12th International Conference on Parallel Processing and Applied Mathematics

PPAM 2017

Lublin, Poland
September 3-6, 2017

Fourth workshop on Models, Algorithms and Methodologies for Hybrid Parallelism in new HPC Systems

Relaxing the correctness conditions on concurrent data structures for multicore CPUs: a numerical case study

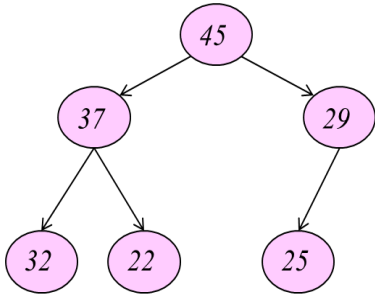
Giuliano Laccetti[1], Marco Lapegna[1], Valeria Mele[1], Raffaele Montella[2]

1) depart. Mathematics and Applications – Univ. of Naples Federico II
2) depart. Science and Technologies – Univ. of Naples Parthenope

Aims of the talk

In this talk we introduce an approach for the management of
heap based priority queues on multicore CPUs

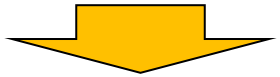
a heap H is a partially ordered binary tree where
each node has a priority higher than its children, so that
the item with highest priority e^* is in the root
(the so called max-heap property).



Dynamical data structure where the
sequence of items with high priority is unpredictable

Heap based priority queues

Heaps are useful when an application needs set of data not requiring a complete ordering, but only the access to some items tagged with high priority.



indispensable tools in almost every scientific field

- operating systems (process selection in scheduling algorithms)
- scientific computing (error reduction in numerical algoritms)
- big data (affinity maximization in clustering algorithms)

computational cost of the basic operations (remove and insert)

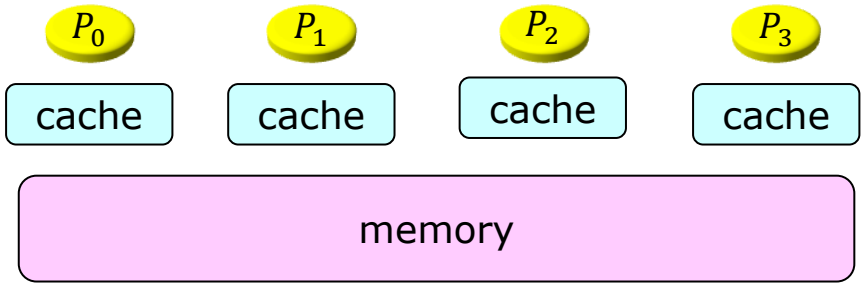
$$T(k) = \log_2(k)$$

```
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
```

general framework for a heap based algorithm

computing and performance model

- N processing elements (the cores)
- assume 1 thread for each core (i.e. N concurrents thread P_i)
- Local caches an a shared main memory



- $T(N,z)$ = total elapsed time to complete a task with dimension z using N threads
- Define scaled efficiency (weak scalability) the ratio

$$R(N,z) = \frac{T(1,z)}{T(N,Nz)}$$

the scaled efficiency measures the ability of the algorithm to solve a N-times larger problem with N concurrent threads in the same time

ideal value: $R(N,z) = 1$

computing and performance model

$T(N,z)$ can be decomposed as

$$T(N,z) = T_s + \frac{T_c(Z)}{N} + T_o(N)$$

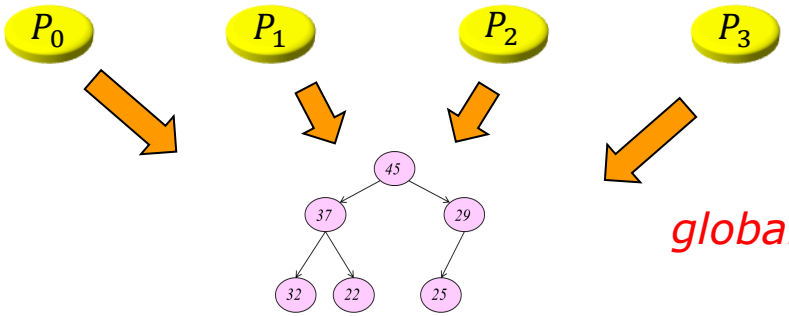
- T_s is the running time for the serial section of the algorithm. It is independent on the problem size z
- $T_c(N)/N$ is the running time for the parallel section of the algorithm. It is composed by N concurrent tasks of equal running time
- $T_o(N)$ is the synchronization overhead. It is a not decreasing function of the number of threads N

