HLPP 2003, Paris, France

ESkIMO

an Easy Skeleton Interface (Memory Oriented)

Marco Aldinucci

Computer Science Dept., Pisa, Italy

www.di.unipi.it/~aldinuc/

Outline

- Motivations
- Programming model
- (Some) experimental results
- The payback of the approach
- if (elaps. time<30min)
  development issues

Motivations

- We developed several skeletal frameworks, both academic and industrial:
  - P3L (Uni Pisa , 1991, C)
  - SkIE (Uni Pisa + QSW ltd., 1998, C++, Fortran, Java)
  - Lithium (Uni Pisa, 2002, Java based, macro-data-flow)
  - ASSIST (Uni Pisa + Italian Space Agency, 2003 ?, GRID-targeted (not GREED)
  - Many variants of them
- Many “real world” applications developed with these frameworks:
  - Massive data-mining, computational chemistry, numerical analysis, image analysis and processing, remote sensing, ...

Lack of expressiveness

- “missing skeleton” problem
- skeletons as “pure” functions
  - enable static source-to-source optimizations, but
  - how to manage large data-sets, possibly accessed in a scattered, unpredictable way?
  - primary targeted to speedup (memory?, bandwidth?)
- No support for dynamic data structures
  - neither for “irregular” problems (B&B)
  - hierarchical organized data (C4.5 classificator ...)

ESkIMO approach

- Mainly a library to experiment solutions to scheduling and mapping
  - for the framework developer more than app dev
- Extend the C language with skeletal ops
- Layered implementation
  - Based on Soft-DSM (exploiting DAG consistency)
  - Targeted to loosely coupled architectures (NUMA)
  - Exploiting multiprocessing (inter-PEs), multithreading (intra-PE), MMX/Altivec fine grained SIMD/vectorial parallelism within the runtime (Intel performance libs / Apple gcc port)
  - Working on Linux/Pentium and PPC/MacOs X equipped with TCP/IP net (homogeneous)

eskimo provides abstraction 1

- on the programming model
  - parallel entities (e-flows)
    - share the memory
    - not limited in number
    - number not fixed at the program run (as in MPI)
  - skeletal operations
    - native foreach (on several data structures)
    - Divide&Conquer
    - ad hoc parallelism (pipes, sockets, ...)

eskimo provides abstraction 2

- on data structures (ADT)
  - seen as single entities (as Kuchen lib)
  - shared among e-flows
  - spread across the system
  - static and dynamic
    - native k-trees, arrays and regions
    - any linked data structure by means of references in the shared address

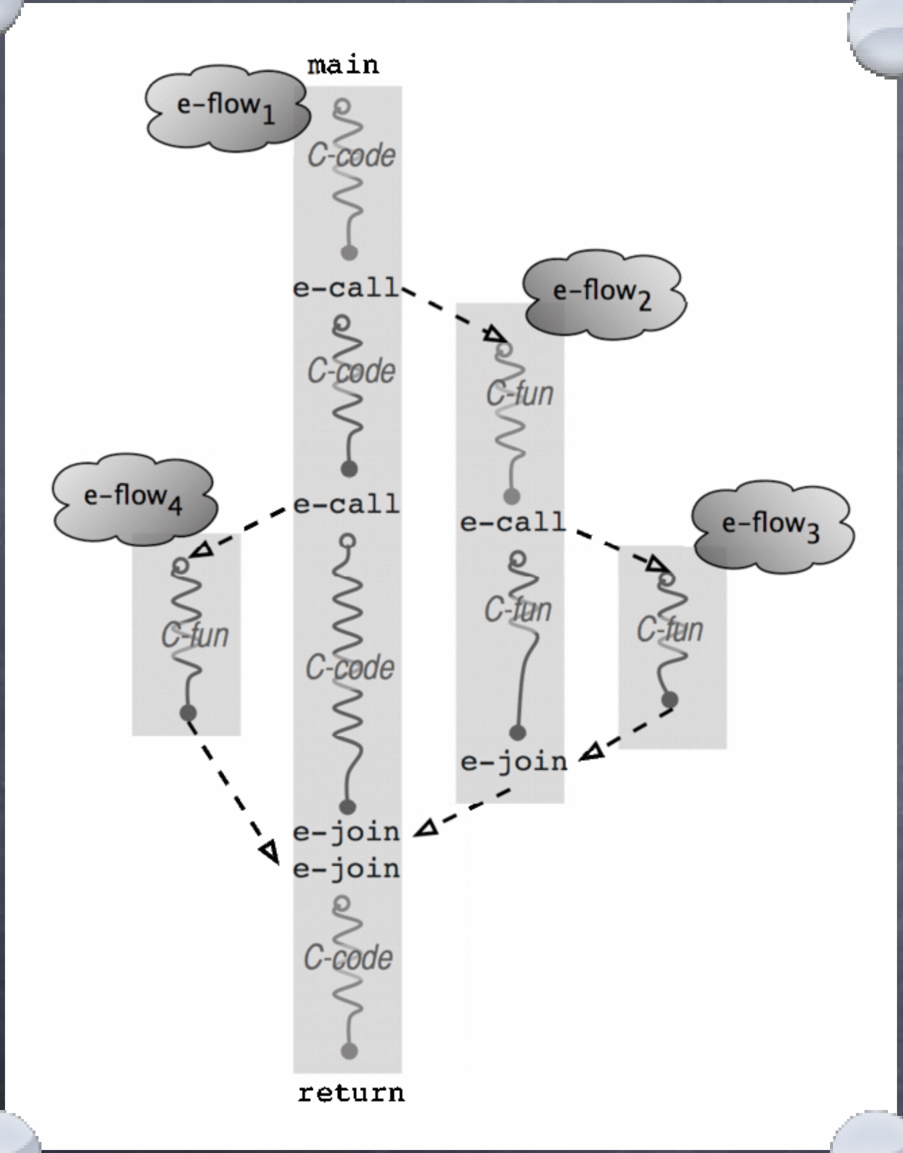
eskimo programming model

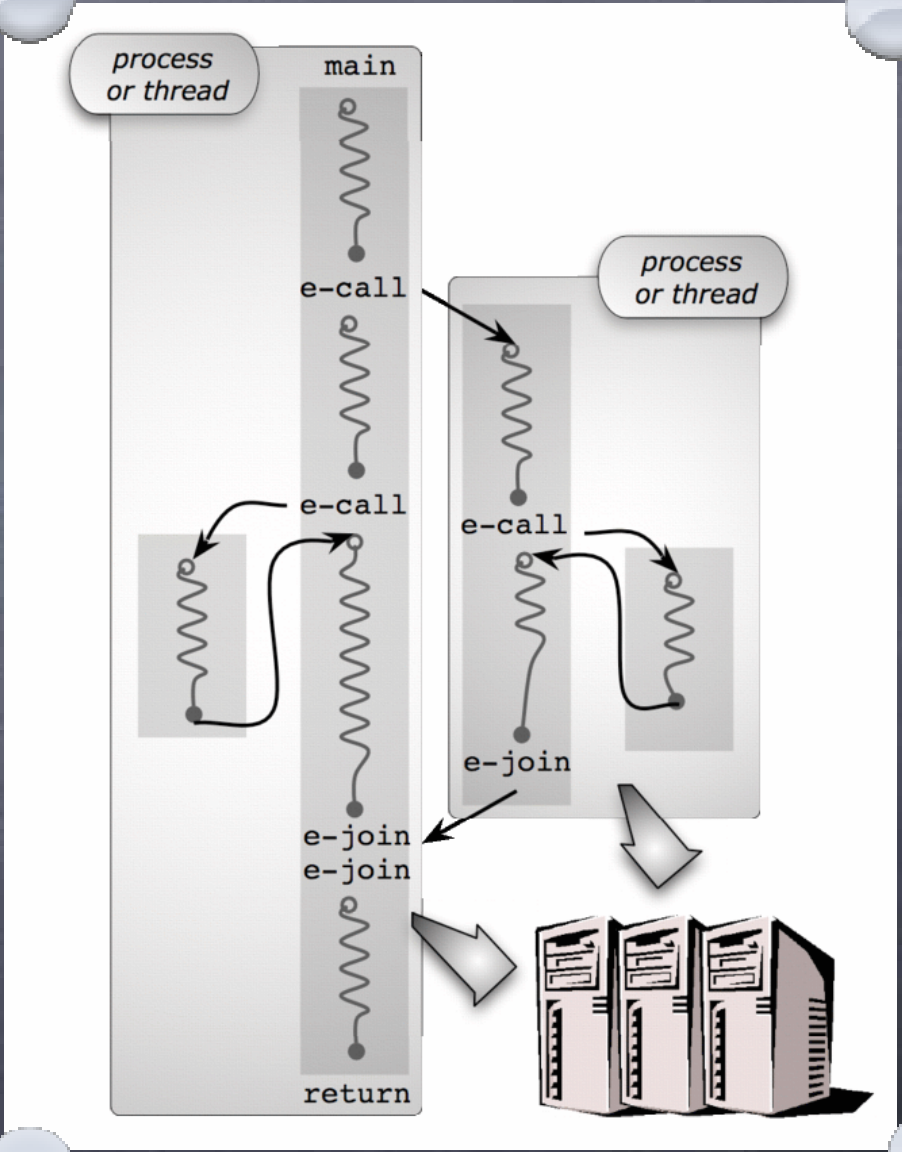
- Programs start with a single flow
- The flow may be split (then joined) with fork/join-like constructs: e_call and e_join

- These constructs originate C fun instances, i.e. e-flows
- e-flows are not processes/threads but abstract entities
  - rather, they are similar to *Athapascan* tasks (JL. Roch et al.)
  - bound to PEs once created (spawned)

- e-flows have a private and a shared memory:
  - private is HW accessed
  - shared memory accesses are software mediated

eskimo e-flows and their execution

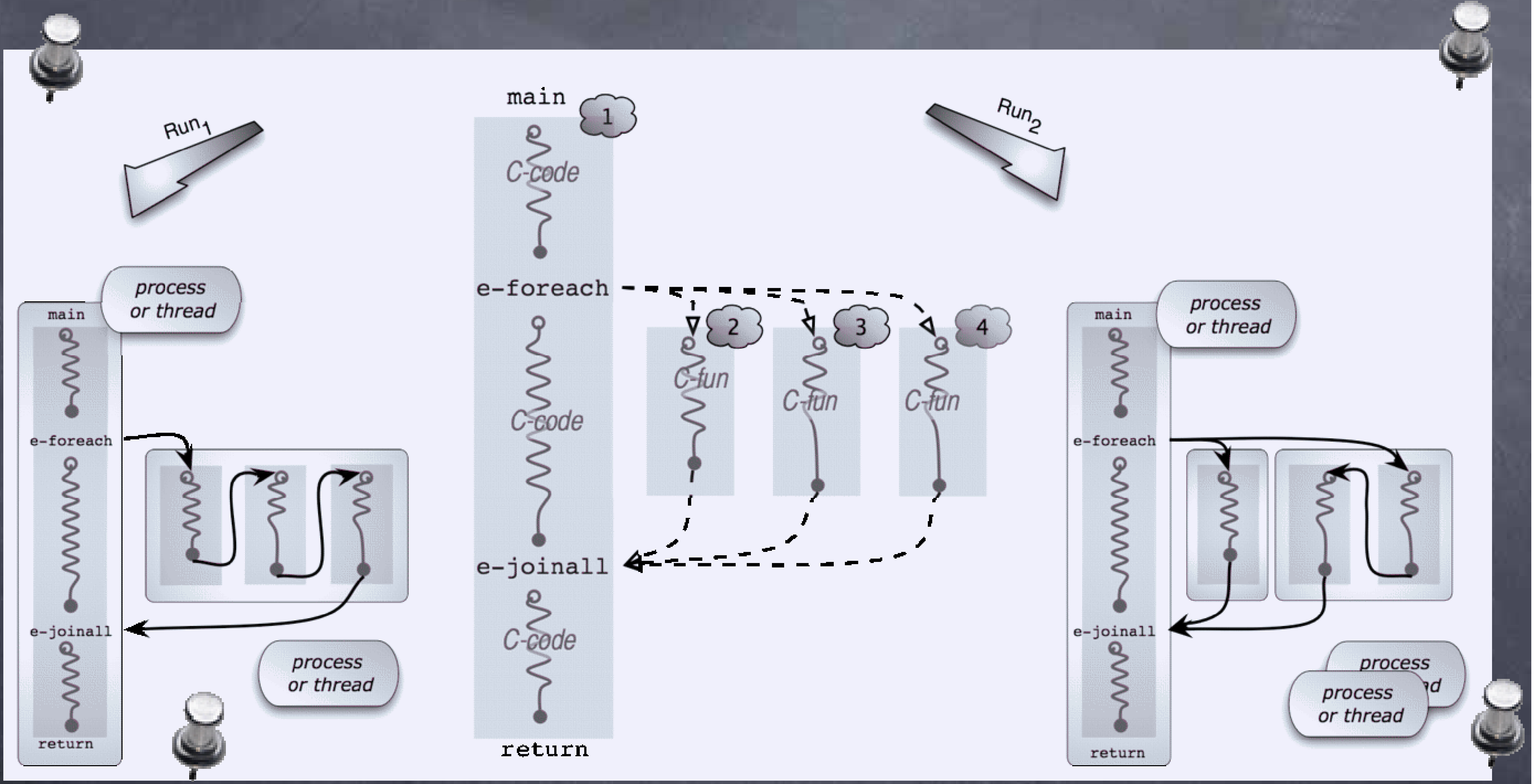




foreach/joinall

- n-way extensions of e_call/e_join
- work on
  - arrays
  - k_trees (e_foreach_child)
  - generic set of references (e_foreach_ref)

Different runs -- same program/data



eskimo data structures

- SADT (Shared Abstract Data Types)
  - simple parametric types,
  - may be instanced with any C type to obtain a SDT
  - SDT typed variables are shared variables
  - C standard vars are private, global/static forbidden within e-flows
  - sh. vars may grow beyond (logical) address space of the platform
- They are:
  - k-trees (because we know the acc. patterns)
  - lists = 1-trees, graphs = spanning tree + refs
  - arrays and regions ... lists = 1-trees, graphs
- In addition:
  - references, addresses in shmem: **`eref_t`**
  - handlers, in order to match call/join: **`ehandler_t`**

Example: a couple of binary trees

```
edeclare_tree(binary_tree_t, int, 2);

binary_tree_t t1 = TREE_INITIALIZER;
binary_tree_t *t2;
t2=(binary_tree_t *)malloc(sizeof(binary_tree_t));
…
etree_init(t2);
```

This yields two shared/spread empty trees t1 and *t2

These can be dynamically, concurrently populated with nodes by using **enode_add** or either joined, split ...

