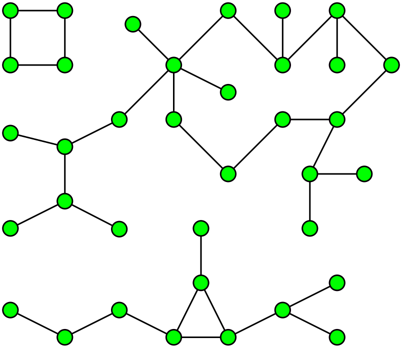
Connected Components

Benjamin Ulmer and Tobias Wicky

December 16, 2013

Connected Components

Determine the **number** and **size** of the connected components of a given graph.



boost graph library[1]

- provides function to compute number of connected components
- uses DFS

---

[1] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

parallel boost graph library [2]

- provides function to compute number of connected components
- uses hooking approach similar to ligra
- implemented using boost MPI for distributed memory

---

[2] Douglas Gregor and Andrew Lumsdaine. “The Parallel BGL: A generic library for distributed graph computations”. In: *Parallel Object-Oriented Scientific Computing (POOSC)* (2005).

Ligra[3]

- lightweight graph processing framework for shared memory
- provides two functions: VertexMap and EdgeMap
- many examples implemented e.g. Connected Components

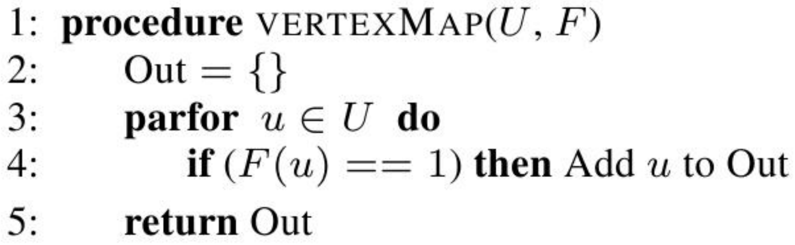
---

[3] Julian Shun and Guy E Blelloch. "Ligra: a lightweight graph processing framework for shared memory". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2013, pp. 135–146.

Ligra: provided Functions

Vertex Map

Returns all vertices $u \in U$ with $F(u) == 1$

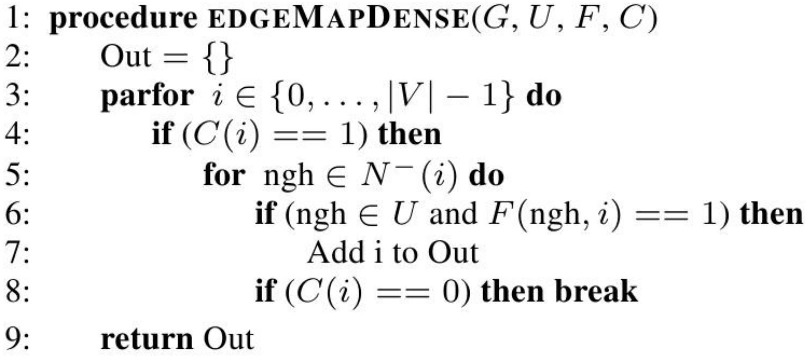


```
procedure VERTEXMAP($U$, $F$)
    Out = {}
    parfor $u \in U$ do
        if ($F(u) == 1$) then Add $u$ to Out
    return Out
```

Ligra: provided Functions

Edge Map Dense

Loops through **all vertices *i* in the Graph**

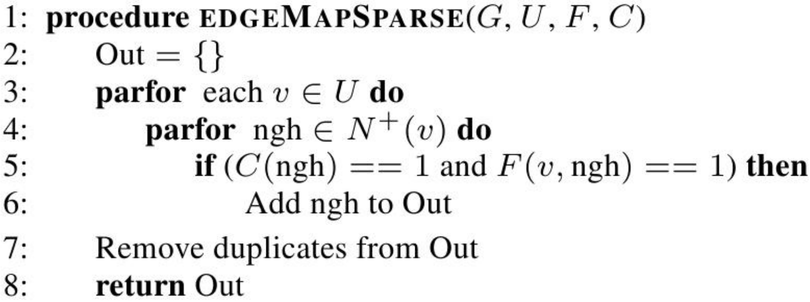


```
procedure EDGEMAPDENSE($G$, $U$, $F$, $C$)
    Out = {}
    parfor $i \in \{0, \ldots, |V| - 1\}$ do
        if ($C(i) == 1$) then
            for ngh $\in N^-(i)$ do
                if (ngh $\in U$ and $F(\text{ngh}, i) == 1$) then
                    Add i to Out
                if ($C(i) == 0$) then break
    return Out
```

