Towards Automatic Parallelization of Object-Oriented Programs

Welf Löwe   Jonas Lundberg   Erik Österlund

Linnæus University

Motivation

- Computational power is nowadays increased by increasing the number of cores per processor.
  - Sequential programs cannot ride the wave of increased clock rates.
  - Parallel programs needed to use multi-core processors effectively.
- Parallel programming is not economic in all software applications:
  - Legacy programs
  - Software technologies developed for sequential computing:
    - object-, aspect- and component-oriented programming
    - predefined reusable frameworks, platforms, libraries.
- Parallelizing sequential programs to exploit parallel hardware.

Automatic Parallelization

Our approach suggests three steps:

1. Analysis of independent parts in sequential programs applying static and dynamic dependence analysis.
2. Aggressive parallelization transformation of these independent parts. Add automatically parallelized components to the original sequential component variants.
3. Context-aware composition composes sequential and parallel components dynamically, depending on the execution context.

Basic idea similar to autotuning

- Generate parallel variants that might be efficient in certain contexts (problem size, number of processors available)
- Assess properties of an execution context (problem size, # of actually available processors) dynamically and select dynamically the champion variant for that context
- Use machine learning to find the champion for each context
  - Offline in a training phase or
  - Online by not always selecting the assumed champion and possibly find a better variant
- Parallelization only needs to generate correct parallel program components. Context-aware composition puts them together to an efficient program.

Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

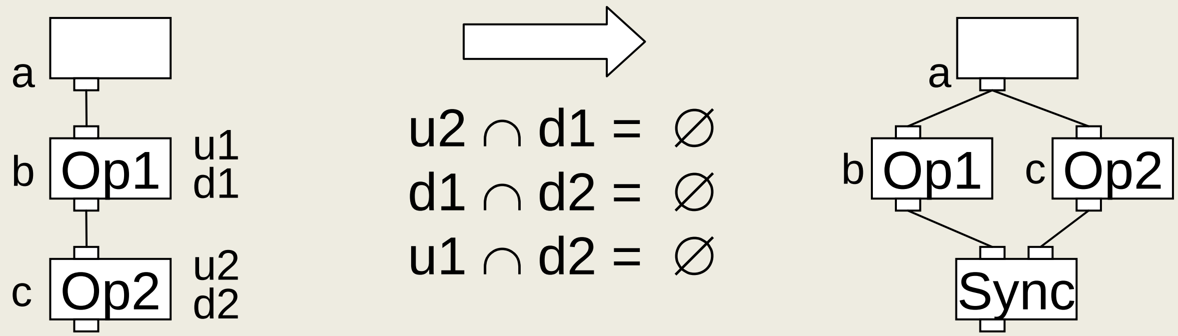
1. Analysis of independent parts

Static analysis

- Program representation: Memory Static Single Assignment form
  - Non-essential dependencies over local variables are removed
  - Explicit memory dependencies (read-write, write-read, write-write) over heap operations

- Well-established analyses
  - Points-to analysis approximates addresses of heap objects
  - Side-effect and heap analyses for task-level parallelism
  - Loop-index analysis identifies loops
  - Loop-dependence analysis for loop parallelism

