GUMSMP: a multi-level parallel Haskell implementation

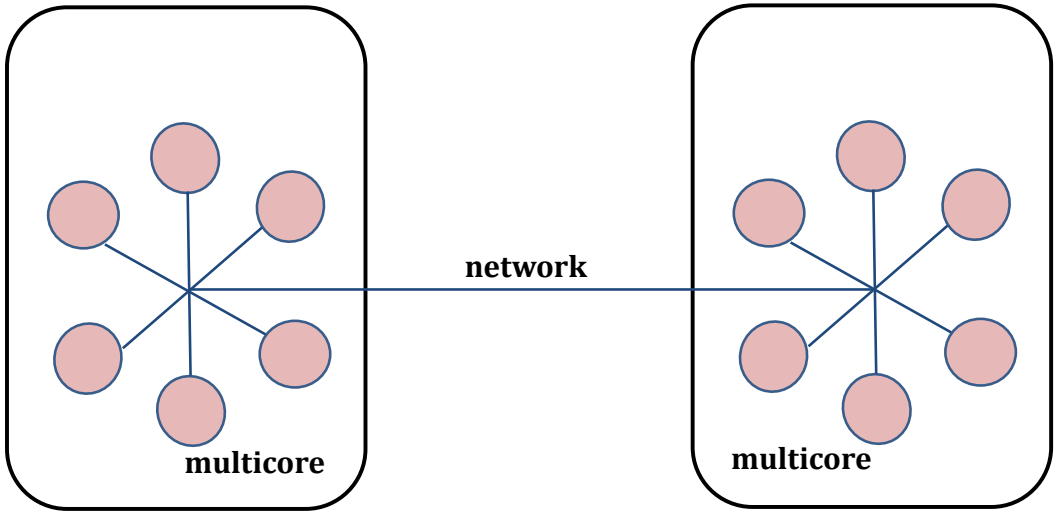
Malak Aljabri, **Hans-Wolfgang Loidl**, and Phil Trinder

The University of Glasgow - Heriot Watt University

Sep 6, 2014

Motivation

- Parallel architectures are increasingly **multi-level** e.g. clusters of multicores.
- A hybrid parallel programming model is often used to exploit parallelism across the cluster of multicores e.g. using MPI + OpenMP.
- Managing two abstractions is a **burden** for the programmer and increases the cost of porting to a new platform.



Contents

GpH (Glasgow Parallel Haskell)

- GpH is a conservative, parallel extension of Haskell, focussing on stateless code.
- **Identify** parallelism, do not control it (semi-explicit)!
- Parallelism is expressed by two primitives added to the Haskell program: **par** and **pseq**.

```
par  :: a -> b -> b    -- parallel composition
pseq :: a -> b -> b    -- sequential composition

            x `par` y => y
```

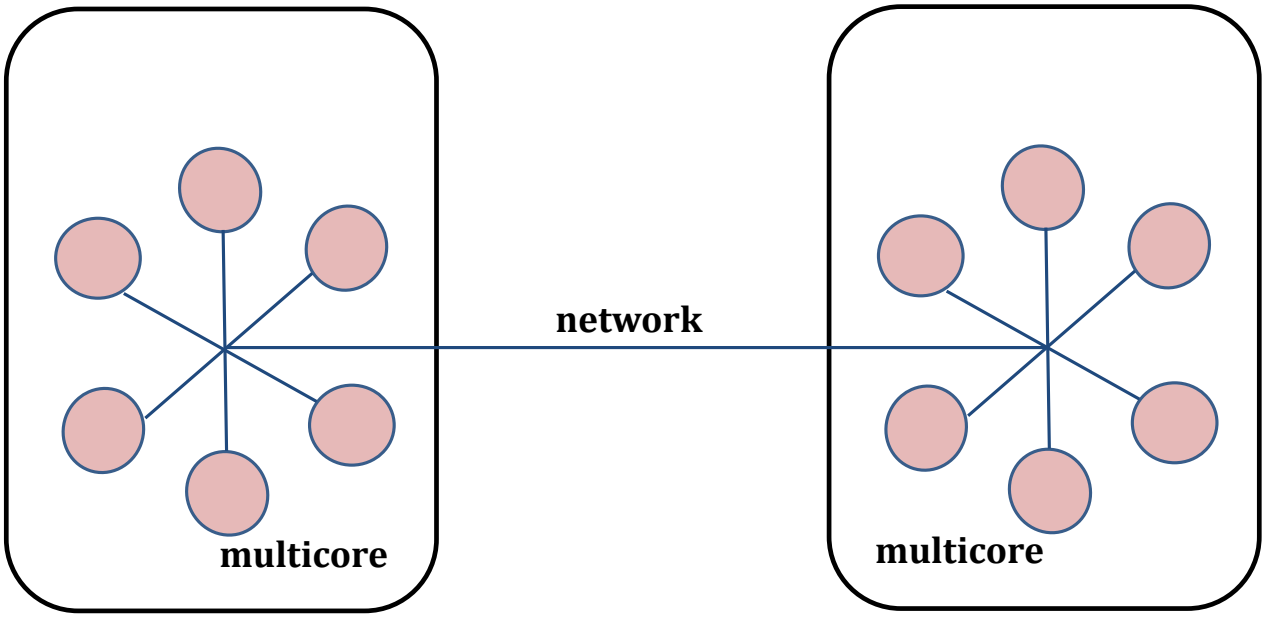
- Evaluation strategies are abstractions over these basic primitives.
- Example[1]: `parmap f xs = map f xs `using` parList rdeepseq`

[1] See AiPL14 summer school and "Seq no more" paper (Haskell'10)

GpH Implementations

Three main GpH implementations (runtime-systems):

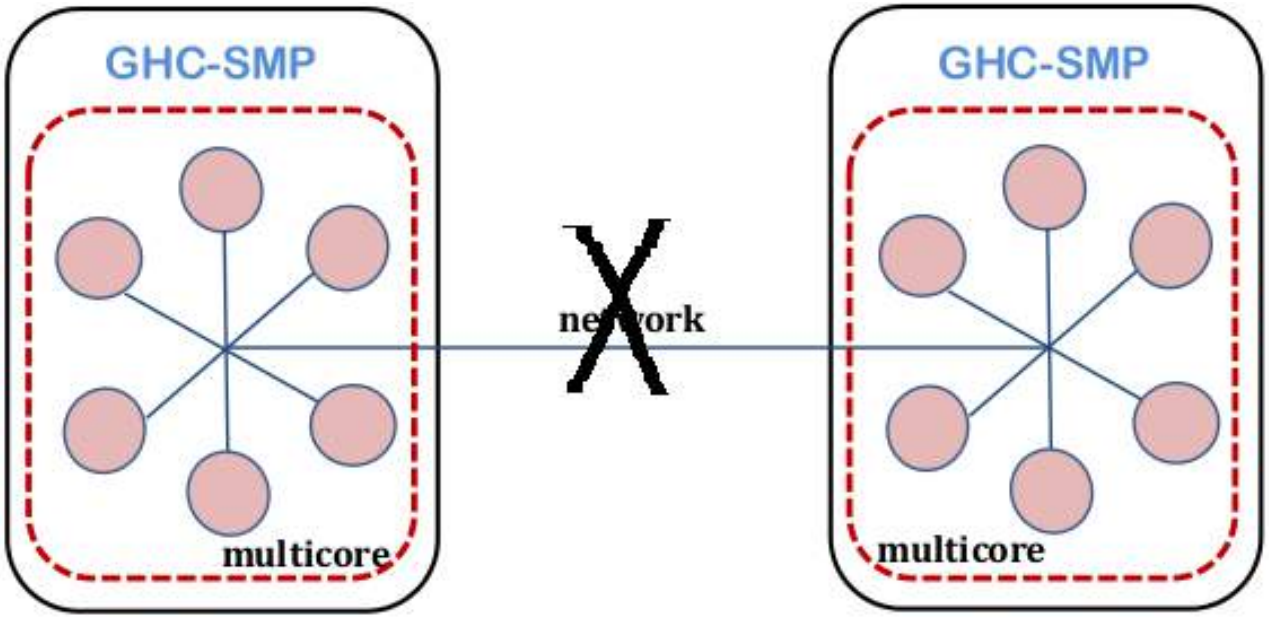
- GHC-SMP - shared memory.
- GHC-GUM - distributed memory.
- GUMSMP - hybrid shared/distributed memory.