$$R(N,z) \leq \frac{T(1,z)}{T(1,z) + T_o(N)} \leq 1$$

First solution (centralized heap)

All threads access a single heap in the shared memory
with the aim to share the items with highest priority

All basic operations on the heap must be carried out in a critical section



global synchronization !!

N threads entering M times in a critical region have an overhead

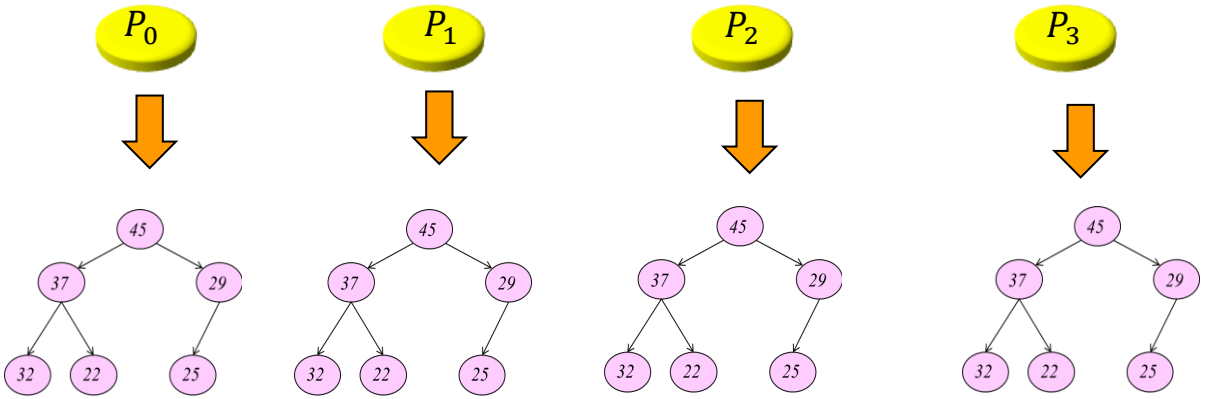
$$T_o(N) = M(N-1)t_c = O(N)$$

$$R(N,z) \leq O\left(\frac{T(1,z)}{T(1,z)+N}\right)$$

very poor scalability !!

Second solution (distributed heap)

N separated data structure, one for each thread,
each of them maccessing its private data structure



no synchronization among threads (natural parallelism) !!!

synchronization overhead
$T_o(N) = const$

$$R(N,z) \leq O\left(\frac{T(1,z)}{T(1,z) + const}\right)$$

perfect scalability !!

BUT...

main problem of the second solution

Dynamical data structures are used for dynamic problems, where the sequence of items with high priority is unpredictable, and it is impossible to distribute them uniformly before the computation

In case of items with priority very poorly distributed among the heaps, there is a high risk that the items with the highest priority in each heap, are not those that globally have the highest priority, so that some threads can process unimportant items with no significant progress for the whole application.

In this cases it should be desirable a periodical redistribution strategy in order to balance the critical items among the threads.

a contraposition

Centralized approach:
All threads access a single data structure.
Basic operations 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

