Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Experiences with Parallel Multi-threaded Network Maximum Flow Algorithm

Seren Soner[a], Can Özturan[a,*]

[a]Computer Engineering Department, Bogazici University, Istanbul, Turkey

Abstract

The problem of computing the maximum flow problem on capacitated networks arises in many application areas. In the area of heterogeneous computing, it arises in job or process scheduling when allocations of resources to jobs/processes need to be tuned. The maximum flow solver is difficult to parallelize. Highly optimized sequential version of maximum flow solvers such as those by Goldberg exists. This work describes how some of the concurrency problems are resolved in our existing Pmaxflow (https://code.google.com/p/pmaxflow) solver. Pmaxflow employs a parallel pre-flow push algorithm to compute the maximum flow. Results of various tests that compare Goldberg's sequential solvers and Pmaxflow on a NUMA shared memory computer are presented.

1. Introduction

Given a network and capacity constraints of the network's links (i.e. edges), the maximum flow problem computes the maximum flow that can be sent from a source vertex s to another vertex t, called the sink. Maximum flow problem is a fundamental problem that arises in many application areas: Computer networks, logistics, bioinformatics, operations research, computer resource allocation and security are just a few areas where maximum flow problem can be made use of. With the coming data deluge, there will be a growing interest from academia and industry for big data analysis tools. Big networks having billions of vertices and edges will soon need to be analysed routinely.

In the area of supercomputing, as we head towards exascale computing, resource allocation problems will be (i) combinatorially more sophisticated because of heterogeneus architectures and (ii) larger in scale because of the increasing number of resources to be managed. As a result, maximum flow solvers need to be made faster and be able to operate on massive networks. Job scheduler algorithms in resource managament systems usually run on a single node. To make them faster, multi-threaded scheduling algorithms can be employed on a single node. In the future, we plan to make use of a maximum flow solver in our Aucsched SLURM scheduler plug-in [1] for deciding how to make heterogeneous resource allocations. This is actually our main reason for working on a maximum flow solver.

This whitepaper is organized as follows: In Section 2, we first present previous work on maximum flow algorithms. In Section 3, we describe modifications we have done to our existing open source Pmaxflow maximum flow solver [2] to solve some concurrency related problems. Results of various tests that compare best sequential solvers and Pmaxflow on a NUMA shared memory computer are presented in Section 4. In Section 5, we present applications of maximum flow solver in scheduling on heterogeneous systems. Finally, in Section 6,the whitepaper is ended with a discussion of results and conclusions.

2. Previous Work

There are many different approaches for solving maximum flow problems. The most notable ones are augmenting path, blocking flow, network simplex method and preflow push-relabel methods [3, p. 240]. Recent efforts to make maximum flow solver faster are progressing along two paths: (i) design of new algorithmic approaches [4, 5]

*Corresponding author.
tel. +90-212-359-7225 fax. +90-212-387-2461 e-mail. ozturaca@boun.edu.tr

and (*ii*) taking existing algorithms and parallelizing them. Highly optimized sequential implementations of a preflow push-relabel network maximum flow solver by Goldberg already exists [6, 7, 8, 9].

Pmaxflow [2] is an open source and LGPL licensed parallel maximum flow solver that was originally developed by Cihan [10, 11] as part of his MS thesis at Bogazici University. It parallelizes Goldberg's pre-flow push based maximum flow algorithm. Unfortunately, the first released version of Pmaxflow code was vulnerable to some subtle concurrency related problems and was also running very slowly on NUMA shared memory computers. This whitepaper describes the modifications we have done to Pmaxflow in order to overcome these problems.

Network maximum flow algorithms are difficult to parallelize. Augmenting path algorithms need to send flow along a path from the source vertex all the way to the target vertex and hence parallel algorithms need to lock all the vertices and edges on the augmenting path. This introduces costly overheads. Pre-flow push based algorithms push flow along edges and not paths. They operate at a finer level which enables (*i*) higher levels of parallelism to be taken advantage of and (*ii*) the use of fast atomic operations to be used instead of expensive locks. On the other hand, due to unstructured nature of networks, it is quite challenging to develop parallel versions that achieve good speedups over the best sequential implementations.

A description of a shared memory parallel maximum flow solver implementation on legacy Sequent Symmetry system appears in Anderson *et al.* [12]. Another parallel implementation of push relabel algorithm is presented in Bader *et al.* [13]. This work was carried out on 14 processor Sun E4500 system. The most notable parallel maximum flow solver work is that of Bo Hong *et al.*'s [14] that contribute an asynchronous multithreaded algorithm with non-blocking global relabeling heuristic. Instead of using expensive locks, this work uses fast atomic operations. In Section 3, we will compare in more detail Bo Hong *et al.*'s algorithm with that employed in Pmaxflow. Recently, an augmenting path maximum flow solver has also been developed using map-reduce by [15] to solve problems on massive social graphs.

