Ankit Goyal
ankitgoyal@utexas.edu

Software Multicore Processors
Homework 3

In this report connected component algorithm has been parallelized using Galois system. First the sequential versions of the algorithm has been implemented using FIFO queue and Priority queue.

Pseudocode:

```
for v in graph.vertices():
  v.compId = v.id
worklist wl(graph. vertices ())
while not wl.empty():
  v = wl.pop()
  for nv in v. neighbors ():
       if v.compId < nv.compId:
            nv.compId = v.compId
            wl.add(nv)
```

Sequential Versions

1. Worklist implemented as FIFO queue

Worklist has been implemented as a FIFO queue using the queue. Following table shows the runtime for each graph.

16777216	13.7021	Rmat
23947347	245.968	usa
33554432	22.2624	Random

2. Worklist implemented as priority queue

Worklist has been implemented as a priority queue where nodes are prioritized based on the component id number. Following table shows the runtime for each graph.

16777216	13.7021	Rmat
23947347	16.90335	usa
33554432	22.2624	Random

Ankit Goyal
ankitgoyal@utexas.edu

Parallel Versions

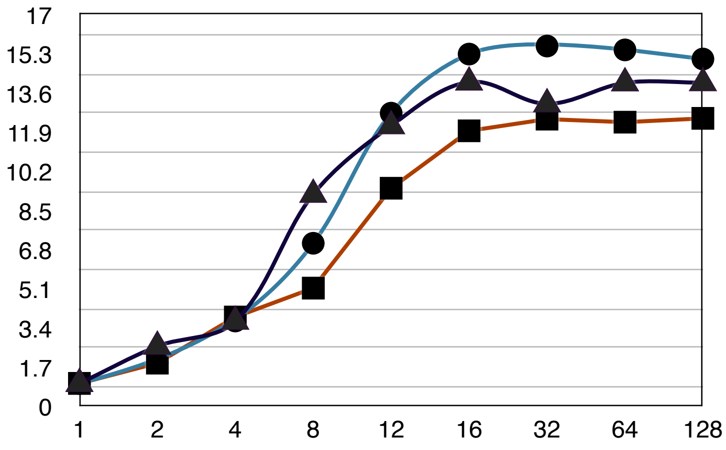
Following parallel versions have been implemented using Galois. Various scheduling policies were tried for worklist. Here are the results for each of them.

1. Worklist implemented as dChunked FIFO queue Ordered by Integer Metric.

Heuristic used for order by integer metric: Each node is assigned priority equal to component_id % chunk_size. This creates lesser number of priority levels and nodes with lower component id are given higher priority.

Table below shows the runtime for each graph for different number of threads.

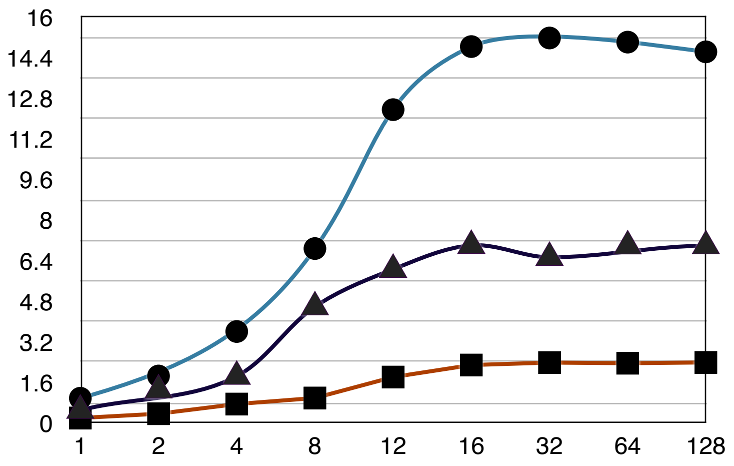
n	1	2	4	8	12	16	32	64	128
$T_n\,(n = 16777210\,)$	27.5978	10.5478	7.45791	2.99623	2.2647	1.95979	2.10062	1.96454	1.96368
$T_n\,(n = 23947347\,)$	253.779	133.296	68.3745	35.8171	19.9314	16.5902	16.2203	16.3963	16.826
$T_n\,(n = 33554432\,)$	116.992	62.123	30.113	22.7609	12.349	9.77683	9.37872	9.47971	9.35529



Scalability

Chunk size: 64.