GpH Implementations

Three main GpH implementations (runtime-systems):

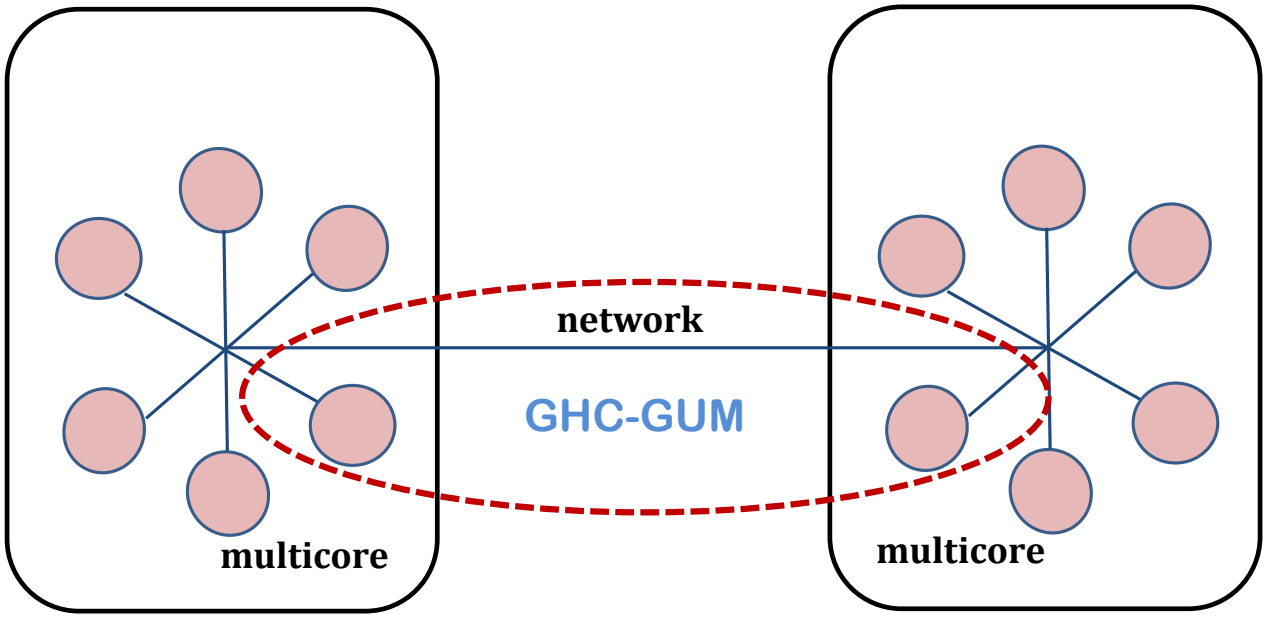
- GHC-SMP - shared memory.
- GHC-GUM - distributed memory.
- GUMSMP - hybrid shared/distributed memory.



GpH Implementations

Three main GpH implementations (runtime-systems):

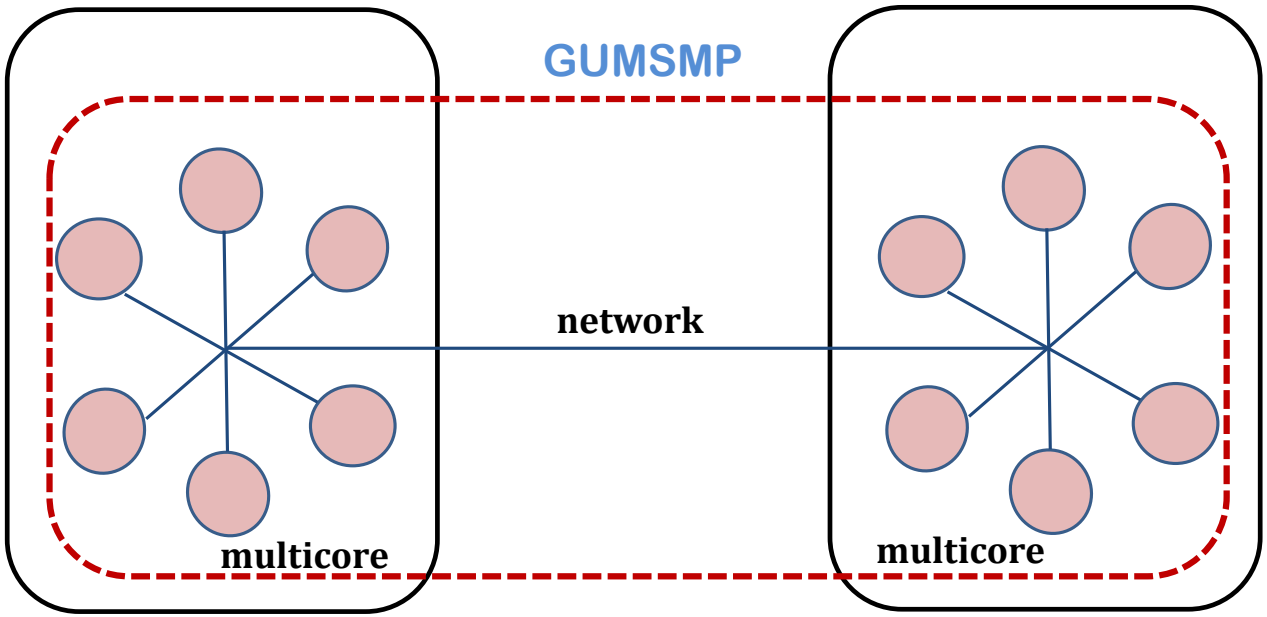
- GHC-SMP - shared memory.
- GHC-GUM - distributed memory.
- GUMSMP - hybrid shared/distributed memory.



GpH Implementations

Three main GpH implementations (runtime-systems):

- GHC-SMP - shared memory.
- GHC-GUM - distributed memory.
- GUMSMP - hybrid shared/distributed memory.



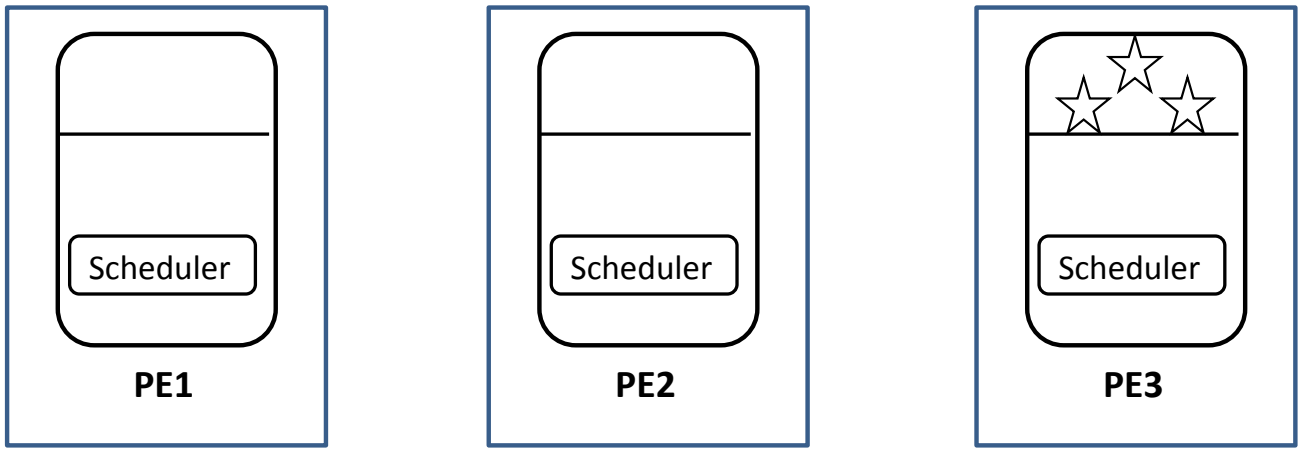
Our System: GUMSMP

- A multilevel parallel Haskell implementation for clusters of multicores.
- Integrates the advantages of the two GpH implementations:
  - Cheap parallelism on one node (GHC-SMP)
  - Scalable parallelism on a cluster (GHC-GUM)
- Implements **virtual shared memory** on a cluster.
- Uses implicit synchronisation and on-demand communication.
- Provides improvements for **automatic load balancing**.
- Provides a **single high-level programming model**.

Work Distribution in GHC-GUM

Load Balancing:

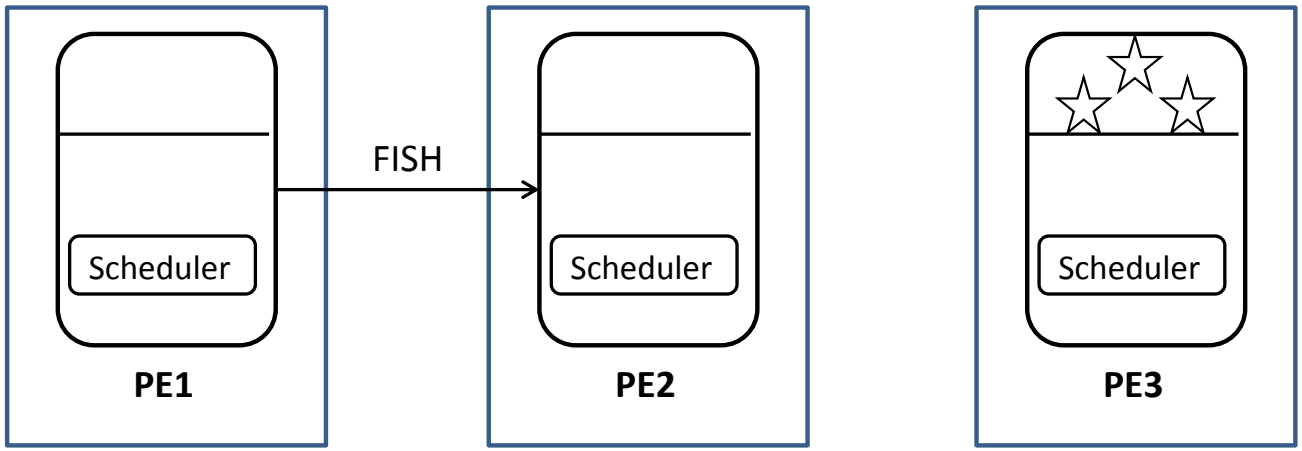
1. Searching for Local Work.
2. Searching for Remote Work.