an effective trade off: a relaxed approach

1: thread organization

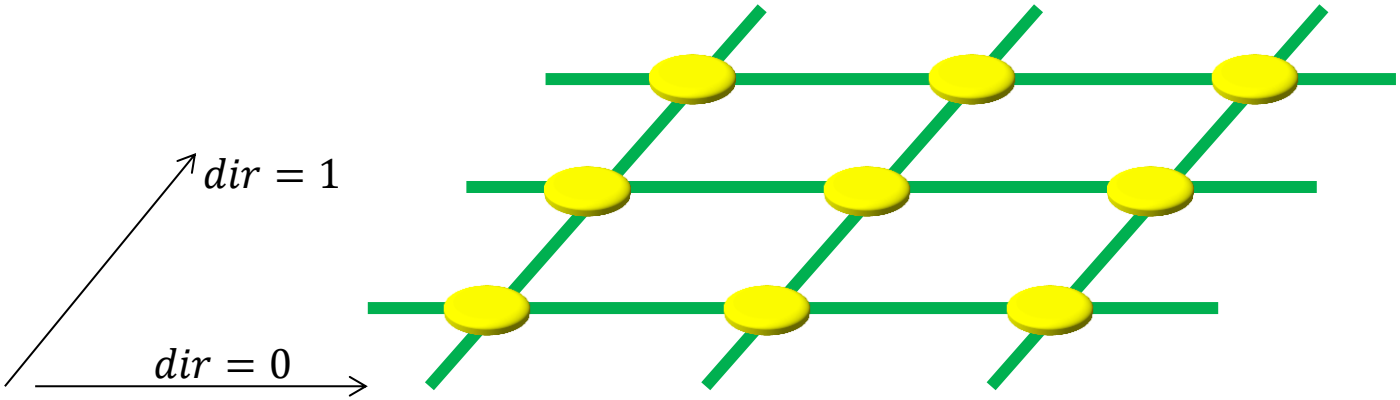
N threads P_i are logically organized according a 2-dimensional periodical mesh M_2

in each direction P_i has 2 neighbors:

- In the horizontal direction ($dir = 0$) P_{i-} and P_{i+} are respectively the leftmost and the rightmost thread of P_i in M_2
- In the vertical direction ($dir = 1$) P_{i-} and P_{i+} are respectively the lowermost and the uppermost thread of P_i in M_2

A shared buffer between each couple of connected nodes is established. The buffer is used to allow exchanging data according a producer-consumer protocol

the threads on the opposite faces of the mesh are connected, so that the mesh is periodical



a effective trade off: a relaxed approach

2. redistribution procedure

We define a loosely coordinated heap (relaxed heap) a collection of N partially ordered binary trees H_i with the max-heap property, where the roots are connected among them according the mesh M_2

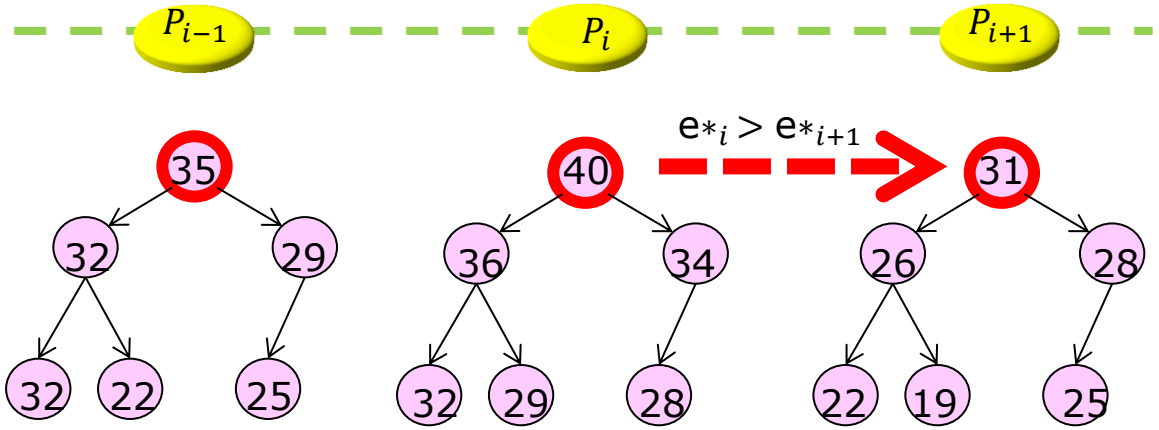
Each thread P_i manages a private sub-structure H_i.

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.

