Work-stealing with Configurable Scheduling Strategies

Martin Wimmer [1] Daniel Cederman [2] Jesper Larsson Träff [1] Philippas Tsigas [2]

[1] Vienna University of Technology, Institute of Information Systems
[2] Chalmers University of Technology, Computer Science and Engineering

Distributed Computing and Systems
Chalmers university of technology
CHALMERS

Parallel Computing
TU WIEN
FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Scheduling Strategies

We propose *scheduling strategies*, as a way to inform a scheduling system about properties of individual tasks.

```
class SsspStrategy : public BaseStrategy {
public:
  SsspStrategy(int distance, int& node_distance)
  : distance(distance), node_distance(node_distance) {}
  inline bool prioritize (SsspStrategy& other) const {
    return distance < other.distance;
  }
  inline bool dead_task() {
    return node_distance < distance;
  }
private:
  int distance;
  int& node_distance;
};
```

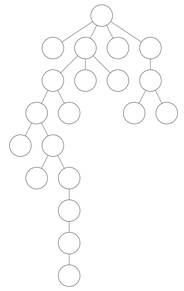
Scheduling strategies can use application-specific information to influence scheduler decisions. The scheduling system stays generic.

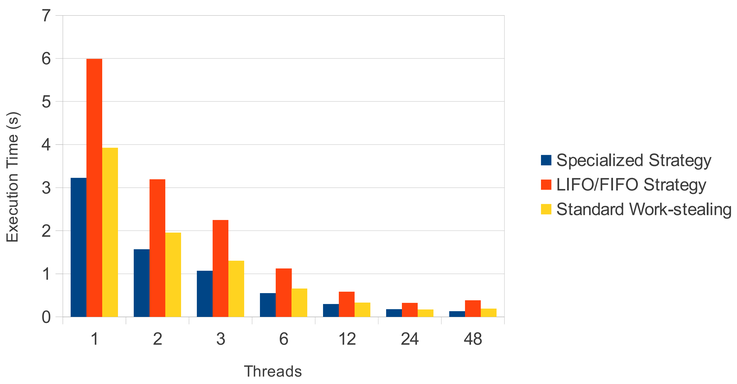
Spawn-to-Call Conversion

With small task granularity the overhead of spawning tasks has a major impact on performance, but only a small fraction of spawned tasks is stolen by other threads. Strategies allow to convert task spawns to function calls at run-time. The decision is made depending on task granularity, available parallelism and number of tasks in the queues.

Example: Unbalanced Tree Search

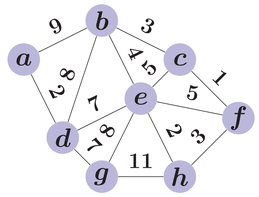
Benchmark to evaluate programmability and performance for applications requiring dynamic load balancing.
A task is created for each node. Tasks are extremely lightweight, so the scheduling overhead is significant.
Spawn-to-call conversion allows to reduce this overhead.





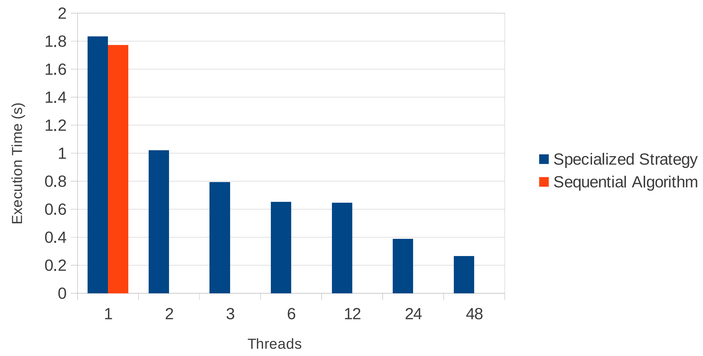
Prioritization of Tasks

Using a simple comparison operator for two strategies of the same type, a priority ordering between tasks is established.



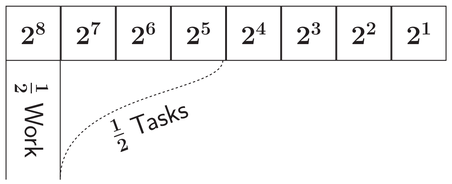
Example: Single-Source Shortest Path

Edges are greedily explored by multiple threads. A task is created for each node when the distance label for this node is updated. With prioritization due to strategies, the scheduling system can take the role of the priority data-structure.



Steal Half the Work, not Half the Tasks

Stealing half the work typically leads to better results in work-stealing. The amount of work per task is not known in classical work-stealing, it is approximated by stealing half the tasks. Strategies allow for a more accurate stealing behaviour by providing the scheduling system with a rough estimate of task granularity.



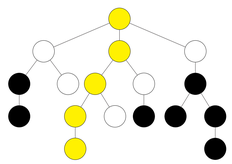
Example:

In divide-and-conquer algorithms, stealing a single task may already yield half the work.

Elimination of Dead Tasks

Strategies allow the scheduling system to recognize *dead tasks*, which are tasks that have become obsolete.

Example: Cutting off branches in branch-and-bound algorithms.



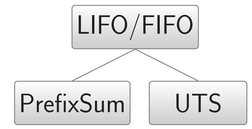
Locality

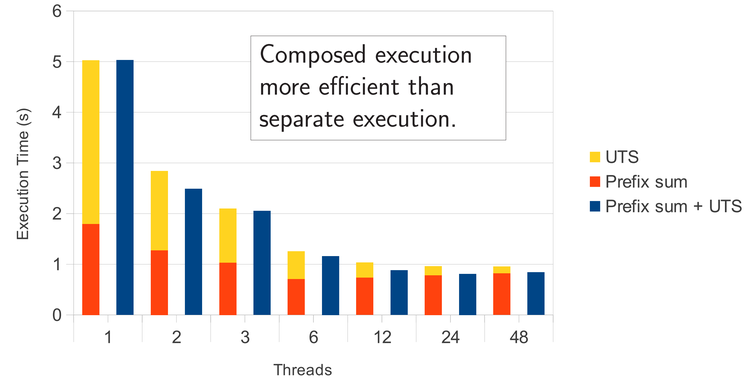
Together with the notion of a *place*, which denotes an execution unit together with its supporting data-structures, the prioritization mechanism can be used to implement per task locality optimizations.

```
bool LocalityStrategy :: prioritize ( LocalityStrategy & other) {
  Place* p = Pheet::get_place ();
  int distance = p->get_distance(last_owner);
  int distance_other = p->get_distance(other.last_owner);
  return distance > distance_other;
}
```

Composability

Composability of algorithms is achieved by organizing strategies in a hierarchy. Tasks with different strategies are composed by the strategy of their common ancestor.





Other Benchmarks

- **Graph Bipartitioning** Promising branches are explored first due to prioritization. Tasks with high uncertainty are preferably stolen. When a new bound is found, some tasks are marked as dead. Spawn-to-call conversion is used on high load.
- **Prefix Sum** Parallel algorithm performs a factor two more operations than sequential. It adapts towards sequential performance when little parallelism is available.
- **Triangle Strip Generation** Improvement on performance and result quality due to prioritization.
- **Quicksort** Improvements mainly due to spawn-to-call conversion.

Pheet

Scheduling strategies and the corresponding schedulers will be released as part of the *Pheet* task-scheduling framework. (`www.pheet.org`)
Public release planned Summer/Fall 2013 under the Boost Software License.