A significant amount of scalability and speedup is achieved using chunked FIFO queue ordered by integer metric.



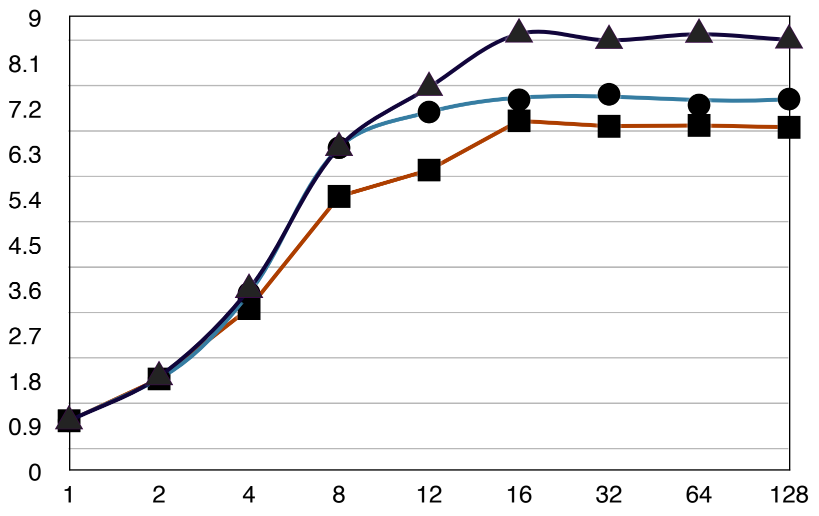
Speedup

Significant amount of speedup can be seen.

Ankit Goyal
ankitgoyal@utexas.edu

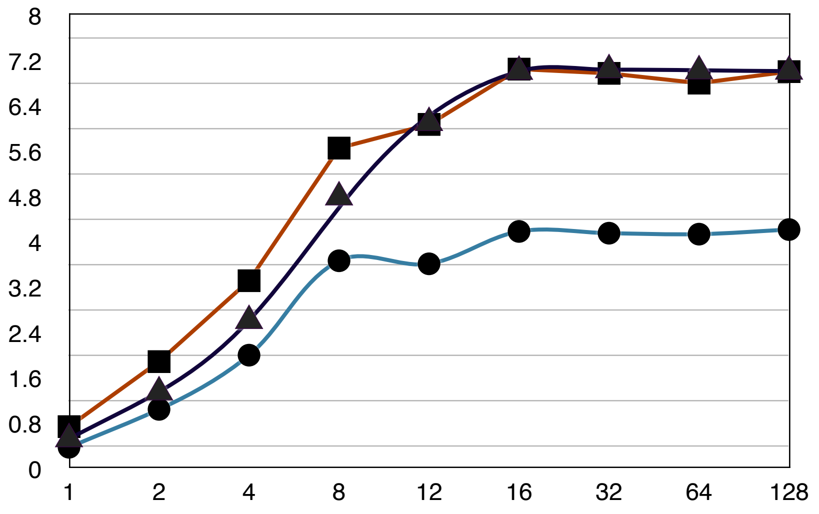
2. ***Worklist implemented as dChunkedFIFO queue without the OBIM.***

n	*1*	2	4	8	12	16	32	64	128
$T_n\,(n = 16777210\,)$	15.4453	8.23198	4.27823	2.40482	2.0283	1.78075	1.8071	1.7818	1.80541
$T_n\,(n = 23947347\,)$	432.281	235.787	122.379	67.3642	60.666	58.6669	57.813	59.505	58.556
$T_n\,(n = 33554432\,)$	21.712	11.8588	6.71972	3.98159	3.6338	3.12376	3.1723	3.1650	3.1821



Scalability

Again the algorithm scales well due to parallelism.