Trees: example

```
typedef struct {
  int foo;
  eref_t next;  //The head of a list for example
} list_cell_t;

sh_declare_tree(bin_tree_ll_t,list_cell_t,2);
bin_tree_ll_t t1 = TREE_INITIALIZER;
eref_t node,root;

root = eadd_node(bin_tree_ll,E_NULL,0); // the root
node = eadd_node(bin_tree_ll,root,0);   // its child
node = eadd_node(bin_tree_ll,root,0);   // another one
```

Reading and writing the shared memory

- A shared variable cannot r/w directly
- It must be linked to a private pointer

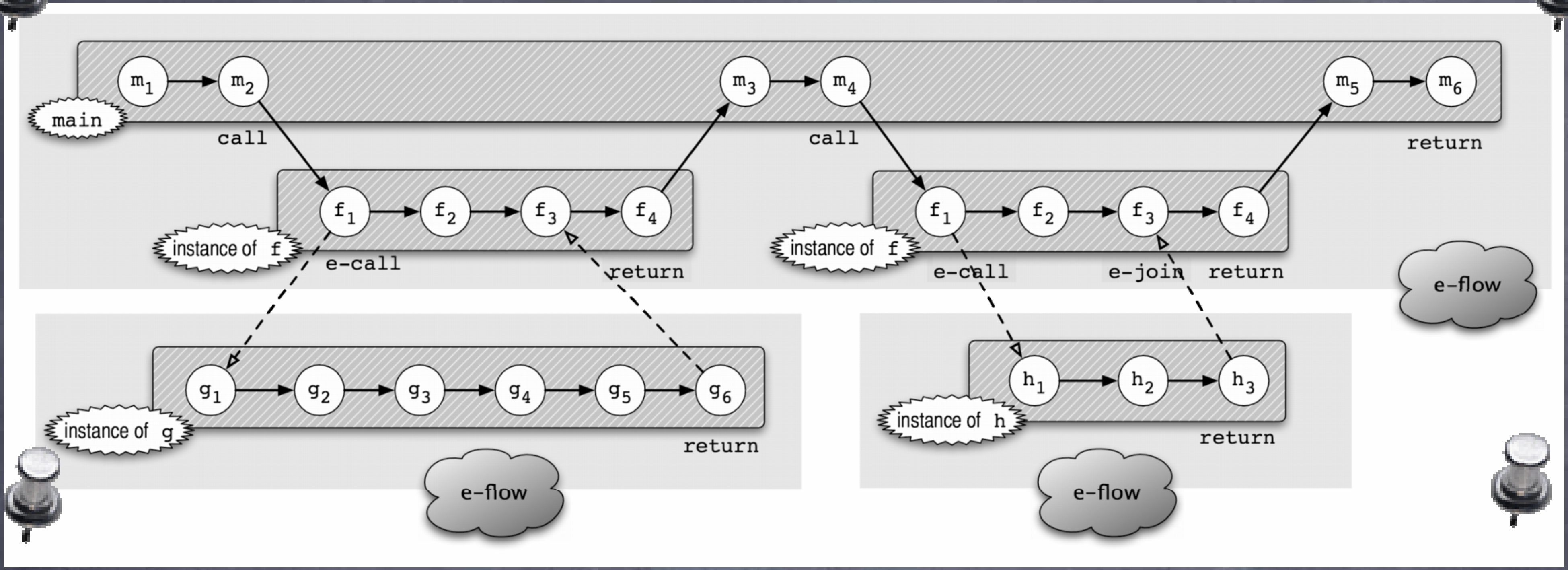
```
list_cell_t *body;  // C (private) pointer
body = (list_cell_t *) r(root)
```

- From r/rw on, the priv. pointer may be used to access shared variable (no further mediation …)

- Shared variables obey to DAG consistency
  no lock/unlock/barrier (Leiserson+, Cilk)

- No OS traps, no signal-handlers, fully POSIX threads compliant, address translation time 31 clock cycles (in the case of cache hit)

DAG consistency



Reads “sees” writes along paths on the eflow graph

- Independent e-flows ought to write different memory words
- A DAG consistency serious problem
- Accumulation behavior can be achieved with reduce used with an user-defined associative/commutative operations (...)

Build & Visit a k-tree

```
edeclare_tree(k_tree_t,int,K);
k_tree_t a_tree = TREE_INITIALIZER;

typedef struct {int child_n; int level} arg_t;

main() {
  eref_t root;
  arg_t arg = {0, 16  /* tree depth */ };
  e_initialize();
  root = tree_par_build(E_NULL,&arg);
  tree_visit(root,&arg);
  e_terminate();
}
```

Visiting a k-tree

```
eref_t tree_visit(eref_t node) {
  int *body;
  ehandler_t it;

  efun_init();
  ehandler_init(it);
  body = r(node);
  *body += *body/3;
  e_foreach_child(hand,tree_visit,body)
  e_joinall(it ,NULL);
return(E_NULL);
}
```

The speedup-overhead tradeoff

To parallelize or not to parallelize

eskimo mission

- exploit enough parallelism to maintain a fair amount of active threads (exploit speedup), but
- not too much in order to avoid unnecessary overheads. They come from many sources:
  - accesses to remote data (network, protocol, cache, ...)
  - parallelism management (synchronizations, scheduling, ...)
- runtime decisions (that depend on programmer hints, algorithm, data, system status ...)

eflows proactive scheduling

- No work-stealing (as cilk, athapascan)
- Policy: at ecall/eforeach time

  The local node is overwhelmed w.r.t. to the others?
  - Yes – spawn it remotely
  - No - The new e-flows will use mostly local addresses ?
    - Yes – enough locally active threads ?
      - Yes – sequentialize it
      - No – map it on a local thread
    - No – Spawn it remotely where data is

eflows scheduling 2

- How known if the PE is overwhelmed w.r.t others
  - keep statistics (#active threads, CPU load, mem) and exchange with others
- How known what data the new flow will access?
  - Expect an hint from the programmer
- If the programmer gives no hints?
  - Use system-wide lazy-managed statistics

The programmer insight

We need a prog. env. where performances improves gradually with programming skills. It should neither requires an inordinate effort to adapt application to ready-made skeletons nor to code all parallelism details (M. Cole)

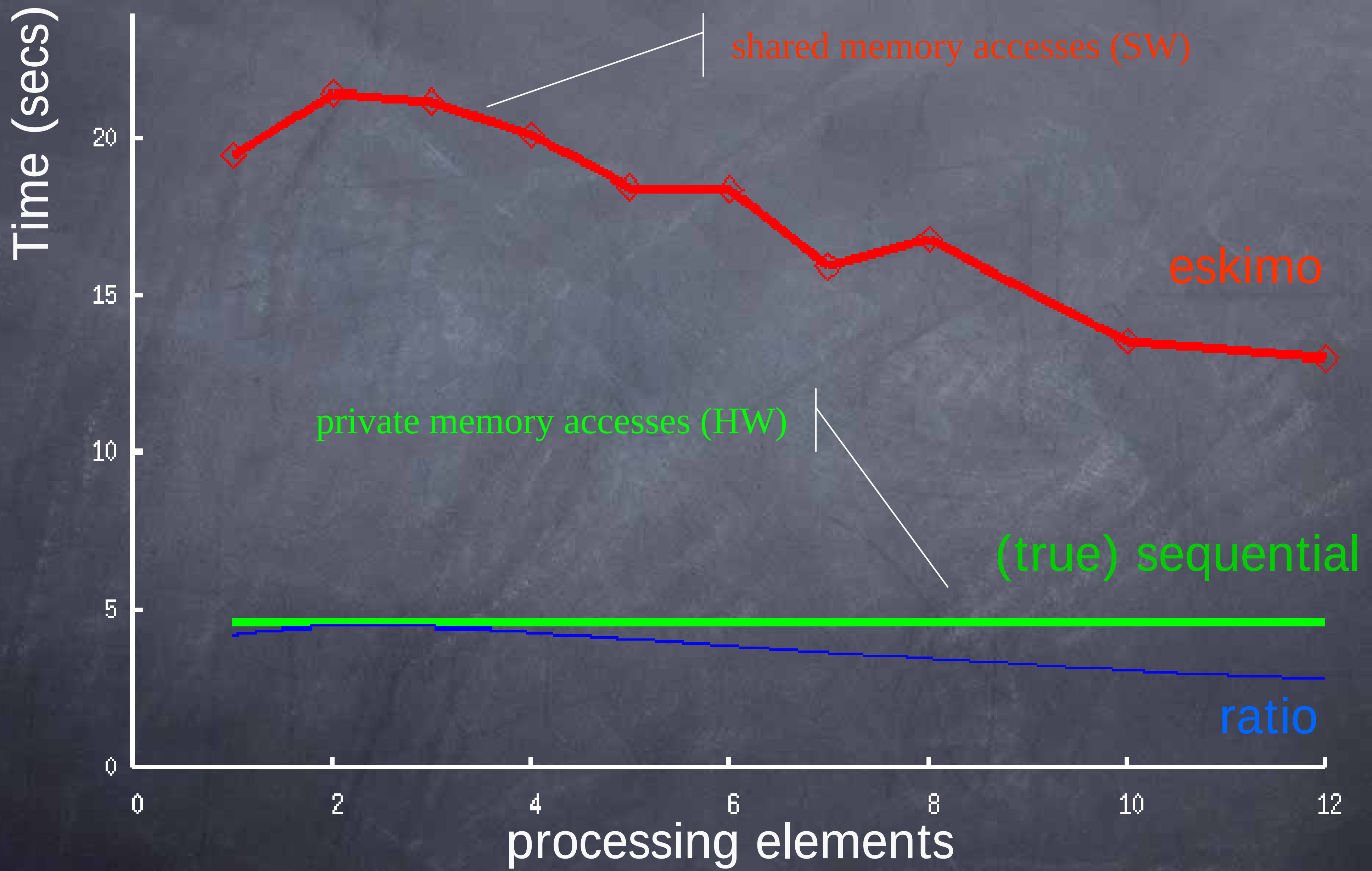
1. Allocate data exploiting accesses spatial locality within the same e-flows
2. Pass the reference of mostly accessed data as the first parameter of functions

- The more you follow these guidelines the faster is the application. The application is “anyway correct”.
- Quite usual in seq. programming. How C programmers navigate arrays? And fortran ones?

