Exploiting Multi-Core Architectures for Fast Modular Synthesis

LAC2008

Feb 29, 2008

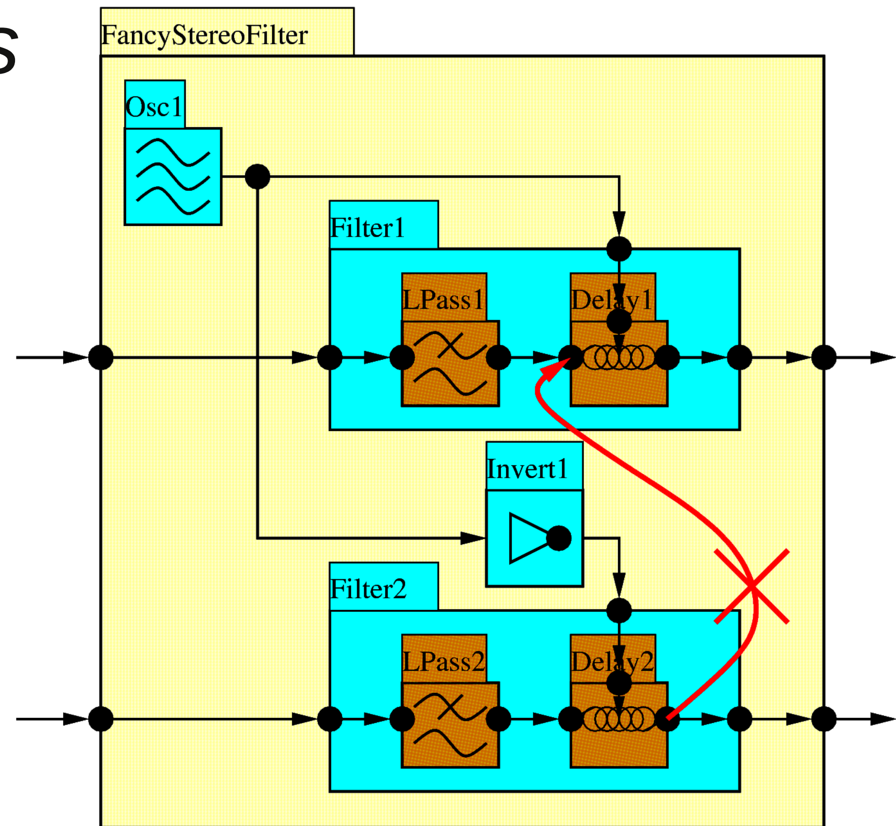
Jürgen Reuter

Multiple Cores

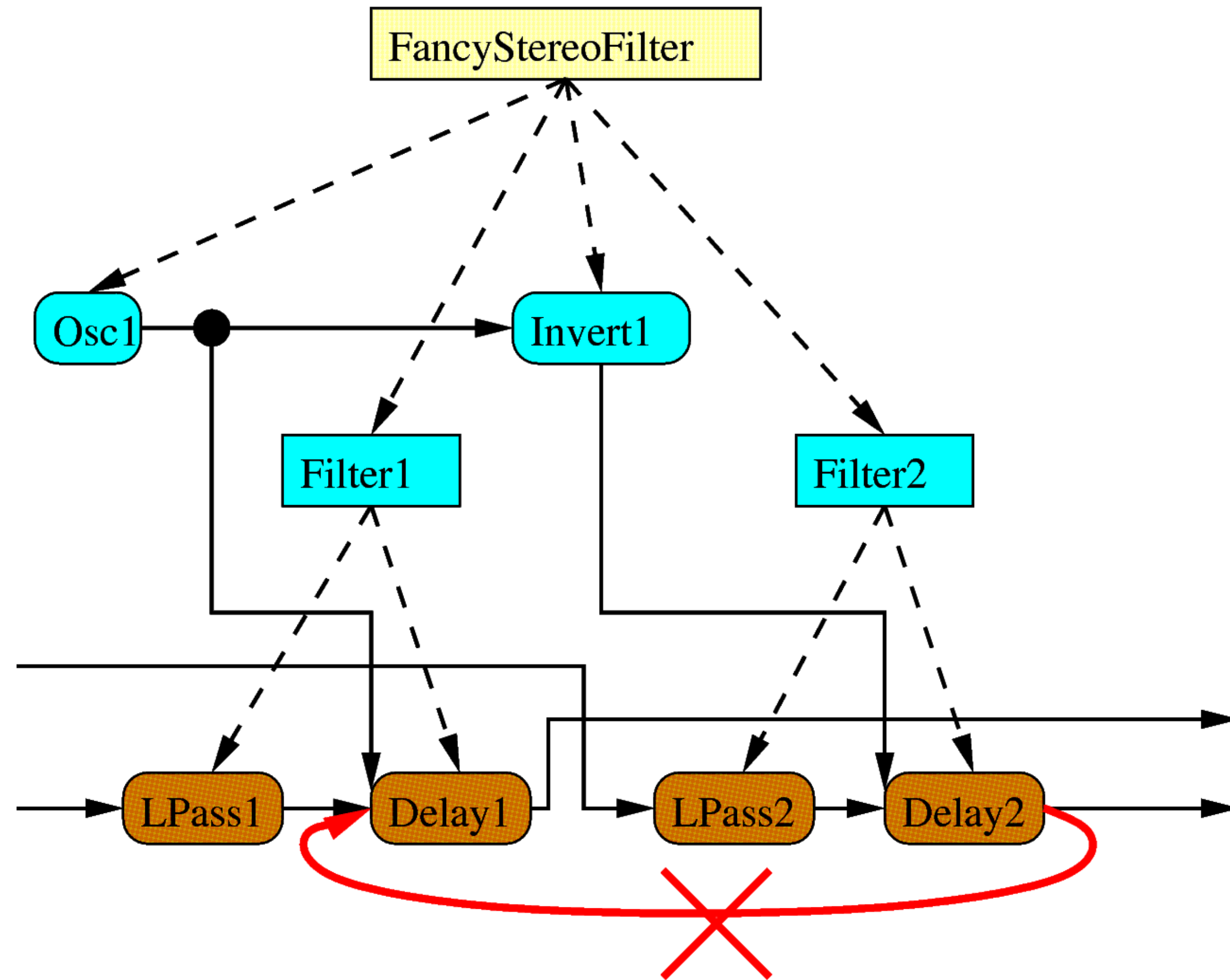
- CPU speed only slowly growing
- Multi-Core CPUs now pervade market
- Ideal for thread-parallel compute-intensive tasks
- Most existing applications not yet parallelized
- Linux kernel support grown from SMP experience
- This talk: *How to parallelize modular synthesis?*

Module Topology Model

- Hierarchy of *modules*
- *Input terminals*
- *Output terminals*
- *Primitive* modules
- *Composed* modules
- No connections between *different* submodules



Module Tree Representation



Module Timing Model

- Goal: sample synchronous operation
  - One time step per sample
- Compute module output from module inputs
- Transfer module output samples to connected module inputs

