DoPipe: An Effective Approach to Parallelize Simulation

Parallel Processing Institute
Fudan University

2008

Outline

- Motivation
- Design Issues
- Experiments
- Work ongoing

Motivation

- General Flow of Simulation

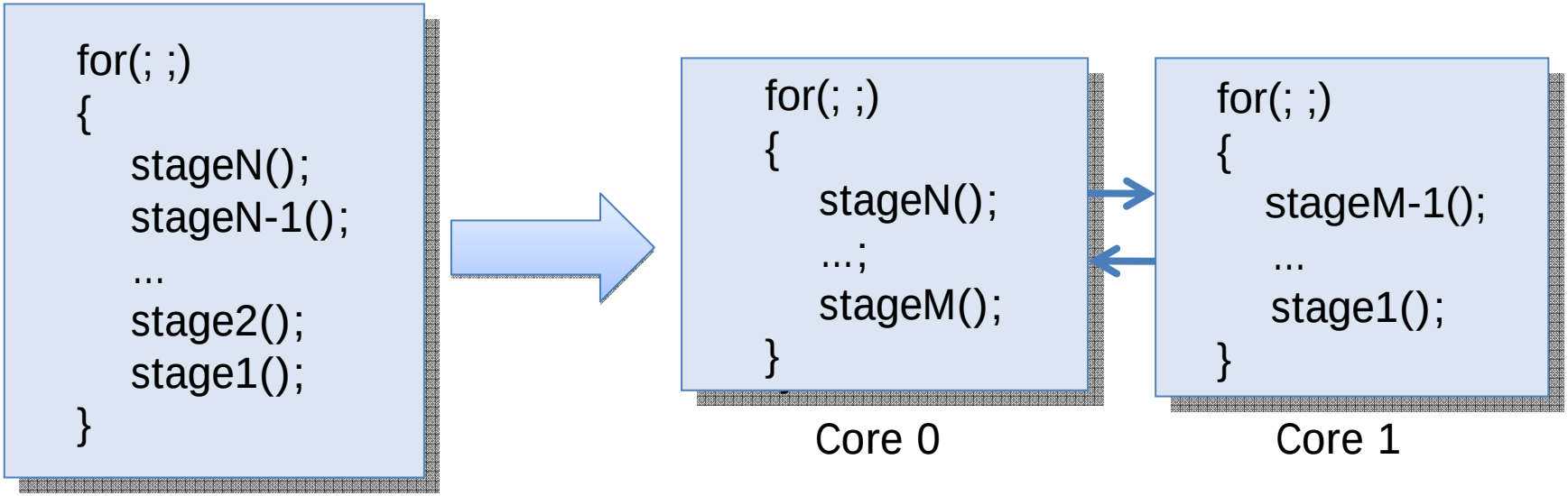
```
for(; ;)
{
    stageN();
    stageN-1();
    ...
    stage2();
    stage1();
}
```

Running on a single-core processor

How to effectively utilize the increasingly popular multicore processors to accelerate the simulation process?

Motivation

- Parallelize the simulation by pipelining the simulated stages



Running on a single-core processor

Running on a dual-core processor

Motivation

- Traditional dopipe
  - Back-edge dependency
    - Lower the opportunity of partition
      - Exasperate the workload imbalance
  - Communication
    - Depend on hardware support
      - Low access latency synchronization array

Design Issues

- Speculation
- Communication & Synchronization

Speculation

- To keep the computing core as busy as possible
  - Simulate the next cycle as long as the correctness is not violated
  - Temporally save the calculated result into a buffer and wait for being consumed by other core(s)
  - Stop the progress when the buffer is full

Speedup percentage will be severely limited when the computing cores are synchronized cycle by cycle.