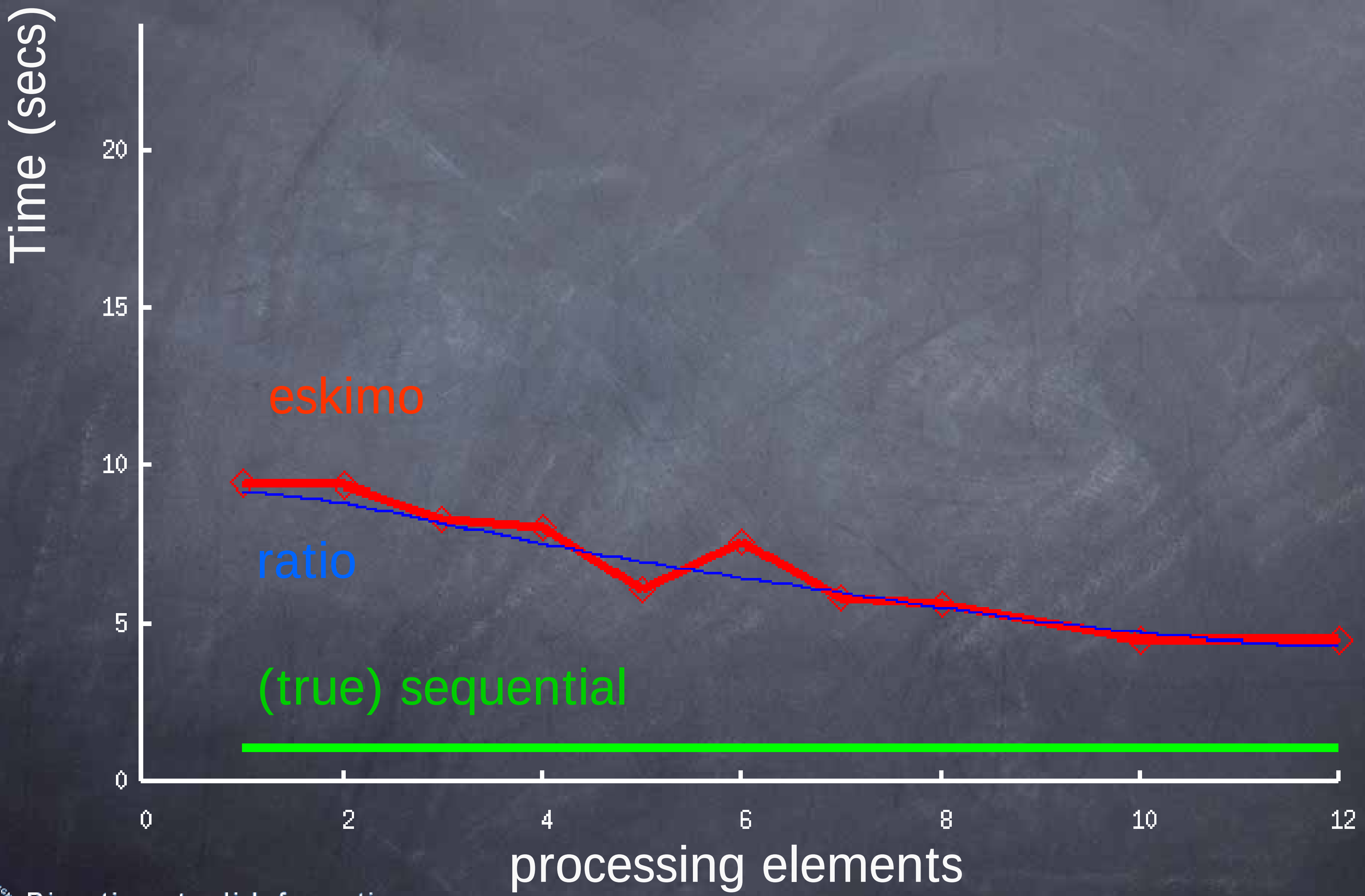
Performances

1. 12 Pentium II @ 233MHz
   Switched Eth 100MB/s
   (exclusive use)
2. 2x2-ways PIII @ 550MHz
   Switched Eth 100MB/s
   (shared with all the dept.)
3. 1 int x node (worst case)

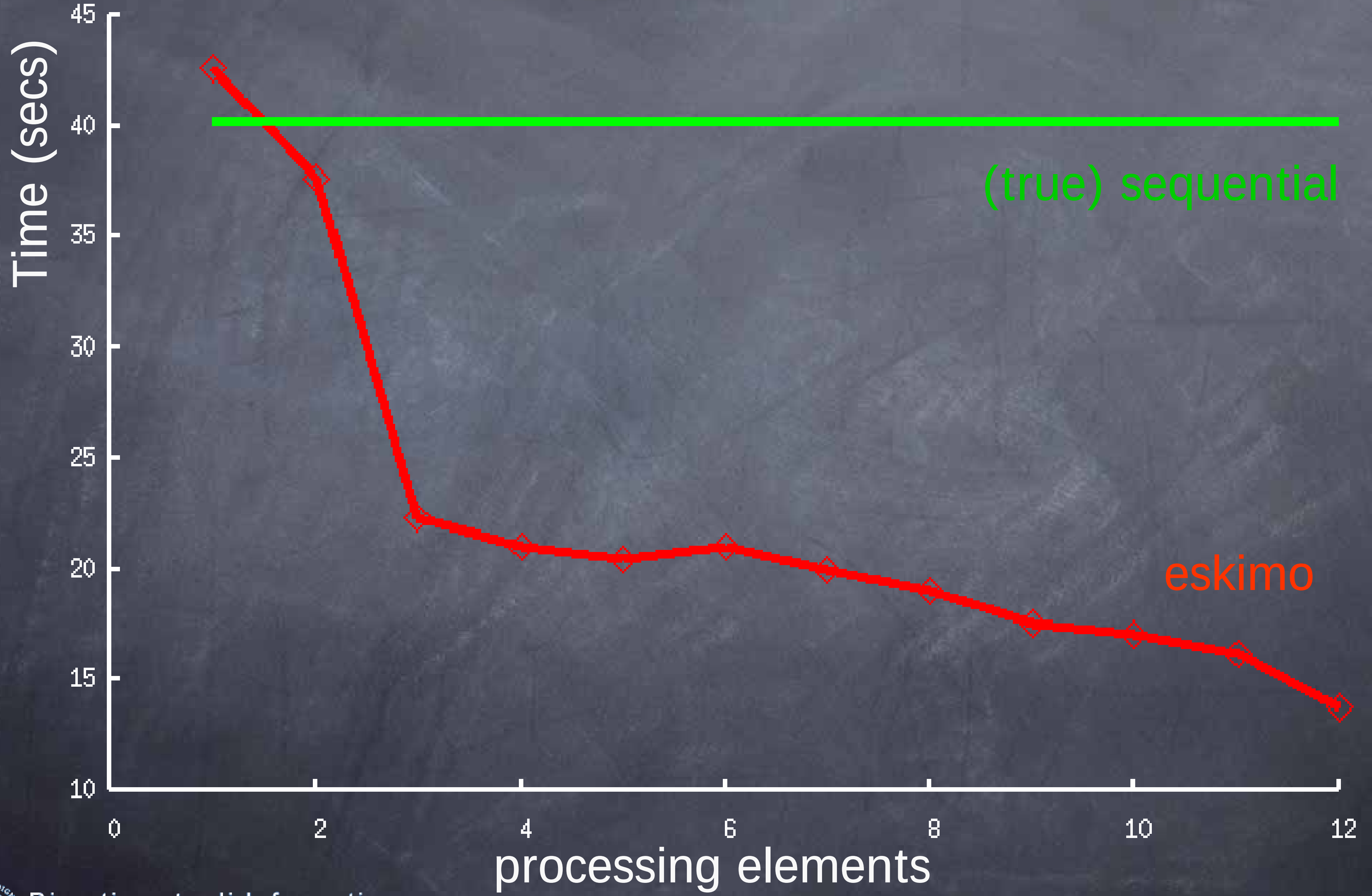
Overhead allocate+write (d22/4Mnodes)



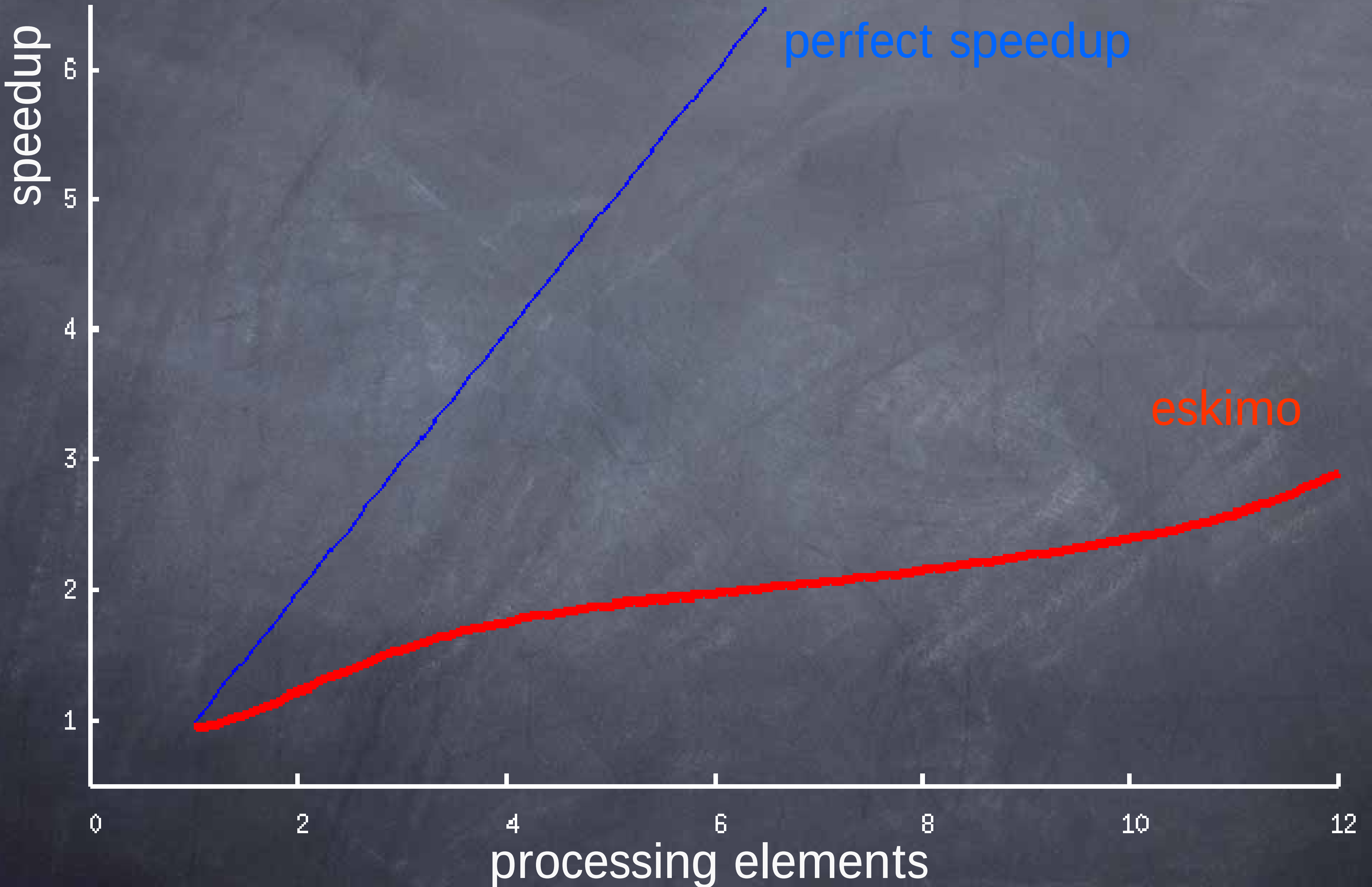
Overhead visit -- read -- (22/4Mnodes)



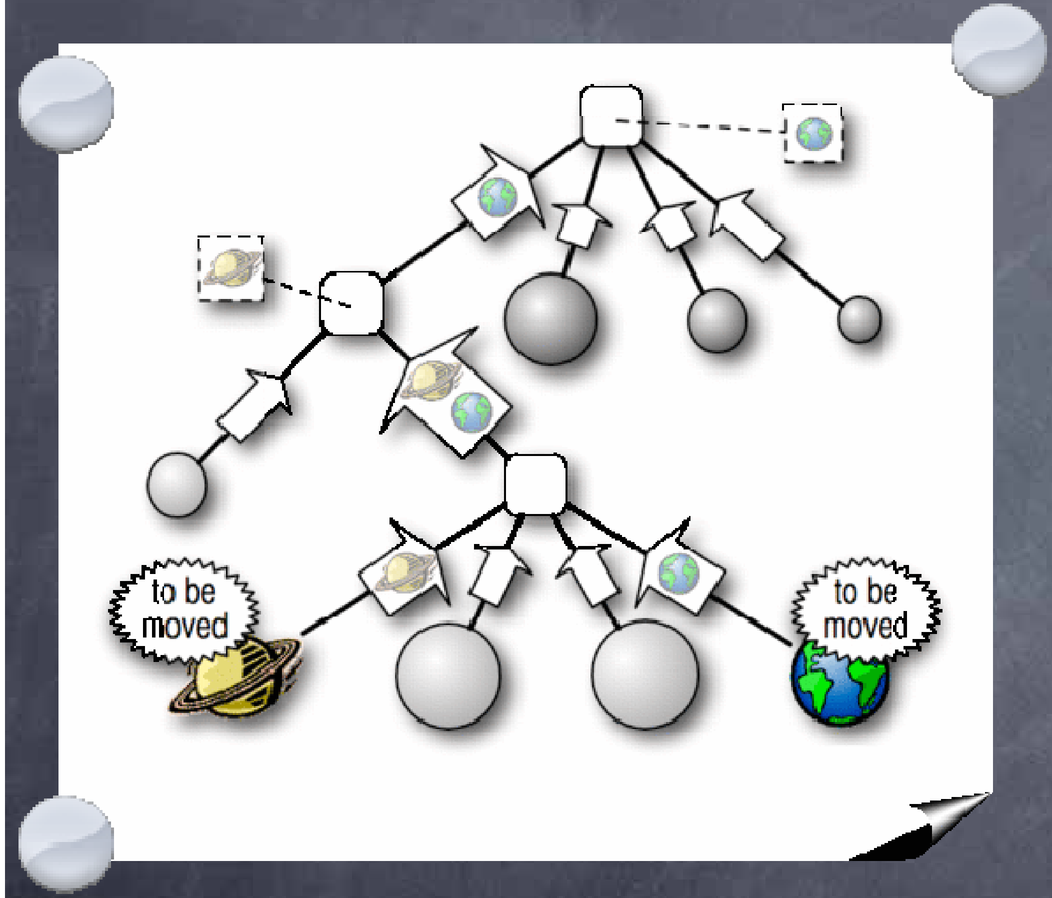
Visit time (depth 20, 1Mnodes, 37μs load)