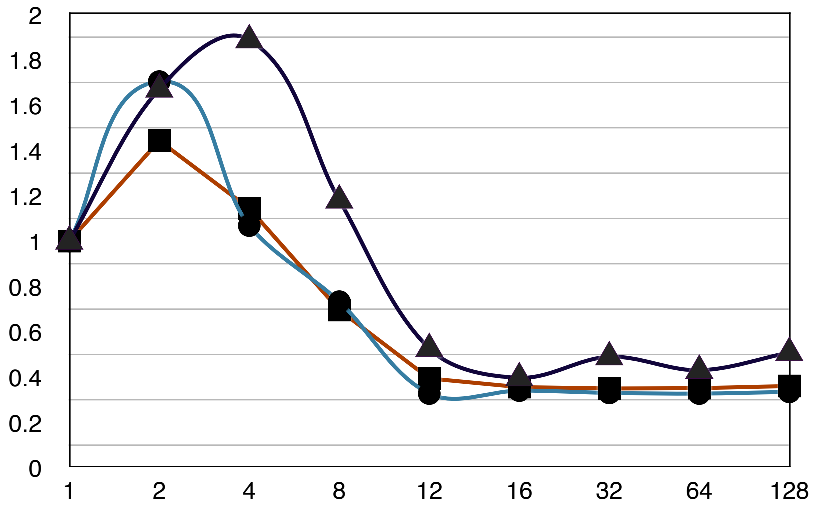
Speedup

A significant amount of speedup can be seen when ran for larger number of threads.

Ankit Goyal
ankitgoyal@utexas.edu

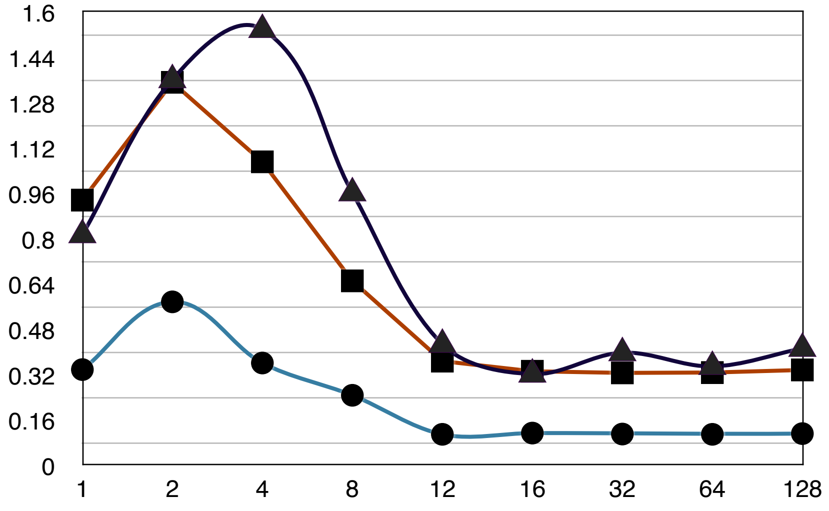
3. *Worklist implemented as* FIFO queue

n(FIFO)	*1*	2	4	8	12	16	32	64	128
$T_n\,(n = 16777210\ )$	16.7943	10.0526	8.89564	14.2192	31.93	42.3024	34.307	39.044	33.036
$T_n\,(n = 23947347\ )$	724.091	425.015	677.792	988.054	2212.2	2127.74	2192.54	2212.54	2162.54
$T_n\,(n = 33554432\ )$	23.7347	16.4475	20.7483	34.1036	60.261	66.6905	67.8724	67.5958	65.8084



Scalability:

Scalability is not achieved due to overhead associated. Since all the threads are competing for the same queue.