Work Distribution in GHC-GUM

Load Balancing:

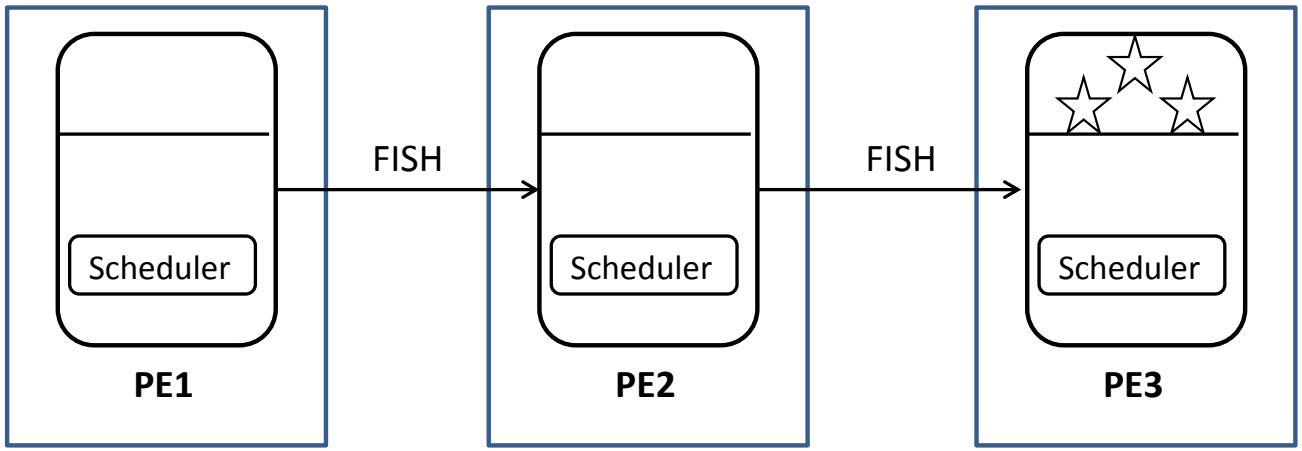
1. Searching for Local Work.
2. Searching for Remote Work.



Work Distribution in GHC-GUM

Load Balancing:

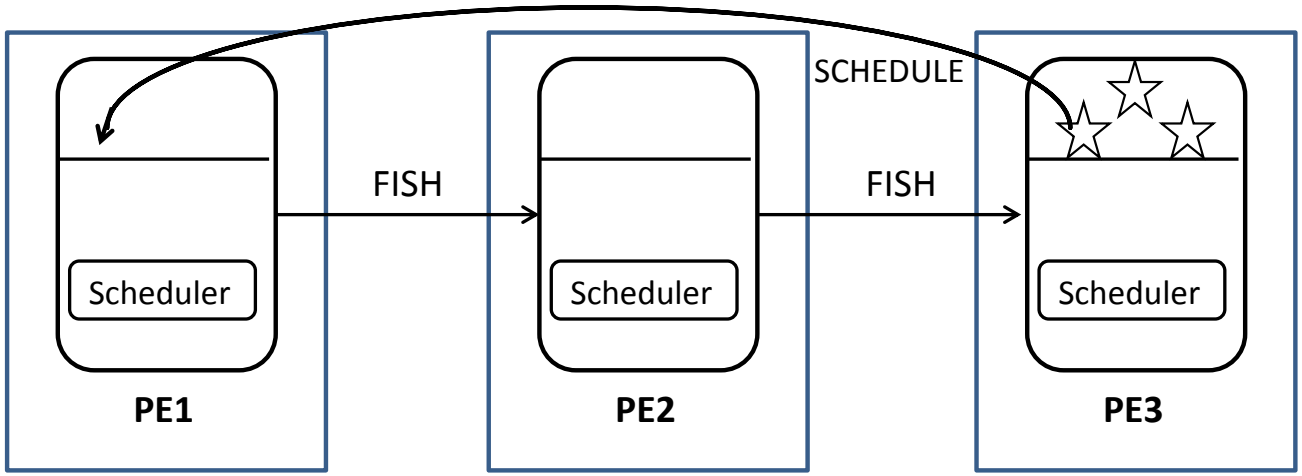
1. Searching for Local Work.
2. Searching for Remote Work.



Work Distribution in GHC-GUM

Load Balancing:

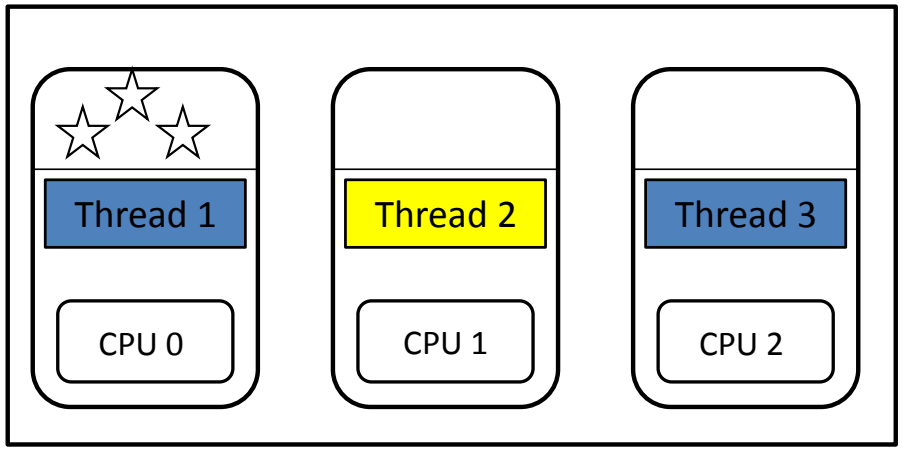
1. Searching for Local Work.
2. Searching for Remote Work.



Work Distribution in GHC-SMP

Load Balancing:

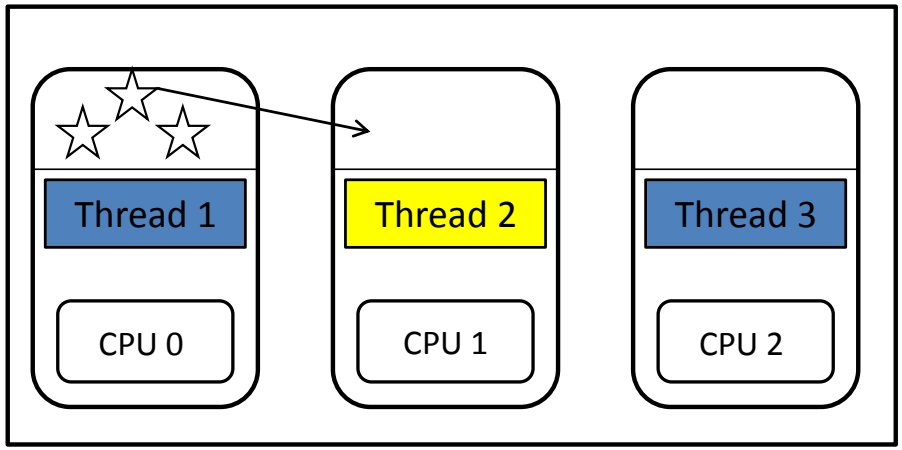
- Processor's Spark Pool is implemented as a bounded work-stealing queue.
- The owner can push and pop from one end of the queue without synchronization.
- Other threads can steal from the other end of the queue.



Work Distribution in GHC-SMP

Load Balancing:

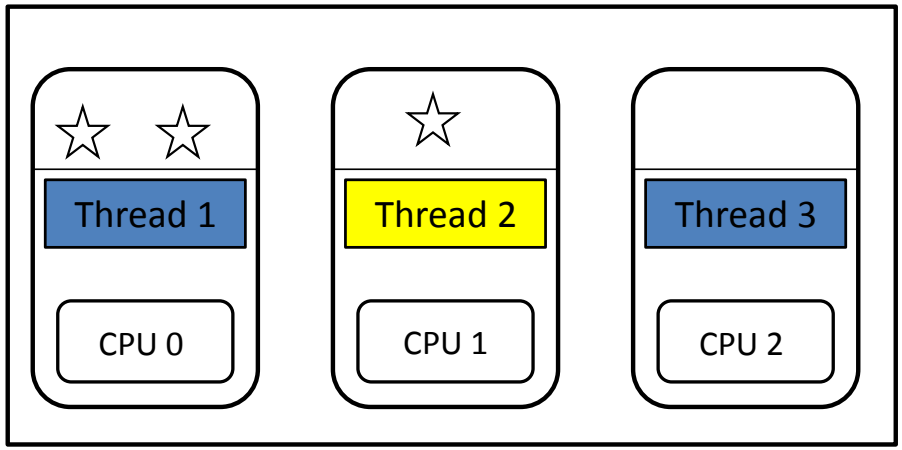
- Processor's Spark Pool is implemented as a bounded work-stealing queue.
- The owner can push and pop from one end of the queue without synchronization.
- Other threads can steal from the other end of the queue.



Work Distribution in GHC-SMP

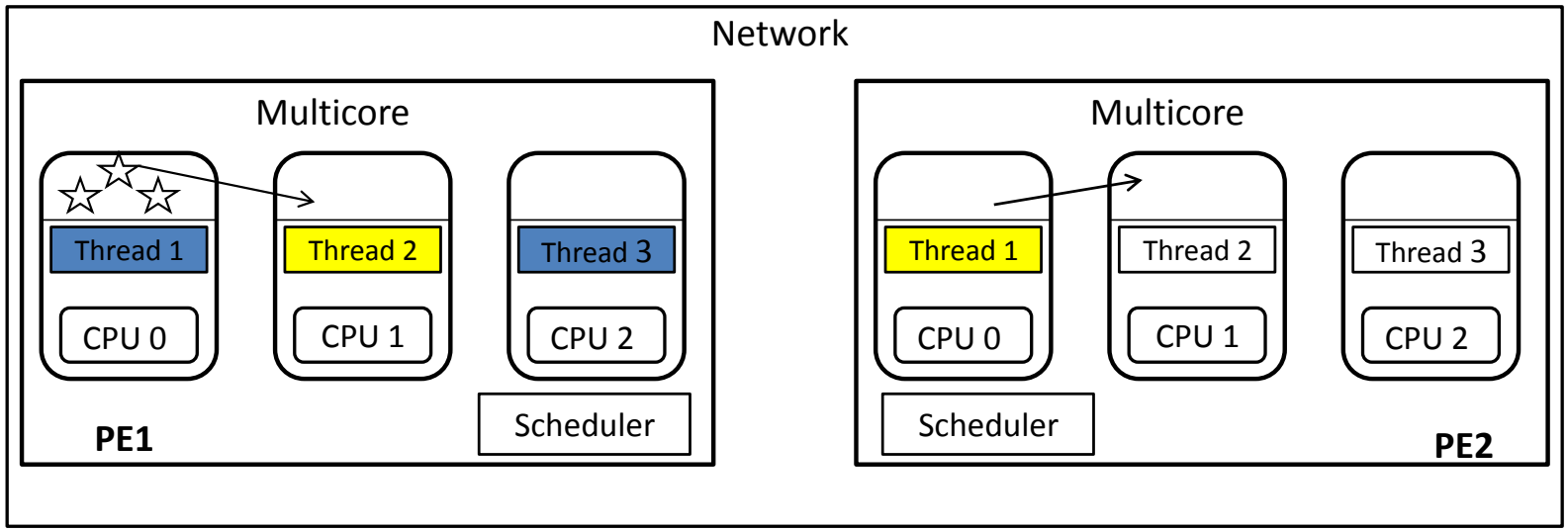
Load Balancing:

- Processor's Spark Pool is implemented as a bounded work-stealing queue.
- The owner can push and pop from one end of the queue without synchronization.
- Other threads can steal from the other end of the queue.