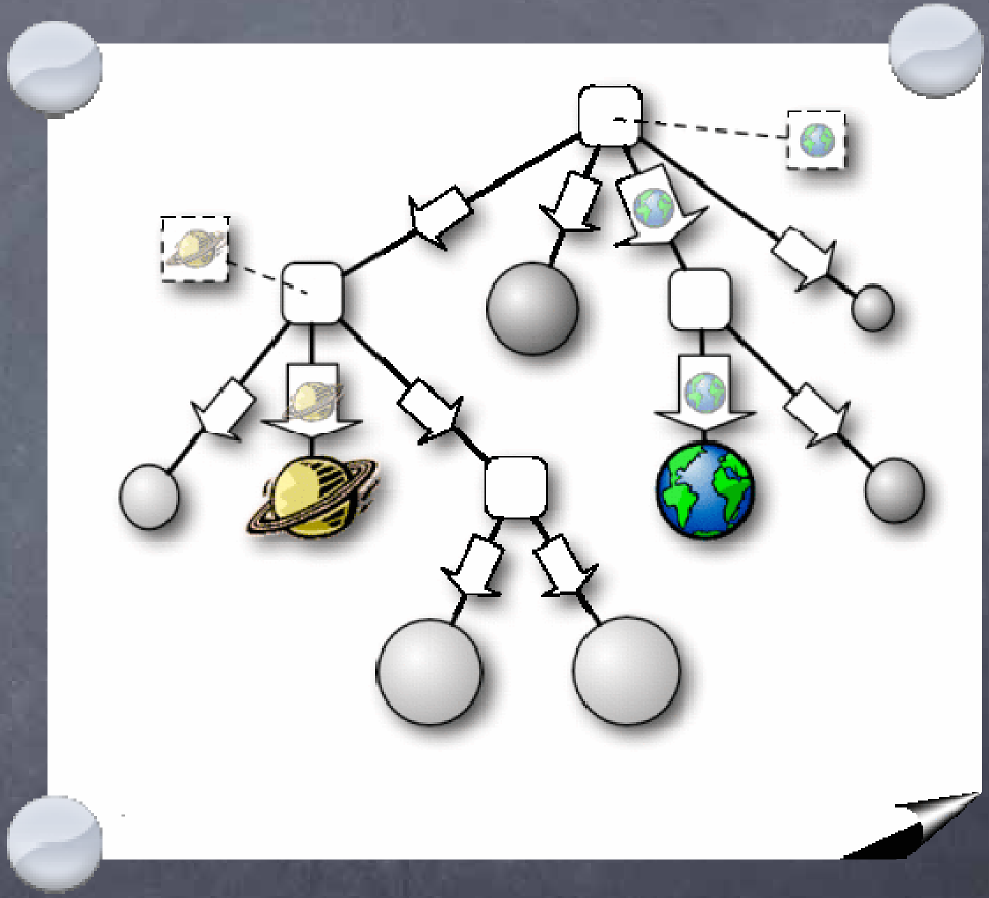


Visit speedup (d20, 1Mnodes, 37μs load)



Barnes-Hut (system step in 2 phases)





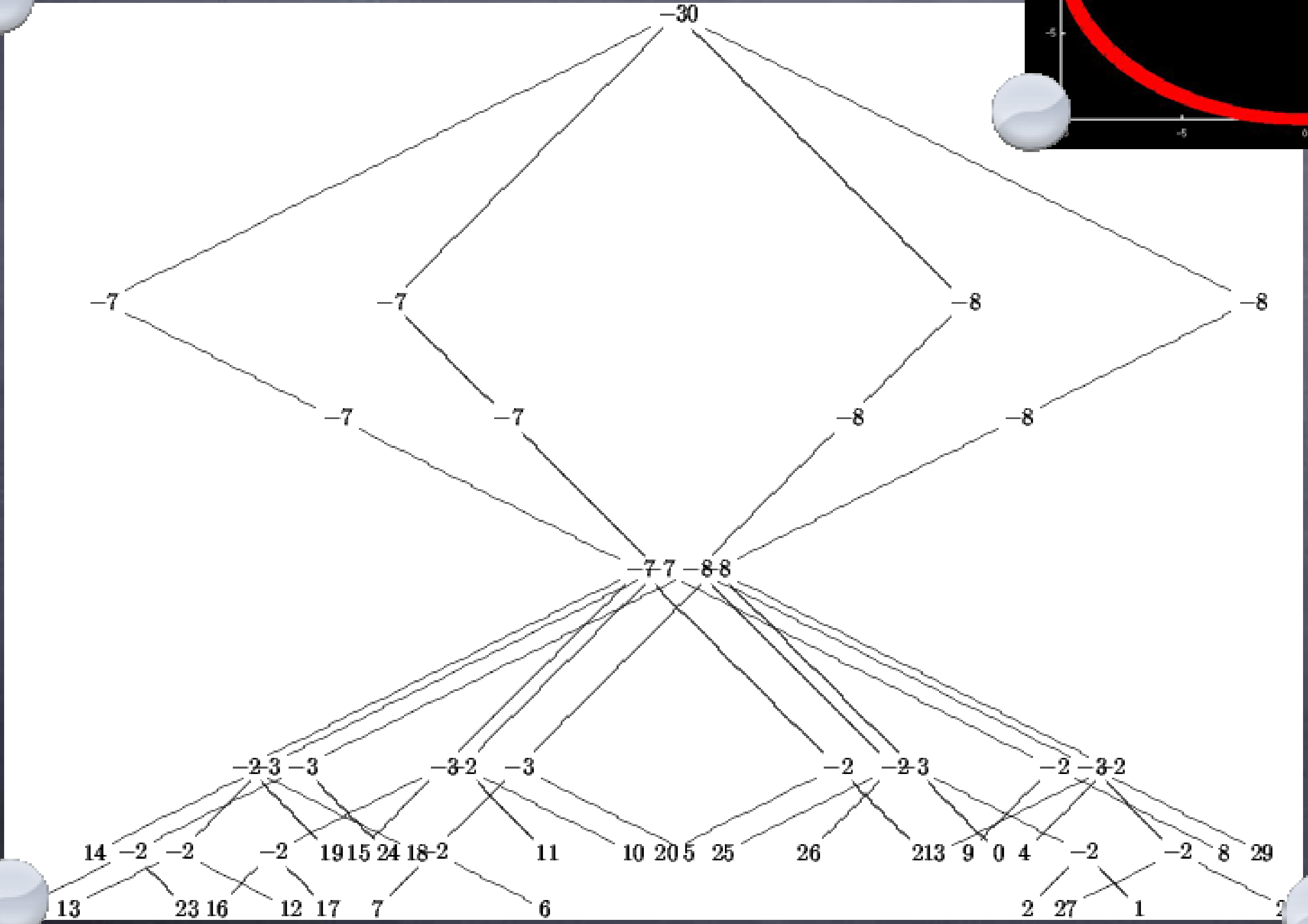
1) bottom-up

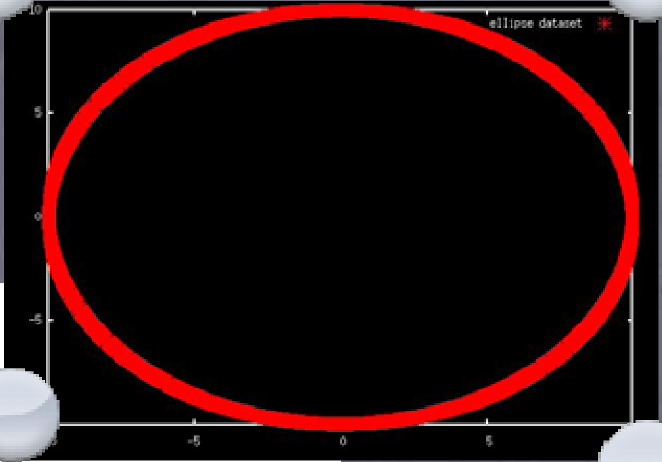
2) top-down

eskimo Barnes-Hut bottom-up phase

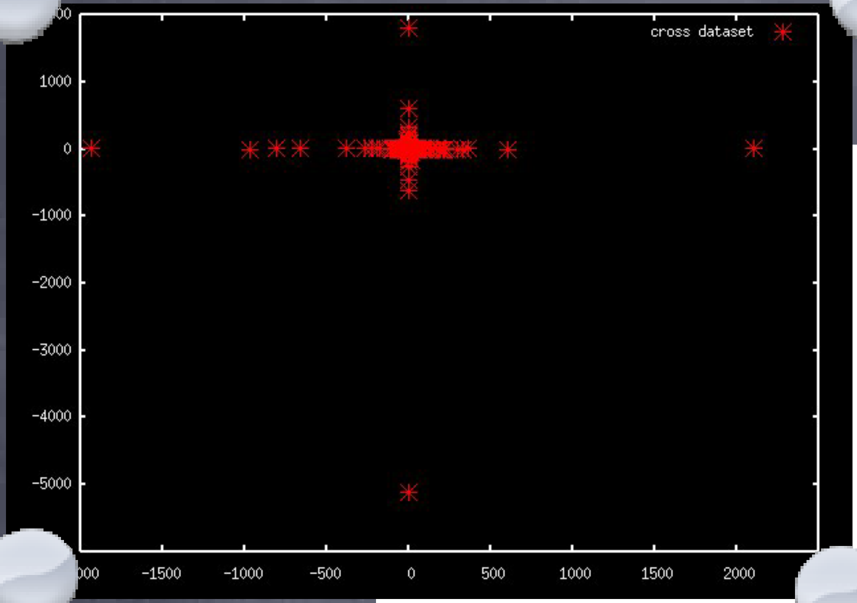
```
eref_t sys_step_bottom_up(eref_t anode){
  eref_t ret_array[4]; ehandler_t hand;
  eref_t float_list, sink_list; node_t *np;
  np = (node_t *) rw(anode);
  if (np->leaf) {
    <figure out acceleration (implies a visit from the root
    update bodies position (np->x = …; np->y = …;)>
    if (!within_borders(anode)) push(float_list,anode);
  } else {
  /* Divide */
  e_foreach_child(hand, sys_step_bottom_up,np);
  e_joinall(hand,ret_array);
  /* Conquer */
  for(i=0;i<4;i++)
    while(elem=pop(ret_array[i]))
    if (within_borders(elem)) push(sink_list,elem);
    else push(float_list,elem);
  np = (node_t *) rw(anode); np->ancestor_list = elem;
return(float_list); }
```

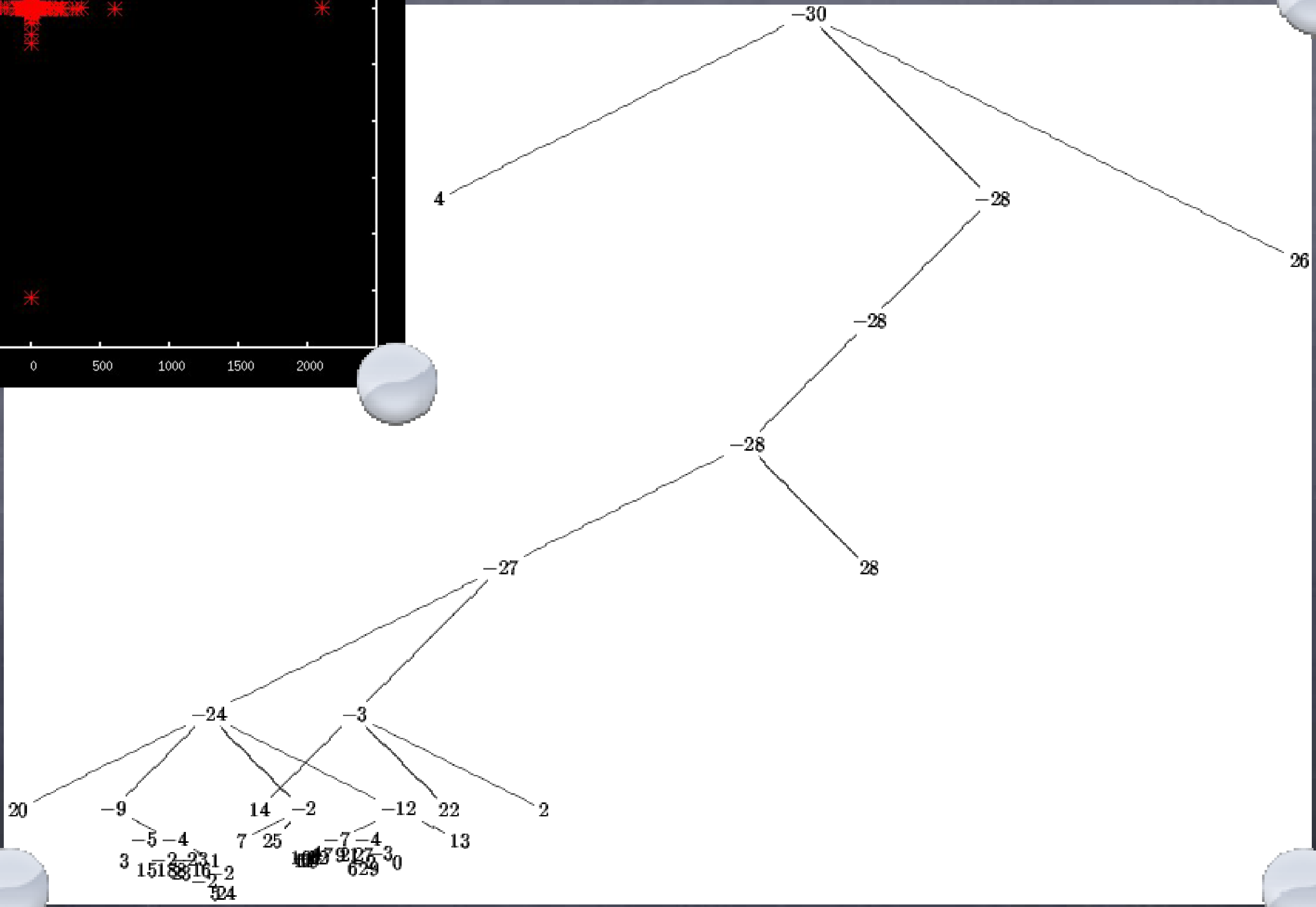
Ellipse dataset (balanced)