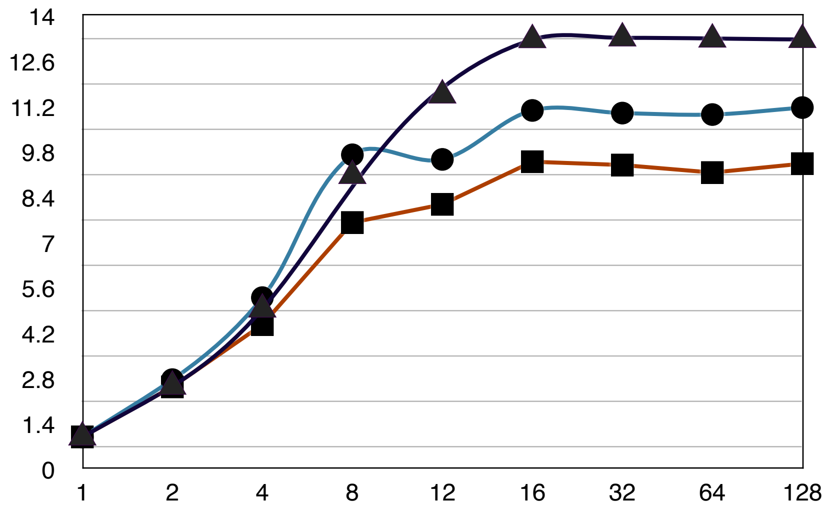
Speedup:

Due to similar reasons as above speedup is not achieved.

Ankit Goyal
ankitgoyal@utexas.edu

4. Worklist implemented as ChunkedFIFO queue

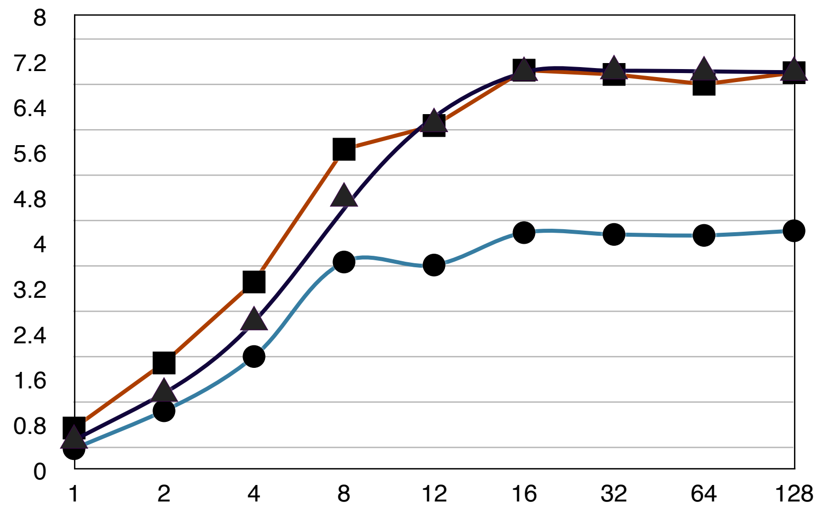
n(chunked fifo)	1	2	4	8	12	16	32	64	128
$T_n\,(n = 16777210\,)$	25.9309	10.0981	5.2382	2.85677	2.2428	1.95398	1.9461	1.9514	1.9531
$T_n\,(n = 23947347\,)$	651.217	235.1	122.813	67.0412	68.039	58.7574	59.192	59.436	58.290
$T_n\,(n = 33554432\,)$	29.9922	11.781	6.71209	3.93461	3.6653	3.15713	3.1913	3.2723	3.1770



Scalability

Again the algorithm scales well due to parallelism.

Chunk size: 64



Speedup

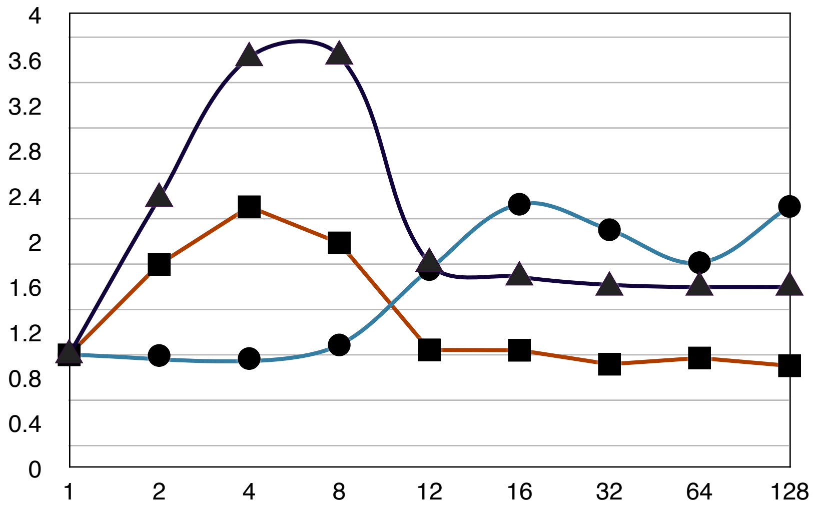
Again the algorithm scales well due to parallelism.