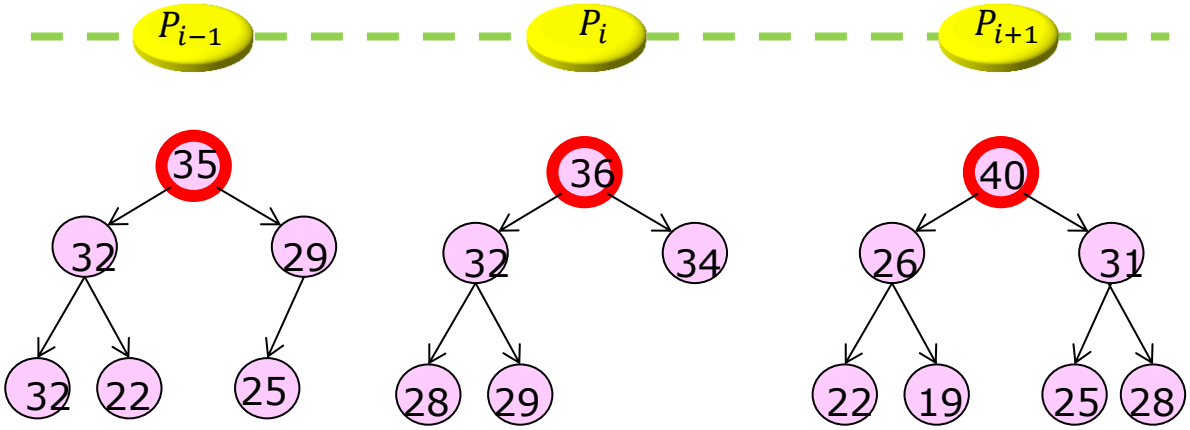
an example

before



$\sigma = 6.3$

after



$\sigma = 2.8$

In general, under little restrictive hypotheses, it is possible to show $\sigma_{before} > \sigma_{after}$

3: The algorithm (SPMD programming model)

```
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
```

Redistribution procedure

a effective trade off: a relaxed approach

4. efficiency

In the proposed algorithm there are no global synchronizations among the threads, and each of them exchanges data only with two connected threads P_{i-} and P_{i+} , so that the synchronization overhead is $T_o(N) = O(1) = const$,

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

the informations about the items with high priority are moved from thread to another one along the 2 directions of the mesh

Such informations reaches all threads after a number of iterations equal to the maximum distance between any pair of threads (the so called diameter of the mesh $D(M_2)$

The diameter of a 2-dimensional periodical mesh is $D(M_2) = 2\left(\sqrt{N} - 1\right)$ when the mesh is symmetric and N is a perfect square

$D(M_2)$ is a slow-growing function of N ensuring a fast distribution of critical items even with large values of N

a parallel adaptive algorithm for multidimensional quadrature

$$I(f) = \int_U f(t_1, \ldots, t_d) dt_1 \cdots dt_d$$

$U = [a_1, b_1] \times \cdots \times [a_d, b_d]$ d-dimensional rectangular region

Algorithm:

- Iterative algorithm refining a partition of U that at each step compute

$$I(f) \cong R^{(j)} = \sum_{s(k) \in P} r(k)$$

$$E^{(j)} = \sum_{s(k) \in P} e(k)$$

P is a partition of U
$r(k)$ is a quadrature rule
$e(k)$ is an error estimate procedure

with the aim

$$\lim_j R^{(j)} = I(f) \qquad \lim_j E^{(j)} = 0$$

the subdomains will be smaller where the error estimate is larger

parallel adaptive algorithm

the convergence rate of this procedure depends on the behavior of the integrand function (presence of peaks, oscillations, etc)

in order to reduce as soon as possible the error, the algorithm splits in two parts the subdomain S^* with maximum error estimate e^*.

The two new subdomains take the place of S^* in P, and in a similar way the approximations $Q^{(j)}$and $E^{(j)}$ are updated.

A natural implementation of a such procedure can be done with a priority queue with the max-heap property, where the nodes of the heap H contain the subdomains of the partition P, and where the priority is represented by the error estimate $e(k)$ in each subdomain.

