B| = 1$
            && $|C| = 1$) {
    $C_{00}$ += $A_{00} \cdot B_{00}$;
  } else {
    ...
  }
}
```

```
/* Cholesky factorization */
chol($A$) {
  if ($n = 1$) {
    return ($\sqrt{a_{11}}$);
  } else {
    ...
  }
}
```

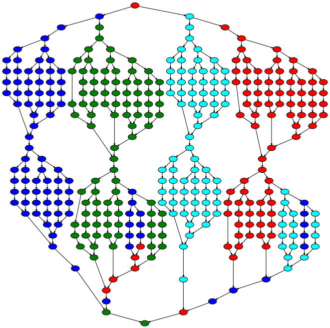
```
/* FFT */
fft(n, x) {
  if ($n = 1$) {
    return $x_0$;
  } else {
    ...
  }
}
```

```
/* triangular solve
   $LX = B$. */
trsm($L, B$) {
  if ($M = 1$) {
    $B$ /= $l_{11}$;
  } else {
    ...
  }
}
```

They all admit "trivial" base case, only if performance is acceptable . . .

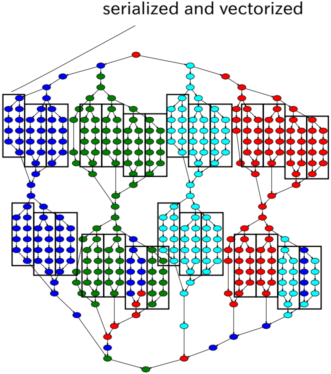
# Static optimizations and vectorization of tasks

- goal: run straightforward task-based programs as fast as manually optimized programs
- write once, parallelize everywhere (nodes, cores, *and vectors*)



# Static optimizations and vectorization of tasks

- goal: run straightforward task-based programs as fast as manually optimized programs
- write once, parallelize everywhere (nodes, cores, *and vectors*)



# What does our compiler do?

1. static cut-off statically eliminates task creations
2. code-bloat-free inlining inline-expands recursions
3. loopification transforms recursions into flat loops (and then vectorizes it if possible)

# Static cut-off

```
$f(a, b, \cdots)$ {
  if ($E$) {
    $L(a, b, \cdots)$
  } else {
    $\cdots$
    spawn $f(a_1, b_1, \cdots)$;
    $\cdots$
    spawn $f(a_2, b_2, \cdots)$;
    $\cdots$
  }
}
```

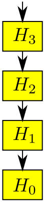
$\Rightarrow$

```
$f_{\mathtt{seq}}(a, b, \cdots)$ {
  if ($E$) {
    $L(a, b, \cdots)$
  } else {
    $\cdots$
    $f_{\mathtt{seq}}(a_1, b_1, \cdots)$;
    $\cdots$
    $f_{\mathtt{seq}}(a_2, b_2, \cdots)$;
    $\cdots$
  }
}
```

key: determine a condition $H_k$, in which the height of recursion from leaves $\leq k$

- $H_0 = E$
- $H_{k+1} = E$ or $\forall i(a_i, b_i, \cdots)$ satisfy $H_k$

when succeeded, generate code that *statically eliminate all task creations*



# Code-bloat-free inlining

- under condition $H_k$, inline-expanding all recursions $k$ times would eliminate all function calls
- but this would result in an exponential code bloat when the function has multiple recursive calls
- code-bloat-free inlining fuses multiple recursive calls into a single call site

```
        $\cdots$
        $f(a_1, b_1, \cdots);$
        $\cdots$
        $f(a_2, b_2, \cdots);$
        $\cdots$
```

$\Rightarrow$