Ligra: provided Functions

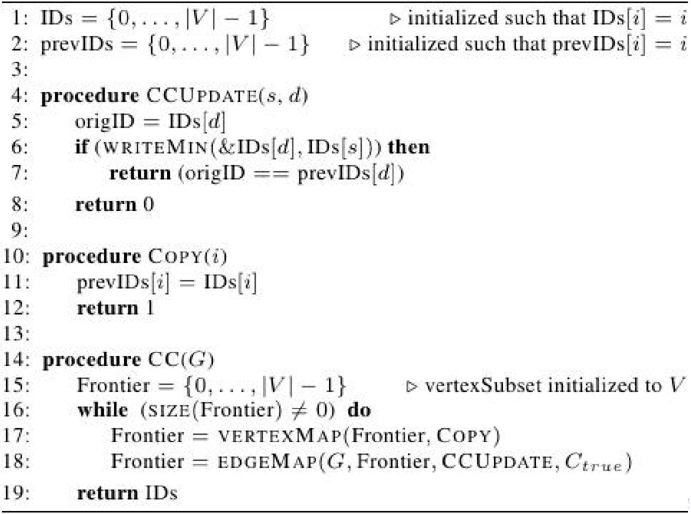
Edge Map Dense

Loops through **vertices in given Subset** U.



```
procedure EDGEMAPSPARSE($G, U, F, C$)
    Out = {}
    parfor each $v \in U$ do
        parfor ngh $\in N^+(v)$ do
            if ($C$(ngh) == 1 and $F(v,$ ngh$)$ == 1) then
                Add ngh to Out
    Remove duplicates from Out
    return Out
```

ligra: connected components



```
$\text{IDs} = \{0, \ldots, |V| - 1\}$                ▷ initialized such that $\text{IDs}[i] = i$
$\text{prevIDs} = \{0, \ldots, |V| - 1\}$         ▷ initialized such that $\text{prevIDs}[i] = i$

procedure CCUpdate($s$, $d$)
    origID = IDs[$d$]
    if (writeMin(&IDs[$d$], IDs[$s$])) then
        return (origID == prevIDs[$d$])
    return 0

procedure Copy($i$)
    prevIDs[$i$] = IDs[$i$]
    return 1

procedure CC($G$)
    Frontier = $\{0, \ldots, |V| - 1\}$          ▷ vertexSubset initialized to $V$
    while (size(Frontier) $\neq 0$) do
        Frontier = vertexMap(Frontier, Copy)
        Frontier = edgeMap($G$, Frontier, CCUpdate, $C_{true}$)
    return IDs
```

adaption of ligra algorithm

Idea

Algorithmic approach stays the same, but EdgeMap function needs not to be as generic as in the ligra version.

adaption of ligra algorithm

Idea

Algorithmic approach stays the same, but EdgeMap function needs not to be as generic as in the ligra version.

- no Condition Function C
- our EdgeMap sparse and dense follow the EMsparse algorithm because our dense version is already significantly faster than ligra
- we change only the data structure for the frontier

Frontier Dense: boolean array

Non-zero entry means vertex in frontier.

+ no race conditions
+ no duplicates in frontier
+ read fully parallelizable
- if frontier gets sparser many 0 entries

Frontier Dense: boolean array

Non-zero entry means vertex in frontier.

- + no race conditions
- + no duplicates in frontier
- + read fully parallelizable
- - if frontier gets sparser many 0 entries

Frontier Sparse: 2D vector

Each thread adds to its own frontier vector.

- + no race conditions, no atomic write needed
- + few for small frontier
- - duplicates may occur. Remove duplicates too expensive
- - implicit barriers slow down read from frontier

Atomic writes

used in two different ways:

- update frontier
    - uses atomic write introduced in OpenMP 3.1[4]

---

[4] ARB OpenMP. *OpenMP Application Program Interface, v. 3.1*. 2008.

Atomic writes

used in two different ways:

- update frontier
    - uses atomic write introduced in OpenMP 3.1[4]
- write minimum id to node
    - uses compare and swap written in inline assembly

---

[4] ARB OpenMP. *OpenMP Application Program Interface, v. 3.1*. 2008.