Elimination of non-essential dependencies



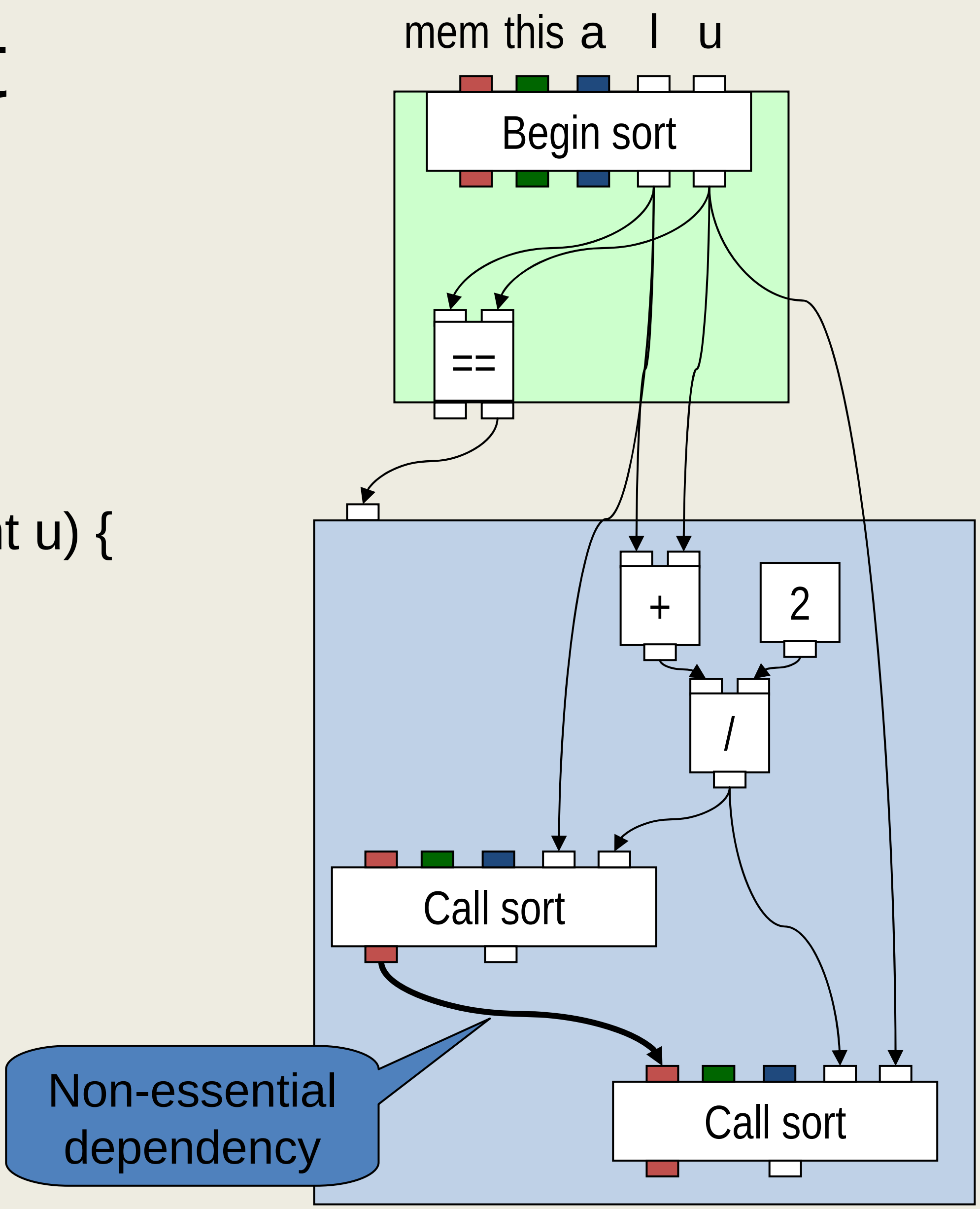
Heap dependency analysis:

- Op1, Op2 memory operations (store, call)
- u1, u2, d1, d2 designate may use/define sets
- They are computed in Points-to Analysis

Example Mergesort

```
E [] a = new E [length];
a.init();
sort(a, 0, length);

public boolean sort(E[] a, int l, int u) {
        // base case
        if (l == u) return true;
        // split
        int q = (l + u) / 2;
        // recursive calls
        sort(a, l, q);
        sort(a, q, u);
        // merge
        …
}
```

