Higher-level implicit parallelism with PASL

Umut Acar
Carnegie Mellon University

Arthur Charguéraud
INRIA

Mike Rainey
INRIA

LaME

01.06.2013
What is PASL?

• PASL is our Parallel Algorithm Scheduling Library.

• It’s a test bed for new ideas relating to implicit parallelism.

• It’s written in C++. 
How do we raise the level of abstraction?

Generalize the implicit-threading model

Primitives for creating and scheduling parallel computations

Address performance

Granularity control by Oracle Scheduling

Dynamic load balancing by work stealing with private deques
Primitives for creating and scheduling parallel computations
The implicit-parallelism zoo

- spawn/sync
- futures
- parallel loops
- TBB flow graphs
- reducers / hyperobjects
- map-reduce
- clocks / phasers
- concurrent revisions
- etc.
Computation DAGs

work

{ span 

Thursday, July 4, 13
Computation DAGs at runtime

- ready
- already executed
- suspended
- not yet created

Thursday, July 4, 13
Almost-complete programming interface

node* create_node(closure*)

void add_node(node*)

void add_edge(node*, node*)
Edge capture

\[ n \text{ calling } a \]
(with continuation \(k\))

\[
\begin{align*}
\text{transfer\_outedges\_to}(a)
\end{align*}
\]

`void transfer_outedges_to(node*)`
Encoding binary fork join

fork_join

as last
instruction

void fork_join(closure* a, closure* b, closure* j)
node* na = create_node(a)
node* nb = create_node(b)
node* nj = create_node(j)
transfer_outedges_to(nj)
add_edge(na, nj)
add_edge(nb, nj)
add_node(na)
add_node(nb)
add_node(nj)
Encoding graph traversal using a big join

big join

processed node

a.k.a. async/finish parallelism
Encoding futures

future (producer)

\[ \Rightarrow \]

consumer demanding the future be forced

\[ \Rightarrow \]

future executed

\[ \Rightarrow \]

become ready
Four key ingredients for efficiency

1. Granularity control
2. Dynamic load balancing (work stealing)
3. Number of incoming edges (a.k.a. join counter)
4. Continuation (list of edges)

join counter = 8
Small-arity joins with atomic counters

join counter = 2  join counter = 1  join counter = 0

fetch_and_add(-1)
Big-arity joins

- use one counter per processor:
- \( \# \) edges added - \( \# \) edges removed
- periodic check by one particular processor to see if the sum is zero

owner = core #4
counters = [23; -9; 97; 67; 20]
Representation of nodes and edges

• We use an `instrategy` for representing the number of incoming edges

• and an `outstrategy` for representing the list of outgoing edges

```c
node* create_node(closure*, instrategy*, outstrategy*)
```
Summary

Dynamic DAGs, with per-node specification of edge representation:

```c
node* create_node(closure*, instrategy*, outstrategy*)
void add_node(node*)
void add_edge(node*, node*)
void transfer_edges_to(node*)
```

Find other examples of custom instrategies in paper, e.g.,

- distributed
- owner based
- optimistic
Automatic granularity control by Oracle Scheduling
Do we need to tame DAG-related overheads?

• Yes:
  • Parallel fib in PASL is typically 100x slower than sequential fib.
  • Parallel fib in PASL is no more than a few percent slower.
  • It’s not that bad, because we can ensure the costs are well amortized by granularity control.
Taming DAG overheads

fat sequentialized leaves
Oracle scheduling

Idea: 1. Pick a target leaf run time $t$.

```c
void quicksort(int A[], int s , int e) {
    if (e - s < 2)
        return;
    int p = partition(A, s , e);
    fork_join {
        quicksort (A, s , p );
        quicksort (A, p + 1, e );
    }
}
```

2. Make calls:
- parallel, if combined run time prediction $> t$
- sequential, otherwise
Our theoretical contribution

• Suppose we have an oracle predicting run times with error always less than certain ratio.

• Then, the total cost of creating nodes is well amortized.

• See paper for precise formal bound.
void quicksort(int A[], int s, int e) {
    cost {
        int n = e - s;
        return n * log(n)
    }
    if (e - s < 2)
        return;
    int p = partition(A, s, e);
    fork_join {
        quicksort(A, s, p);
        quicksort(A, p + 1, e);
    }
}
Runtime profiling

Let $n$ be asymptotic complexity of a call.

Let $r = e / n$.

We use running average of past few measurements of $r$ to make predictions.
Summary

• A few issues:
  • Outlier measurements increase error.
  • Our approach assumes that average case complexity matches worst case.
  • Our approach works well for a wide range of computations.
  • Please see our paper for performance study.
Dynamic load balancing by work stealing with private deques
Scheduling parallel tasks
Scheduling parallel tasks

set of cores
Scheduling parallel tasks

pool of tasks
Scheduling parallel tasks

- Goal: dynamic load balancing
- A centralized approach: does not scale up
- Popular approach: work stealing
- Our work: study implementations of work stealing
Work stealing
Work stealing

deque

28
Work stealing
Work stealing

pop \uparrow \downarrow \text{push}

pop \uparrow \downarrow \text{push}

pop \uparrow \downarrow \text{push}

pop \uparrow \downarrow \text{push}
Work stealing
Work stealing
Work stealing
Concurrent deques

- Deques are shared.
- Two sources of race:
  - between thieves
  - between owner and thief
- Chase-Lev data structure resolves these races using atomic compare&swap and memory fences.
Concurrent deques

• **Well studied:** shown to perform well both in theory and in practice ...

however, researchers identified two main limitations

• **Runtime overhead:** In a relaxed memory model, \texttt{pop} must use a memory fence.