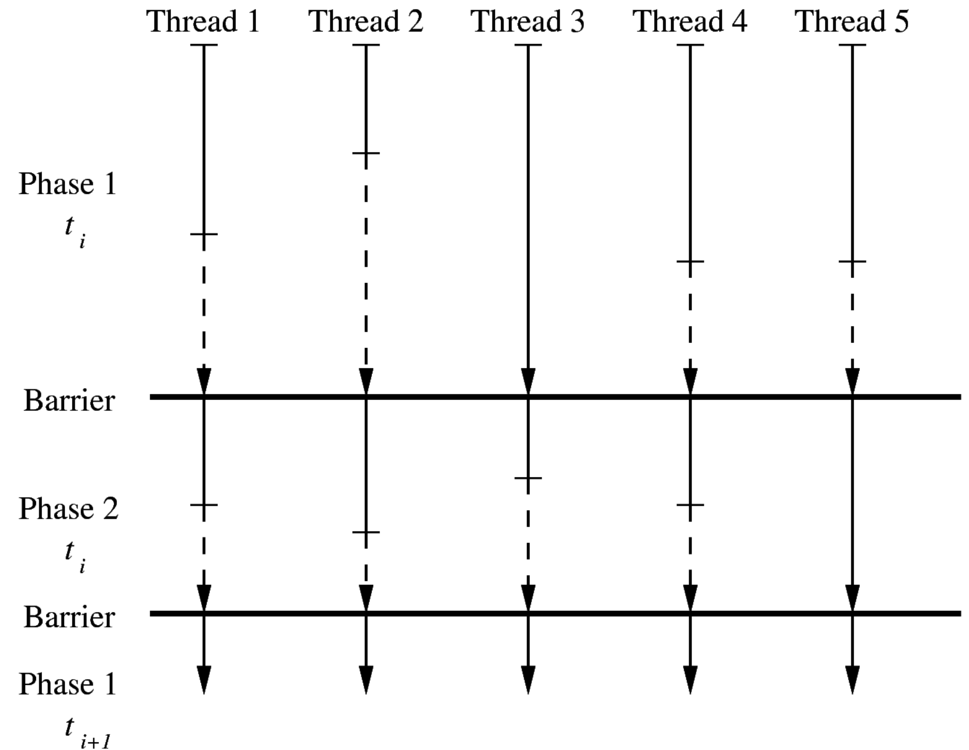
Two-Phase Compute / Update

- Use multiple threads
- Goal: sample synchronous update
- But: dependencies between modules
- => Order of update significant
- Separate into phases *compute & update*

```
while (true) do {
  // Phase 1: Compute
  for all modules do {
    compute outputs for next time step
    in terms of other module's outputs,
    but keep results private to this
    module
  }
  // Phase 2: Update
  for all modules do {
    publish outputs to other modules
  }
}
```