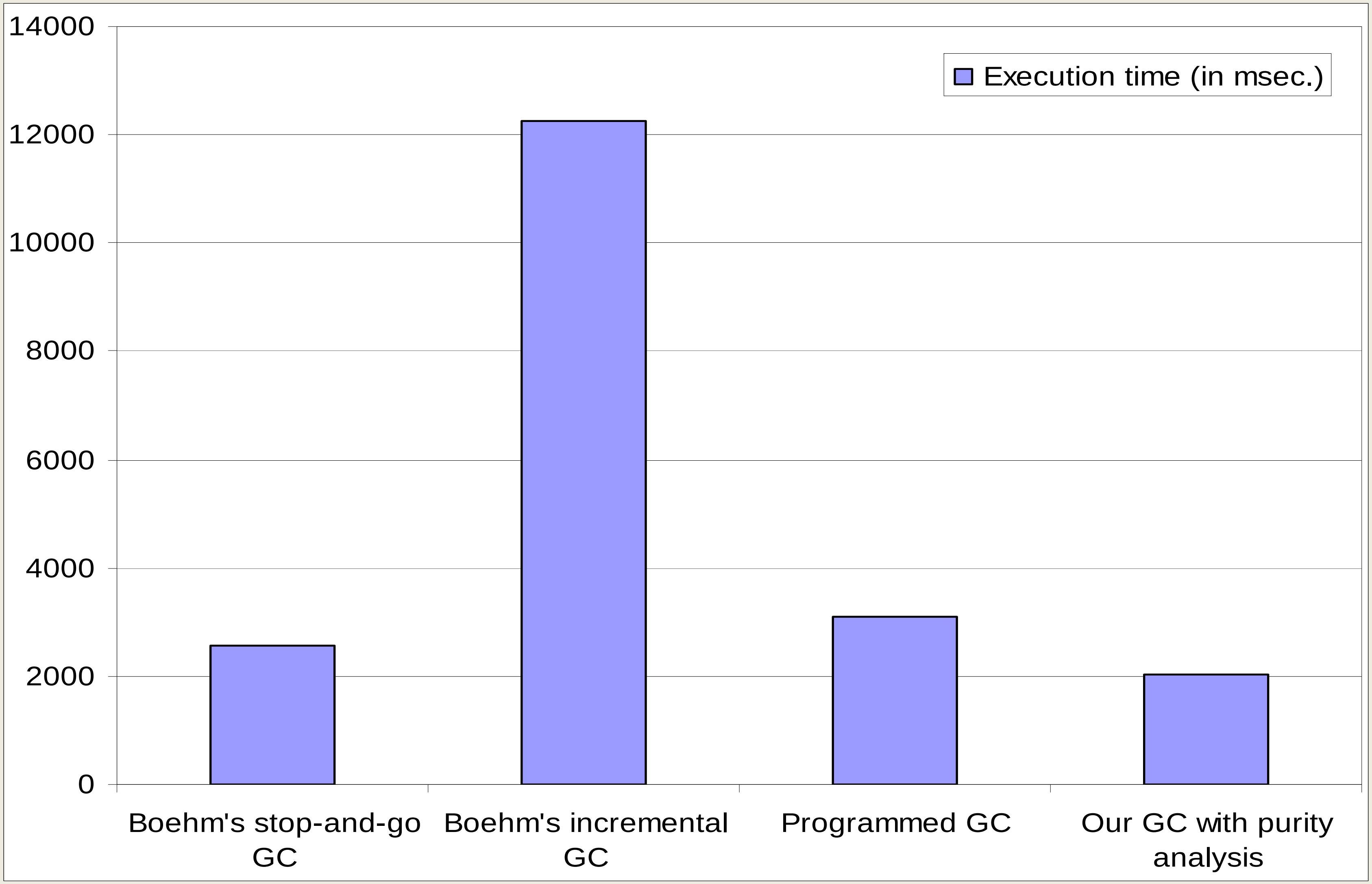


Analysis of independent parts (cont'd)

Dynamic analysis:

- Piggybacking on a (concurrent, replicating) garbage collector
- Find "pure" objects, i.e., objects that do not change state
  - Methods of "pure" objects are pure functions
  - Can be executed in parallel with other code if there are not input nor output dependencies
- Optimistic purity analysis guesses temporarily "pure" objects, i.e., objects that did not change since last collection cycle and do not (transitively) point to such objects
  - Adapted Tarjan's SCC algorithm (just one round, no stack)
  - Roll-back is cheap:
    - trap methods falsely guessed pure before they change state
    - then sequentially restart subsequently called methods

Purity analysis is for free



Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

2. Parallelization

Well-established parallelization transformation, e.g.:

- Code motion/placement moves independent statements to same block
  - Result: Basic blocks are malleable task-graphs (executable on more than one processor)
  - Nodes are simple tasks or calls to methods (malleable tasks)
  - Edges are essential dependencies
- Loop parallelization transformations
- ...

Clustering and Scheduling

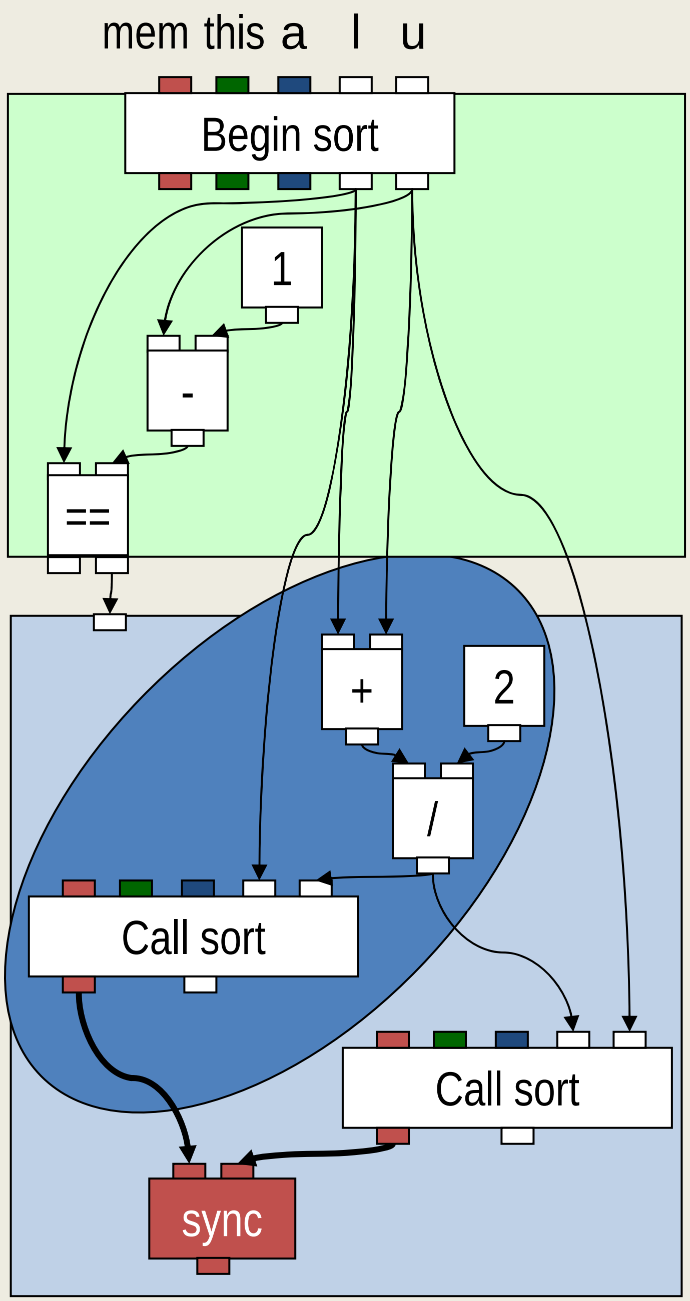
Clustering and scheduling for malleable task-graphs:

- Cluster simple tasks with malleable tasks (calls to methods) to avoid too lightweight processes
- Schedule the clusters with any (malleable task-graph) scheduling strategy
- Result are sequential and parallel components containing
  - Original sequential code
  - Several automatically generated parallel variants thereof

Example Mergesort

```
E [] a = new E [length];
a.init();
sort(a, 0, length);

public boolean sort(E[] a, int l, int u) {
        // base case
        if (l == u-1) return true;
        // split
        int q = (l + u) / 2;
        // parallel recursive calls
        sort(a, l, q) || sort(a, q, u);
        // merge
        …
}
```



Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

3. Context-aware composition

Select sequential or any parallel variant depending on context:

- Problem size
- Number of processors available for a sub-problem

Observation:

- Context, hence, optimum variant may change dynamically
- Requires runtime decision

Add infrastructure for dynamic variant selection

- Assesses context properties (e.g., problem size and number of processors) before each selection
- Dynamically selects the champion variant for each context
- Machine learning to update the champion for each context
  - learn a generic classifier for champion selection
  - based on monitoring data from executions (offline or online)

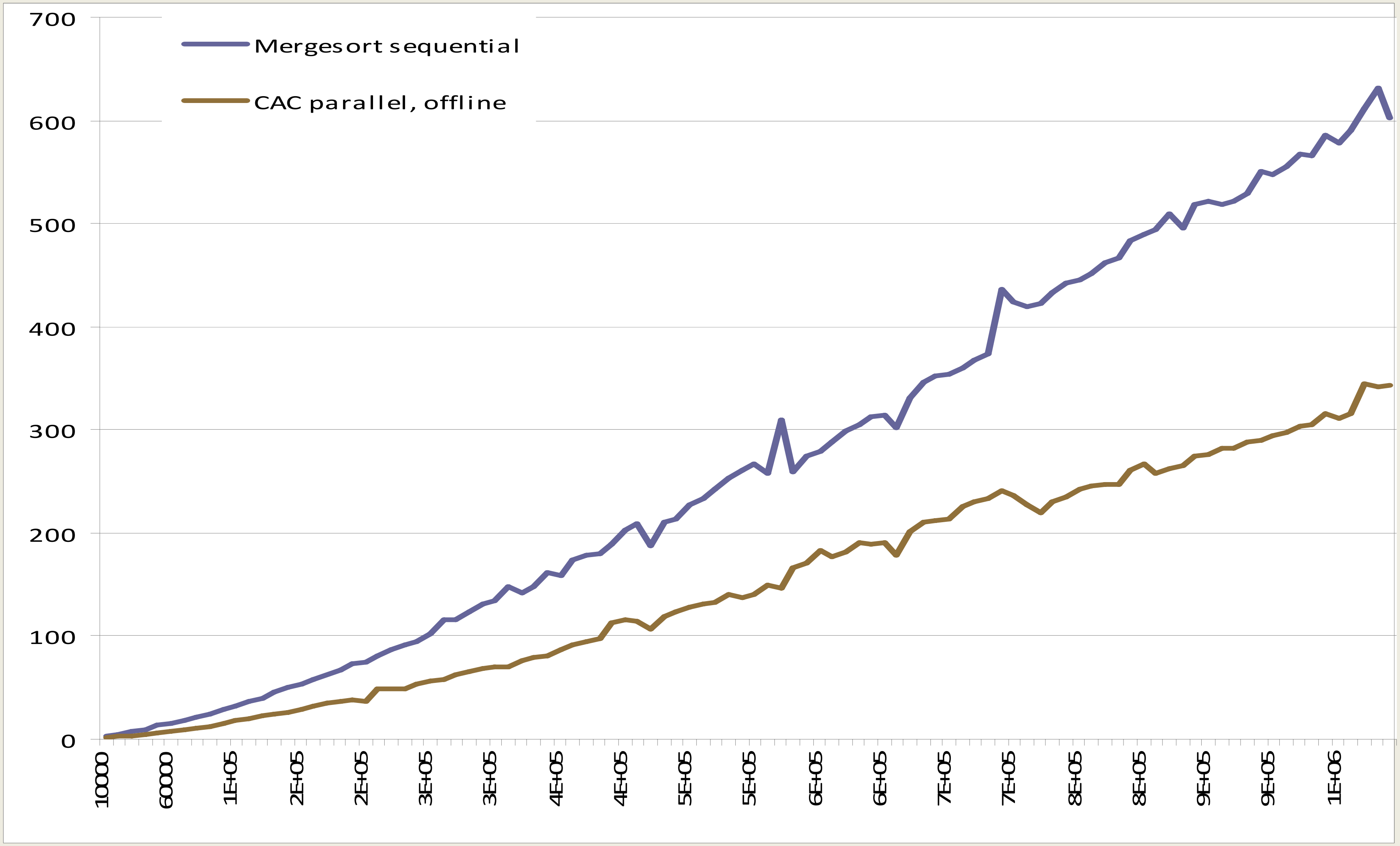
Example Mergesort

```
public boolean sort(E[] a, int l, int u) {
        …
        if (selectSequentialSchedule(a, l, q, u, numberOfProcessors)) {
                tic = getTime();
                Sort.sort(a, l, q); Sort.sort(a, q, u); //sequential schedule
                toc = getTime();
                updateSelection(toc-tic, a, l, q, u, numberOfProcessors)
        }
        else {
                tic = getTime();
                dec(numberOfProcessors);
                Sort.sort(a, l, q) || Sort.sort(a, q, u); //parallel schedule
                inc(numberOfProcessors);
                toc = getTime();
                updateSelection(toc-tic, a, l, q, u, numberOfProcessors);
        }
        …
}
```