3. Parallel Pre-flow Push Algorithm and Concurrency Problems

In this section, we will assume that the reader is familiar with pre-flow push algorithms and the terminology such as active vertices, excess flow, distance labels, push and relabel associated with them. Ahuja *et al.*[3, p. 207] can be consulted for further details.

When processing active vertices (with excess flow), Pmaxflow uses a FIFO selection strategy. Each thread has its own local queue and a task transfer mechanism is used to load balance the threads. Pmaxflow is implemented using Pthreads but it also makes use of Threading Building Blocks (TBB) constructs for atomic variables. The main routine of Pmaxflow basically implements a while loop that is repeated until no more active vertices remain on the local queues of threads. Within the body of the main while loop one or more of the following operations can be executed:

- *Parallel global relabeling* : this is carried out periodically on the whole network by all the threads to compute distance labels of each vertex from the target t vertex. This operation is basically a parallel unweighted breadth-first traversal of the network starting from the sink t vertex. All threads synchronize via a barrier before and after the global relabeling operation. Global relabeling is normally carried out after n (no. of vertices) relabeling operations. However, it can also be triggered by a thread if distance labels property is violated. This can occur as a result of concurrent interleaved executions of push/relabel operations (described in detail later).
- *Task transfer*: a thread can wait until a task is transferred to it by other threads. This is done so as to balance the loads of threads.
- *Push operation*: pushes flow over an edge (p, q) from an excess (active) vertex p to its neighbor vertex q which has lower distance label, i.e. the adjacent neighbor which is closer to the sink t.
- *Relabel operation*: Increases the distance label of an active vertex p to be one higher than the distances of its adjacent vertexs, i.e.,

$$d(p) = min\{d(r) + 1 : (p, r) \ \ is \ \ an \ \ outcoming \ \ arc \ \ of \ \ p\}$$

Figure 1 and Figure 2 show steps of push and relabel operations respectively. Here, we concentrate on concurrent interleaved operations of push and relabel operation which may lead to a violation of distance label properties. This problem was first noticed by Bo Hong *et al.*'s [14]. Consider the example in Figure 3 which illustrates how invalid distance labels may arise when a thread is performing push operation on vertex a while another one is concurrently performing a relabel operation on vertex b. Here, the notation $e(a)$ and $d(a)$ denote excess flow and distance label respectively on vertex a. In Goldberg's algorithm, vertex labels should satisfy the following invariant throughout the execution of the algorithm: Given any arc (p, q), then we should have $d(p) \leq d(q) + 1$. Invalid labeling can potentially happen as follows: While thread 2 that relabels vertex b sees only vertices c and d as its only neighbors, thread 1 can push flow out of a over the arc (a, b) at the same time and hence create a reverse residual (b, a) arc without the relabeling thread 2 being aware of it. The trace that leads to this is as follows:

```
Method push(Thread $t$, Vertex $v$)
 1: for all $arc$ in $v.adj$ do
 2:   if $arc.capacity - arc.flow > 0$ then
 3:     if $arc.w.d < v.d - 1$ then
 4:       $labelingcounter \leftarrow 2 \cdot n$ // trigger global relabeling
 5:       return 0
 6:     end if
 7:   end if
 8: end for
 9: for all $arc$ in $v.adj$ do
10:   if $arc.cap - arc.flow > 0$ then
11:     if $arc.w.d = v.d - 1$ then
12:       $flow \leftarrow min(arc.capacity - arc.flow, v.e)$
13:       $arc.flow \leftarrow arc.flow + flow$
14:       $arc.other.flow \leftarrow arc.other.flow - flow$
15:       $oldE \leftarrow arc.w.e.fetch\_and\_add(flow)$
16:       if $oldE = 0$ and $arc.w.d > 0$ then
17:         push $arc.w$ to $t.queue$
18:       end if
19:       $v.e \leftarrow v.e - flow$
20:       if $v.e = 0$ then
21:         return 0
22:       end if
23:     end if
24:   end if
25: end for
26: return 1
```

Fig. 1: Pseudo-code for push

1. Thread 1: On line 11 in *push*, $arc.w.d$ is accessed atomically to test the condition $arc.w.d = v.d - 1$ which evaluates to true.
2. Thread 2: On line 1 in *relabel*, $v.d$ (i.e. $w.d$ in view of thread 1) is atomically initialized to the maximum value $2 \cdot n$.
3. Thread 2: $arc.flow$ is accessed atomically on line 4 in *relabel* and the condition $arc.cap - arc.flow > 0$ evaluates to true or false (depending on distance value). This is done twice, once for vertex c and once for vertex d.
4. Thread 1: $arc.other.flow$ is updated atomically on line 14 in *push*.
5. Thread 2: distance label $v.d$ (i.e. $w.d$ in view of thread 1) is updated atomically on line 10 in *relabel*.