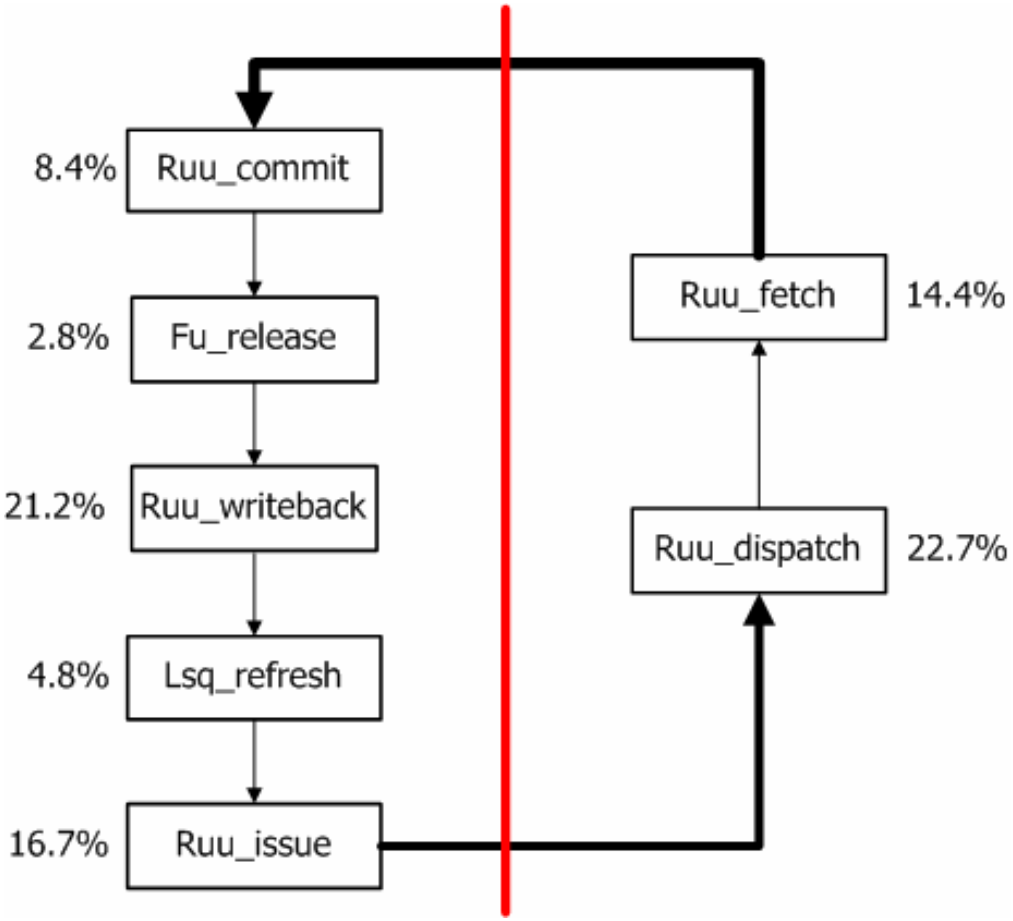
Communication & Synchronization

- Notify the other core(s) immediately after the data is completely produced
  - Reduce the other cores' waiting time.
- Access the data produced by the other core(s) as late as possible
  - Reduce the current core's waiting time.

A step to further improve the performance.