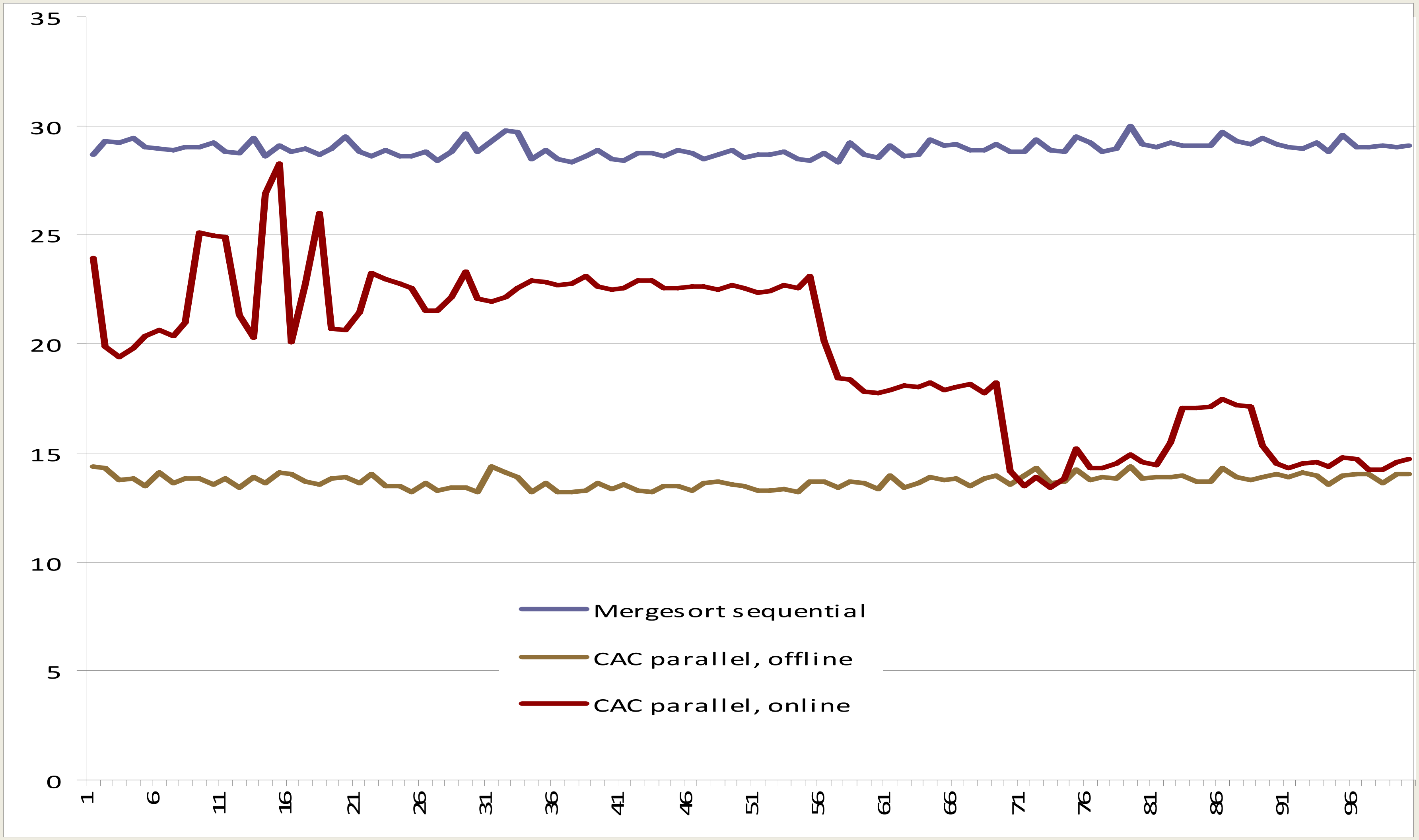
Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