Pmaxflow code was vulnerable to this subtle problem. In this work, Pmaxflow implementation was redesigned to fix this concurrency problem. Note that Pmaxflow parallelizes Goldberg's original pre-flow push algorithm whereas Bo Hong *et. al.* contributes a modified algorithm. Bo Hong's modification involves changing of a condition under which the push operation is carried out. Goldberg's original algorithm pushes flow from excess vertex to an adjacent vertex whose distance label is one lower. In Bo Hong *et. al.*'s algorithm, however, a push from an excess vertex can be done to an adjacent vertex whose label can be one or more lower. This modification is done in order to correct the the outcome of the problematic concurrent relabeling operation that causes an invariant property obeyed by distance labels to be violated. Pmaxflow still performs pushing to a vertex with a label exactly one lower as in original Goldberg's algorithm and resolves outcome of problematic of concurrent relabeling operations by initiating a global relabeling process. Since global relabeling recomputes distances all over again, the problem of invalid distances are resolved. Note that, the fact that between (1) and (4) events in the trace above, thread 1 executes just a few instructions (namely lines 12 and 13) means that events (2) and (3) of thread 2 should be squeezed between these very close successive events - a situation whose likelyhood can be said to be very low, albeit possible. Therefore, if this situation does indeed occur, correction of it by an expensive process like global relabeling can be justified.

Hence our solution to this problem basically lets the invalid distances situation occur, but introduces statements to detect this later so as to trigger global relabeling. After relabeling of an excess vertex, push operation

```
Method relabel($Vertex\ v$)
$v.d \leftarrow 2 \cdot n$
$wd \leftarrow 2 \cdot n$
for all $arc$ in $v.adj$ do
    if $arc.capacity - arc.flow > 0$ and $arc.w.d < wd$ then
        $wd \leftarrow arc.w.d$
    end if
end for
$wd \leftarrow wd + 1$
if $wd < 2 \cdot n$ then
    $v.d \leftarrow wd$
end if
return $wd$ // new label for $v$
```

Fig. 2: Pseudo-code for relabel

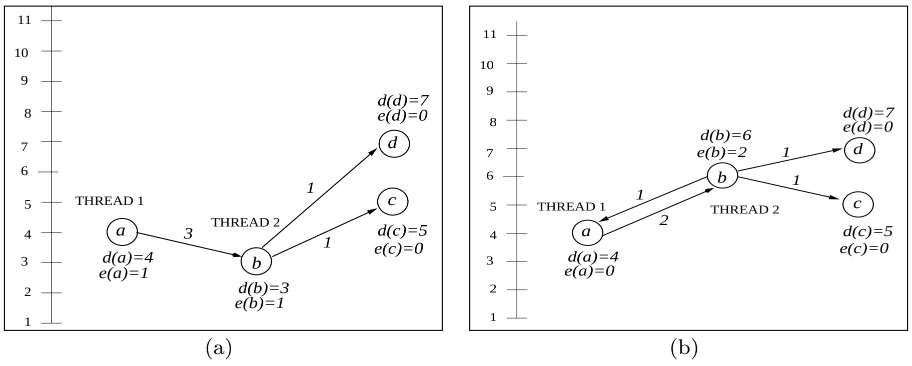


Fig. 3: Formation of invalid distance labels: Flow is pushed on (a, b) while b is being relabeled in figure (a) and reverse arc (b, a) with $d(b) > d(a) + 1$ is formed in figure (b)

can be applied. The statements on lines 1-8 that have been added in the push operation go through the list of neighbors of incorrectly relabeled vertex and checks the distance condition. If it is violated, then the *labelingcounter* is set to a large number which later triggers global relabeling. Note that no flow is pushed from this vertex at this moment and the return statement is executed immediately. After global relabeling, flow can be pushed from this vertex as usual.

Whereas this solution may sound ad-hoc, it has the advantage that it allows us to parallelize Goldberg's original algorithm. Bo Hong et al.'s algorithm finds the maximum flow with $O(n^2 \cdot m)$ push and relabel operations whereas Goldberg's FIFO based algorithm which we have parallelized has lower $O(n^3)$ operations. Note that m is the number of arcs which in the worst case can be $O(n^2)$. Therefore, our implementation can offer advantages especially when the number of arcs is quite large.

4. Tests

We compare the performance of Pmaxflow to Goldberg's sequential F_PRF, H_PRF, HI_PR, HI_PRO and HI_PRW implementations. The characteristics of these implementations are as follows:

- Pmaxflow and F_PRF both employ FIFO active vertex selection scheme and global relabeling.
- H_PRF employs highest label active vertex selection, global relabeling and gap relabeling. The other (HI_PR, HI_PRO and HI_PRW) implementations are derivatives of H_PRF, with heuristic parameters adjusted for better performance on certain network instances [6]. Note that highest label selection refers to taking the vertex with the greatest distance label for processing. Gap relabeling involves recognising those vertices from which the sink is not reachable and increasing their labels to n.