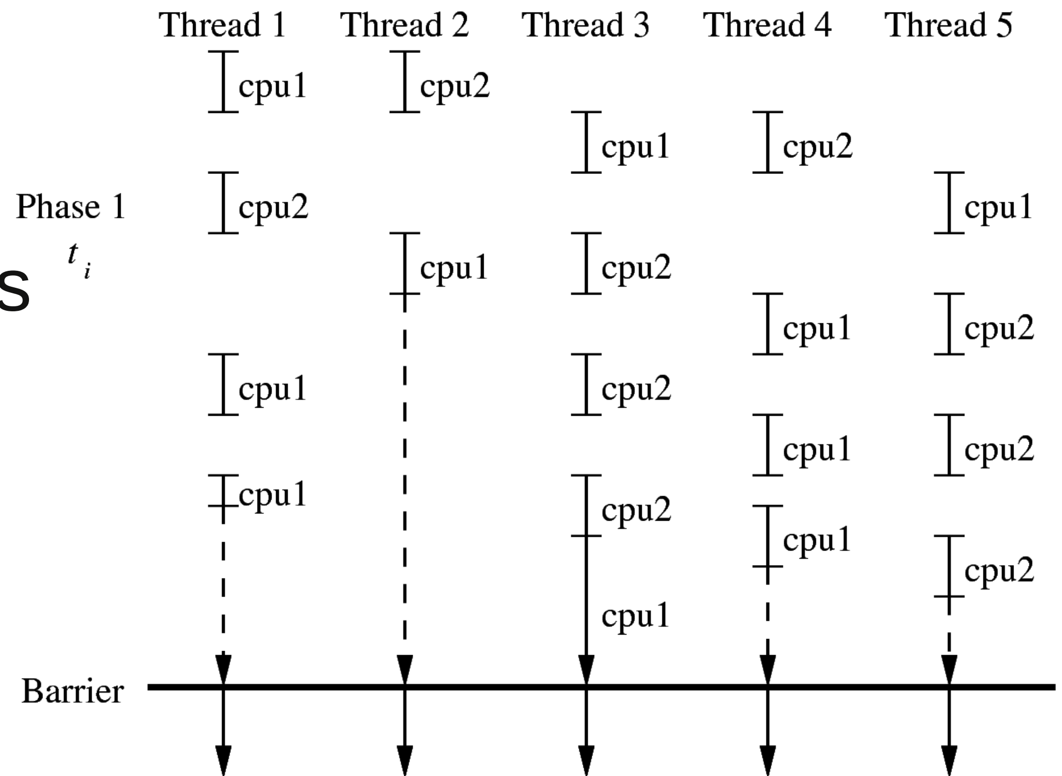
Barrier Synchronization

- Start phase 2 only after phase 1 completed in all threads
- And vice versa
- => Use barriers to synchronize threads



Round-Robin Scheduling

- Spawn one thread per module?
- Bad idea:
  - OS schedules threads onto CPUs