• **Lack of flexibility:** Simple extensions (e.g., steal half) involve major challenges.
Previous studies of private deques

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feeley</td>
<td>1992</td>
<td>Multilisp</td>
</tr>
<tr>
<td>Hendler &amp; Shavit</td>
<td>2002</td>
<td>C</td>
</tr>
<tr>
<td>Umatani</td>
<td>2003</td>
<td>Java</td>
</tr>
<tr>
<td>Hirashi et al.</td>
<td>2009</td>
<td>C</td>
</tr>
<tr>
<td>Sanchez et al.</td>
<td>2010</td>
<td>C</td>
</tr>
<tr>
<td>Fluet et al.</td>
<td>2011</td>
<td>Parallel ML</td>
</tr>
</tbody>
</table>
Private deques

- Each core has exclusive access to its own deque.
- An idle core obtains a task by making a steal request.
- A busy core regularly checks for incoming requests.
Private deques

Addresses the main limitations of concurrent deques:

• no need for memory fence
• flexible deques (any data structure can be used)

but

• new cost associated with regular polling
• additional delay associated with steals
Unknowns of private deques

• What is the best way to implement work stealing with private deques?

• How does it compare on state of art benchmarks with concurrent deques?

• Can establish tight bounds on the runtime?
Unkn0wns of private deques

• What is the best way to implement work stealing with private deques?

We give a receiver- and a sender-initiated algorithm.

• How does it compare on state of art benchmarks with concurrent deques?

We evaluate on a collection of benchmarks.

• Can establish tight bounds on the runtime?

We prove a theorem w.r.t. delay and polling overhead.
Receiver initiated

1

-1

2

-1

3

-1

4

-1
Receiver initiated

-1 → CAS

1 → 2 → 3 → 4
Receiver initiated

2

-1

-1

-1

CAS

1

2

3

4
Receiver initiated
Receiver initiated
Receiver initiated
Receiver initiated
From receiver to sender initiated

- Receiver initiated: each idle core targets one busy core at random
- Sender initiated: each busy core targets one core at random
- Sender initiated idea is adapted from distributed computing.
- Sender initiated is simpler to implement.
Sender initiated
Sender initiated

... 0 ...

1 2 3 4

Thursday, July 4, 13
Sender initiated

1

3

CAS

2

3

4
Sender initiated

... → CAS

1 2 3 4
Sender initiated
Sender initiated

...       ...       ...       ...

1          2          3          4
Sender initiated
Analytical model

\[ P \quad \text{number of cores} \]
\[ T_I \quad \text{serial run time} \]
\[ T_\infty \quad \text{minimal run time with infinite cores} \]
\[ T_P \quad \text{parallel run time with } P \text{ cores} \]
\[ \delta \quad \text{polling interval} \]
\[ F \quad \text{maximal number of forks in a path} \]
Our main analytical result

Bound for greedy schedulers:

\[ T_P \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty \]

Bound for concurrent deques (ignoring cost of fences):

\[ \mathbb{E}[T_P] \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(F) \]

Bound for our two algorithms:

\[ \mathbb{E}[T_P] \leq \left( \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(\delta F) \right) \cdot \left( 1 + \frac{O(1)}{\delta} \right) \]
Our main analytical result

Bound for greedy schedulers:

\[ T_P \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty \]

Bound for concurrent deques (ignoring cost of fences):

\[ \mathbb{E}[T_P] \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(F) \]

Bound for our two algorithms:

\[ \mathbb{E}[T_P] \leq \left( \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(\delta F) \right) \cdot \left( 1 + \frac{O(1)}{\delta} \right) \]
Our main analytical result

Bound for greedy schedulers:

\[ T_P \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty \]

Bound for concurrent deques (ignoring cost of fences):

\[ \mathbb{E} [T_P] \leq \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(F) \]

Bound for our two algorithms:

\[ \mathbb{E} [T_P] \leq \left( \frac{T_1}{P} + \frac{P-1}{P} T_\infty + O(\delta F) \right) \cdot \left( 1 + \frac{O(1)}{\delta} \right) \]

\[ \text{cost of steals} \]

\[ \text{cost of steals polling overhead} \]
Performance study

- We implemented in PASL:
  - our receiver-initiated algorithm
  - our sender-initiated algorithm
  - our Chase-Lev implementation
- We compare all of those implementations against Cilk Plus.
Benchmarks

• Classic Cilk benchmarks and Problem Based Benchmark Suite (Blelloch et al 2012)

• Problem areas: merge sort, sample sort, maximal independent set, maximal matching, convex hull, fibonacci, and dense matrix multiply.
Performance results

Intel Xeon, 30 cores
polling period = 30µsec

Shared deques
Recv. − init.
Sender − init.

Cilk Plus
normalized execution time

Normalized run time

matmul
cilksort(exptintseq)
cilksort(randintseq)
fib
matching(egrid2d)
matching(eglrg)
matching(egrmat)
MIS(grid2d)
MIS(rlg)
MIS(rmat)
hull(plummer2d)
hull(uniform2d)
Summary

• We presented two new private-deque algorithms, evaluated them, and proved analytical results.

• In the paper, we demonstrated the flexibility of private deques by implementing the steal half policy.
Our papers

• **Efficient primitives for creating and scheduling parallel computations**
  By U. Acar, A. Charguéraud, and Mike Rainey
  DAMP’12

• **Oracle scheduling: controlling granularity in implicitly parallel languages**
  By U. Acar, A. Charguéraud, and Mike Rainey
  OOPSLA’11

• **Scheduling parallel programs by work stealing with private deques**
  By U. Acar, A. Charguéraud, and Mike Rainey
  PPoPP’13