The subdomain to be split at the iteration j, with maximum error estimate e^* is in the root of the tree.

For such a reason, to implement the algorithm in a multicore based computing environment, it is possible to use the loosely synchronous approach for the heap management.

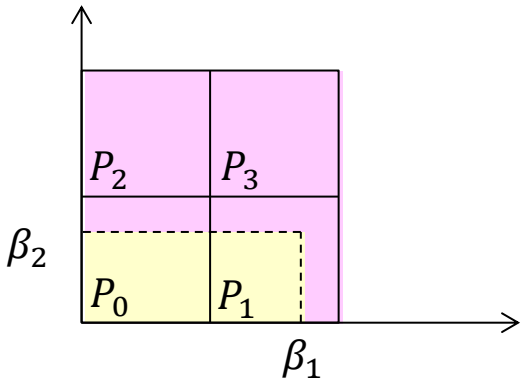
test function and computing environment

test function

$$f(x) = \begin{cases} 0 & if\ x_1 > \beta_1\ \ or\ \ x_2 > \beta_2 \\ \exp(\alpha_1 x_1 + \cdots + \alpha_1 x_1) & otherways \end{cases}$$

α_i *and* β_i are random values (10 functions)

strong discontinuity (and large error)
in the subdomains containing
the edge of the yellow region



es. with d=2 dimension

computing environment

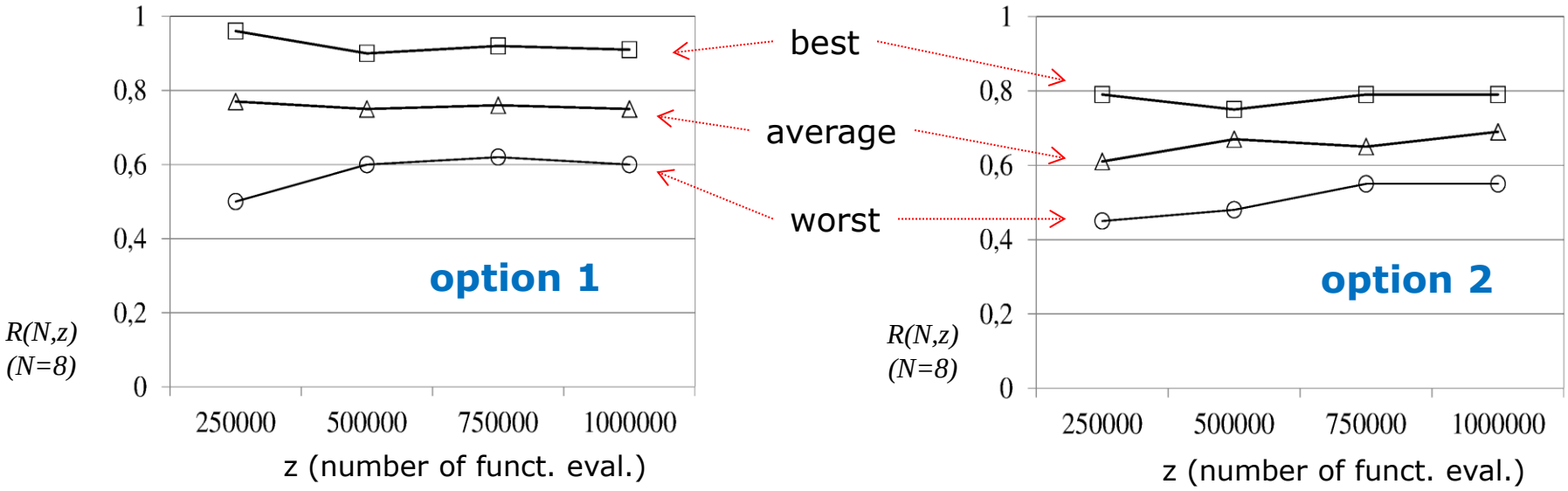
- Intel Xeon E5-4610 v2 with 8 core @2.33 GHz
- DDR3 256 GB shared memory
- Scientific Linux 6.2 OS
- C compiler with POSIX Thread Library

* Image is a representation of this product

aim: to measure the scaled efficiency

Option 1. Without redistribution procedure: the integration domain U is equally distributed among the N threads P_i and the calculation goes on without interaction among threads. In this case any difficulties in the integration domain are not shared among the threads.