We have conducted our tests on the Curie supercomputer [16], which is owned by GENCI and operated by CEA in France. We have used two types of nodes for our tests: (i) thin nodes and (ii) extra large nodes. The thin nodes have two Intel E5-2680 processors on each node, consisting of 16 cores, and 64 GB of memory [17]. These processors have a peak frequency of 2.7 GHz, therefore we mainly ran our tests on these nodes. The extra large nodes consist of 16 Intel X7560 processors on each node, resulting in 128 cores in total [17]. These fat nodes have 512 GB memory on each node, which allows us to run our parallel maxflow solver on larger graphs. Processors on fat nodes have a peak frequency of 2.26 GHz.

Both sequential Goldberg implementations and Pmaxflow have been compiled with Intel C Compiler version 12.1.7.256. During Pmaxflow compilation, we have also used Intel Thread Building Blocks version 12.1.7.256. The use of ordinary memory allocation malloc/calloc made the code run very slow. To resolve this, we use *numa_alloc* instead of calloc, in order to allow NUMA-aware memory allocation policy. Also, we use CPU affinity mapping, with the core mapping described as in [17] for thin nodes and extra large nodes to optimize Pmaxflow.

We test Pmaxflow by running it on synthetically generated and real life graphs. These are:

- Graphs produced by generators, namely, genrmf, washington and acyclic-dense, which are described in [18].
- Graphs bl06-gargoyle-lrg , bl06-gargoyle-med, lb07-bunny-med, babyface.n6c.100, babyface.n6c.10, bl06-camel-lrg and bl06-camel-med that come from the vision area [19].

For each test given in Tables 1 and 2, we have conducted 9 experiments. Some of the tests in Table 2 could not be performed on thin nodes, since these large graphs did not fit the 64 GB memory on these nodes. In the tables, we report the best timing obtained with Pmaxflow (whatever number of cores it corresponds to). We also report the speed up of Pmaxflow for the large graphs in Figure 4 and Figure 5 on both thin and fat nodes. Note that, on thin nodes, the speed-ups are computed by using Pmaxflow timing on 1 core divided by timing on multiple cores. On fat nodes, the speed-ups are computed by using timing on 8 cores divided by timing on multiple cores. It should also be noted that, HI_PRO cannot find the optimal solution in some cases, for example genrmf-wide 2 in Tables 1 and 2. We have recorded the timings as N/A for these cases.

Table 1: Timings in seconds obtained from Pmaxflow and Goldberg's solvers on Curie's thin nodes

Test	No. of vertices (n)	No. of edges(m)	Pmaxflow	F_PRF	H_PRF	HI_PR	HI_PRO	HI_PRW
genrmf-wide 1	2,560,000	12,614,400	41.27	94.04	44.17	39.02	37.66	36.37
genrmf-wide 2	40,000,000	198,840,000	1471.54	3927.79	3164.24	3658.08	N/A	3554.11
genrmf-long	1,048,576	5,110,784	6.03	9.61	3.1	2.74	2.63	3.05
washington-wide	2,097,154	6,258,688	11.82	22.25	11.07	12.28	10.42	10.45
washington-long	4,194,306	12,582,848	34.23	34.87	0.95	1.9	1.01	1.06
washington-mod	262,146	33,521,502	0.62	0.85	0.55	1.3	0.83	0.84
ac-dense 1	4,096	8,386,560	2.83	6.83	3.72	4.09	5.43	6.39
ac-dense 2	20,000	199,990,000	52.91	131.12	59.71	70.5	102.67	121.5
bl06-gargoyle-lrg	17,203,202	86,175,090	93.55	270.03	216.31	150.29	162.49	134.1
bl06-gargoyle-med	8,847,362	44,398,548	40.45	87.51	75.76	56.05	61.52	49.35
lb07-bunny-med	6,311,088	38,739,041	15.91	21.82	25.26	26.64	21.22	29.01
babyface.n6c.100	5,062,502	30,386,370	18.6	43.1	52.63	42.14	37.61	35.37
babyface.n6c.10	5,062,502	30,386,370	16.8	30.77	32.53	26.23	23.92	25.04
bl06-camel-lrg	18,900,002	93,749,846	115.67	191.01	215.74	199.91	205.91	211.6
bl06-camel-med	9,676,802	47,933,324	49.85	80.08	86.79	73.66	88.54	72.66

When looking at the results obtained on thin nodes, we observe:

- Pmaxflow performs better than Goldberg's implementations on all vision graphs. Vision graphs come from data arranged in a grid. Fewer number of neighbors in such graphs probably imply less atomic operations for locking of shared data, hence, resulting in faster operation.
- On long graphs, flow must be pushed over longer distances to the sink. There is more serial dependence and less parallelism. As a result, this results in long running times of Pmaxflow. Golberg's highest label and gap relabeling implementations perform the best on long graphs.
- For wide, moderate and ac-dense graphs having edges up to a few tens of millions in number, Goldberg's highest label and Pmaxflow performance is more or less similar. When the number of edges are in the 200 million range, then Pmaxflow performs better on wide and ac-dense graphs.
- On thin nodes, we can observe an increasing speed up for ac-dense graphs (see Figures 4(a) and 4(b)). But the speed up falls radically after 8 cores for genrmf-wide graphs (see Figures 4(c) and 4(d)) due to