  - => numerous task switches
- Solution: handle multiple modules per thread



Module-to-Thread Mapping (1)

- How many threads to spawn?
  - Few threads: bad exploitation of cores
  - Many threads: thread switch overhead
  - => find trade-off

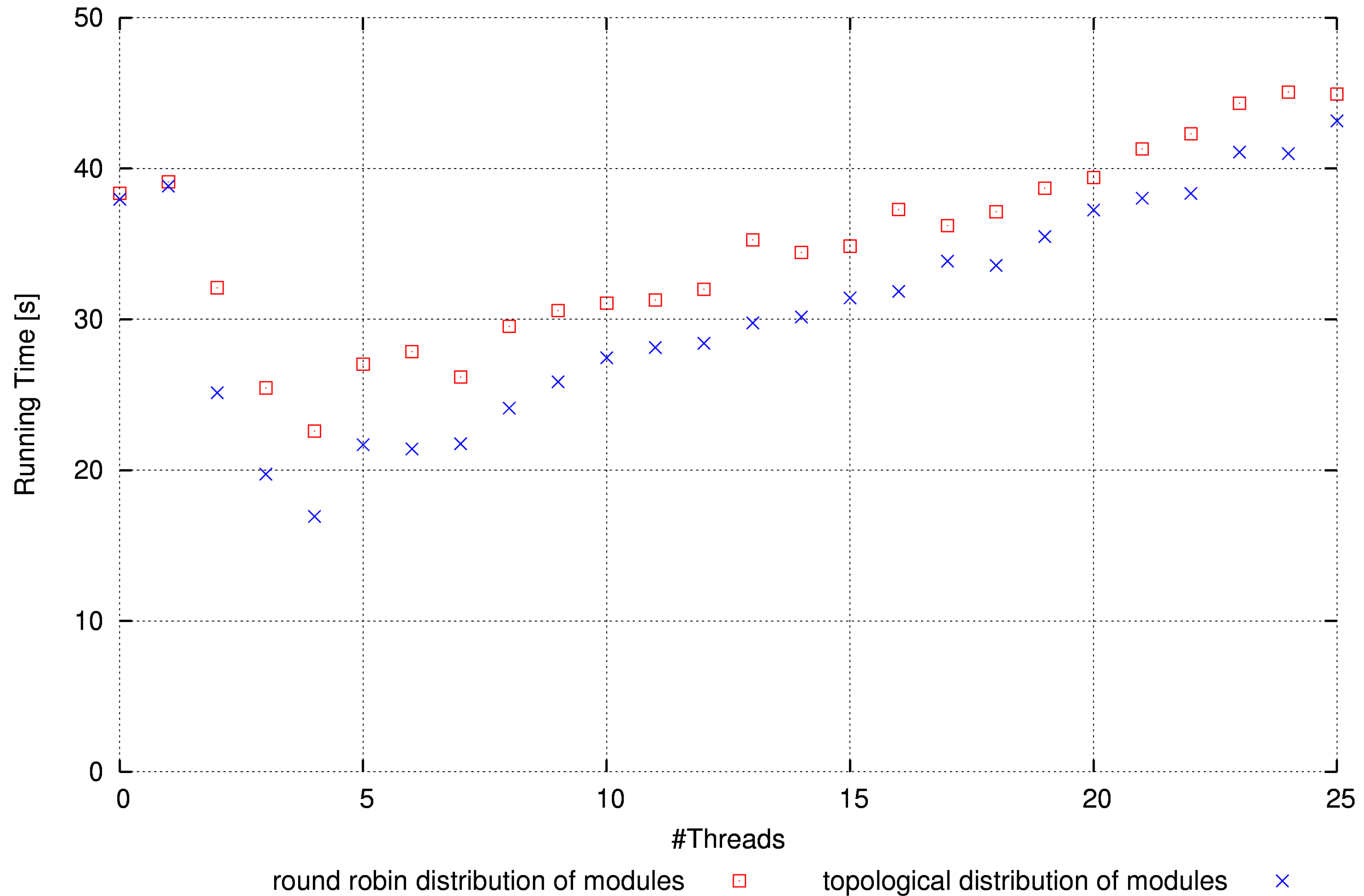
Module-to-Thread Mapping (2)