4. Offline learning (time in msec.)



First results on Online learning (time in msec.)



Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

5. Related Work

- Automatic program parallelization has a long history back to the 1970s.
  - Overwhelming majority focuses on static parallelization of programs dealing with numerical computations.
- Little work on automatic parallelization of object oriented programs
  - many from Java Grande focusing again on numerical computations
  - few on general applications responding to the multi-core development
    - require manual source code annotations
    - Duarte *et al.* use algebraic laws to define static source code patterns possible to parallelize; no analysis detecting these patterns automatically.
    - Bradel *et al.* identify and parallelized computational intensive and parallelizable loops using dynamic analysis.
- JIT and speculative approaches aggressively parallelize statements regardless of their independency. In practice, some programs benefit but other significantly decrease performance.

Outline

1. Analysis of independent parts
2. Aggressive parallelization transformations
3. Context-aware composition
4. Experimental results
5. Related Work
6. Conclusions and Future Work

6. Conclusion

- Idea: separately analyze
  - The inherent parallelism (static and dynamic analyses)
  - Efficiency of sequential vs. parallelized variants (context-aware composition: offline or online learning, dynamic composition)

- Experimentally showed some speed up based on the tools available:
  - For the analysis part:
    - Java frontend generating Memory SSA code
    - efficient inter-procedural and context-sensitive Points-to and Side-effect analyses
    - Purity analysis based on GC
  - For the transformation part:
    - Malleable task graph scheduling
    - AOP infrastructure for context-aware composition
  - For the context-aware composition part:
    - Fully implemented and tested using both offline and online learning

6. Future Work

- More experiments needed parallelizing real world software instead of lab examples.
- Therefore, tool chain needs to be completed.
  - For the analysis part, we lack
    - index and loop-dependency analyses
    - integration static and dynamic analysis
  - For the transformation part, we lack code motion and loop parallelization.
  - For the context-aware composition part, we lack interleaving of monitoring / learning and scheduling.
- Finally, we need to put together the loose strings to a fully automatically parallelizing compiler and runtime system.

- Partially funded by the Swedish Research Council VETENSKAPSRÅDET THE SWEDISH RESEARCH COUNCIL