Table 2: Timings in seconds obtained from Pmaxflow and Goldberg's solvers on Curie's extralarge nodes

Test	No. of vertices (n)	No. of edges(m)	Pmaxflow	F_PRF	H_PRF	HI_PR	HI_PRO	HI_PRW
genrmf-wide 1	2,560,000	12,614,400	99.92	302.26	116.35	84.32	75.11	78.75
genrmf-wide 2	40,000,000	198,840,000	1576.72	3914.22	4526.04	3152.75	N/A	2570.94
ac-dense 1	4,096	8,386,560	19.8	39.23	18.56	12.37	12.82	14.85
ac-dense 2	20,000	199,990,000	291.38	867.03	331.38	219.2	320.73	394.33
ac-dense 3	30,000	449,985,000	851.86	1866.83	864.5	733.86	971.05	1125.08
ac-dense 4	40,000	799,980,000	1018.65	2841.2	1642.25	1281.8	1746.46	2281.95
ac-dense 5	50,000	1,249,975,000	810.21	6366.45	2938.1	2343.91	3789.9	4431.2
genrmf-long	1,048,576	5,110,784	13.75	25.67	7.62	6.57	5.6	6.79
washington-wide	2,097,154	6,258,688	32.6	63.66	27.89	32.71	22	25.94
washington-long	4,194,306	12,582,848	53.53	93.19	3.05	5.37	2.77	2.91
washington-mod	262,146	33,521,502	14.6	3.72	3.24	3.67	2.34	2.15
bl06-gargoyle-lrg	17,203,202	86,175,090	503.28	1134	798.38	463.91	620.4	391.29
bl06-gargoyle-med	8,847,362	44,398,548	202.6	438.85	263.03	191.5	173.57	106.79
lb07-bunny-med	6,311,088	38,739,041	100.36	93.23	88.4	88.15	60.31	90.77
babyface.n6c.100	5,062,502	30,386,370	89.97	86.77	146.28	90.59	96.28	81.83
babyface.n6c.10	5,062,502	30,386,370	92.31	90.86	119.16	81.49	90.75	93.58
bl06-camel-lrg	18,900,002	93,749,846	408.29	835.48	906.96	671.96	521.66	539.9
bl06-camel-med	9,676,802	47,933,324	215.93	297.72	386.28	324.87	326.28	274.58

NUMA memory issues. With over 8 cores threads are distributed over two sockets, each adjacent to a local memory controller and memory bus (32 GB NUMA node). Whereas the reduction in performance can be expected for wide graphs, exactly why it happens in one but not the other needs to be further investigated.

On fat nodes, we have performed the analysis for 8, 16 and 32 cores for Pmaxflow. Our main motivation for the use of fat nodes is to actually run Pmaxflow on extremely large graphs that do not fit on thin nodes. Peak frequency of the processor cores in fat nodes are lower compared to those of thin nodes, hence the solution times may be longer than those on thin nodes. Due to the slowdown implied by the switch between different sockets in the fat node, the maximum number of cores we used is 32. When looking at results obtained on fat nodes, we observe:

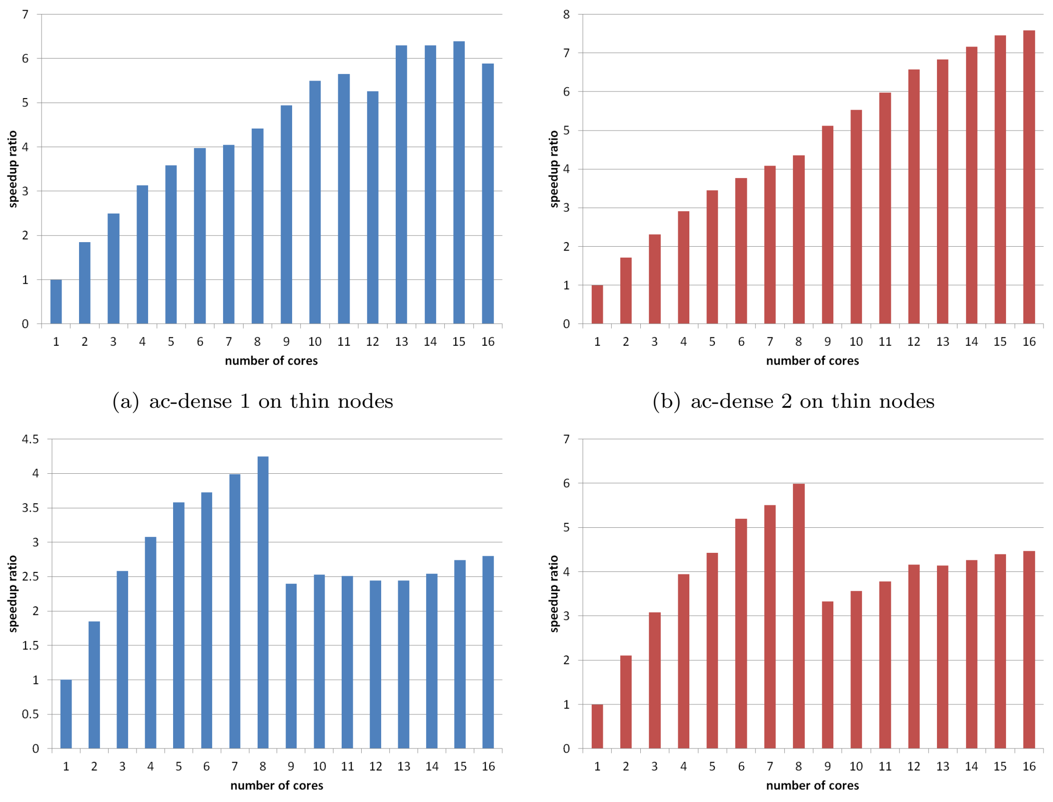
- Maximum flow on a massive 1.25 billion edge ac-dense5 graph can be solved in 13.5 minutes with Pmaxflow whereas Goldberg's HI_PR implementation takes 39.05 minutes.
- On fat nodes, in general, it is becoming more difficult for Pmaxflow to beat HI_PR implementation performance.