Cross dataset (unbalanced)





Barnes-Hut speedup

	unbalanced		balanced		
#bodies	10k	20k	10k	20k	optim
MPI 1 x 2 SMP/2	0.9	1.0	1.9	1.8	2
MPI 1 x SMP/2	0.9	1.0	3.2	3.1	4
eskimo 1 x SMP/2	1.2	1.1	1.9	1.8	2
eskimo 2 x SMP/2	1.6	1.8	3.1	3.0	4

A non-trivial MPI implementation (thanks to C. Zoccolo)

Payback of the approach

data and tasks

- an e-flow is bound to a PE for the life
  - no stack data migration (no cactus stack)
- e-flows and data orthogonalized
  - e-flows may be spawned towards data, or
  - data may migrate towards requesting e-flow, or
  - both
  - it depends on programs, input data, system status, ...

Skeletons

- foreach ("dynamic" data parallelism)
  - exploit nondeterminism in e-flows scheduling by executing first e-flows having data in cache
- build your own using both ecall/ejoin/...
  - As for example Divide&Conquer in many variants
- programmer does not deal with load balancing, data mapping but with an abstraction of them

Summary

- A platform to experiment, mainly
- Introduces dynamic data structures
- Introduces data/task co-scheduling
  - parallel activities not limited in number nor bound to a given processing elements
  - extendible to support some flavors of hot-swappable resources (...)
- Frames skeletons in the shared address model
- Implemented, fairly efficient

To Do

- Move to C++ framework:
  - It simplify syntax through polymorphism
  - It provides static typ checking
  - It enables the compilation of some part through templates and ad-hoc polymorphism
- Improve language hooks:
  - many parts of the runtime are configurable but there are no hooks at the language level (as for example cache replacing algorithm)

"eskimo works if and only if you absolutely believe it should work"

My kayak maestro

Questions ?

www.di.unipi.it/~aldinuc

Building a k-tree

```
eref_t tree_par_build(eref_t father,void *argsv){
  arg_t myvalue = *argsv;
  efun_init();
  if ((myvalue.level--)>0) {
    ehandler_t h[K]; ehandler_init(h, K);
    node = eadd_node(a_tree,father,myvalue.child_n);
    body = ((int *) rw(node));  *body= … ;
    for (i=0;i<K;i++) {
      myvalue.child_n=i;
      e_call_w_arg(&h[i],tree_par_build,node,
                         &myvalue,sizeof(arg_t));
    }
    e_joinall(a_child,tid,K);
    for (i=0;i<K;i++)
      e_setchild(k_tree_t,node,i,a_child[i]);
return(node);
}
```

Some implementation details

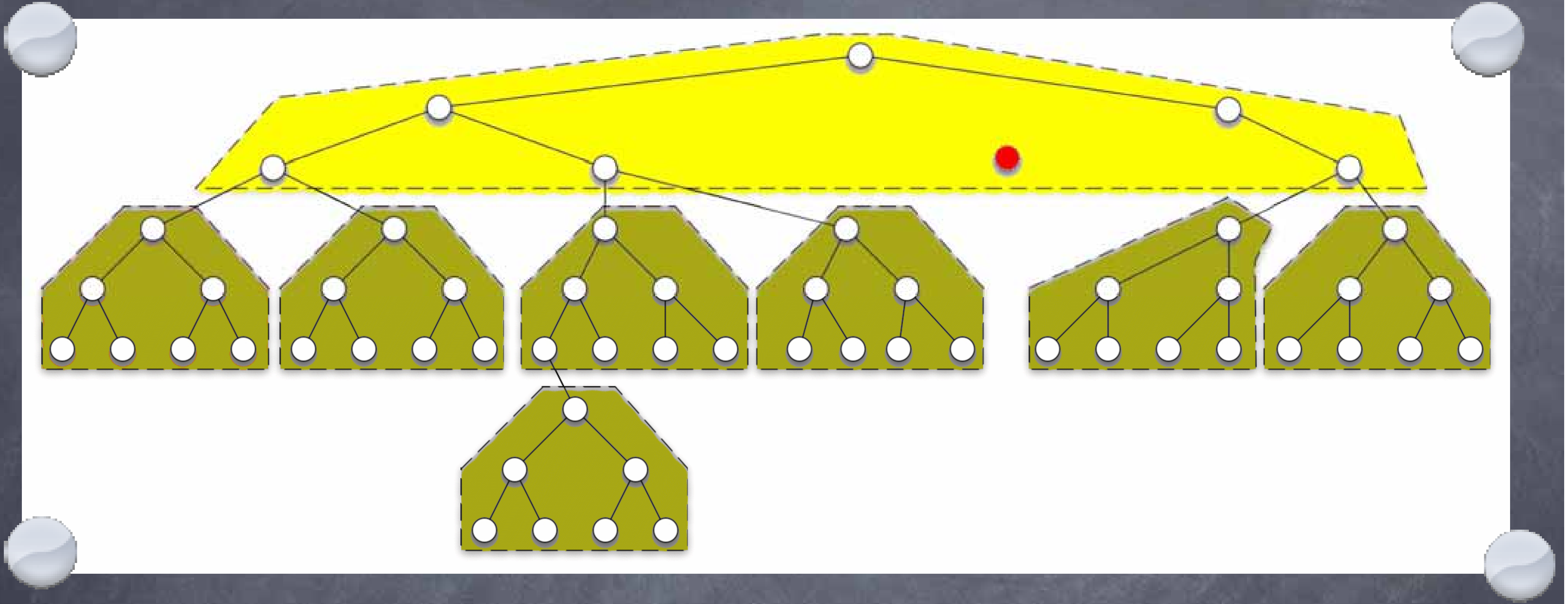
Trees are stored blocked in segments

- of any size (no mmap allocation), even within the same tree
- better if size match arch. working-grain (cpu/net balance)
- have internal organization (configurable, programmable at lower level)
- segms with different organizations can be mixed, even in th same tree
- their size may match architecture working-grain
- is the consistency-unit (diff+twin)
- segms boundaries trigger scheduling actions

Tree visit (d18, 256knodes)

	load	0 μs o	37μs	73 μs	optim
time (secs)	seq	0.03	9.95	19.01	--
	1 x SMP/2	0.30	7.03	12.07	--
	2 x SMP/2	0.15	4.80	8.51	--
speedup	1 x SMP/2	0.10	1.35	1.57	2
	2 x SMP/2	0.20	1.98	2.23	4

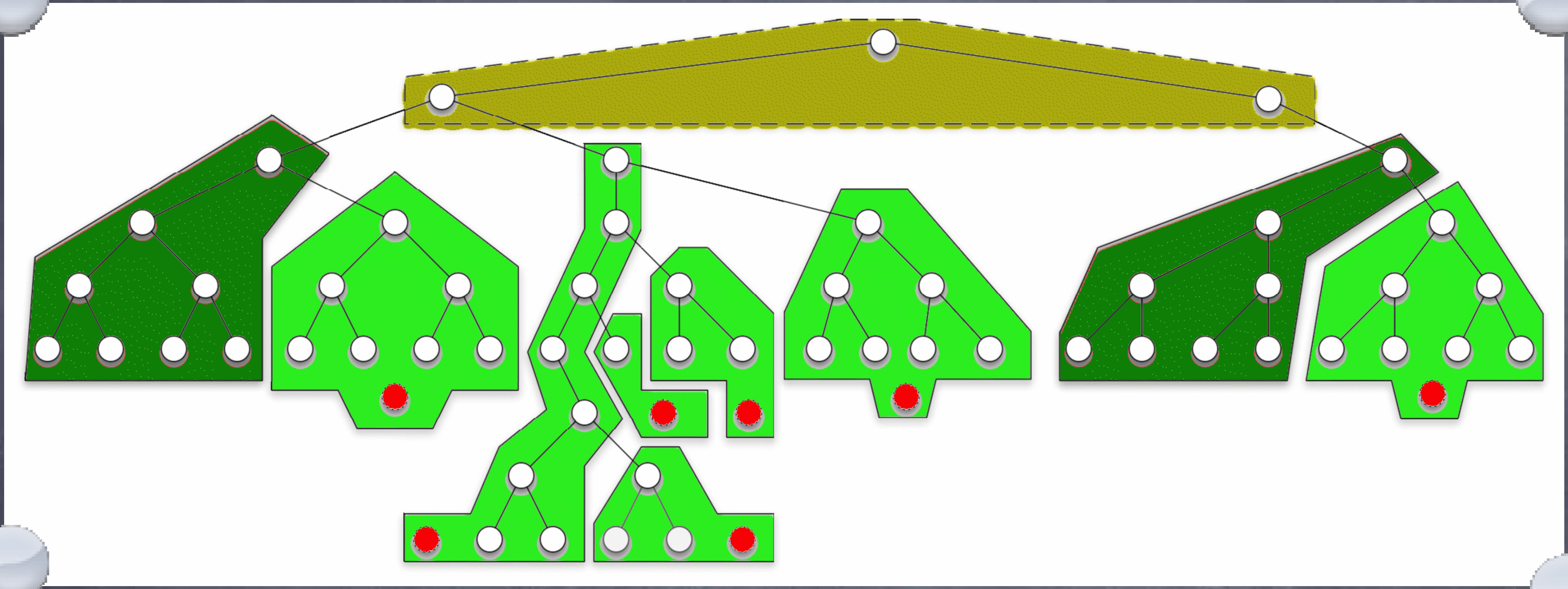
Tree organizations (heap)



- good for random accesses
- internal fragmentation

  rebuild with +1 level = + 56 segms (fill perc. 98% → 25%)

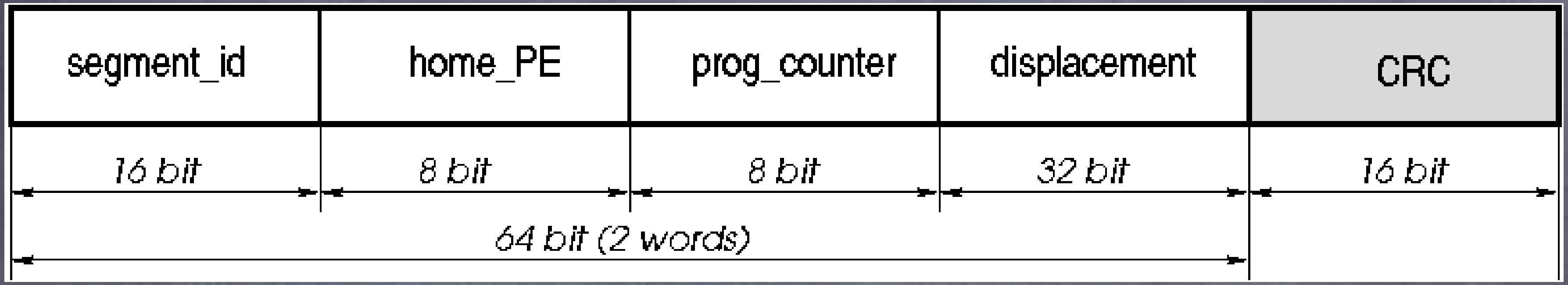
Tree organizations (first-fit)