```
for (i = 0; i < 2; i++) {
  switch (i) {
   case 0: $\cdots$
   case 1: $\cdots$
  }
  $f(a_i, b_i, \cdots);$
}
```

# Loopification

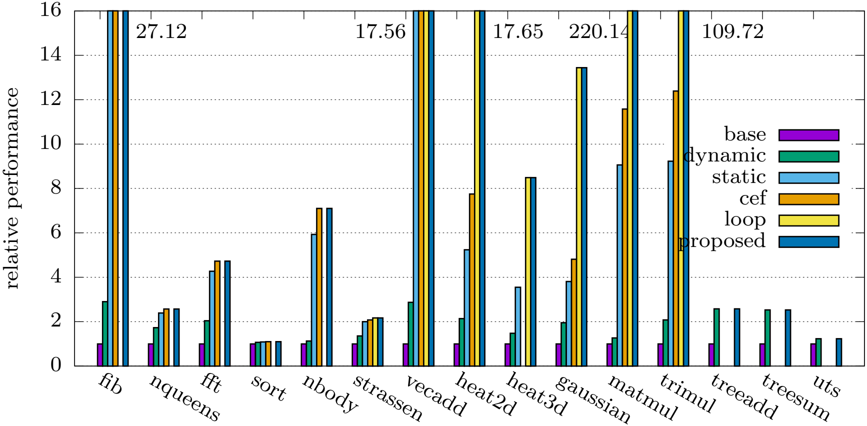
```
$f_{\texttt{seq}}(a, b, \cdots)$ {
  if ($E$) {
    $L(a, b, \cdots)$
  } else {
    $\cdots$
    $f_{\texttt{seq}}(a_1, b_1, \cdots)$;
    $\cdots$
    $f_{\texttt{seq}}(a_2, b_2, \cdots)$;
    $\cdots$
  }
}
```

$\Rightarrow$

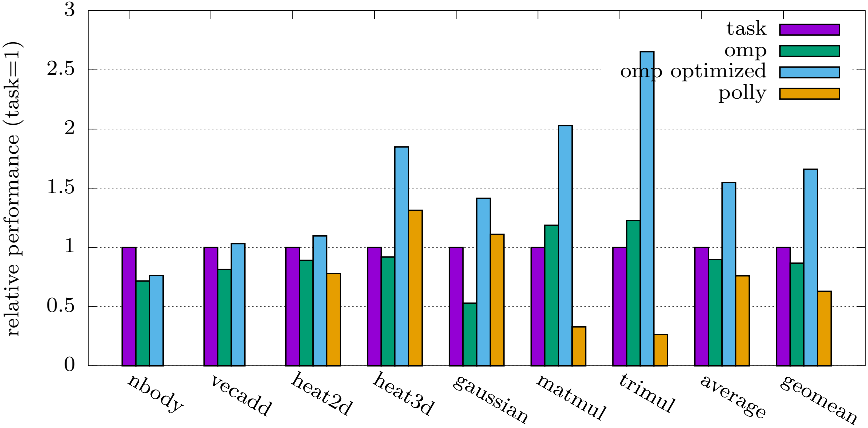
```
for $i \in P$ {
  $L(x_i, y_i, \cdots)$
}
```

- instead of code-bloat-free inlining, loopification attempts to generate a flat (or shallow) loop directly from recursive code
- it tries to synthesize hypotheses that the original code is an affine loop of leaf blocks
- the loopified code may then be vectorized

# Results: effect of optimizations



# Results: remaining gap to hand-optimized code



# Agenda

Intra-node Task Parallelism

Task Parallelism in Distributed Memory

Need Good Performance Analysis Tools

Compiler Optimizations and Vectorization

Concluding Remarks

# Future outlook of task parallelism

- the goal: offer both programmability and performance
- long way toward achieving acceptable performance on distributed memory machines. why?
  - dynamic load balancing → random traffic
  - global address space → fine-grain communication
- OK in shared memory today. why not on distributed memory (at least for now)?
  - checking errors and completion everywhere
  - doing mutual exclusion everywhere
  - no hardware-prefetching analog
  - or lack of bandwidth to tolerate random traffic and aggressive prefetching

*Thank you for listening*