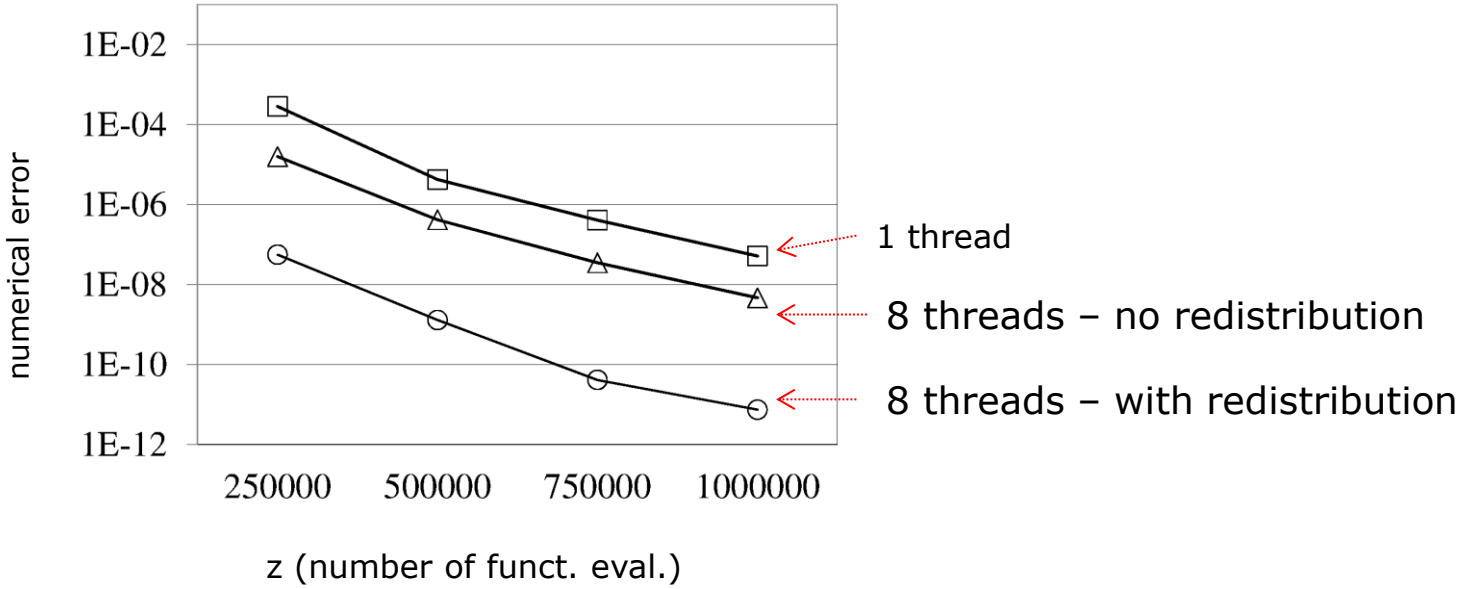
Option 2. With redistribution procedure: after the same distribution of U among the threads, the computation attempts to balance the work load among the local data structures H_i of the loosely coordinated heap H. In this case the difficulties in the integration domain are shared among the threads.



second set of experiments

aim: to measure the benfit of the proposed redistribution procedure on the accuracy of the results.

This is a critical experiment because it is tested the ability of the loosely coordinated heap H to supply effectively high-priority items to the threads



conclusions

We proposed a **relaxed model for heap-based priority queues in multicore environments**. The work is motivated by the need to achieve a balance between two contrasting requirements on the data structure: **correctness and scalability**.

To this end, we have developed an approach based on a **distribution of the data structure** among the computing units. At the same time we introduced **a periodical redistribution** of the high-priority nodes, based on a **synchronization strategy** involving only a small (and constant) number of processing units.

Our experiments show that such a strategy is able to realize an **effective compromise between the two requirements**.