Read Graph from file

- compute CC $\mathcal{O}(V+E)$ and read graph as well $\mathcal{O}(V+E)$
- $\Rightarrow$ banchmark time is algorithm time only

Read Graph from file

Requirement for reads to not count them in the benchmarking

- No information gain except list of neighbors and out degree while reading
- No sorting of nodes

If that is not fulfilled one can write algorithms that calculate the number of connected components while reading the graph

Sketch of $\mathcal{O}(1)$ CC-Algo

```
Number Components = N
Node(i).id = i

add edge(i,j){
    o1=find orig id(i)
    o2=find orig id(j)
    if(o1!=o2){
        o1=o2
        Number of Components - -
    }
}
```

---

[5] J. Hoshen and R. Kopelman. "Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm". In: *Phys. Rev. B* 14 (8 1976), pp. 3438–3445. DOI: `10.1103/PhysRevB.14.3438`. URL: `http://link.aps.org/doi/10.1103/PhysRevB.14.3438`.

Problem with this

Read with calculation	**Read normal**
Flickr-Graph (19'674'428 Edges)	
2'668'856 μs	1'458'167 μs
Wikipedia Graph (90'060'778 Edges)	
2'238'214'340 μs	11'420'153 μs

Environment and Testgraphs

Test environment

Experiments were preformed on a 32-Core Intel machine with 4x 2.13 GHz Intel 8 Core-Xenon E7 4830 Processors.
The programms were compiled with gcc 4.9.0 with the -O3 flag

Environment and Testgraphs

Test environment

Experiments were preformed on a 32-Core Intel machine with 4x 2.13 GHz Intel 8 Core-Xenon E7 4830 Processors.
The programms were compiled with gcc 4.9.0 with the -O3 flag

rMat Graphs

All GraphX and rMat24 Graph are created with a random Graph generator from problem based benchmark suite[a] with parameters

$$a = 0.5,\ b = 0.1,\ c = 0.1,\ d = 0.3$$

---

[a] Julian Shun et al. "Brief announcement: the problem based benchmark suite". In: *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*. ACM. 2012, pp. 68–70.

Boost parallel preconditions

Tested on small graphs with a given number of cores that communicate over MPI

Graph Parameters:

Graph0:

- V: 131072
- E: 2508284
- CC: 762

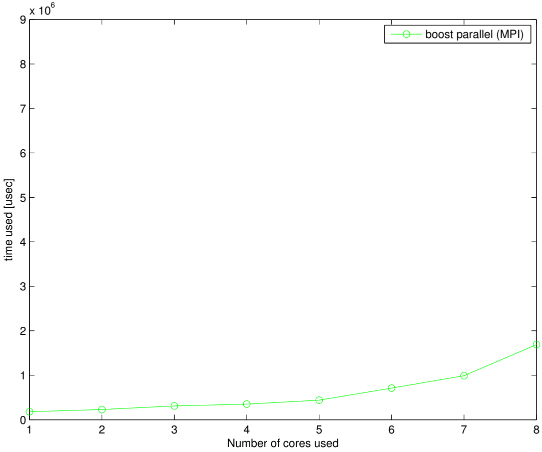
Graph1:

- V: 262144
- E: 5066324
- CC: 1765

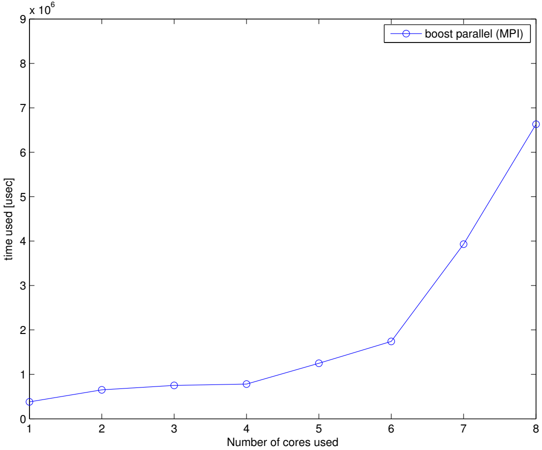
Graph2:

- V: 524288
- E: 10211482
- CC: 4190

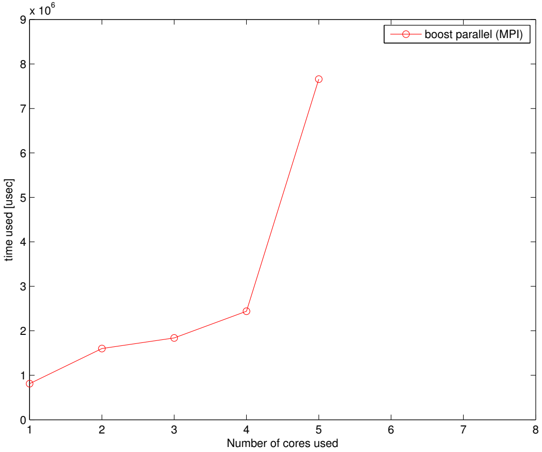
Boost parallel results I



Boost parallel results II



Boost parallel results III



Serial vs Parallel preconditions

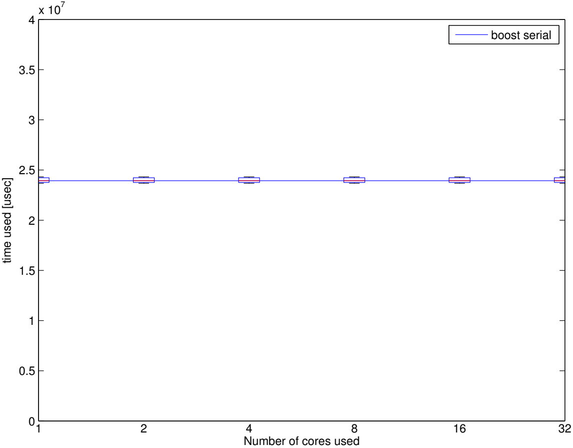
Testing Boost serial against our parallel alogrithm to see if parallelizing is worth:

Graph Parameters:

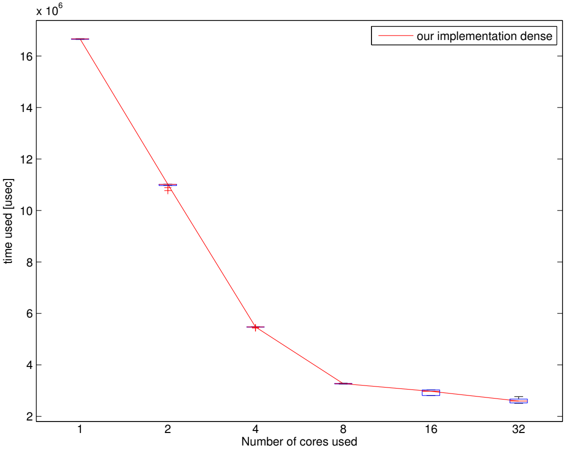
rMat24:

- V: 16777216
- E: 166976680
- CC: 935879

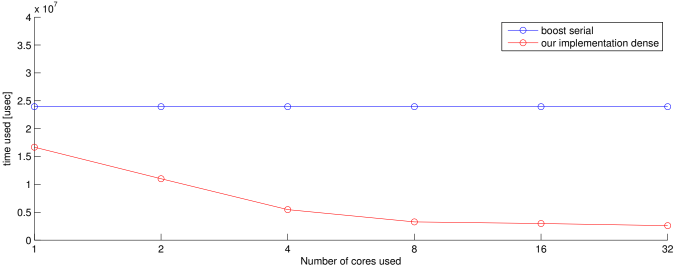
Serial vs Parallel results I



Serial vs Parallel results II



Serial vs Parallel results II



Sparse vs Dense vs Mix preconditions

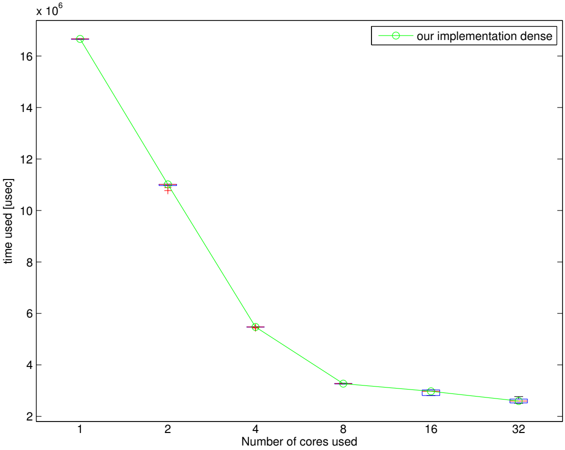
Having a sparse and a dense implementation we wanted to see how they preform against each other

Graph Parameters:

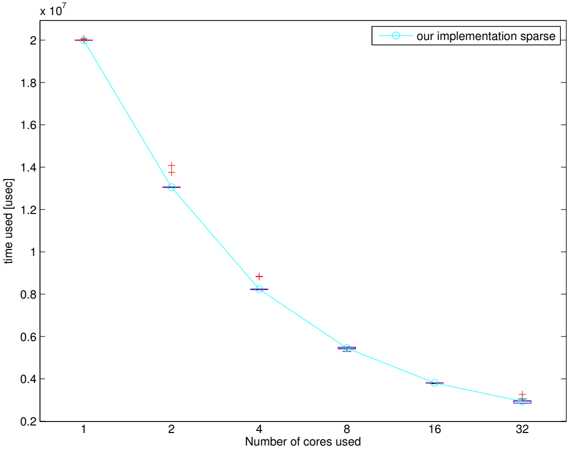
rMat24:

- V: 16777216
- E: 166976680
- CC: 935879

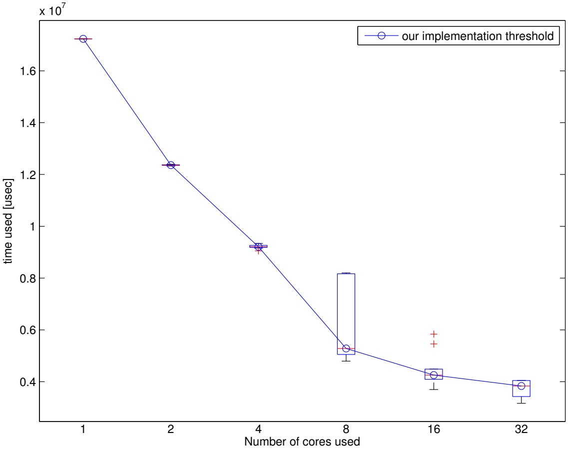
Sparse vs Dense vs Mix results I



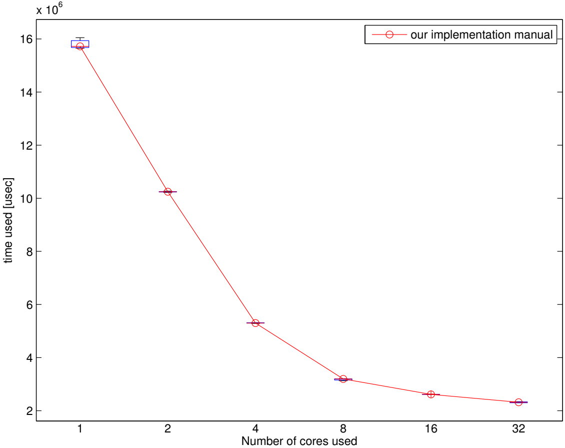
Sparse vs Dense vs Mix results II



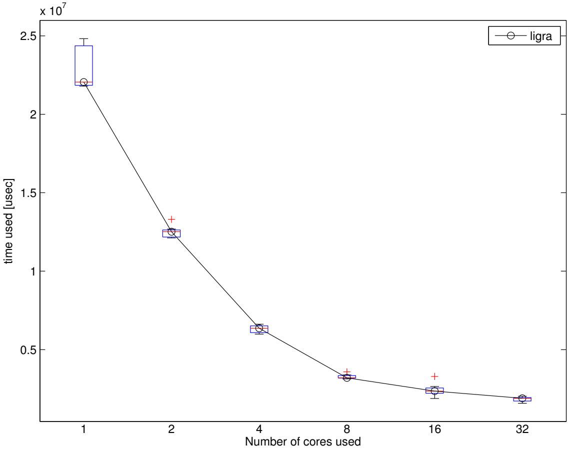
Sparse vs Dense vs Mix results III



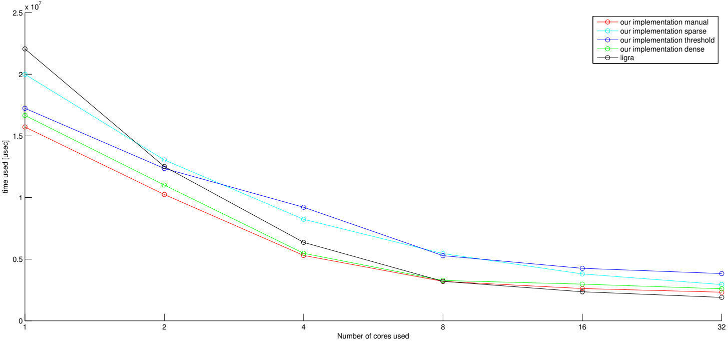
Sparse vs Dense vs Mix results IV



Sparse vs Dense vs Mix results V



Sparse vs Dense vs Mix results VI



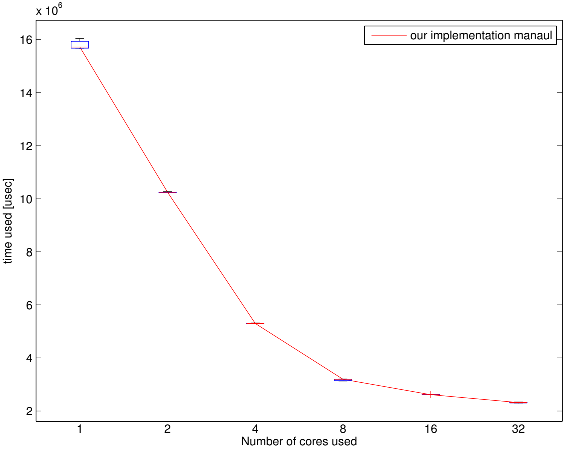
Strong Scaling preconditions

Analyzing the scaling of the algorithms
Graph Parameters:

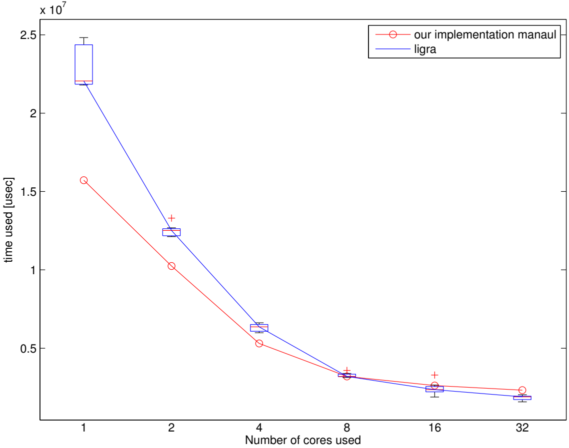
rMat24:

- V: 16777216
- E: 166976680
- CC: 935879

Strong Scaling results I



Strong Scaling results II



Weak Scaling preconditions

Analyzing the scaling of the algorithms:

$$E_w(p) = \frac{T(1)}{T(p)}$$

Graph Parameters:

Graph1:

- V: 262144
- E: 5066324
- CC: 1765

Graph2:

- V: 524288
- E: 10211482
- CC: 4190

Graph3:

- V: 1048576
- E: 20544690
- CC: 9803

Graph4:

- V: 2097152
- E: 41280250
- CC: 22753

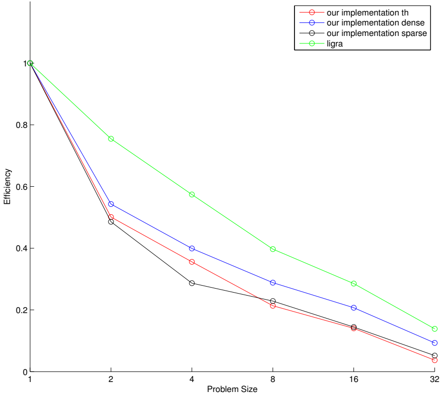
Graph5:

- V: 4194304
- E: 82857080
- CC: 51611

Graph6:

- V: 8388608
- E: 166177454
- CC: 116372

Weak Scaling results



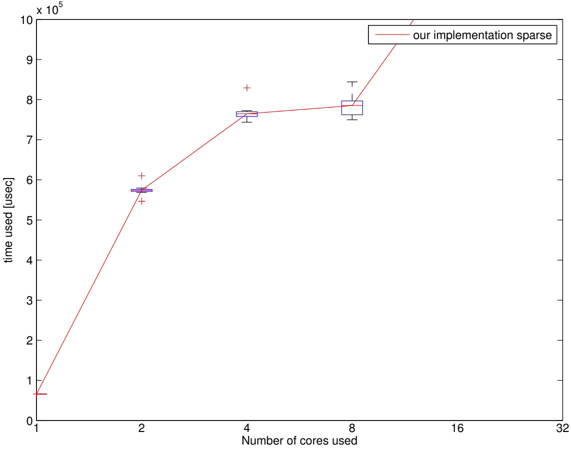
US Roads preconditions

Adapting the algorithms to real world problems:
Graph Parameters:

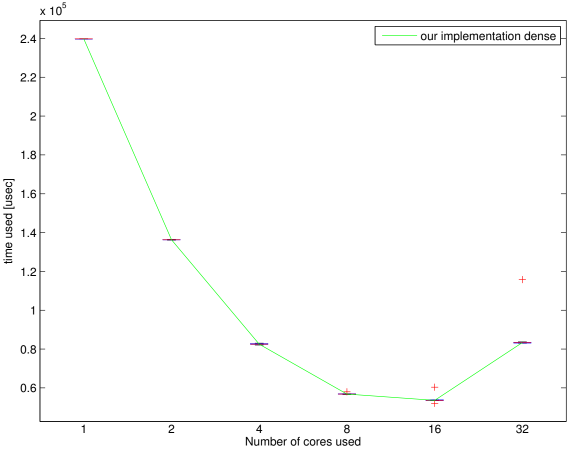
US-Roads:

- V: 129164
- E: 330870
- CC: 56

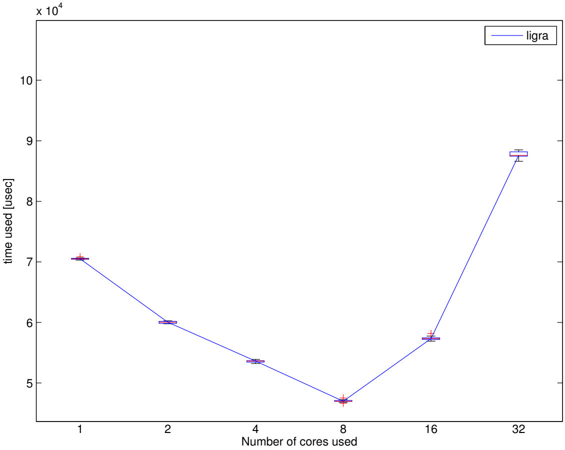
US Roads results I



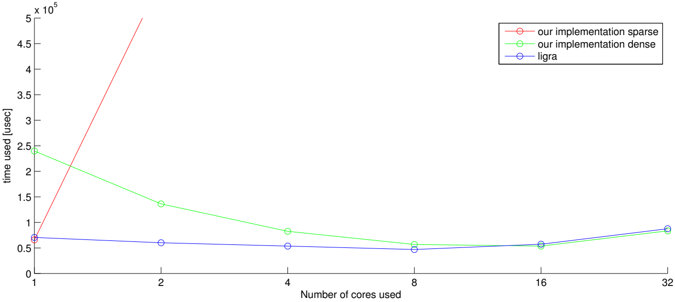
US Roads results II



US Roads results III



US Roads results IV



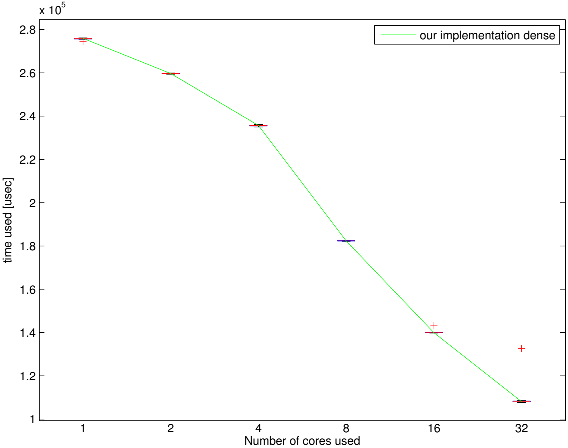
Flickr preconditions

More real world problems: **Graph Parameters:**

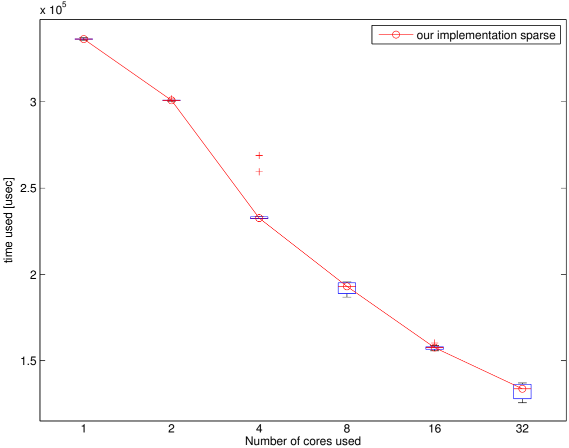
Flickr:

- V: 820878
- E: 19674428
- CC: 1

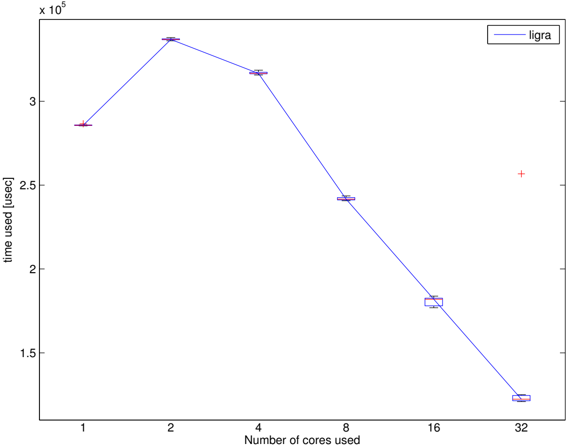
Flickr results I



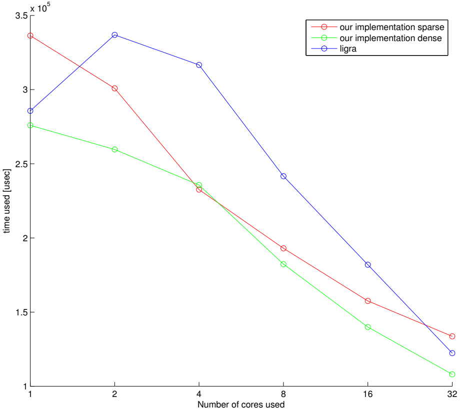
Flickr results II



Flickr results III



Flickr results IV



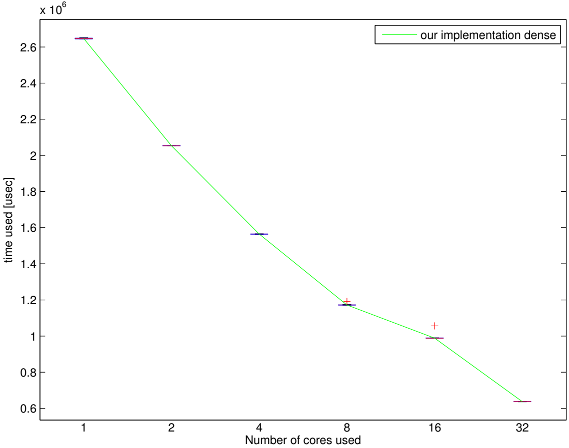
Wikipedia preconditions

More real world problems: **Graph Parameters:**

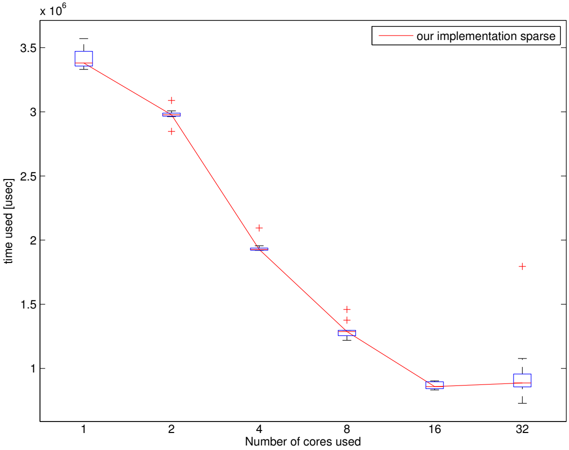
Wikipedia:

- V: 3566907
- E: 90060778
- CC: 52922

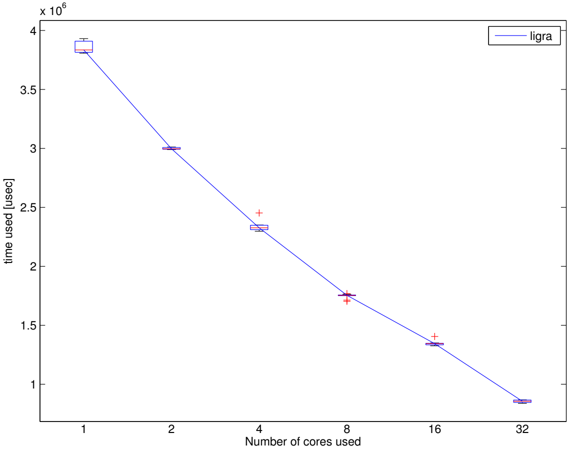
Wikipedia results I



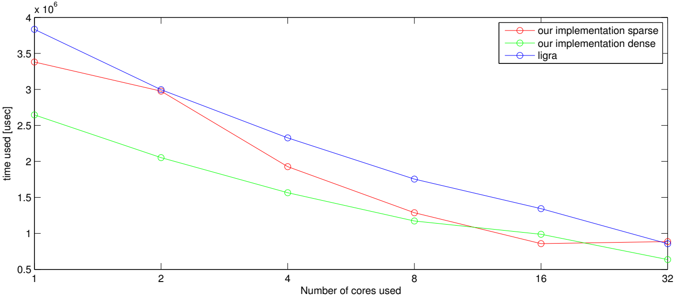
Wikipedia results II



Wikipedia results III



Wikipedia results IV



Conclusion

- we found an implementation which is faster than ligra on some real world graphs
- ligra has a better scaling behavior on rMat graphs, but consumes twice as much memory
- sparse representation has only little advantage because of big overhead for switching between representations