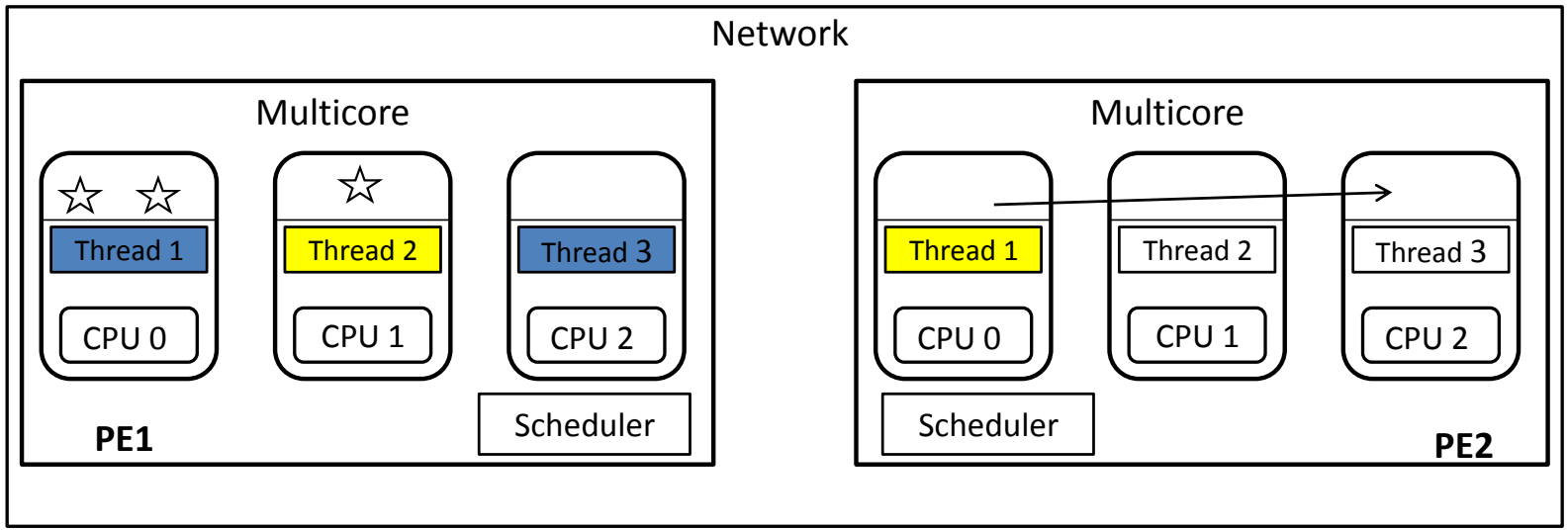
GUMSMP Work Distribution Mechanism

- Work distribution of GUMSMP is hierarchy aware.
- It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GHC-GUM).
- Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP).



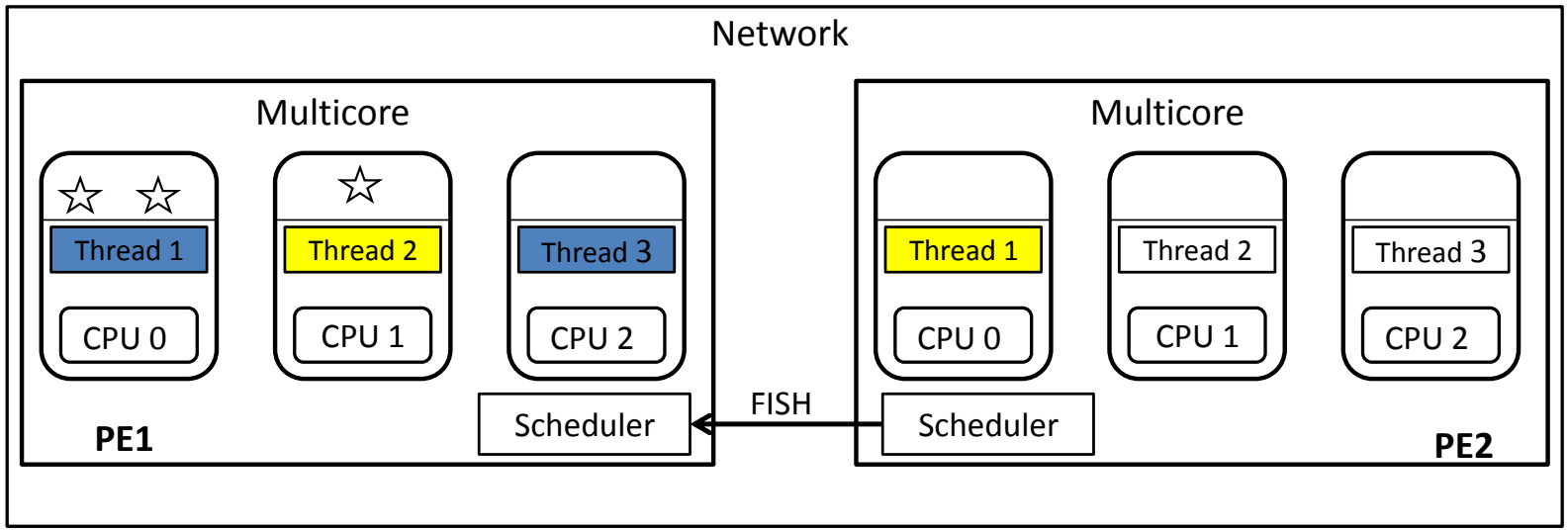
GUMSMP Work Distribution Mechanism

- Work distribution of GUMSMP is hierarchy aware.
- It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GHC-GUM).
- Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP).



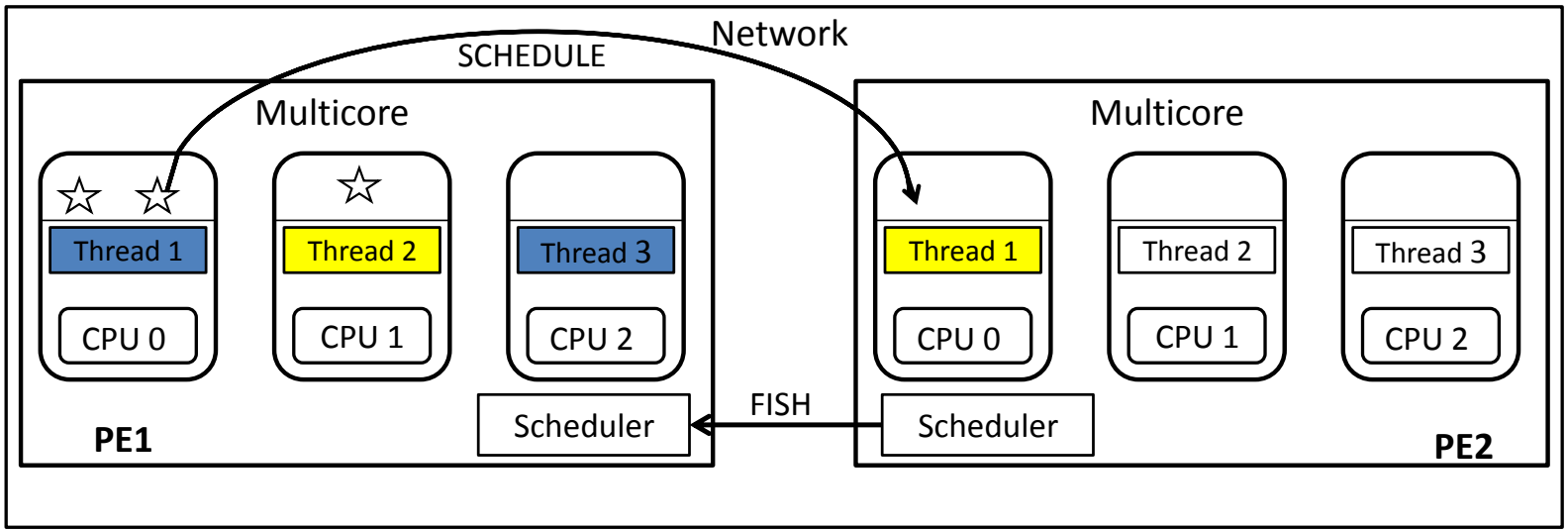
GUMSMP Work Distribution Mechanism

- Work distribution of GUMSMP is hierarchy aware.
- It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GHC-GUM).
- Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP).

