Functional Parallelism in SAC

Aram Visser

Informatics Institute
University of Amsterdam

May 27th 2010 / SAC DevCon 7
Outline

1. Introduction
   - Why Functional Parallelism
   - Techniques

2. Implementation
   - Detail
   - Phases

3. Future
Outline

1. Introduction
   - Why Functional Parallelism
   - Techniques

2. Implementation
   - Detail
   - Phases

3. Future
Currently

- Parallelism in SAC
  - The With Loop
  - …?
Currently

- With loop highly optimized for multiple threads
- Outside with loop, no parallelism
- SAC is purely functional language, lot of potential parallelism
Functional Parallelism

- Function Parallelism used besides with loop
- Spawn functions in parallel
- Useful for:
  - Long running computations
  - Divide and Conquer Algorithms
- Three area’s to focus on
  - Scheduler
  - Efficient execution of tasks
  - Interaction between functional and data parallelism
Outline

1. Introduction
   - Why Functional Parallelism
   - Techniques

2. Implementation
   - Detail
   - Phases

3. Future
Start with $N$ workers
Every worker has own task queue
New tasks are added to own queue
Execute newest task from queue
Steal from other queues when own is empty
Double-ended Queue (deque)

- Tasks added and removed at head of queue by worker itself
- Tasks stolen at tail by other workers
- Worker doesn’t lock queue when size larger then 1
\[ a = \text{spawn } a(); \]
\[ \text{...} \]
\[ \text{sync}(a); \]
\[ \text{...} \]
Wool

- Create more workers than cores
- Block all but N workers
- On sync and value not available
  - Stall current worker
  - Wake up new worker
  - Resume when value available
- When work is done, look at number of workers active. More than N? Block current worker
Leapfrogging

- On sync, value not available
- Find out who you are waiting for
- Steal work from worker doing your work
- Will make sure your work is done soon
- Sync becomes function, nested syncs

```java
sync(Task t) {
    while (t.running) {
        steal from t.worker
        execute
    }
    return t.value
}
```
Resuming

- Allows function to resume halfway
- Store a entry counter in task frame
- If sync not completed
  - Save live variables in frame
  - Set entry counter to current location and exit
- At start of function, look at entry counter
  - If zero, start like normal
  - If not zero, goto label matching entry
Cilk

- Idea that spawning is expensive and stealing not common case
- Immediately execute spawned function, saving current state
- Two versions of same code, fast and slow clone
- Fast clone is for sequential code
  - Before spawn, save location in frame
  - After spawn, check to see if frame stolen
- Slow clone allows for resuming of functions
  - Is executed when frame is stolen
Continuations

- Split function up into multiple functions
- Sync point is another function
Outline

1. Introduction
   - Why Functional Parallelism
   - Techniques

2. Implementation
   - Detail
   - Phases

3. Future
Specifics

- Like Cilk, execute spawn first
- Sync on individual spawns
- Does not support loops
  - Need parallelism in loop? Use *with* loop!
- Either resuming or continuations, not decided yet
  - Assume resuming for now
- Reuse existing worker code
Task Frame

A frame tracks all data about a task
- State
- Entry counter
- Result
- Space for all live variables
- Waiting and ready flag for every spawn

```c
void a() {
    setup frame
    ...
    queue frame
    b();
    check frame
    ...
    save result
    return
}
```
Sync point

- On sync point, value not ready
  - Set waiting flag to true in frame
  - Do other work

- When spawn completes
  - Set ready flag to true in frame
  - If frame stolen, look for waiting flag
  - If waiting, resume beyond that sync
Outline

1. Introduction
   - Why Functional Parallelism
   - Techniques

2. Implementation
   - Detail
   - Phases

3. Future
Phases

- Parsing input file
- Category: Functional Parallelism
  - Find sync for spawn
  - (Optimizations)
  - Analyze live variables
- Code Generation
Adding Sync

Keep code in SSA

```plaintext
a = a();  // with spawnable flag
...
a' = sync(a);
```
Optimizations

- Move spawn up and syncs down in code
- Spawn followed by its sync is useless
- Before adding sync, introducing spawn?
  
  ...
Open Questions

- Resuming or continuations?
- Data parallelism
  - Affinity Scheduling
Open Questions

- Resuming or continuations?
- Data parallelism
  - Affinity Scheduling
Dreams

- Automatic parallelization
  - Use Tim’s work?
- Spawns become future types
  - Only evaluated if really needed or worker has time

```c
a = spawn a();
b = spawn b();
x = choose(a, b);
```
Dreams

- Automatic parallelization
  - Use Tim’s work?
- Spawns become future types
  - Only evaluated if really needed or worker has time

```latex
a = spawn a();
b = spawn b();
x = choose(a, b);
```
Questions?