- Assign which modules to which application thread?
  - Bad data locality => less cache hits
  - Bad load balancing => CPUs idle at barrier
- Here two approaches
  - Round-robin (i.e. pseudo-random) assignment of modules to threads => better load balancing?
  - Assignment according to Module tree representation => better data locality?

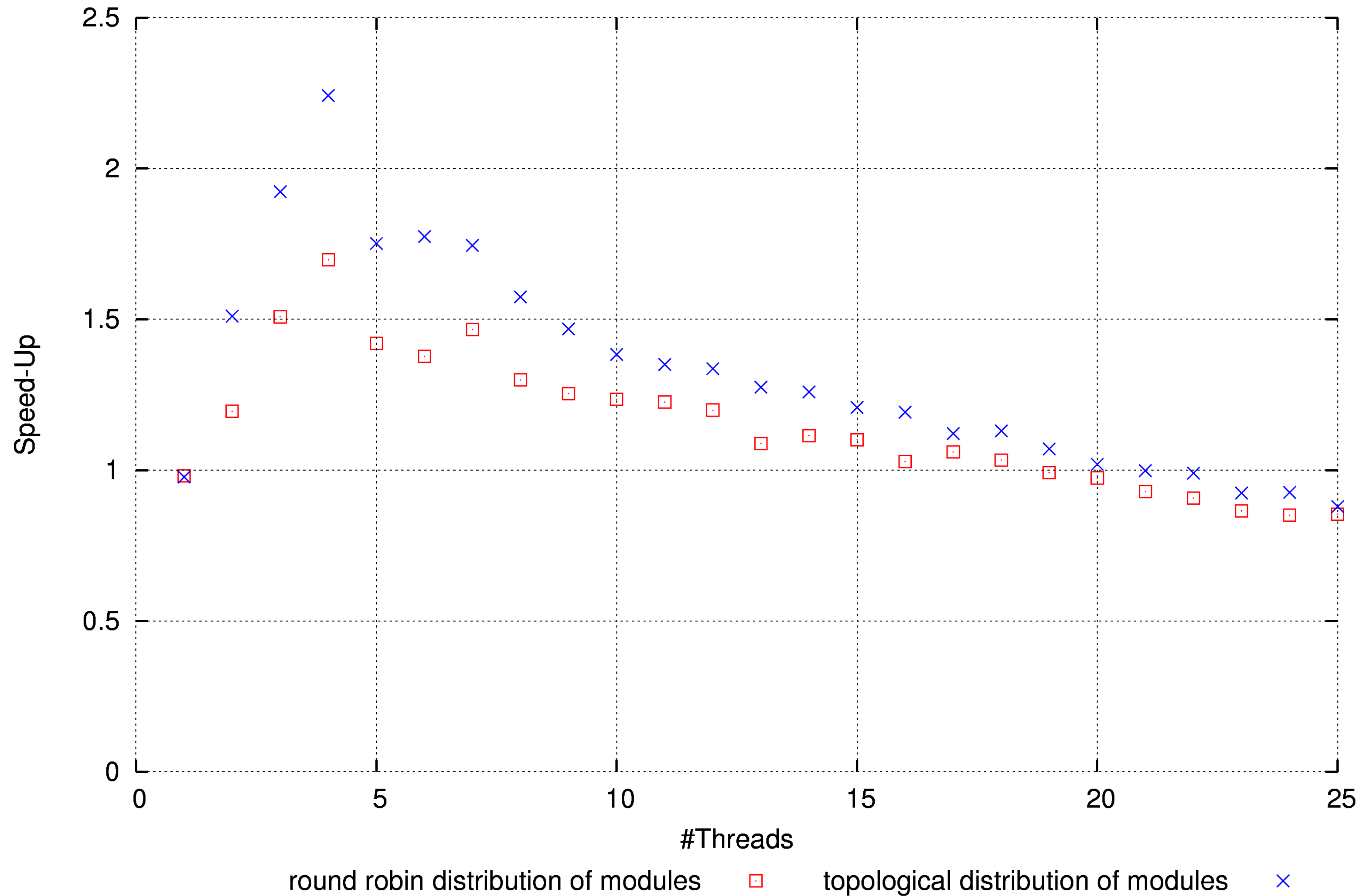
Evaluation

- Implementation in Java
  - Adjustable number of application threads
  - Java threads map to Linux native threads
- Compare distributions of modules among threads
  - Round-robin distribution
  - Topological distribution
- Dummy synth with array of ~2000 oscillators
- B/W SoundPaint run with ~1650 modules
- Run on Core Quad CPU Q6600 @ 2.40 GHz