- little internal fragmentation

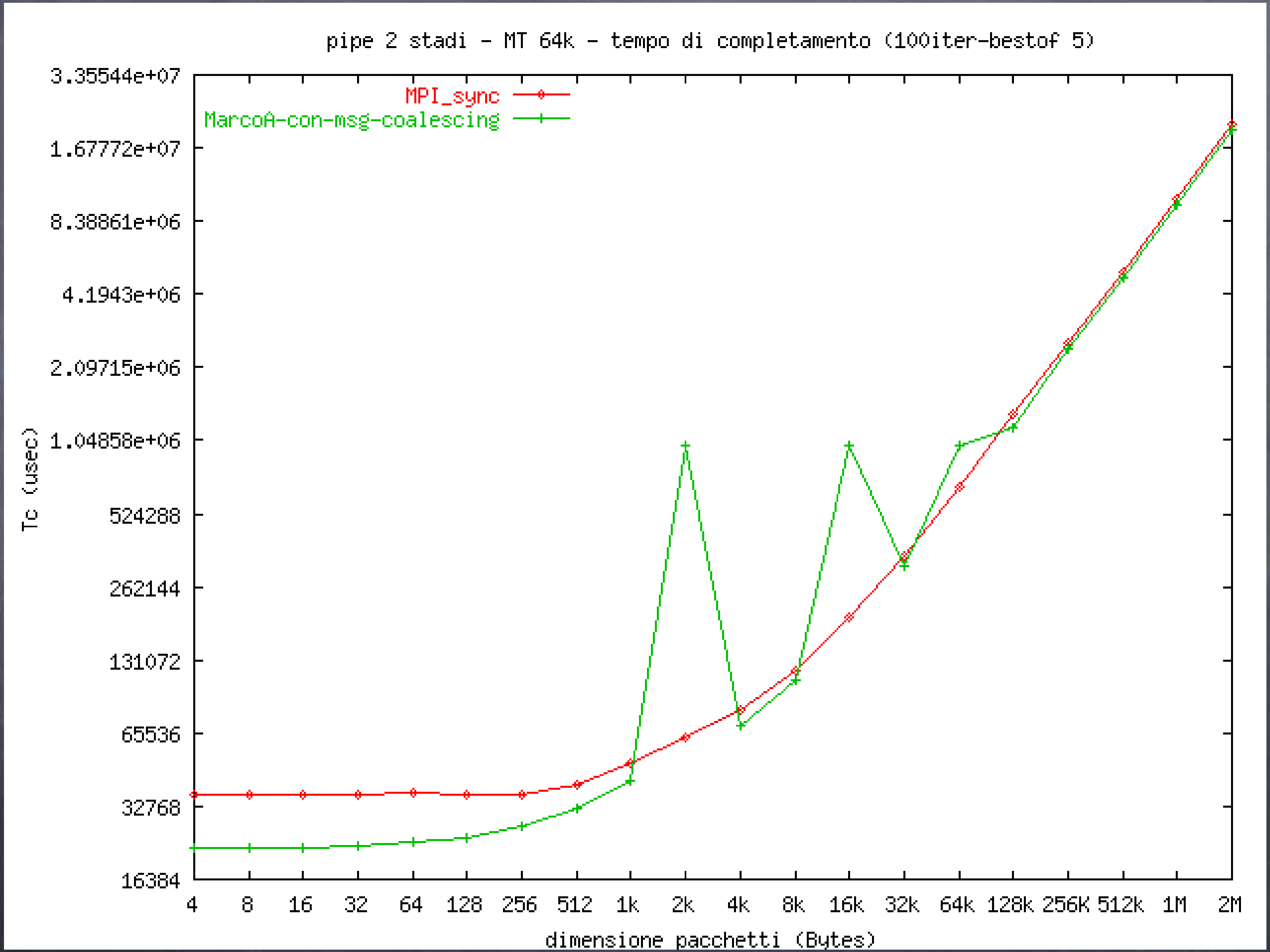
  rebuild with +1 level = + 8 segms (fill perc. 73% → 80%)
- good if allocated as visited (but it is a not rare case)
- heap-root block improves scheduling (because ...)

Shared Addresses

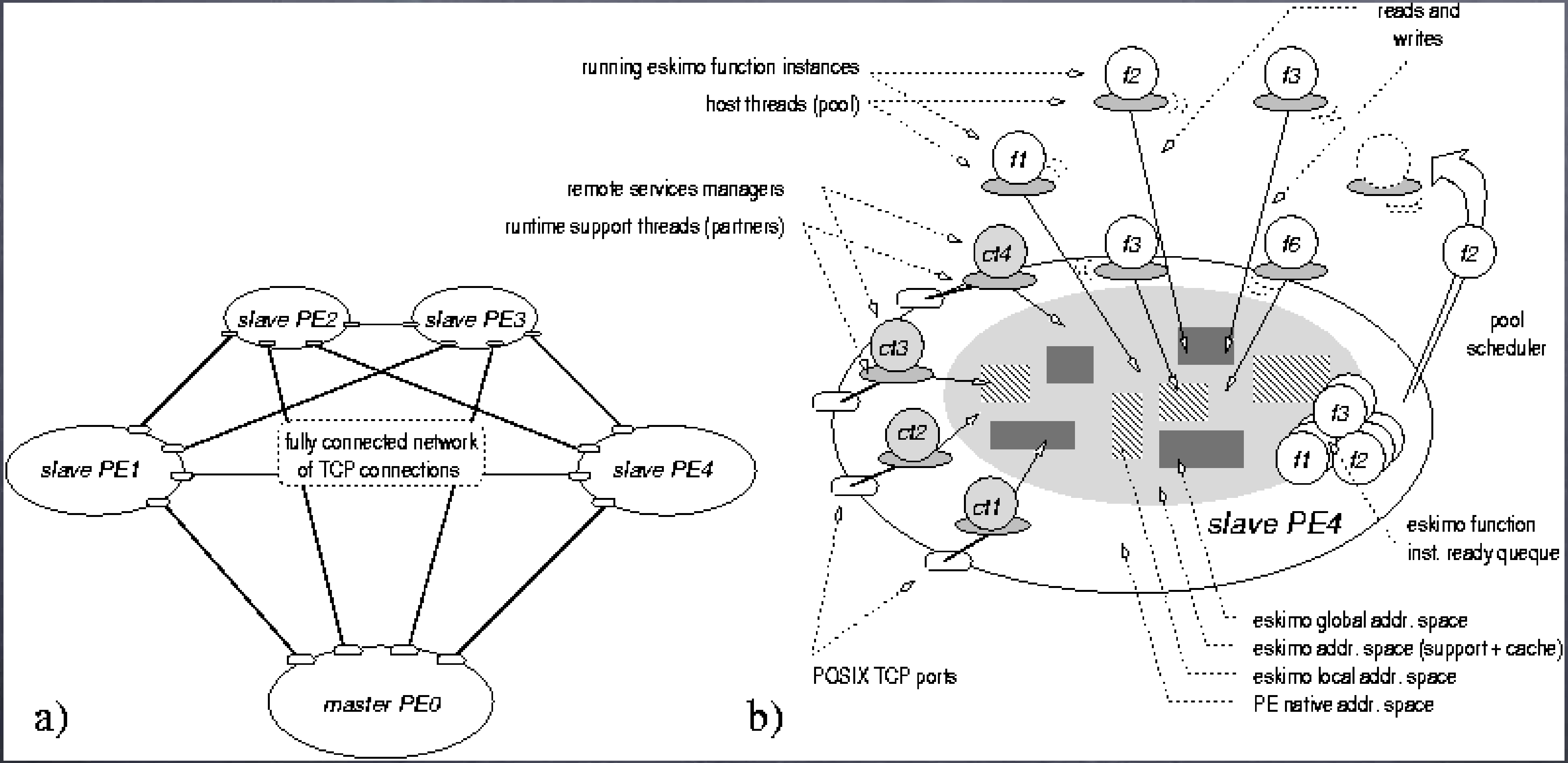


- memory in segments
- Independent from machine word
- Configurable
- Addr. Trasl. 31 clock cycles (PIII@450MHz), hit.
  - Miss time higher, but it depends on other factors
- Zero copy