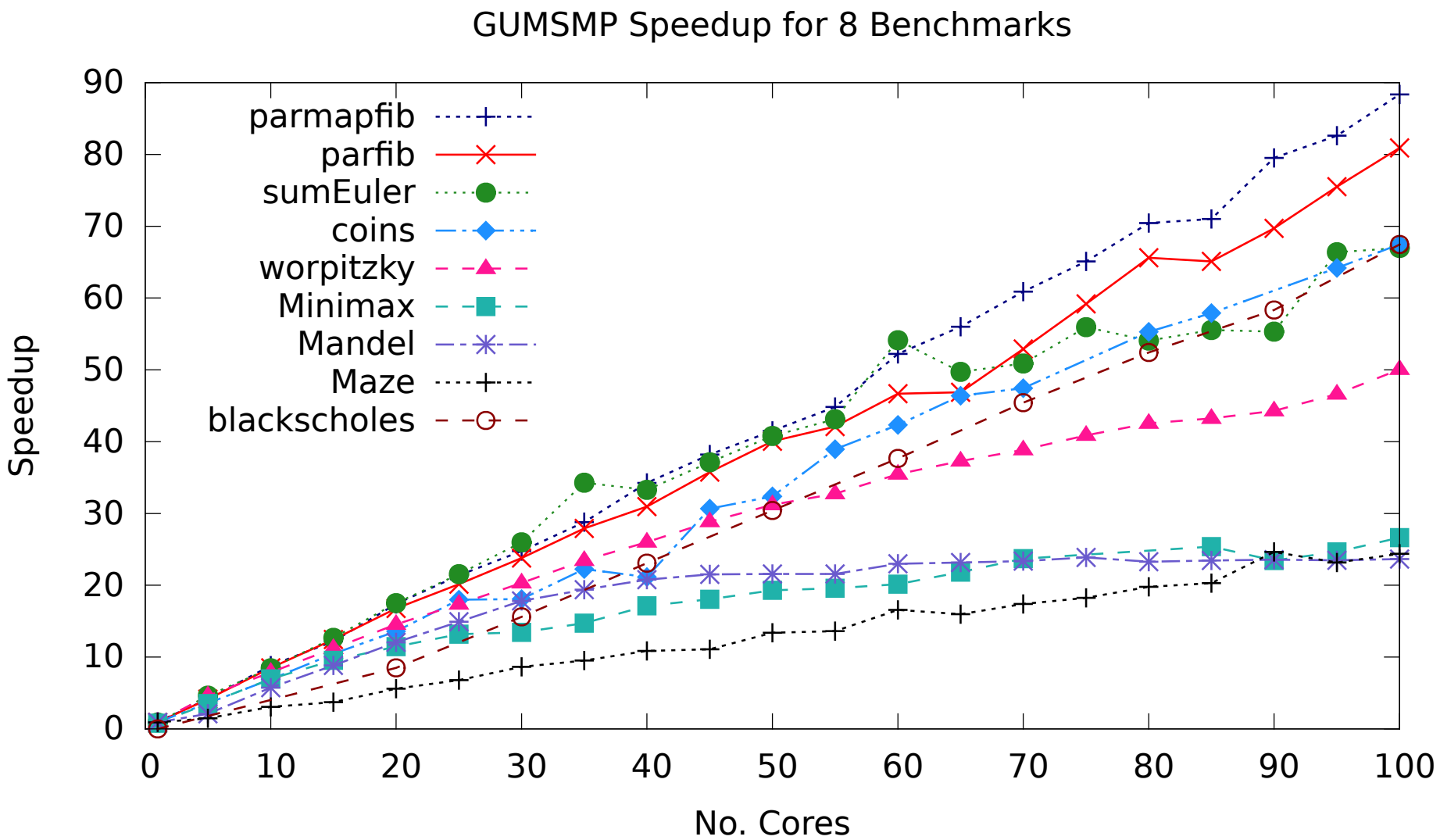


GUMSMP Work Distribution Mechanism

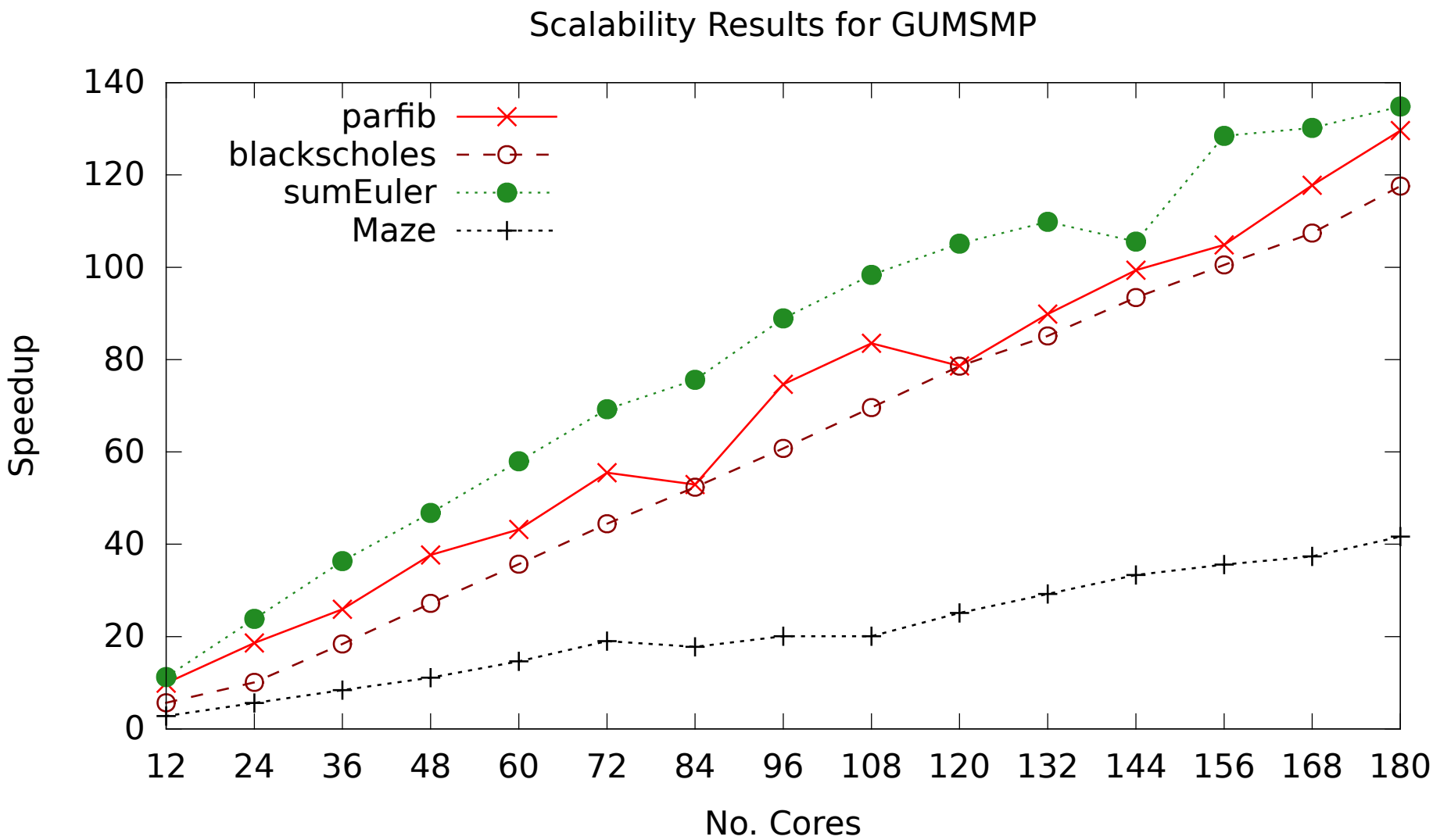
- Work distribution of GUMSMP is hierarchy aware.
- It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GHC-GUM).
- Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP).



Speedup Results on a Multicore cluster



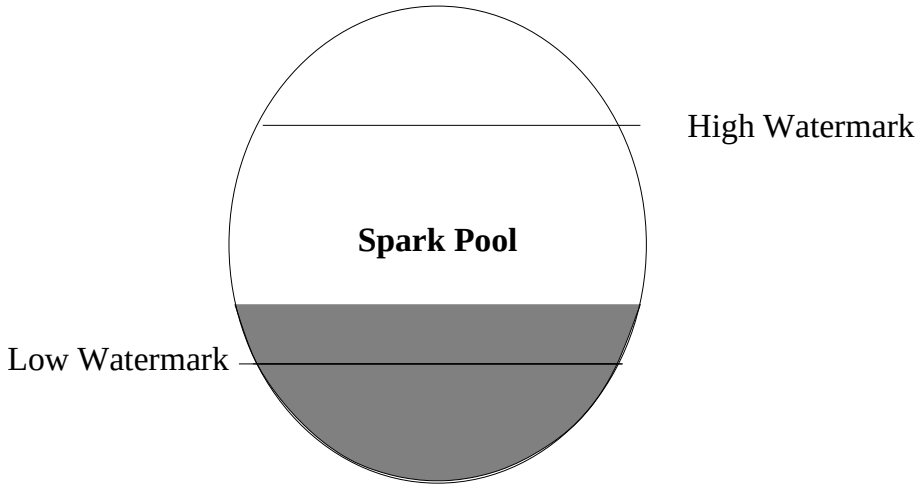
Scalability Results on a Multicore cluster