In summary, due to highly unstructured nature of the maximum flow problem, we can state that algorithmic heuristics are very important.

5. Applications of Maximum Flow Algorithm to Scheduling on Heterogeneous Systems

Perhaps, one of the most cited work for the application of maximum flow algorithm in the area of heterogeneous computing is Stone's [20]. Stone looks into the problem of optimized assignment of program modules to two processors (not necessarily identical) in a multiprocessor system. Both interprocessor communication as well as execution costs of each module on each processor are considered in the problem. Stone utilizes maximum flow solver to compute a minimum cut in the network constructed for this problem. The minimum cut gives the assignment of modules to processors with the least overall cost. When considering a state-of-the-art heterogeneous system consisting of a CPU core as a host and a GPU as an accelerator, one can possibly make use of Stone's approach to decide the assignment of program modules to the CPU core or the GPU based on expected CPU/GPU execution times and CPU/GPU data transfer times. Our interest in maximum flow solvers is basically motivated by such applications in the area of heterogeneous computing.

We now explain how we can make use of a fast maximum flow solver. Aucsched scheduler plug-in for SLURM [1] implements an auction based scheduler. A window of jobs is taken and bids are generated for each job representing possible allocations. An allocation problem is then solved optimally as an integer programming (IP) problem. Aucsched plug-in also provides support for GPU ranges [21]. This can be very useful to runtime auto-tuning applications/systems that can make use of variable number of GPUs. A maximum flow based heuristic can also be used to implement GPU ranges as well as core ranges. Consider an example heterogeneous system with the following available resources:



(a) ac-dense 1 on thin nodes

(b) ac-dense 2 on thin nodes

(c) genrmf-wide 1 on thin nodes

(d) genrmf-wide 2 on thin nodes

Fig. 4: The speedup ratio of the large graphs on thin nodes

- 3 nodes each having 8 cores and 4 GPUs

and the bids (requests), b_i, for resources as shown in Table 3. Note that such bids may be generated automatically by Aucsched plug-in based on user supplied job submission commands.

Table 3: Bids of jobs on resources (cores and GPUs on nodes 1,2 and 3)

	N_1^{core}	N_2^{core}	N_3^{core}
b_1	4	4	
b_2		2	2
b_3	$2-4$	$2-4$	$2-4$

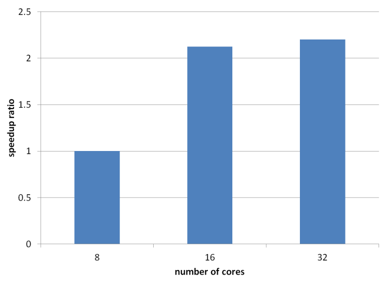
	N_1^{gpu}	N_2^{gpu}	N_3^{gpu}
b_1	$2-4$	$2-4$	
b_2		$2-4$	$2-4$
b_3	$0-4$	$0-4$	$0-4$

If we make the following definitions:

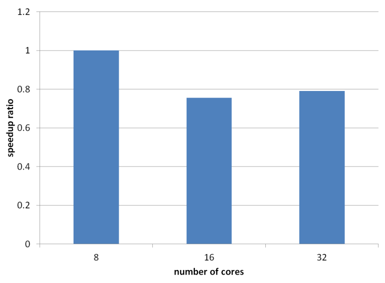
- $R_{j,n}^{mincore}$ and $R_{j,n}^{maxcore}$: minimum and maximum number of cores requested by job j on node n respectively.
- $R_{j,n}^{mingpu}$ and $R_{j,n}^{maxgpu}$: minimum and maximum number of GPUs requested by job j on node n respectively.

a maximum flow network $G(V, E, l, c)$ with vertices V, edges E, lower bound l on each arc and capacity (upper bound) c on each arc can be constructed as follows:

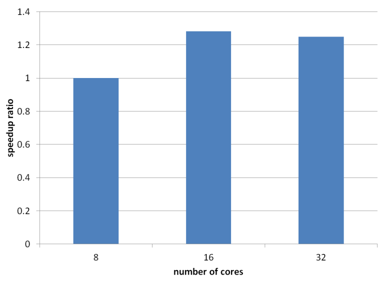
- The vertex set V contains the source vertex s, sink vertex t, the job bids, core and GPU resources on each node (two vertices per node) and the bids of each job.
- The arc set E contains the following arcs:
  - Arcs going from source s to job bid with both lower bound and capacity set to $\sum_{n\in N} R_{j,n}^{mincore} + R_{j,n}^{mingpu}$, $\sum_{n\in N} R_{j,n}^{maxcore} + R_{j,n}^{maxgpu}$, respectively.
  - Arcs going from each job bid to the core vertices with lower bound and capacity set to $R_{j,n}^{mincore}$ and $R_{j,n}^{maxcore}$ respectively.
  - Arcs going from each job bid to the GPU vertices with lower bound and capacity set to $R_{j,n}^{mingpu}$ and $R_{j,n}^{maxgpu}$ respectively.



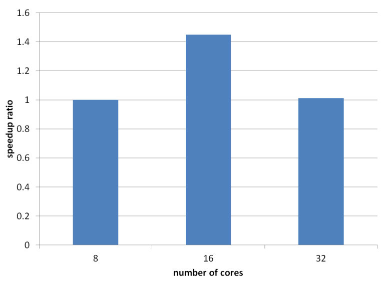
(a) ac-dense 1 on fat nodes



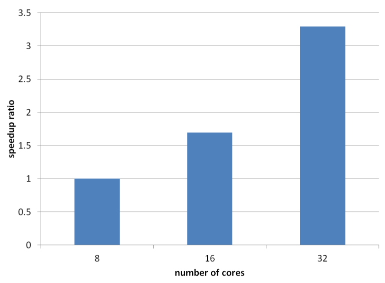
(b) ac-dense 2 on fat nodes



(c) ac-dense 3 on fat nodes



(d) ac-dense 4 on fat nodes



(e) ac-dense 5 on fat nodes

Fig. 5: The speedup ratio of the large graphs on fat nodes

– Arcs going from core and GPU resource vertices to t with lower bound set to 0 and capacity set to the available number of cores and GPUs.

For the example given in Table 3, Figure 6 shows the maximum flow network $G(V, E, l, c)$. A maximum flow network problem with lower bounds can be solved by using a maximum flow solver that works with zero lower bounds (see for example [3, p. 191]).

Note that given a number of job bids, our described procedure will find a feasible maximum flow if it exists. Otherwise, it will conclude that it is not feasible. It will *not* select the subset of bids that lead to feasible maximum flow. This is actually an NP-hard problem that is solved as an IP problem in Aucsched [1]. In SLURM's first-come-first-served (FCFS) scheme, SLURM allocates resources to the first job and then the second etc. Once the allocation decision for the first job is done, it is fixed. SLURM takes the second job and tries allocating resources to it from the available resources at that moment. Note that at this point, there are actually three possibilities (i) the second job can be allocated resources (ii) the second job can be allocated resources if resource allocation of the first job can be adjusted (iii) the second job cannot be allocated resources. For SLURM FCFS scheme, option (ii) is not available since allocation of first job is fixed. Hence, for SLURM, (ii) and (iii) are essentially the same case, i.e. the second job cannot be allocated. In our proposed approach, we can first run the maximum flow solver on the first job only. Then run it on the first and second jobs and repeat the process. If an added job leads to infeasible solution, then it is not allocated resources and not included in the next run. The feasibility checks can be implemented as shown in Figure 7.
Note that the ordered list of jobs can come from a FCFS queue. They can also come from a heuristic solver like a linear programming (LP) relaxation of an IP formulation. The LP solver can return non-integer solutions that can be sorted and the sorted list can be used for repeated application of maximum flow solver.

6. Discussion and Conclusions

Existing sequential pre-flow push based maximum flow solvers work fast. Coming up with parallelized versions of these algorithms that will beat sequential versions and also scale with increasing number of cores is quite challenging. The use of atomic operations on shared data items at the instruction level instead of costly locks over larger code segments is necessary to make the parallel code to run fast. On the other hand, subtle concurrency related problems may arise which may lead to breakdown of assumed conditions in the algorithms.

Our Pmaxflow maximum solver does have better performance than the sequential version on large acyclic-dense and wide graphs. In the future, we plan to use Pmaxflow in our Aucsched scheduler. When we look at the network for the heterogeneous resource allocation application (i.e. Figure 6), we see that the network is acyclic and wide. Therefore, we can expect Pmaxflow to work quite fast on our allocation networks.