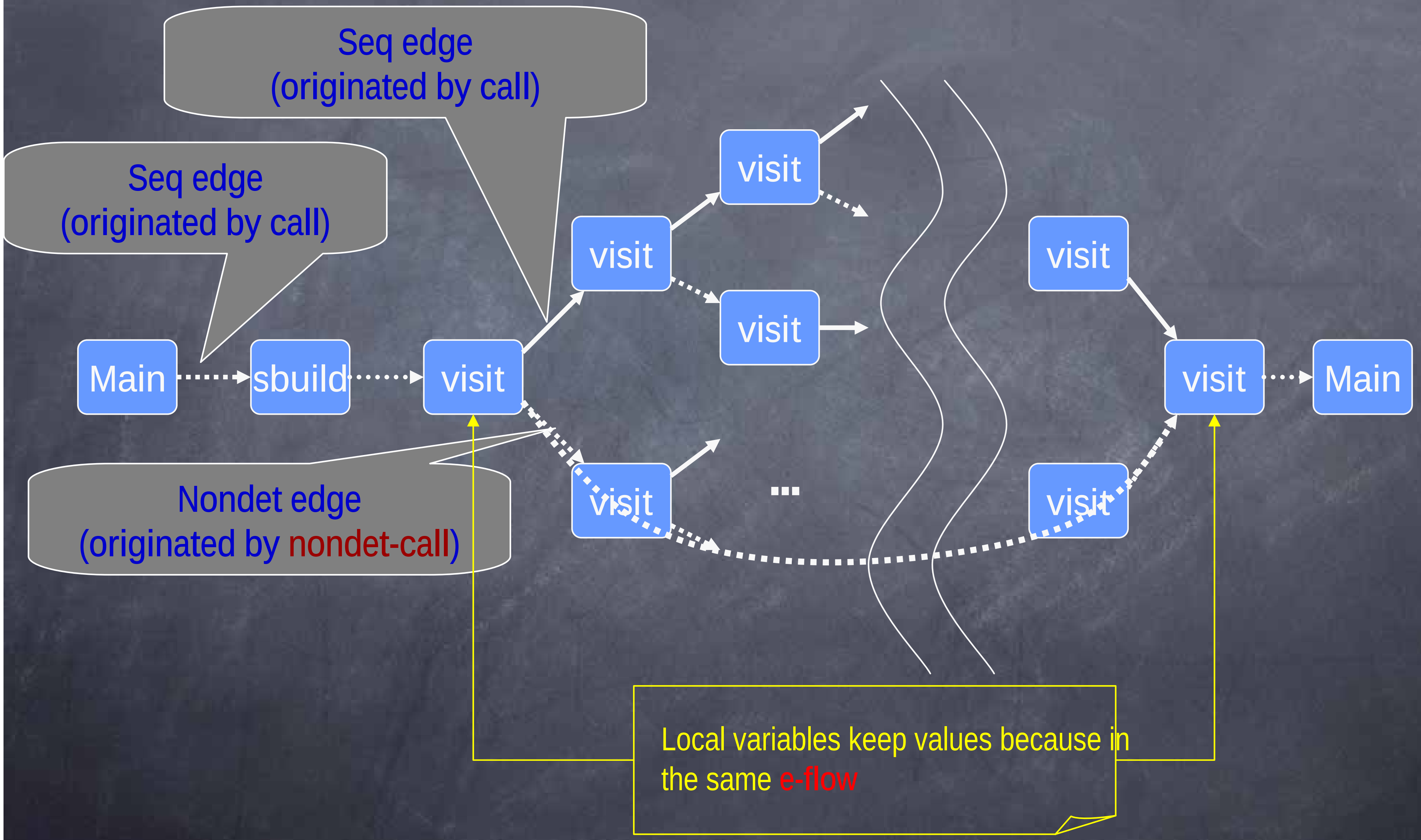
L1 TCP coalesing



Runtime - schema



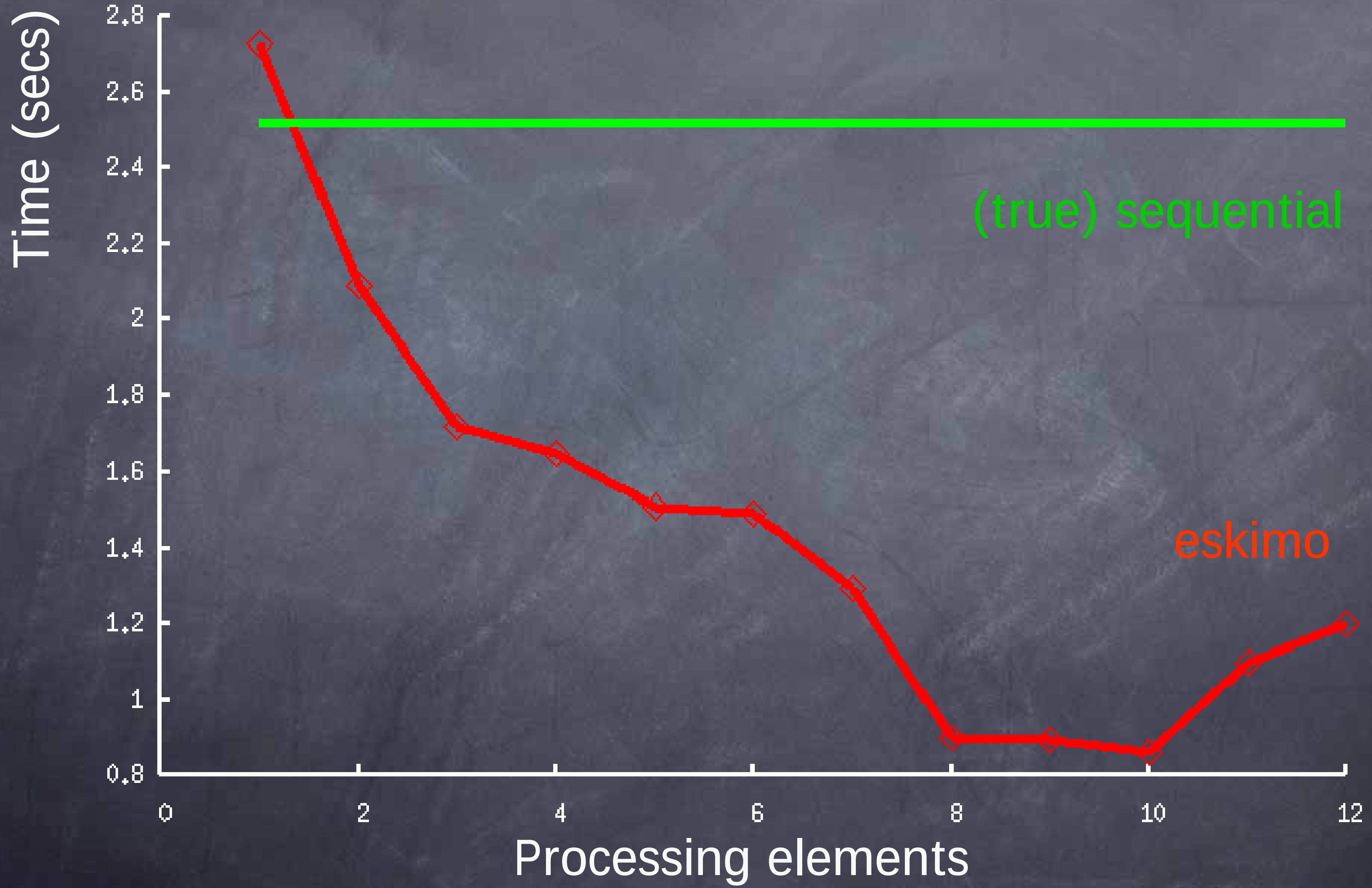
Flow of control (unfolds dynamically)



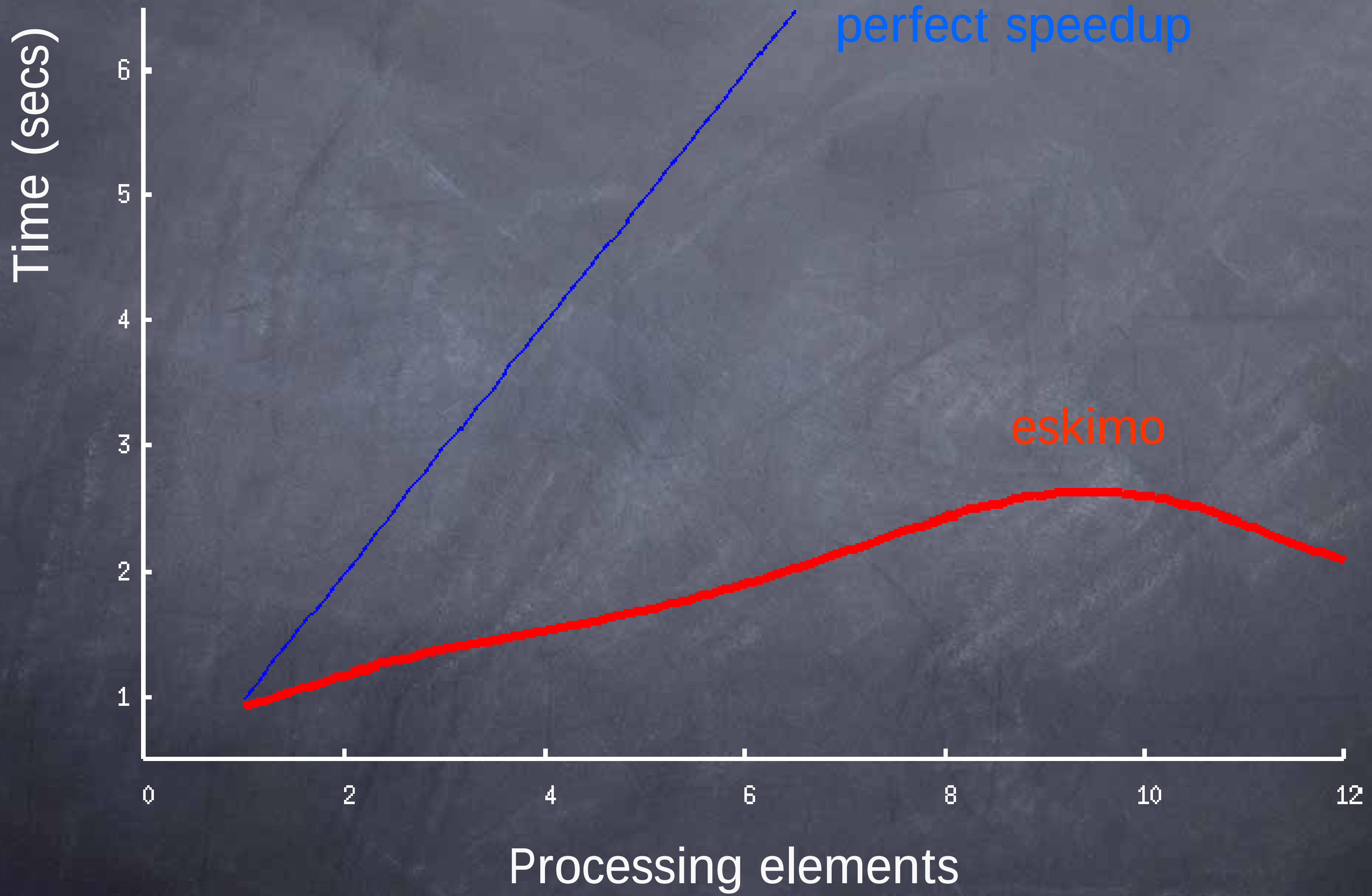
Tree visit overhead (zero load)

tree depth	16	18	20
# nodes	64k	256k	1M
size (MBytes)	768k	3M	12M
seq (secs)	0.01	0.03	0.15
1 x 2-way SMP (secs)	0.80	0.30	1.50
2 x 2-way SMP (secs)	0.40	0.15	0.70

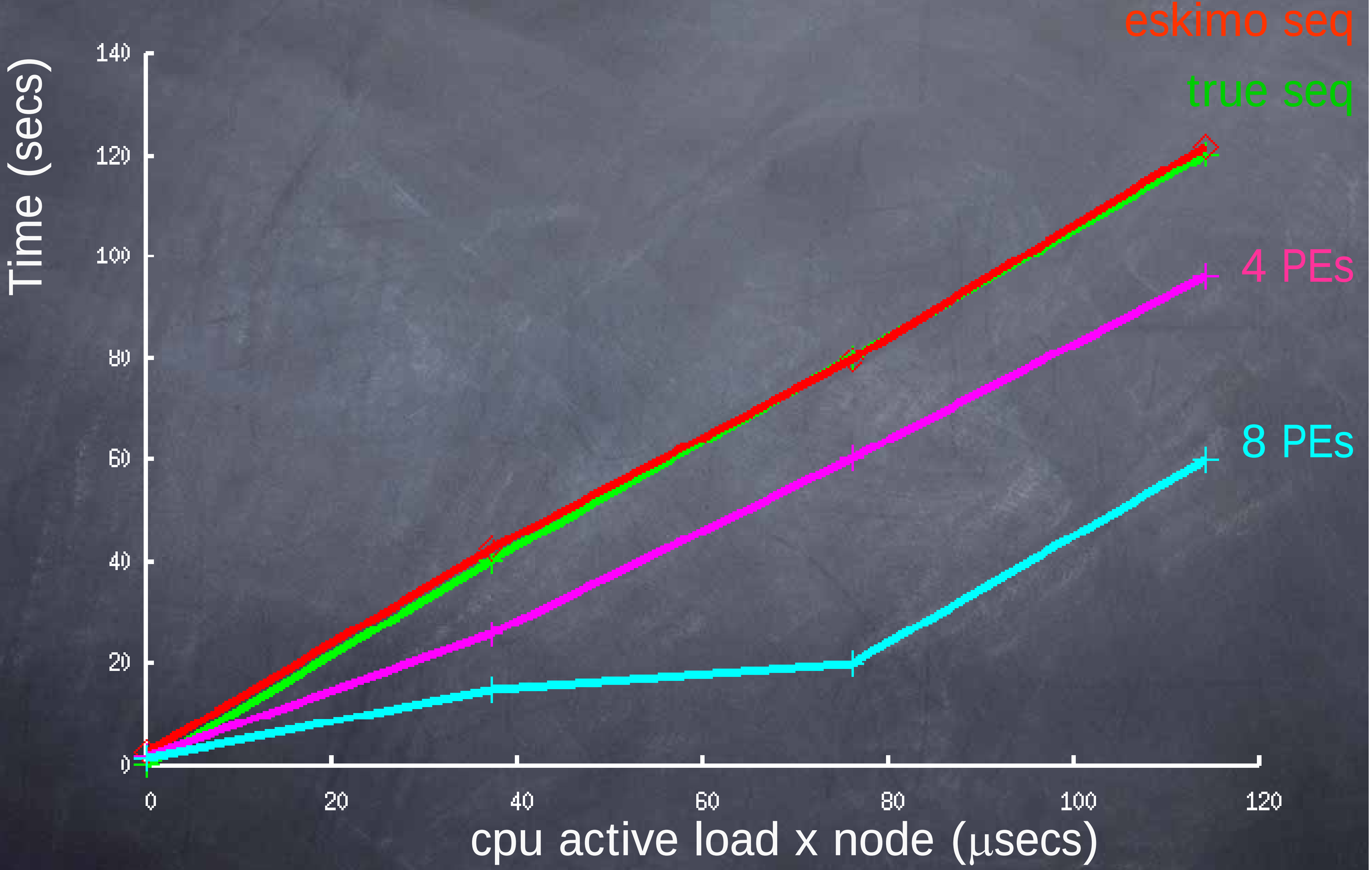
Visit time (d16, 64knodes, 37μs load)



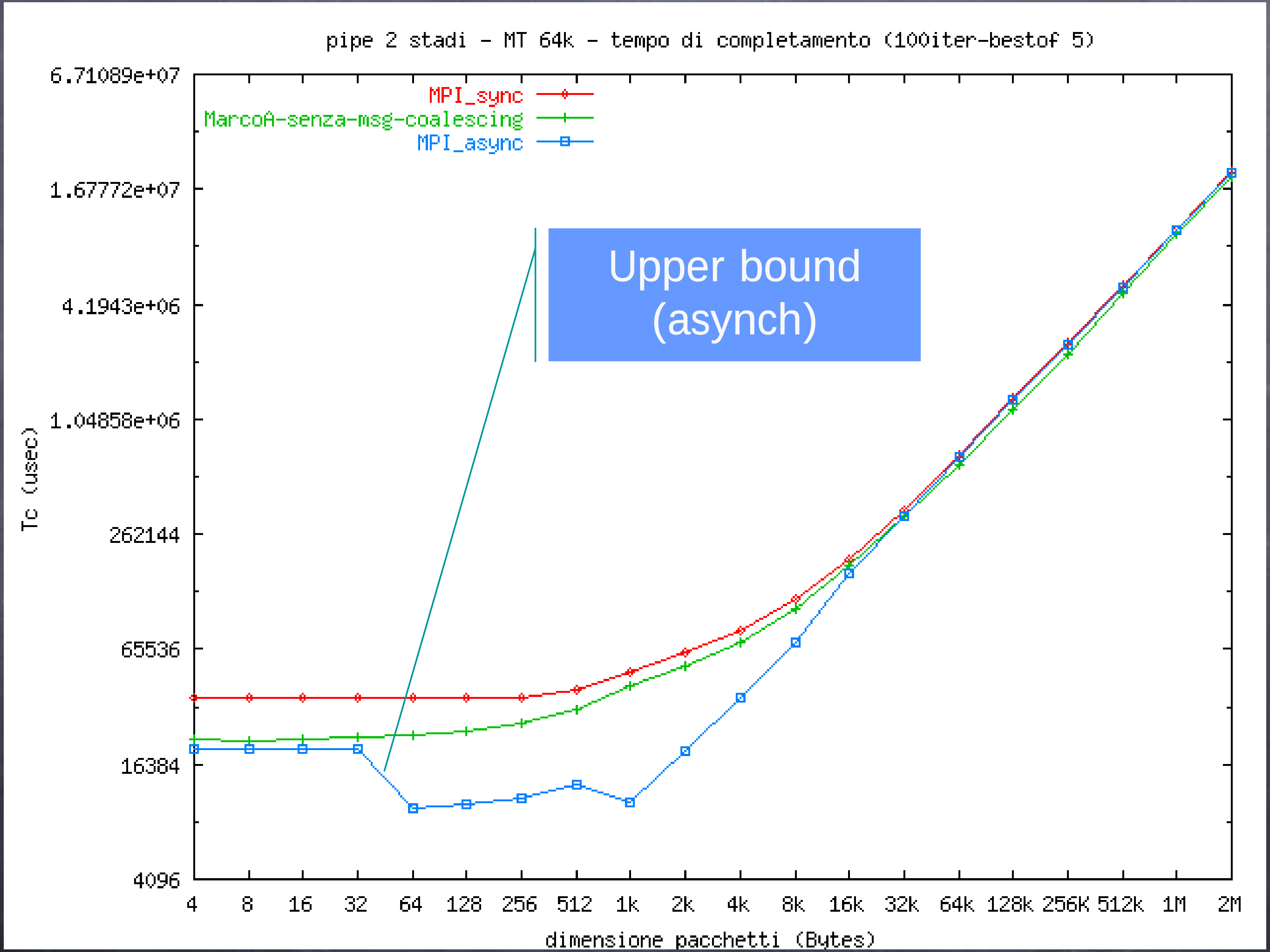
Visit speedup (d16, 64knodes, 37μs load)