Implementation

- Stage cutting



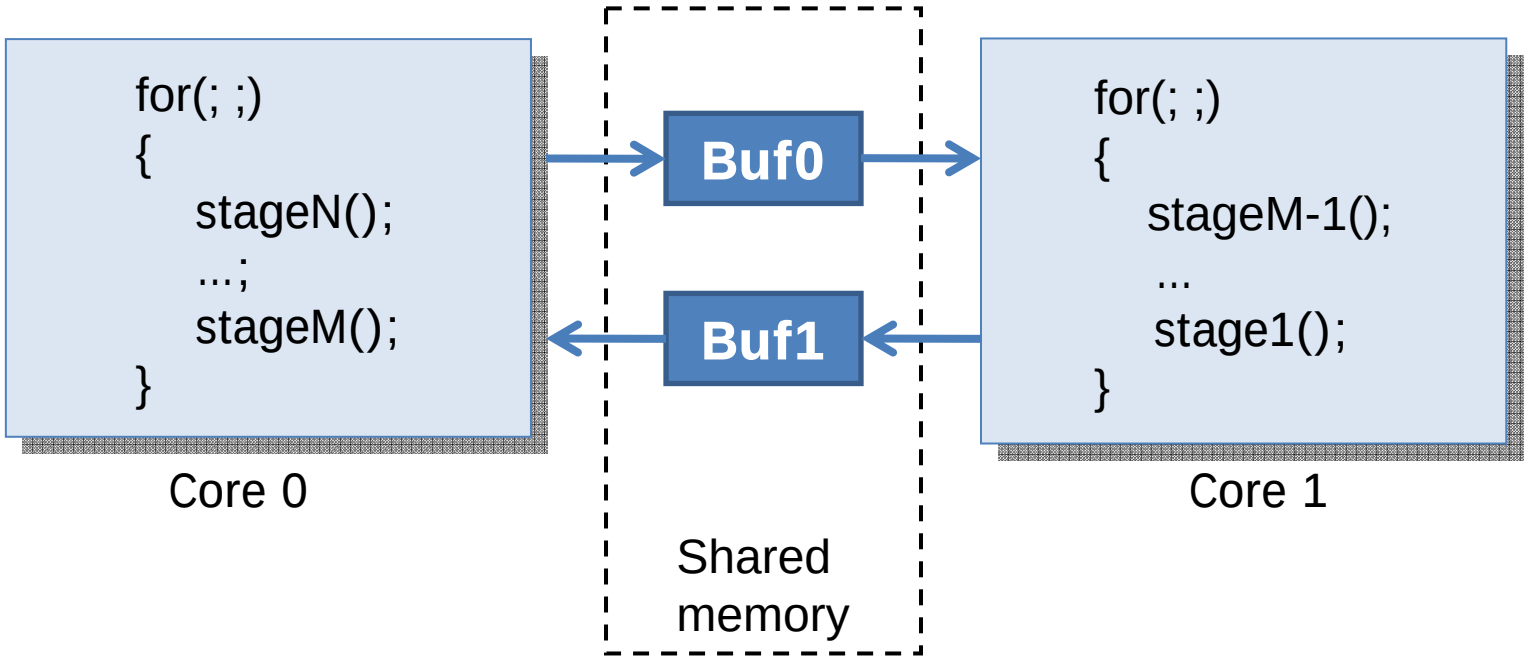
Communication

- Candidates available in Linux
  - Message
  - Pipe
  - FIFO
  - Shared Memory √
  - ...

Most effective!
shmget
shmat
Shmdt
...

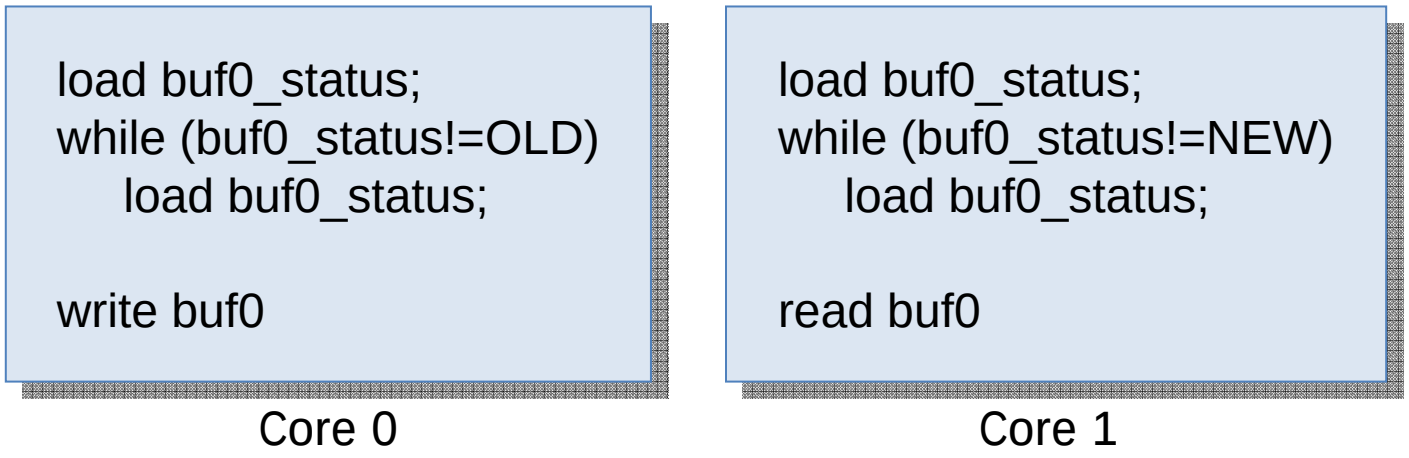
Communication (cont.)