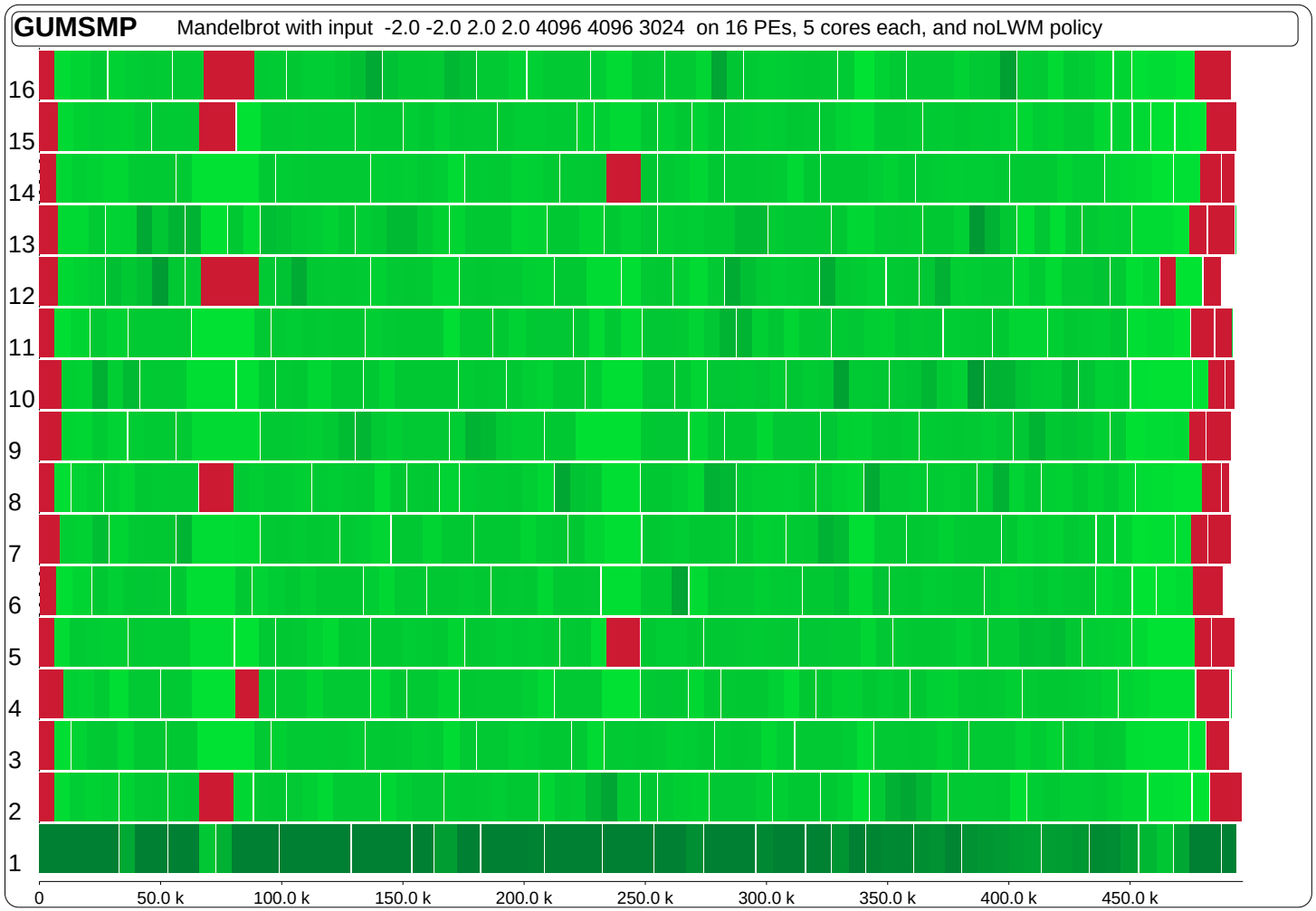
GUMSMP's Improved Work Distribution

We use **watermarks** for more flexible load balancing, with **pre-fetching**:

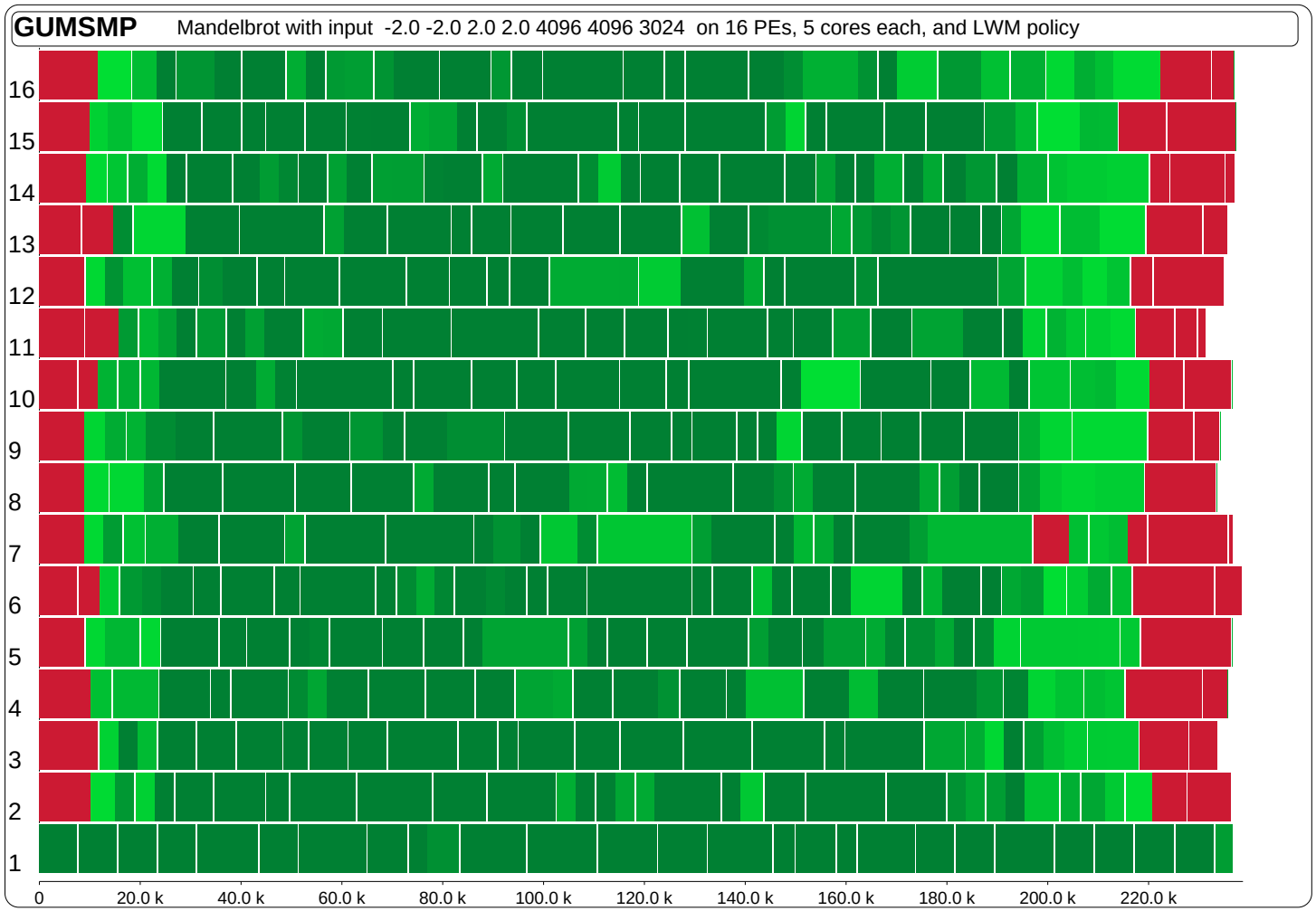
- The system aims to keep the spark pool size between low- and high-watermark.
- Below low-watermark: pre-fetch work from a another processor.
- Above high-watermark: off-load work to another processor.