Multi-Array Synth Parallel Synthesis

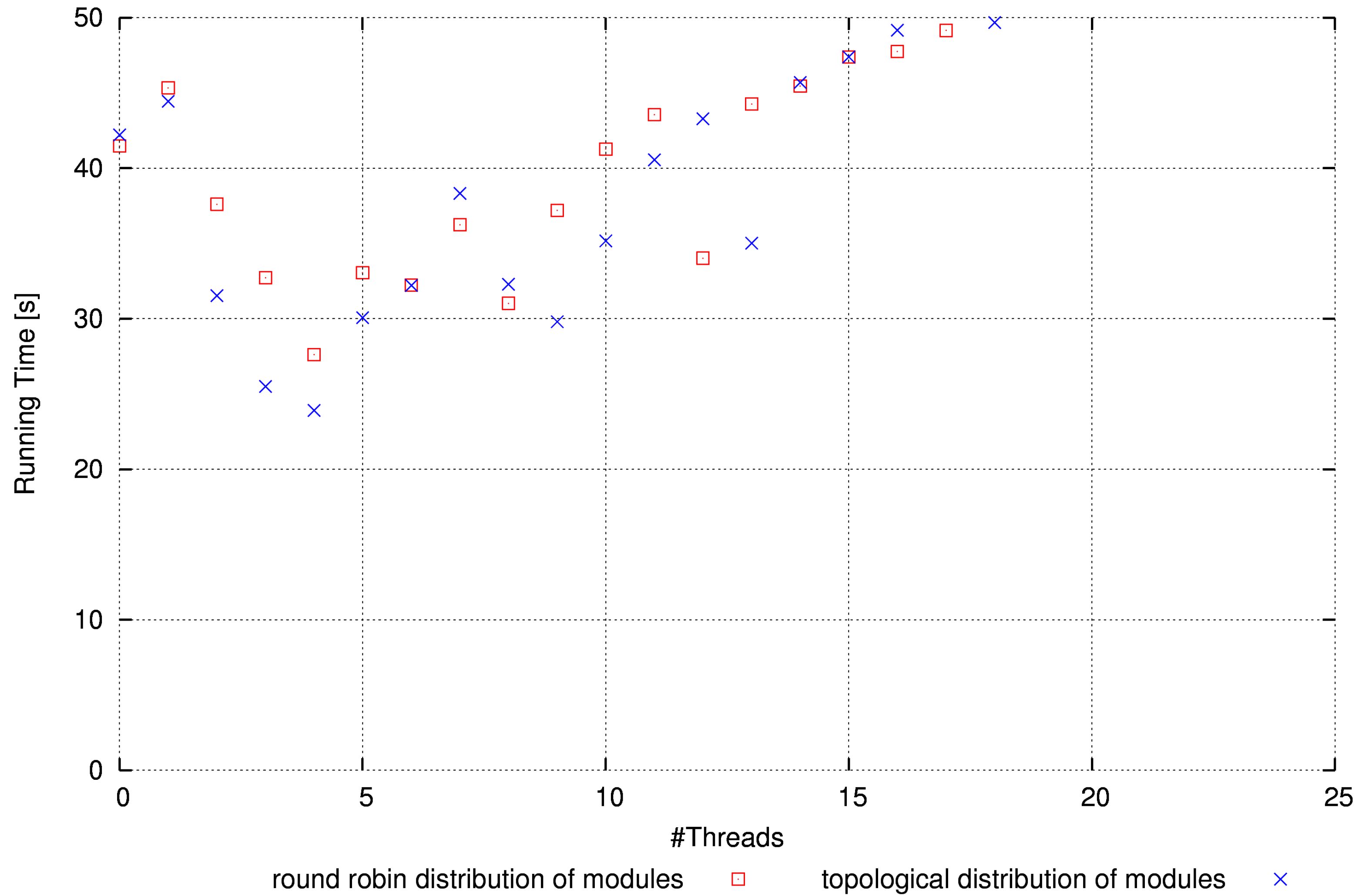


Multi-Array Synth Speed-Up

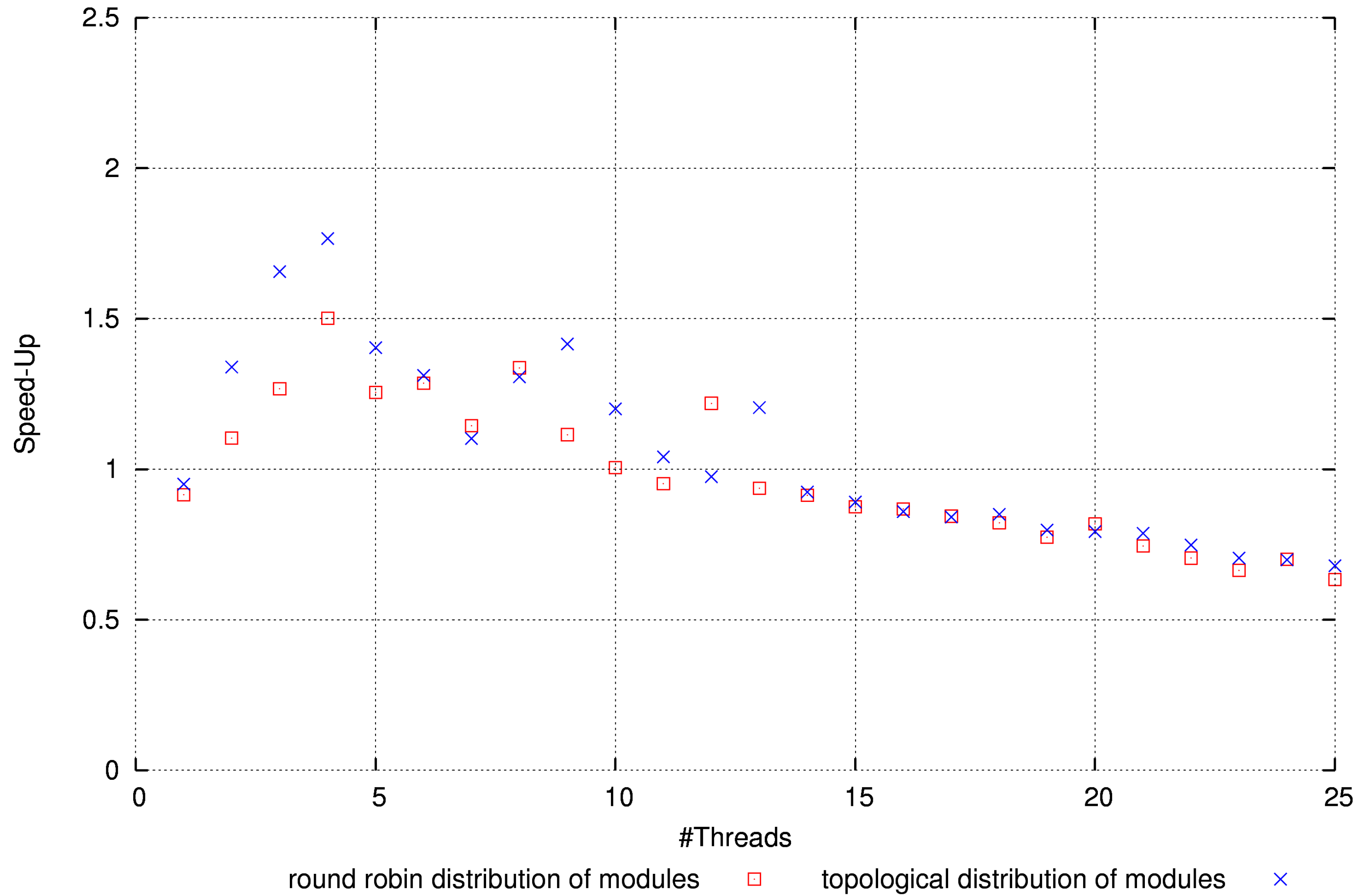


B/W SoundPaint Performance

Performance Costs of Parallel Synthesis on Core2 Quad CPU @ 2.40GHz



B/W SoundPaint Speed-Up



Observations

- Only small overhead of multi-threaded algo (run with 1 thread) over sequential algo
- Optimal speed @ 4 threads (=number of cores)
- “Real-life” SoundPaint data not as clear as dummy synth array
  - Perhaps due to more irregular compute time?
- Topological distribution on the average much better than round-robin distribution

Future Work

- Reason for higher performance of topological distribution of modules still unclear
  - Bad data locality of round-robin distribution?
  - CPUs running idle at barriers?
- Overall speed-up still not satisfying
  - Investigate load balancing / idle CPUs at barriers
  - Let idle CPUs pre-compute samples (e.g. for modules without input terminals)
  - Merge local lightweight modules into heavier ones
- Complete SoundPaint implementation

Conclusion

- Spawn multiple threads to exploit multi-cores
- Don't spawn too many threads (thread switching overhead!)
- => Support an adjustable number of threads
- Carefully distribute the work among the threads
  - Data locality (avoid cache misses)
  - Load balancing (avoid idle CPUs at barriers)

Questions?

- Code (still very experimental) available at **www.soundpaint.org/modsynth**
- Relevant code for barrier synchronization and module distribution currently in class **org.soundpaint.modsynth.syntest.Master**