- Example: 2-core dopipe
  - Producer/Consumer relationship



Synchronization

- Candidates available in Linux
  - Lock, Mutex, Semaphore…
- Our choice
  - Test-and-set shared variable
  - Example: 2-core dopipe



Experiment Platform

- Simulator
  - SimpleScalar3.0d
  - Config: shared & private L2 cache
- Benchmark
  - gcc,gzip,art
  - Input: ref

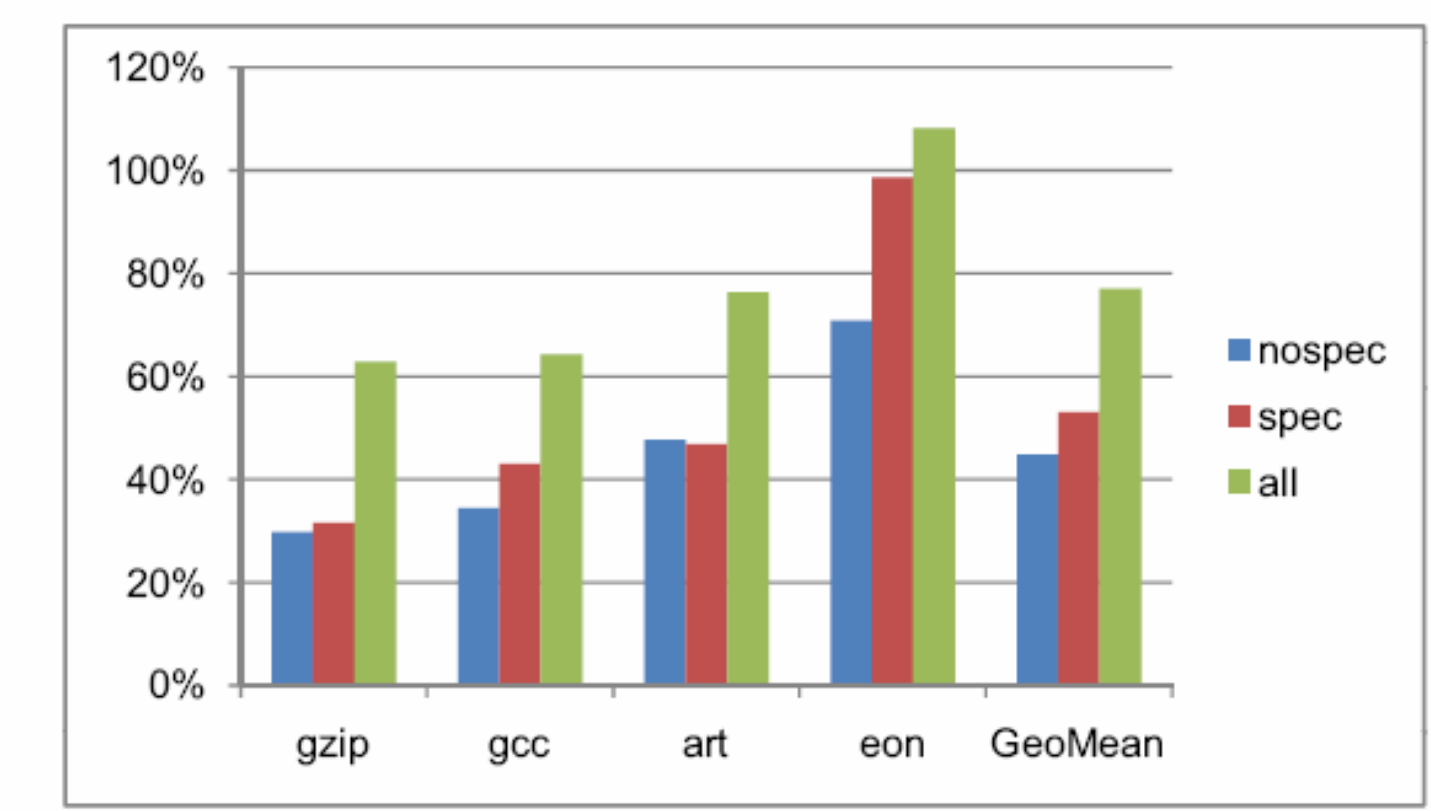
Hardware Platforms

- Two different intel processors

	E6300	Q6600
Tech.	65nm	65nm
#cores	2	4
Clock Speed	1.86GHz	2.40GHz
Trace Cache	2 x 32KB	4 x 32KB
Data Cache	2 x 32KB	4 x 32KB
L2 Cache	2MB	2 x 4MB
Front Side Bus	1066.7 MHz	1066 MHz
Memory Frequency	333.3 MHz	400 MHz
Memory Size	2GB	4GB

