# HIGH PERFORMANCE DATA STRUCTURES FOR MULTICORE ENVIRONMENTS

Giuliano Laccetti, Marco Lapegna
Department of Mathematics and Applications, Università degli Studi di Napoli Federico II (Italy),
Via Cintia Monte S. Angelo, 80126 Naples, Italy

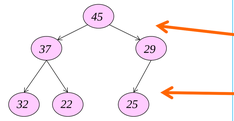
## ABSTRACT

The rise of multicore CPUs introduced new challenges in the process of design of concurrent data structures: in addition to traditional requirements like correctness and progress, the scalability is of paramount importance. It is a common opinion that these demands are partially in conflict each other, so that it is necessary to relax the requirements about a behavior identical to the corresponding sequential data structures in order to achieve high performances.

In this work we introduce a relaxed approach for the management of heap based priority queues on multicore CPUs, with the aim to realize a tradeoff between efficiency and sequential correctness. The approach is based on dynamic arrangement of the data structure among the cores, with a step-by-step redistribution procedure that shares information only among cores directly connected in a virtual mesh.

## HEAPS IN COMPUTATIONAL SCIENCE

- A heap is a dynamical data structures used when there is need to process high priority items (e.g. with large numerical error) produced at run time, with an order depending on the application data and that cannot be predicted.
- Each node is tagged with a priority higher than its children. The node with highest priority $e^*$ is in the root
- Two efficient operations are defined on a heap: remove(root) and insert(item)
- In many scientific applications, the heap is periodically updated in an iterative section of the algorithms




```
initialize data structure
while (stopping criterion == false) do
      do some work
      remove(max_priority_item)
      process data
      generate new items
      insert( new items)
      do some work
endwhile
```

## HEAP MANAGEMENT IN MULTICORE ENVIRONMENTS

two traditional approaches

**Centralized approach:**
All threads $P_i$ access a single data structure.
All basic operations must be carried out in a critical section.

**PRO**: correctness, access to item with highest priority, linearizability (same behaviour of the sequential data structure)
**CONS**: synchronization time depending on the number of threads, low performance, poor scalability

**Distributed approach:**
Each thread manages a private sub-structure.
Access to the data without syncronization.

**PRO**: high performance, high scalability
**CONS**: threads can process unimportant items if the priority are not fairly distributed, risk of no significant progress for the application

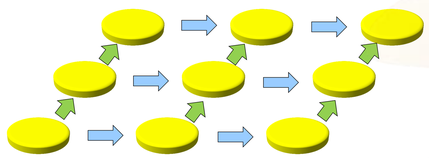

## A HIGH PERFORMANCE DATA STRUCTURE

local redistribution of highest priority items (relaxed approach)

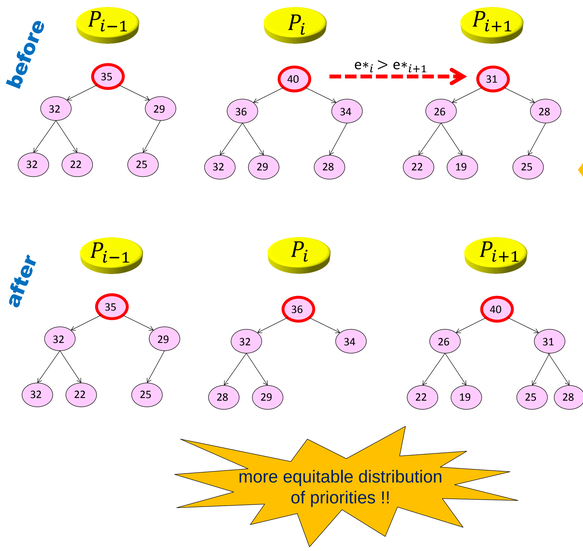
No centralized data structure!
*N* threads are logically organized according to a 2-dimensional periodical mesh $M_2$.

We define a High Performance Heap a collection of partially ordered binary trees $H_i$ with the max-heap property, where the roots are connected among them according the mesh $M_2$

Each thread $P_i$ manages a private sub-structure $H_i$. If $e*_i > e*_{i+1}$ then the item with largest priority $s*_i \in H_i$, is moved forward to a connected thread in the mesh $M_2$, according a producer-consumer protocol, alternatively in the two directions. In this way the critical items with highest priority are passed from thread in thread, an iteration after the other, through all nodes of the mesh, with a better distribution.



even iteration. odd iteration.




```
initialize private data structure   H_i
while (stopping criterion == false)  do iteration j
      define  DIR = mod(j,2)
      share  e*_i with the closest threads P_{i-1} and P_{i+1} along DIR
      if ( e*_i > e*_{i+1} ) then
            remove(max_priority_item) from H_i
            produce item  for P_{i+1}
      endif
      if ( e*_{i-1} > e*_i ) then
            consume  item produced by P_{i-1}
            insert(new item) in H_i
      endif
      remove(max_priority_item)
      process data
      generate new items
      insert( new items)
      do some work
endwhile
```

more equitable distribution of priorities !!

MORE INFO ? scan!

## SCALABILTY ANALYSIS AND TEST RESULTS

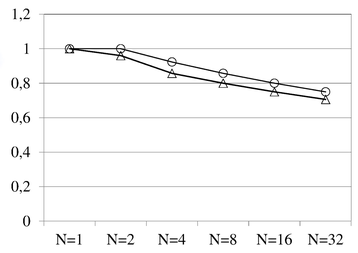
In the proposed algorithm, at each iteration, there are no global synchronizations among threads $P_i$, and each of them exchanges data only with the two threads $P_{i-1}$ and $P_{i+1}$, so that the synchronization overhead is $T_o = O(1) = const$, because it does not depends on the number of threads *N*

$$\text{Scal} = \frac{T(1,z)}{T(N,Nz)} = \frac{T(1,z)}{T(1,z)+T_o} = const$$

**distributed approach vs relaxed approach**

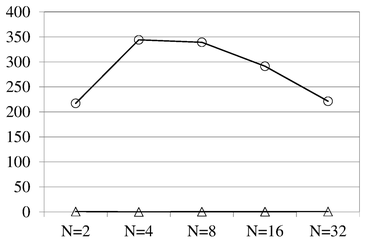
- The private structures are initialized with items with random piorities $e*_i < 10000$
- At each iteration the heap root is removed and it is replaced with two items with priority $e \in [\frac{e*_i}{2}, e*_i]$
- Each node is of 128 byte of data

- test executed on a system with 4 CPUs Intel Xeon E5-4610 v2 with 8-core (Harpertown) running at 2.33GHz, for a total of 32 cores
- Scientific Linux 6.2 operating system ; GNU C compiler; POSIX thread library




Scaled efficiency with a number of threads from N=1 through N=32.
○= without redistribution
△= with redistribution

same performance !




Standard deviation of the highest priorities in each thread, with a number of threads from N=1 through N=32.
○ = without redistribution
△ = with redistribution

better distribution !