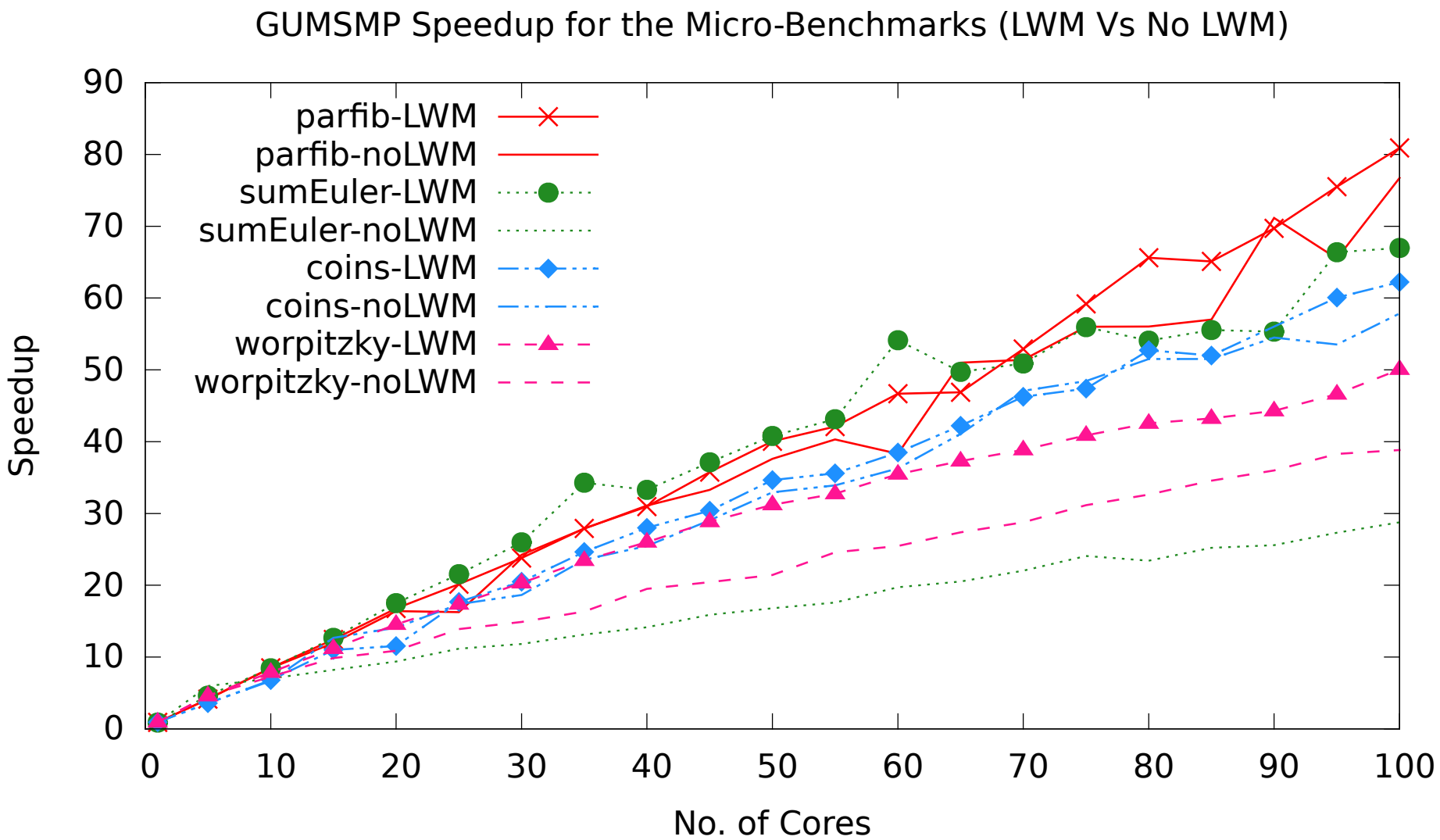
Load Balance **without** low watermarks



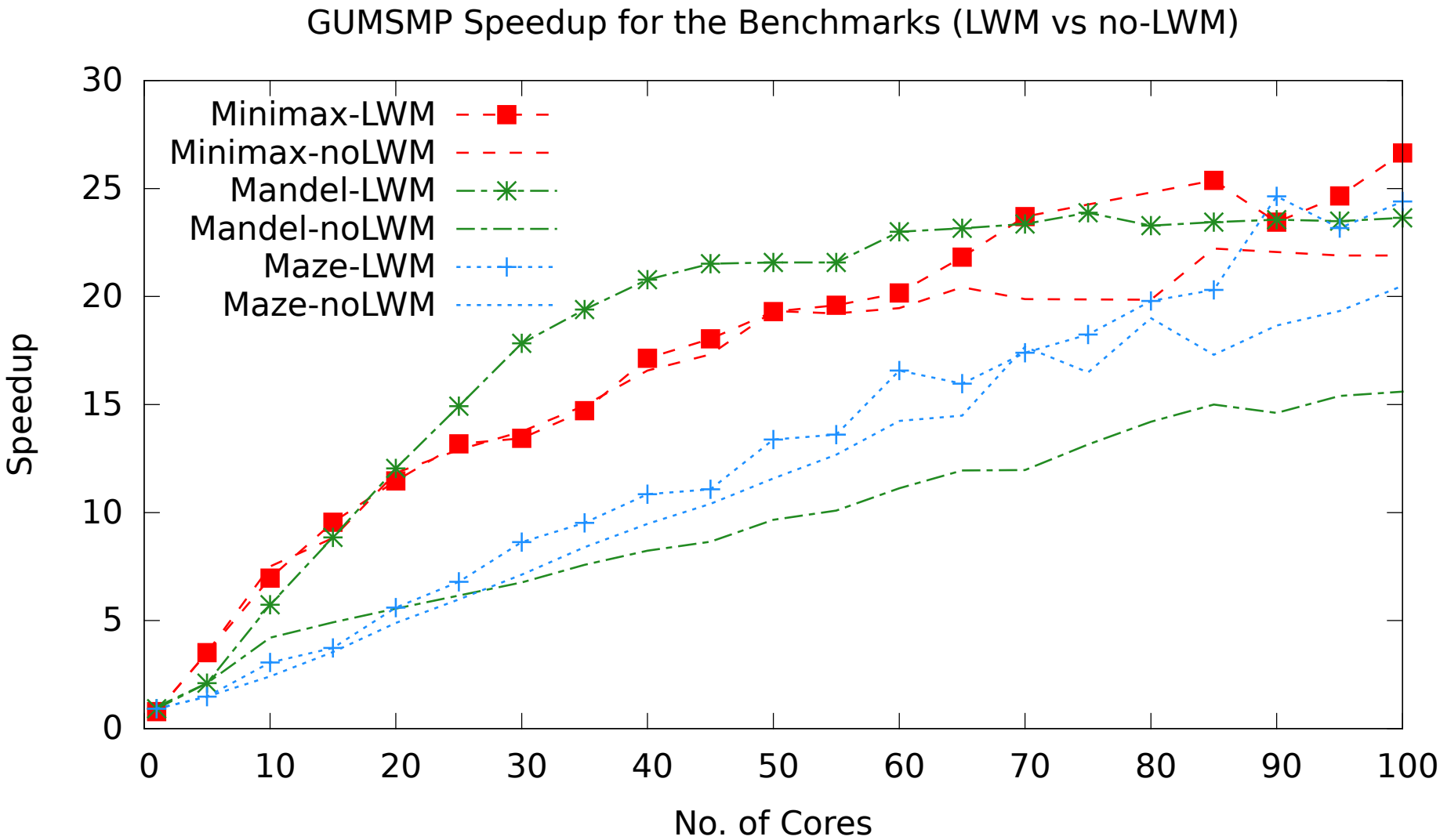
Effectiveness **with** of low watermarks



Speedups with and without low watermarks



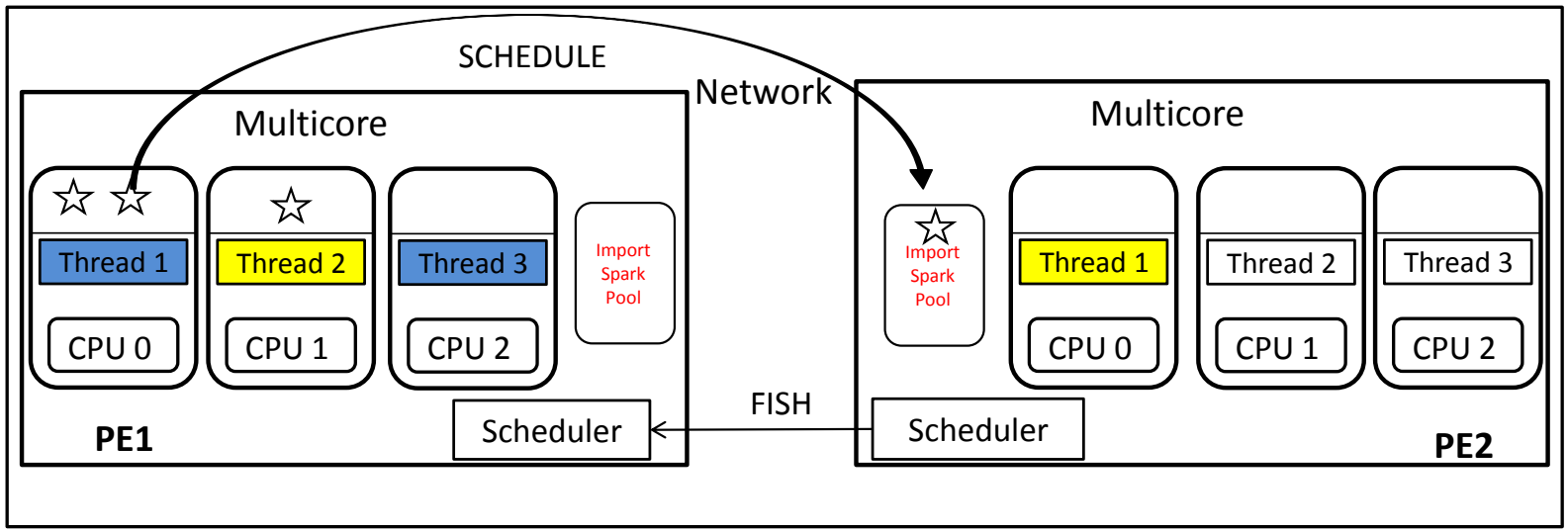
Low watermarks: load balance



GUMSMP's Improved Data Locality

We use **spark segregation** to distinguish work by origin:

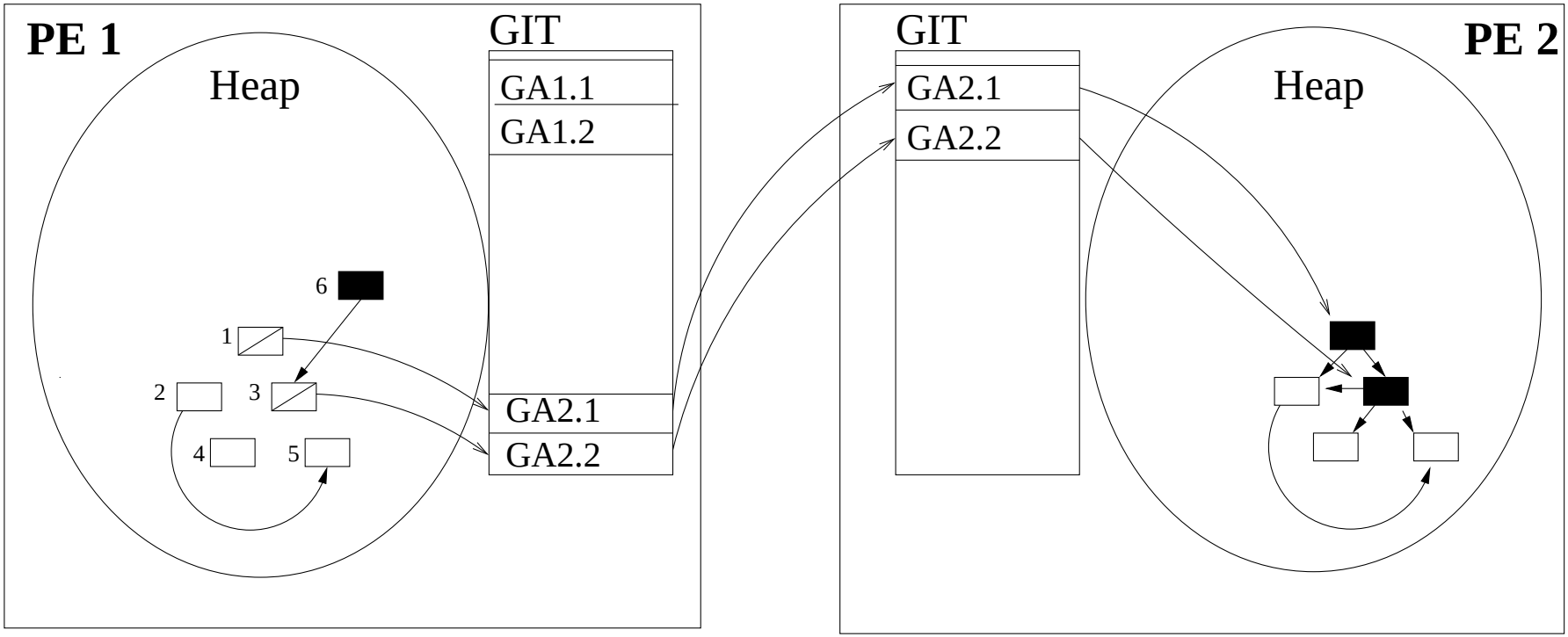
- Original GUM design: all sparks are equal
- Hierarchical GUMSMP design: use a separate **import spark pool** to segregate sparks received from other processors from local sparks
- Prefer either global or local sparks on export or thread creation (tunable).
- Intuition: prefer local sparks where possible, to tackle heap fragmentation.



Heap Fragmentation

- One problem in a virtual shared heap is **heap fragmentation**: **related data-structures** are on **different nodes** of the distributed system
- High heap fragmentation results in frequent messaging.
- We can measure heap fragmentation as the size of our internal GIT-tables.
- An **import spark pool** is designed to reduce heap fragmentation.
- Initial results show a reduction in the GIT-table sizes around 11%.

GUMSMP


PE 1
Heap
GIT
GA1.1
GA1.2
GA2.1
GA2.2
6
1
2
3
4
5
GIT
GA2.1
GA2.2
PE 2
Heap
Thunk (computation)
Fetchme (global indirection)
Normal Form (data)

Distrib. vs. Shared Heap on NUMA

- NUMA architectures pose a challenge to parallel applications.
  - Asymmetric memory latencies
  - Asymmetric memory bandwidth between different memory regions.

Memory access times between different NUMA regions[2]

node	0:	1:	2:	3:	4:	5:	6:	7:
0:	10	16	16	**22**	16	**22**	16	22
1:	16	10	**22**	16	**22**	16	**22**	16
2:	16	**22**	10	16	16	**22**	16	22
3:	**22**	16	16	10	**22**	16	**22**	16
4:	16	**22**	16	**22**	10	16	16	**22**
5:	**22**	16	**22**	16	16	10	**22**	16
6:	16	**22**	16	**22**	16	**22**	10	16
7:	**22**	16	**22**	16	**22**	16	16	10

- **Our goal:** compare the performance of

  parallel Haskell applications using

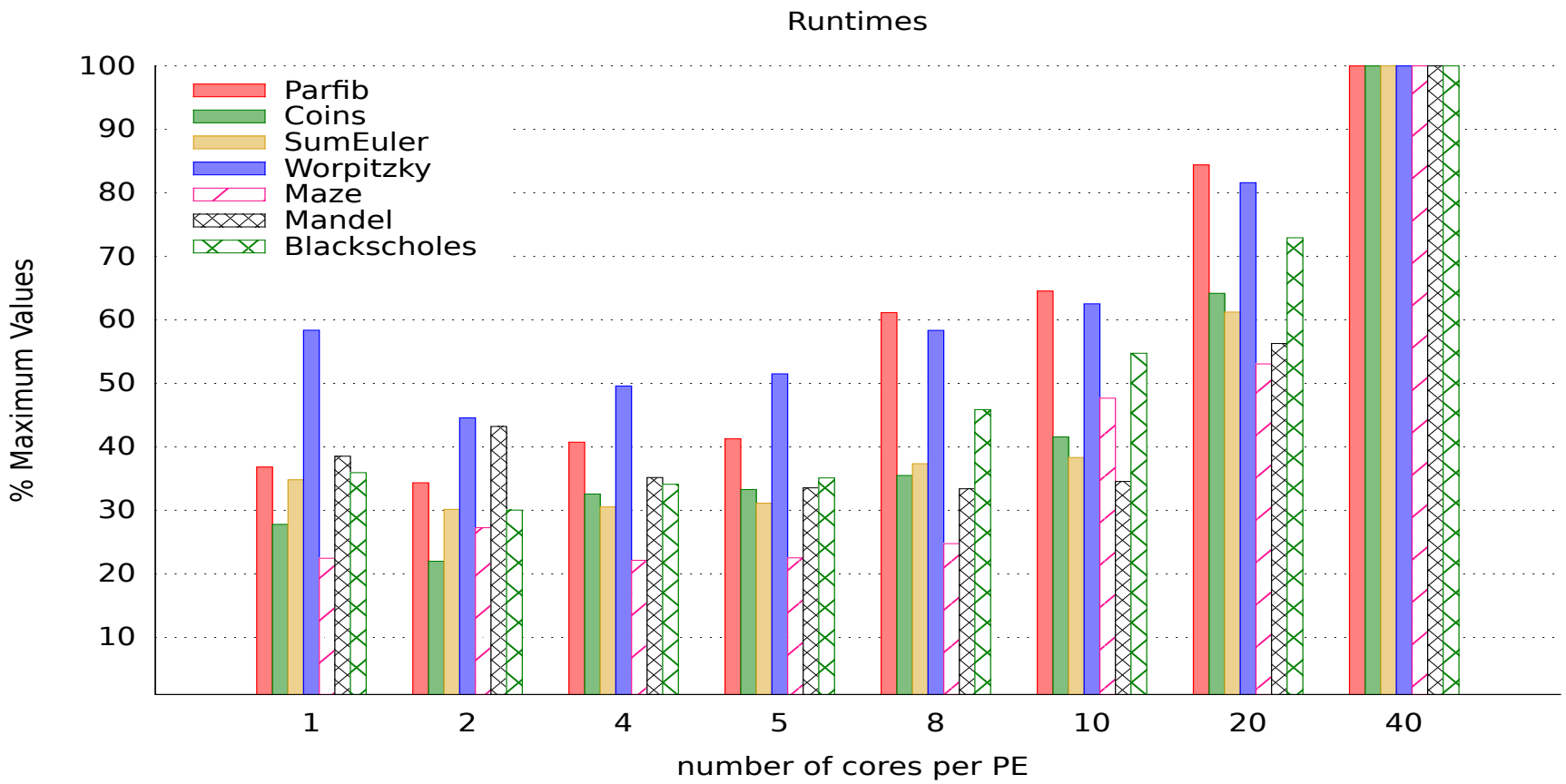
  shared memory vs. distributed memory systems on

  physically shared memory **NUMA** architectures.

[2] Measured using `numactl -H`

Performance results

- In each case, a **total of 40 cores** is used, and the difference is only in the number of cores that are used per PE.



Distrib. vs. Shared Heap on NUMA

On a 48-core, shared-memory NUMA architecture we observe:

- Improved runtimes with GUMSMP using 10+ cores, compared to GHC-SMP.
- Significantly improved performance with GUMSMP using **up to only 5 cores** per PE
- Drastic increase in GC percentage in GHC-SMP for large core numbers, due to a **larger live heap**.
- Lower allocation rate of GHC-SMP compared to GUMSMP, due to the **locking** of the first generation
- $\Rightarrow$ Using several small heaps, rather than one large heap, is consistently better
- $\Rightarrow$ Specifically, use **8 SMP-instances on 8 NUMA regions**

[2] see paper submitted to TFP14

Conclusion

- GUMSMP was designed for high-performance computation on multilevel architectures e.g. networks of multicores.
- One design goal is: **hierarchy aware load balancing**
- The main benefits of GUMSMP:
  - Scalable model
  - Efficient exploitation of distributed and shared memory on different levels of the hierarchy
  - Single programming model
- Improvements to work distribution mechanisms:
  - **Low Watermark**: reduces runtime by up to 57%.
  - **Spark Segregation**: ongoing work to reduce heap fragmentation.
- On clusters speedups between 40 and 135 on up to 180 cores.
- A distributed heap model is beneficial even on physical shared memory systems $\Rightarrow$ use **8 SMP-instances on 8 NUMA regions**

Ongoing Work

- Tune **spark segregation** to keep related data together (initial improvements of heap fragmentation around 11%).
- Evaluation of different spark select and spark export policies. In particular, study:
  1. The success rate of a policy.
  2. Its effectiveness in improving performance and heap fragmentation.
- Message Batching to reduce communication.

GUMSMP

Thanks for Listening ..
Questions?