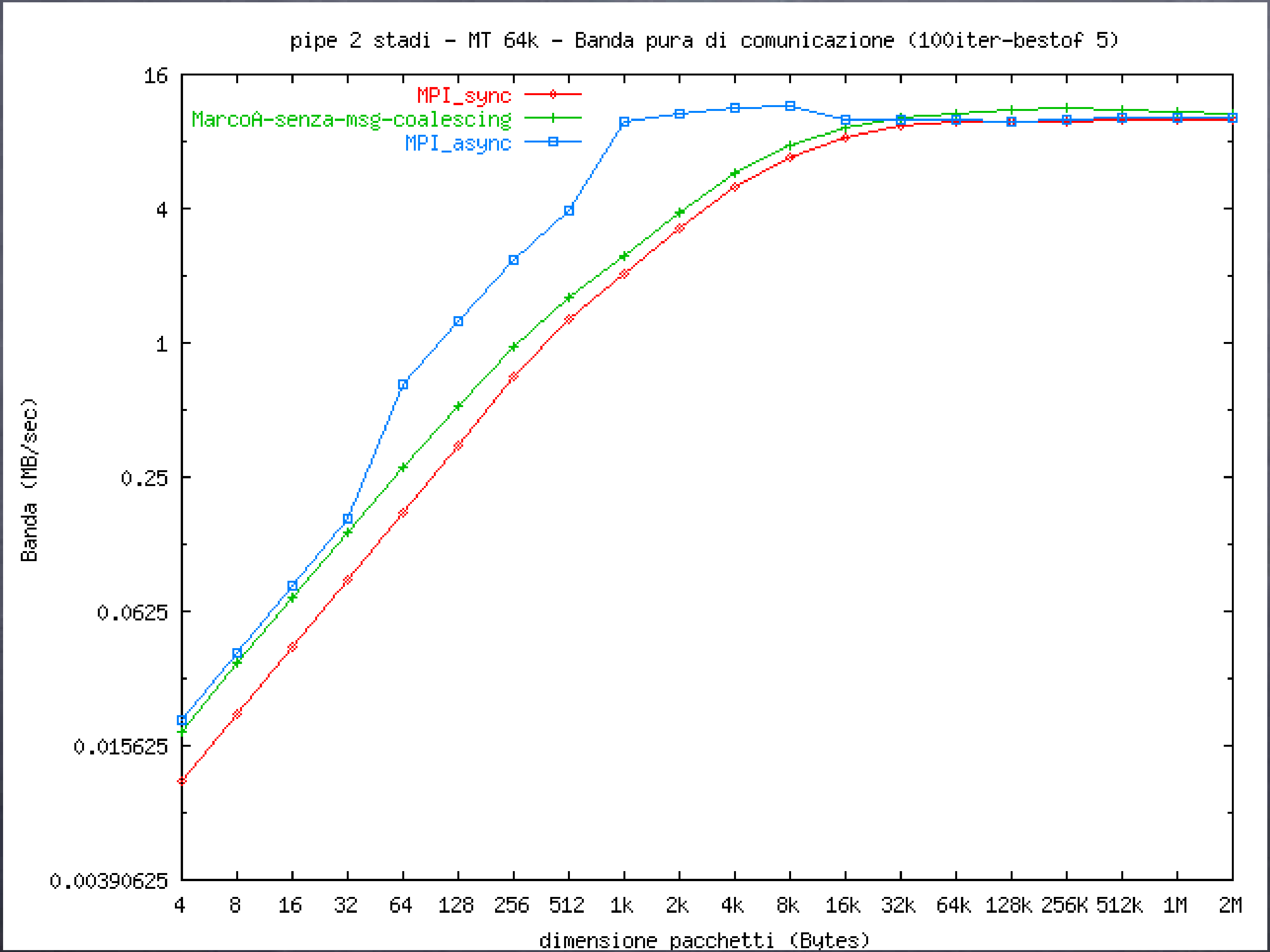
Visit time vs load (d20, 1Mnodes)



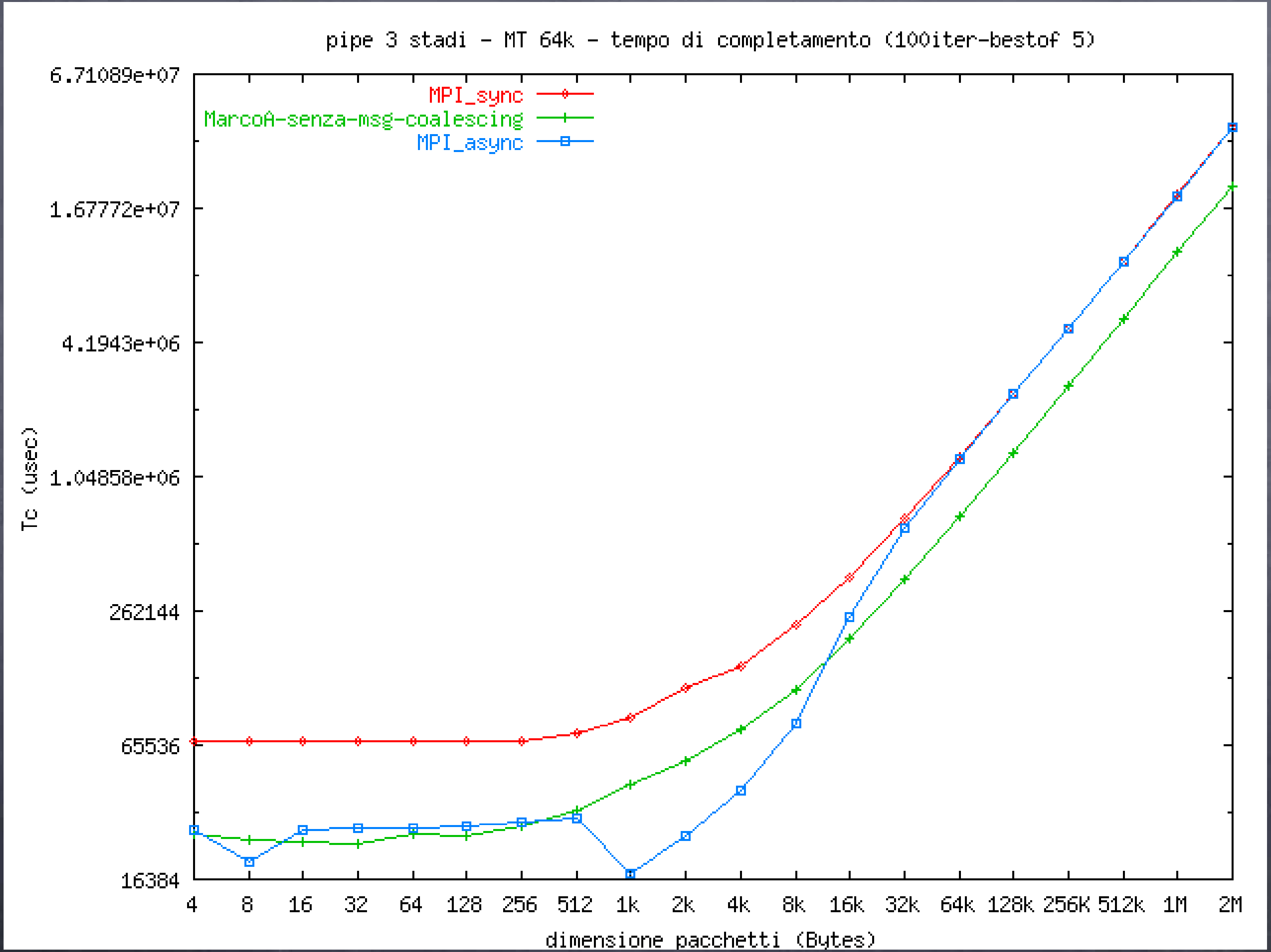
tier0 (producer-consumer sync)



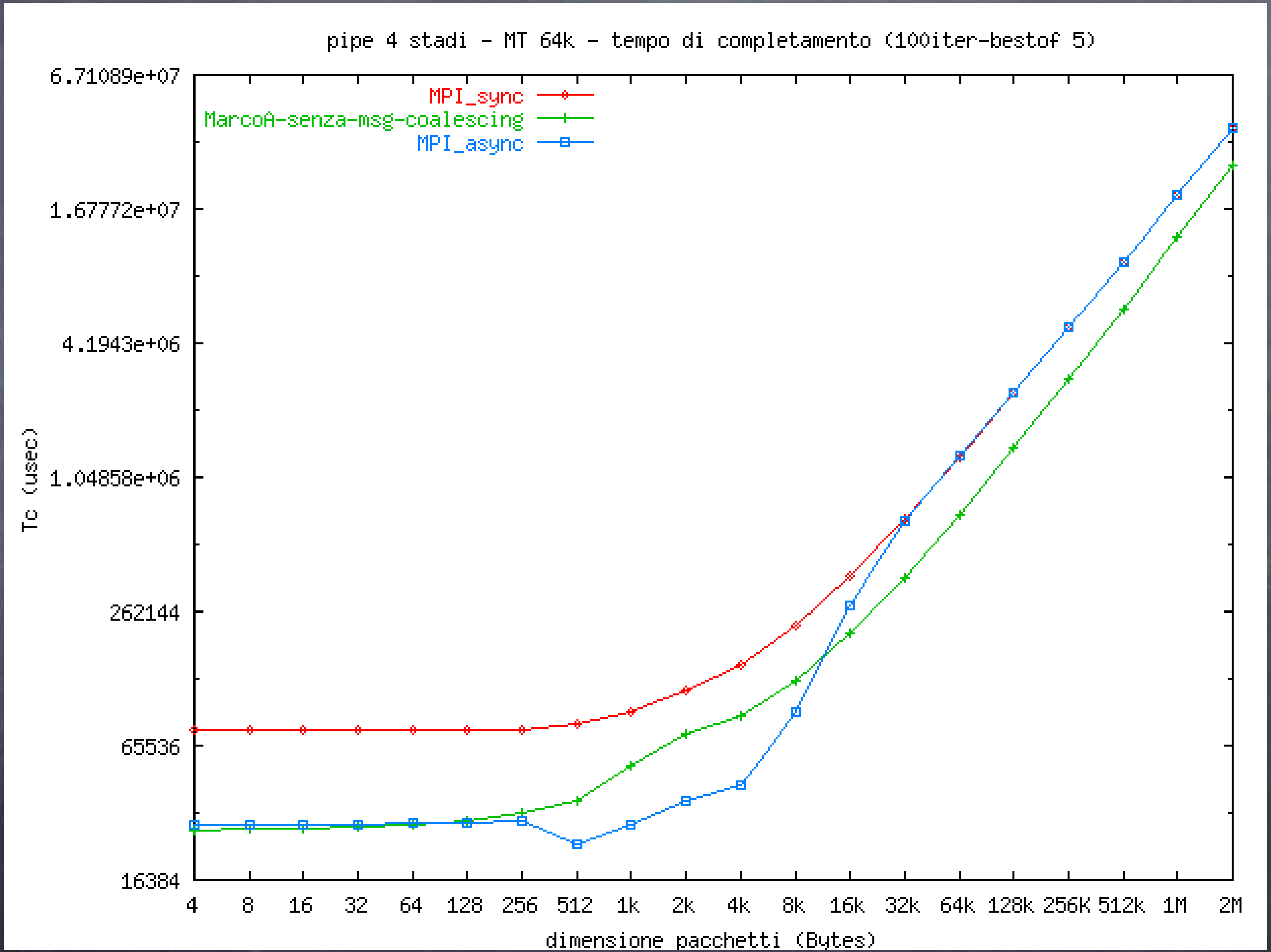
tier0 – throughput (prod-cons)



etier0 three stages pipeline



etier0 four stages pipeline


pipe 4 stadi - MT 64k - tempo di completamento (100iter-bestof 5)
MPI_sync
MarcoA-senza-msg-coalescing
MPI_async
Tc (usec)
6.71089e+07
1.67772e+07
4.1943e+06
1.04858e+06
1k
2k
4k
8k
16k
32k
64k
128k
256K
512k
1M
2M
dimensione pacchetti (Bytes)