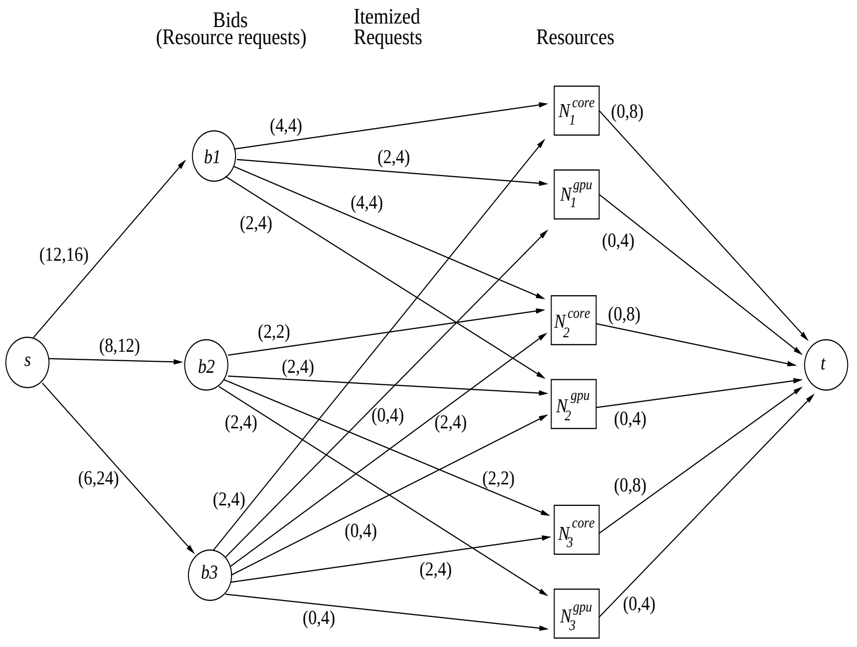


Fig. 6: Maximum flow network for checking feasibility of bids

Allocation Selection Heuristic

```
1: $B \leftarrow [b_1, b_2, \ldots, b_k]$     // a given sequence of bids
2: $S \leftarrow \emptyset$
3: for all $i$ in $1, \ldots, k$ do
4:    $G \leftarrow$ maximum flow network for bids in $S \cup b_i$
5:    if feasible maximum flow solution exists in $G$ then
6:       $S \leftarrow S \cup \{b_i\}$
7:    end if
8: end for
9: Output $S$
```

Fig. 7: Repeated feasibility checks with maximum flow

Acknowledgements

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The work was carried out using the PRACE Research Infrastructure resources (the Curie supercomputer at CEA in France).

References

1. S. Soner and C. Ozturan. An auction based slurm scheduler for heterogeneous supercomputers and its comparative performance study. Technical report, PRACE, 2013. `hhttp://www.prace-project.eu/IMG/pdf/wp59_an_auction_based_slurm_scheduler_heterogeneous_supercomputers_and_its_comparative_study.pdf`.
2. Parallel push - relabel maximum flow solver for multi-core machines. Online. `https://code.google.com/p/pmaxflow/`.
3. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, New Jersey, NJ, USA, 1993.
4. Larry Hardesty. Maximum plus. *MIT Technology Review*, 2010. http://www.technologyreview.com/article/422106/maximum-plus/.
5. P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S. Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings*

of the 43rd annual ACM symposium on Theory of computing, STOC '11, pages 273–282, 2011. http://www.technologyreview.com/article/422106/maximum-plus/.

6. Andrew Goldberg. Andrew goldberg's network optimization library. Online. `http://web.archive.org/web/20080305024609/http://avglab.com/andrew/soft.html`.
7. A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.
8. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.
9. B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.
10. S. Cihan. Parallel maximum flow solver for multi-core machines. Master's thesis, Bogazici University, Computer Engineering Department, 2010.
11. S. Cihan and C. Ozturan. Parallel maximum flow solver for multicore architectures. In *II. High Performance and Grid Computing Conference (Basarim 2010)*, 2010.
12. R. J. Anderson and J. C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of Parallel and Distributed Computing*, 29(1):17–26, 1995.
13. D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. *Proc. 18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, 2005.
14. Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1025–1033, 2011.
15. F. Halim, R.H.C. Yap, and Yongzheng Wu. A mapreduce-based maximum-flow algorithm for large small-world network graphs. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 192–202, 2011.
16. TGCC Curie. Online. `http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm`.
17. Best Practice Guide - Curie, version 1.16. Online. `http://www.prace-project.eu/Best-Practice-Guide-Curie-HTML`.
18. Synthetic Maximum Flow Families. Online. `http://web.archive.org/web/20080508053755/http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm`.
19. Computer Vision at Western - Max-flow problem instances in vision. Online. `http://vision.csd.uwo.ca/maxflow-data/`.
20. Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *Software Engineering, IEEE Transactions on*, SE-3(1):85–93, 1977.
21. S. Soner and C. Ozturan. An auction based slurm scheduler for heterogeneous supercomputers and its comparative performance study. Technical report, PRACE, 2013.