Experimental Results

- Speedup on 4-core Q6600



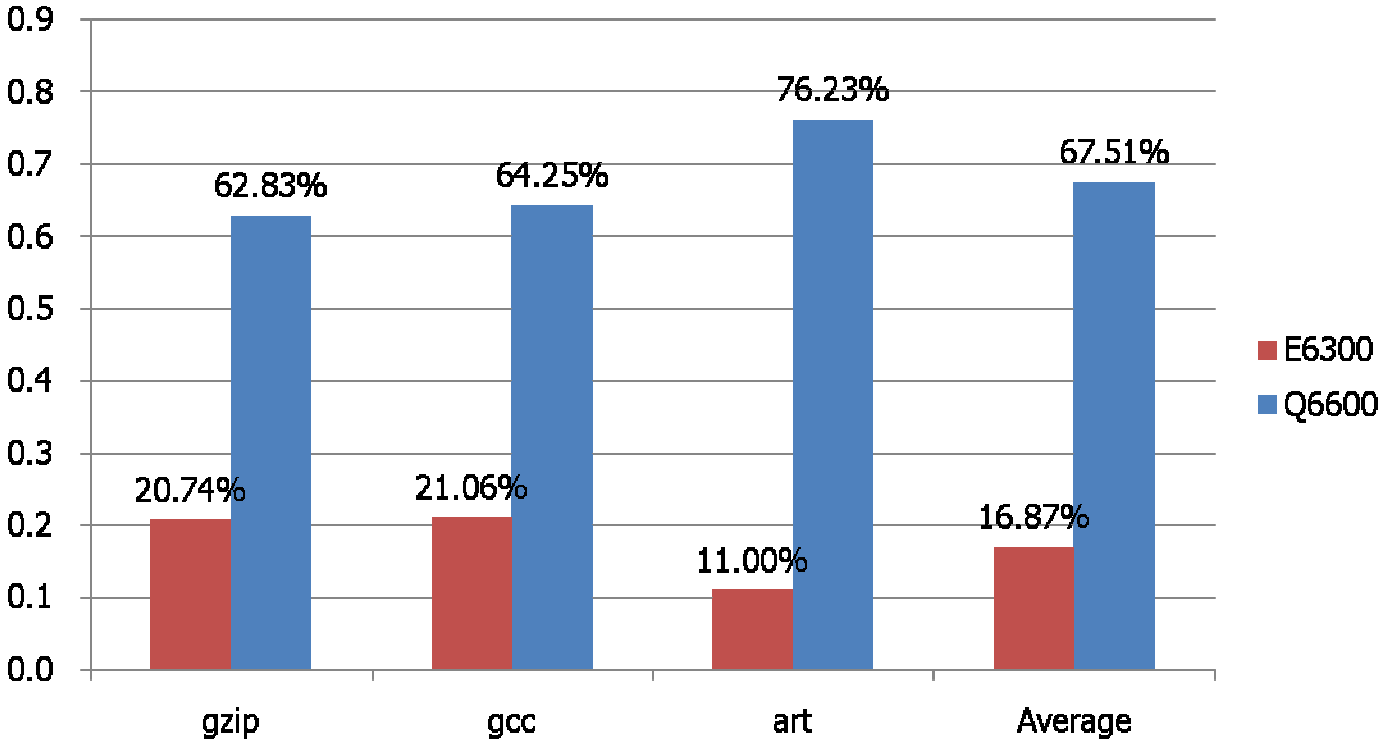
Private L2 vs Shared L2

- Private L2
  - More aggressively optimization can be done since il2 and dl2 are exclusively accessed by different cores
- Shared L2
  - Careful checks must be performed before any speculation.