Chunk size: 64

Ankit Goyal
ankitgoyal@utexas.edu

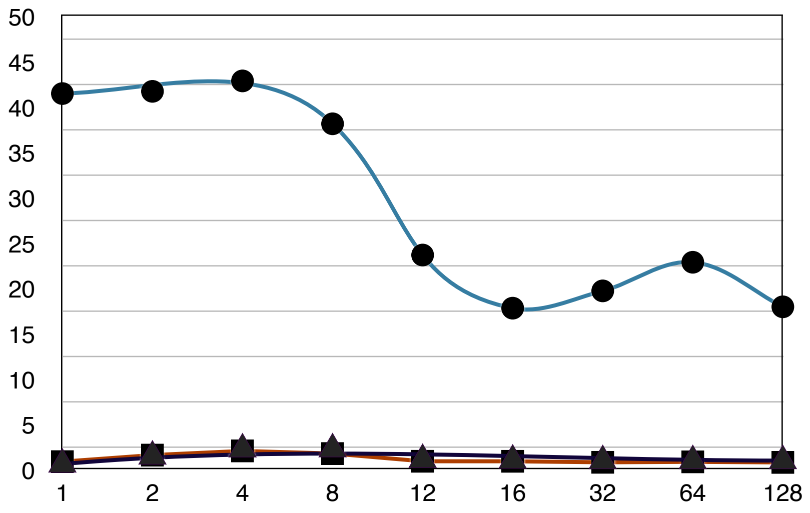
5. Worklist implemented as LocalQueue queue

n(rmat local queue dchunk fifo)	*1*	2	4	8	12	16	32	64	128
$T_n\,(n = 16777210\,)$	21.4701	9.01762	5.93116	5.9072	11.884	12.7285	13.441	13.441	13.441
$T_n\,(n = 23947347\,)$	5.92695	5.89223	5.73414	6.4447	10.382	13.7954	12.4623	10.7439	13.6801
$T_n\,(n = 33554432\,)$	24.6082	13.6828	10.6759	12.3865	23.571	23.6487	26.818	25.3101	27.2372



<u>Scalability</u>

It scales quite well for lesser number of threads in case case of random and rmat graph.
However it scales well for US graph since there are clusters in graph and nodes in the local queue are updated more frequently.



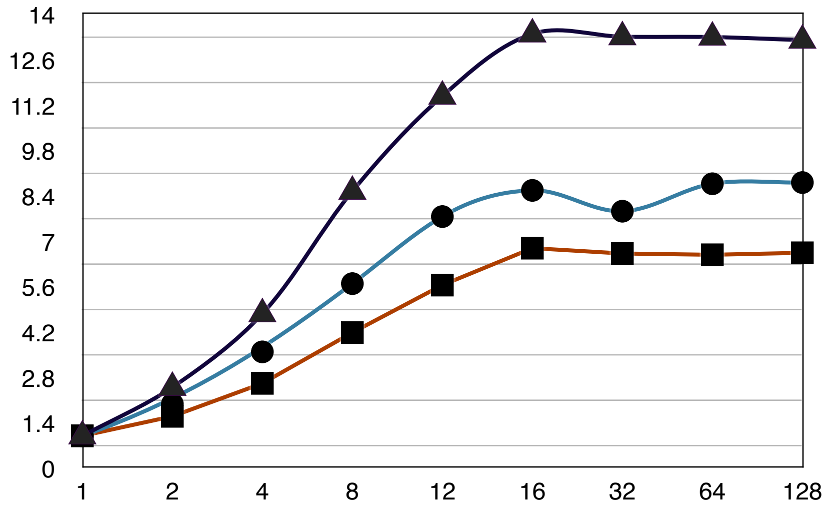
<u>Speedup</u>

Speed is obtained but the overhead takes over in later stages when as there are more threads competing for the same queue.