So, Dopipe should theoretically reach relatively higher speedup under private L2 cache configuration.

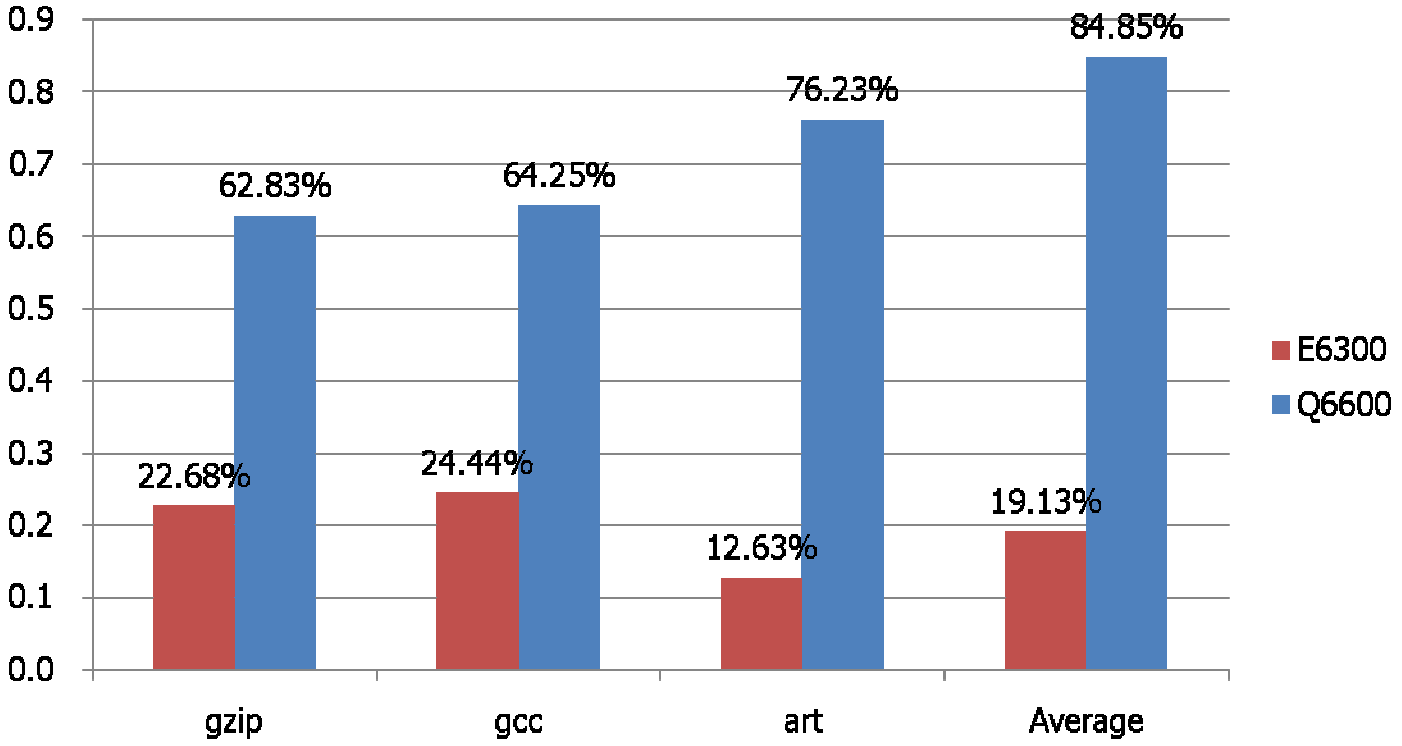
Experimental Results

- Speedup Percentage (Shared L2 Cache)



Experimental Results

- Speedup Percentage (Private L2 Cache)



Reasons for Speedup Gap

- Q6600 over E6300
  - More powerful computation capability
  - Larger L2 Cache size
  - Faster inter-core communication
  - Faster memory access

Critical factors since our dopipe relies on shared memory to implement communication and synchronization

To Conclude, Q6600 saves much less time on waiting for shared data, which means much better CPU utilization (also higher speedup)!

Work Ongoing

- Develop a 4-core version of dopipe SimpleScalar