Ankit Goyal
ankitgoyal@utexas.edu

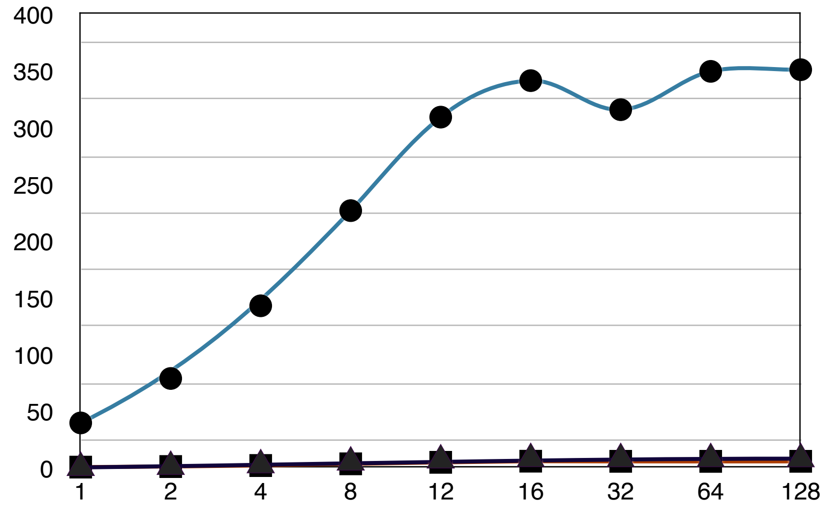
6. Worklist implemented as LocalQueue with dChunkFIFO<64>

n(rmat local queue dchunk fifo)	*1*	2	4	8	12	16	32	64	128
$T_n \, (n = 16777210\ )$	21.7539	8.70306	4.56197	2.54503	1.8945	1.62324	1.6337	1.6351	1.6468
$T_n \, (n = 23947347\ )$	6.17394	3.11272	1.71899	1.08367	0.7949	0.72011	0.77847	0.70328	0.70034
$T_n \, (n = 33554432\ )$	24.8104	15.4735	9.46749	5.92441	4.3896	3.6532	3.74414	3.76669	3.73196



Scalability:

Huge scalability can be seen with the LocalQueue implementation with dChunkedFIFO scheduling.



Speedup:

Speedup for USA road network is exceptionally high since it has many clusters and using this scheduling policy is the optimal.

Ankit Goyal
ankitgoyal@utexas.edu

Observations

The data structure used for parallel implementation of the algorithm in Galois.
Graph was stored as LC_CSR graph, since we only have local computations and the component id was stored as a node data. Various scheduling policies that were experimented with are:

1. dChunkedFIFO with OBIM
2. dChunkedFIFO
3. FIFO
4. ChunkedFIFO
5. LocalQueue with dChunkedFIFO as scheduling policy
6. LocalQueue with default scheduling policy.

It can be seen from the above results that the scheduling policy is input and algorithm dependent. For some inputs like USA-road a huge amount of speedup can be seen in case of LocalQueues with scheduling policy as dChunkedFIFO as compared to the other algorithms.

It can be seen that the scheduling policy of priority queue for USA-road network graph has a dramatically low runtime than the FIFO counterpart. A similar result can be seen in case of parallel implementations also. This is due to the fact that there are very few connected components in the graph(means large clusters) and this means that there is lot of work that needs to be done and since we are initializing the wordlist to be all the nodes then the nodes with updated component id which is smaller are added to the end in case of non-priority scheduling and thus more work is generated. Hence a scheduling policy of priority queue is optimal for the given algorithm. In addition due to lower overhead of threads, LocalQueue with dChunkedFIFO performed quite well since the tasks were ran in parallel.

Overall Galois provides a very easy way to try out different scheduling policies and actually focussing on the algorithm in hand rather than the implementation details and parallelization part. However, it is important to choose the right amount of scheduling policy for a given input.